Diplomarbeit

*

Design and Implementation of a FCC Schedule Planner for an unmanned aircraft demonstrator

*

Matthias Friedrich

13.03.2006

angefertigt bei der Fa. EADS, München

ausgegeben von

Prof. Dr. rer. nat. Klaus Schneider

Fachbereich Informatik * Universität Kaiserslautern

Postfach 3049 * 67653 Kaiserslautern

Erklärung

Hiermit erkläre ich, Matthias Friedrich, Matrikelnummer 336284, dass ich diese Arbeit selbständig verfasst habe, noch nicht anderweitig für Prüfungszwecke vorgelegt sowie keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt habe.

Matthias Friedrich - München, den 13.03.2006

Inhaltsverzeichnis

1 Einleitung **3**
- 1.1 Motivation . . . 3
- 1.2 Aufgabenstellung . . . 3
- 1.3 Gliederung der Arbeit . . . 4

2 Grundlagen **5**
- 2.1 Scheduling . . . 5
- 2.2 Backtracking . . . 6

3 Problembeschreibung **9**
- 3.1 FCS Hardware . . . 9
- 3.2 FCS Software . . . 10
  - 3.2.1 Scheduler und Dispatcher . . . 10
  - 3.2.2 Busnachrichten . . . 12
  - 3.2.3 Constraints . . . 12

4 Konzept **15**
- 4.1 Übersicht . . . 15
- 4.2 Systemmodell . . . 18

5 Implementierung **22**
- 5.1 Modellierung und Visualisierung . . . 22
- 5.2 Schedule Planner . . . 25
- 5.3 Schedulability Analyzer . . . 36
- 5.4 WCET Tracker . . . 41

6 Ergebnisse **47**
- 6.1 Test des FCC-Schedules . . . 47
- 6.2 Exemplarische Schedules . . . 49

7 Schlussbetrachtung **55**
- 7.1 Zusammenfassung . . . 55
- 7.2 Ausblick . . . 57

Tabellenverzeichnis **59**

Abbildungsverzeichnis **60**

Glossar **61**

Literatur **62**

1 Einleitung

1.1 Motivation

Die Forderung nach immer leistungsfähigeren Fluggeräten führt zwangsläufig zu einer stetigen Komplexitätssteigerung der dafür eingesetzten Kontrollcomputer. Dies gilt insbesondere für Flugzeuge neuerer Generationen, die ohne ein funktionierendes Flight Control System (FCS) nicht flugfähig wären.

Die hohe Komplexität spiegelt sich sowohl in der Hardware als auch in der verwendeten Software wieder. Das Flight Control System des Unmanned Aircraft Demonstrators (UAD) , das für diese Arbeit den praktischen Hintergrund darstellt, besteht aus drei identischen Flight Control Computern (FCC) und einer Vielzahl von Sensoren und Aktuatoren, die mittels eines Bussystems angekoppelt sind.

Die Vielzahl an Tasks und Busnachrichten, für die eine zeitliche Planung unabdingbar ist, deren Abhängigkeiten untereinander sowie die Kopplung mehrerer Prozessoren und FCCs machen das manuelle Erstellen solcher Pläne zu einem zeitaufwändigen und fehleranfälligen Verfahren. Auch der Beweis, dass ein auf manuellem Weg generierter Schedule korrekt ist, muss mit Hilfe von langwierigen Reviews geführt werden. Aufgrund dieser Tatsachen entstand die Forderung nach einem Tool, das den Softwareentwickler bei der Entwicklung der Schedules weitestgehend entlastet und ihn bei der Verifizierung unterstützt.

1.2 Aufgabenstellung

Ziel dieser Arbeit ist es, ein Tool zu entwickeln, welches das Erstellen von Schedules für Tasks und Busnachrichten eines Flight Control Computers automatisiert. Außerdem ist es notwendig, die so erzeugten Pläne unabhängig von dem Generierungsprozess gegen die systembedingten Abhängigkeiten zu verifizieren.

Die Arbeit unterteilt sich in einen theoretischen und einen praktischen Teil. Der theoretische Teil beinhaltet folgende Punkte:

- Das Design eines generischen Systemmodells, mit dem die Komponenten eines FCC und deren Beziehungen untereinander modelliert werden können,
- die Beschreibung dieses Systemmodells mittels einer graphischen Beschreibungssprache, beispielsweise UML, und als XML-Schema, um die Interoperabilität des Modells zu gewährleisten,
- Konzept und Design des Schedule-Planning-Algorithmus
- Konzept und Design des Schedulability-Analyse-Algorithmus

Der praktische Teil behandelt

- die Instanziierung eines Flight Control Systems des UADs,
- die Implementierung des Schedule-Planners in Java ,
- die Implementierung des Schedulability-Analyzers als eigenständiges Tool in Java und
- die exemplarische Planung und Analyse der FCS-Schedules.

Da Software, die in Flugzeugen Steuer- und Regelungsaufgaben übernimmt, besonderen Anforderungen in punkto Sicherheit unterliegt, soll der Schedulability -Analyzer nach Kriterien entwickelt werden, die seine spätere Qualifikation nach RTCA/DO-187B [7][23] ermöglichen.

1.3 Gliederung der Arbeit

Die Arbeit gliedert sich in sechs Teile. Kapitel 2 erörtert die nötigen Grundlagen, die für Design und Entwicklung des Schedule-Planners notwendig sind. Abschnitt 1 behandelt Scheduling im Allgemeinen und bisherige Lösungsansätze in diesem Themenbereich, Abschnitt 2 geht näher auf Backtracking ein, eine Problemlösungsmethodik, die bei einem großen Teil der existierenden Scheduling-Algorithmen zum Einsatz kommt.

Kapitel 3 beschreibt den Kontext, in dem sich die Aufgabenstellung bewegt. Sowohl die Hardware, auf der die generierten Pläne zum Einsatz kommen sollen, als auch die Software des FCS haben entscheidenden Einfluss auf Design und Implementierung.

In Kapitel 4 wird das Konzept vorgestellt, nach dem die Aufgabenstellung umgesetzt wird. Eine Übersicht erläutert die vollständige Werkzeugkette, in die sich die zu entwickelnden Module einfügen. Das Systemmodell, das im zweiten Abschnitt beschrieben wird, dient als Beschreibungsgrundlage des Systems für alle im weiteren Verlauf der Arbeit erstellten Softwareprodukte.

Kapitel 5 geht im Detail auf die eigentliche Umsetzung des Konzeptes ein. Für die entwickelten Module werden die verwendeten Algorithmen, Methoden und Strategien zur Umsetzung des Konzepts präzise beschrieben.

Kapitel 6 fasst die gewonnenen Ergebnisse zusammen und Kapitel 7 gibt einen Ausblick auf weiterführende Entwicklungen und Verbesserungen.

2 Grundlagen

2.1 Scheduling

Unter Scheduling (englisch für "Zeitplanerstellung") versteht man ein Verfahren zur Erstellung eines Planes, der die zeitliche Zuteilung eines oder mehrerer Prozessoren zu Prozessen festlegt. Umgesetzt wird dieser Plan von einem Dispatcher.

Unter einem Prozess versteht man im Allgemeinen den Ablauf eines Programms. Ein System kann aus mehreren Prozessen bestehen, die nacheinander oder parallel abgearbeitet werden. Häufig wird auch der Begriff "Task" für einen gekapselten Programmteil im Zusammenhang mit Scheduling verwendet.

Folgende Kriterien beeinflussen das Scheduling-Verfahren:

Harte und weiche Echtzeitanforderungen: Abhängig von der Klassifizierung des Systems, auf dem die vom Scheduler generierten Pläne zum Einsatz kommen, ändern sich auch die Anforderungen an diese Pläne. Ein Echtzeitsystem wird allgemein als System definiert, das innerhalb einer definierten zeitlichen Schranke reagieren muss. Sämtliche Ergebnisse müssen also rechtzeitig vorliegen. Von harter Echtzeit spricht man, wenn die definierten Zeitschranken unter keinen Umständen verletzt werden dürfen, da dies katastrophale Auswirkungen haben könnte. Dies ist bei Systemen, die bei der Steuerung von Fluggeräten zum Einsatz kommen der Fall.

Bei Systemen mit weichen Echtzeitanforderungen führt eine verspätete Reaktion nicht direkt zur Katastrophe, vermindert jedoch die Qualität der Ergebnisse. Beispiel hierfür wäre ein Bankautomat, bei dem der Benutzer eventuell länger als geplant auf eine Ausgabe warten muss.

Ein Scheduler muss sich nach der Klassifizierung des Systems richten und die entsprechende Anforderung gewährleisten.

Statisches und dynamisches Scheduling: Beim dynamischen Scheduling fallen die für die Planung relevanten Daten, wie beispielsweise Bereitzeiten oder Zeitschranken erst zur Laufzeit an. Der Scheduler muss also während der Laufzeit Scheduling-Entscheidungen treffen. Je nach Algorithmus kann die dadurch benötigte Rechenleistung nicht unerheblich sein.

Im Gegensatz dazu stehen beim statischen Scheduling sämtliche benötigten Daten vor dem Start der Software bereit. Der eigentliche Scheduling-Algorithmus muss nicht auf der Zielplattform implementiert sein; es ist möglich, sämtliche Pläne offline, also vor der eigentlichen Laufzeit zu erzeugen. Dabei werden die vollständig geplanten Schedules an den Dispatcher übergeben, der den dort festgelegten zeitlichen Ablauf umsetzt.

Explizites und implizites Scheduling: Wird vom Scheduler ein fertiger Plan an den Dispatcher übergeben, bezeichnet man dies als explizites Scheduling. Generiert der Scheduler im Gegensatz dazu nur Planungsregeln, wie beispielsweise Prioritäten für Tasks, nennt man dieses Verfahren implizites Scheduling. Dies kann vor allem dann sinnvoll sein, wenn neue Prozesse dynamisch eingeplant werden müssen oder wenn ein Prozess häufig unterbrochen wird.

Scheduling für unterbrechbare und nicht-unterbrechbare Prozesse: Ein weiteres Unterscheidungskriterium, das Einfluss auf den Scheduling-Algorithmus hat, ist die Frage, ob die einzuplanenden Prozesse unterbrochen werden dürfen. Ist dies der Fall, kann dem gerade aktiven Prozess jederzeit der Prozessor entzogen werden.

Ein nicht-unterbrechbarer Prozess entscheidet selbständig, wann er die belegten Ressourcen wieder freigibt. In der Regel geschieht das zu dem Zeitpunkt, an dem der Prozess seine Arbeit vollendet hat und terminiert.

2.2 Backtracking

Ein möglicher Weg, Schedulingprobleme zu lösen, ist das so genannte Backtracking. Backtracking bezeichnet eine Problemlösungsmethode innerhalb der Algorithmik, die nach dem "Versuch-und-Irrtum-Prinzip" (*trial and error*) vorgeht. Dabei wird versucht, eine Teillösung schrittweise zu einer Gesamtlösung zu entwickeln. Falls sich herausstellt, dass eine Teillösung nicht mehr zur Gesamtlösung führen kann, wird der letzte Schritt zurückgenommen und ein anderer Weg probiert. Dieses Vorgehen stellt sicher, dass alle vorhandenen Möglichkeiten durchprobiert werden. Die Ausgabe des Algorithmus ist also entweder die Gesamtlösung oder die Aussage, dass diese nicht existiert.

Backtracking hat im schlechtesten Fall mit $O(z^n)$ exponentielle Laufzeit, wobei z die Anzahl möglicher Verzweigungen und n die maximale Tiefe des Lösungsbaums beschreiben. Ist n groß, ist es meistens nicht praktikabel, den kompletten Lösungsraum zu durchsuchen, da dies zu viel Zeit in Anspruch nehmen würde. Die Zeitkomplexität lässt sich jedoch durch Heuristiken oder der Akzeptanz einer Näherungslösung verringern.

Der Backtracking-Algorithmus wird typischerweise rekursiv programmiert, lässt sich jedoch auch iterativ formulieren. Dies hat vor allem in punkto Testbarkeit einige Vorteile, setzt jedoch die explizite Verwendung eines Stacks voraus, der den Verlauf der Lösungsfindung speichert[25]:

Listing 1: Backtracking

```
boolean solve(node n) {
    put node n on the stack;
    while the stack is not empty {
        if the node at the top of the stack is a leaf {
            if it is a goal node, return true
            else pop it off the stack
        }
        else {
            if the node at the top of the stack has untried
                                                    children
                push the next untried child onto the stack
            else pop the node off the stack

    }
    return false
}
```

Falls der aktuelle Knoten, nämlich der Knoten, der obersten Stelle des Stacks liegt, ein Blatt ist und somit das Ende eines Lösungspfades erreicht wurde, kann dies zwei Ursachen haben: Die gefundene Lösung ist die Gesamtlösung, dann wird der Algorithmus beendet, oder es handelt sich um eine Sackgasse, dann wird der letzte Lösungsschritt vom Stack genommen.

Ist der aktuelle Knoten kein Blatt und hat noch Nachfolger, die bei der Lösungsfindung nicht betrachtet wurden, wird einer dieser Nachfolger auf den Stack gelegt, in der nächsten Iteration zum aktuellen Knoten und damit Teil der Teillösung. Ist der aktuelle Knoten kein Blatt, hat jedoch keine Nachfolger, wird der Knoten vom Stack genommen, da dieser Weg nicht mehr zu einer Gesamtlösung führen kann. Diese Schritte werden durchgeführt, solange der Stack nicht leer ist. Tritt dieser Fall ein, bedeutet dies, dass der Lösungsraum komplett durchsucht wurde und keine Lösung existiert.

Terminiert der Algorithmus erfolgreich, repräsentiert der Stack einen möglichen Lösungsweg.

Zur Optimierung der Performance von Backtracking-Algorithmen existieren diverse Möglichkeiten. Eine Übersicht über einige dieser Methoden liefert [10].

Erkennt der klassische Backtracking-Algorithmus, dass der aktuell beschrittene Ast im Lösungsraum nicht zu einer Gesamtlösung führt, wird die letzte Entscheidung verworfen. Es kann jedoch vorkommen, dass die Ursache für den Konflikt, der an einem Blatt des Lösungsbaumes erkannt wird, bereits zu Anfang des Backtrackings entstanden ist. In diesem Fall ist es sinnvoll, nicht nur einen Schritt zurückzugehen, sondern den Verursacher des Konfliktes zu identifizieren und direkt zu diesem Punkt im Lösungsbaum zu springen. Da so mehrere Knoten übersprungen werden, bezeichnet

man dieses Verfahren als Backjumping.

Forward Checking verwendet eine look-ahead-Strategie zum frühzeitigen Erkennen von Sackgassen während des Backtrackings. Es ergänzt das Backtracking so um eine vorausschauende Komponente, welche die Auswirkungen der aktuellen Entscheidung auf die noch offenen Entscheidungen untersucht. Durch diese Methode kann unter Umständen sehr früh festgestellt werden, dass eine Teillösung nicht mehr zu einer Gesamtlösung führen kann.

3 Problembeschreibung

3.1 FCS Hardware

Der folgende Abschnitt beschreibt die für die Entwicklung des Schedulers hardwareseitigen Voraussetzungen. Von Komponenten, die keinen Einfluss auf die Betrachtungen des Schedulers haben, wird abstrahiert.

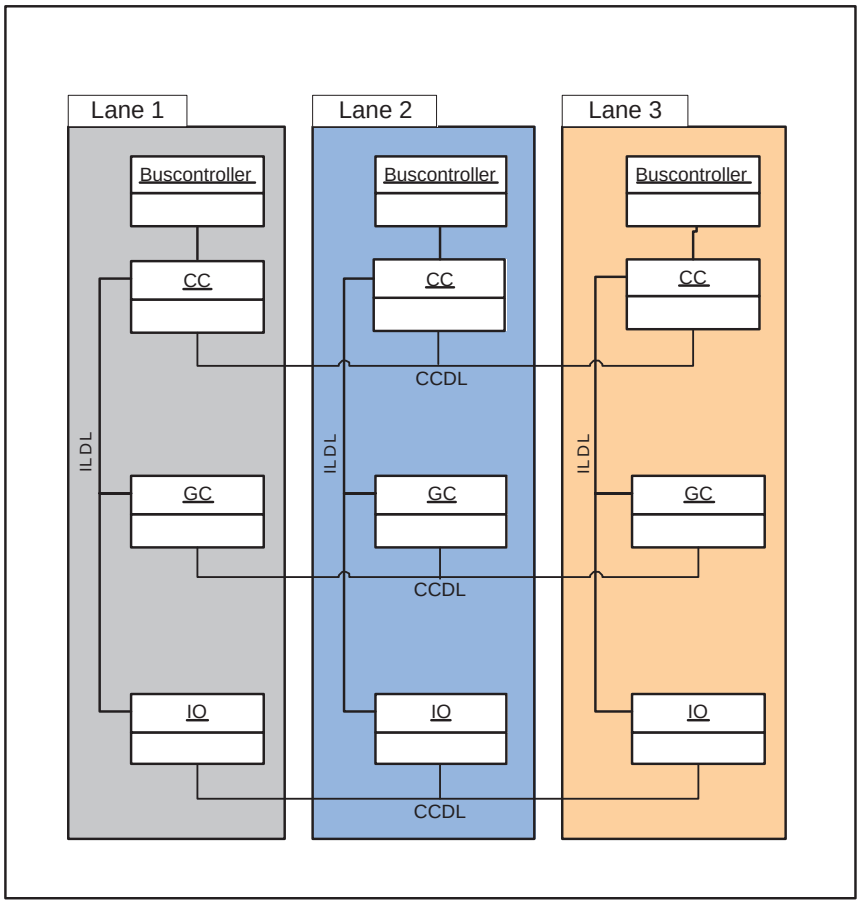

Abbildung 1: Aufbau des FCC

Abbildung 1 zeigt den physikalischen Aufbau des Flight Control Computers, der für den Demonstrator verwendet wird. Das System ist triplex aufgebaut. Das heißt, dass der Steuerungscomputer dreifach vorhanden ist, um die Redundanz zu erhöhen. Alle drei Systeme, die so genannten Lanes, sind identisch aufgebaut und bestehen wiederum aus jeweils drei Prozessoren. Jeder dieser Prozessoren hat spezifische Aufgaben: Der Control-Computer (CC) übernimmt die Steuerung und bewältigt Aufgaben wie Navigation, Flugreglung und Konsolidierung der Daten, der Guidance-Computer (GC)

stellt missionsspezifische Funktionen zur Verfügung und der IO-Computer (IO) stellt die Verbindung zu Sensoren und Aktuatoren her.

Die Prozessoren einer Lane, also CC, GC und IO sind über eine serielle Verbindung, einen High Speed Serial Link (HSSL), untereinander verbunden um Daten und Steuerungsinformationen austauschen zu können. Diese Verbindung innerhalb einer Lane heißt Intra Lane Data Link (ILDL). Will ein über HSSL angebundener Teilnehmer mit einem anderen Prozessor kommunizieren, schreibt er das relevante Datum in einen FIFO-Puffer. Die Hardware übernimmt dann das Versenden der Nachricht und schreibt diese direkt in den Speicher des Empfängers. Da dieser FIFO-Puffer eine begrenzte Kapazität besitzt, muss bei der Zeitplanerstellung der Tasks darauf geachtet werden, dass der Puffer nicht überläuft.

Auch die verschiedenen Lanes müssen untereinander verbunden sein, um berechnete Daten auf alle drei Lanes verteilen zu können, um so eine Konsensbildung zu ermöglichen. Auch diese Verbindung ist mittels eines HSSL implementiert, der in diesem Kontext als Cross Channel Datalink bezeichnet (CCDL) wird. Verbunden sind jeweils Prozessoren gleicher Funktionalität der unterschiedlichen Lanes. Auf Prozessoren verschiedener Lanes mit gleichen Aufgaben läuft ein identisches Softwareimage, so dass bei der Generierung der Schedules nicht nach Lane unterschieden werden muss.

Die Control-Computer jeder Lane sind zusätzlich mit jeweils einem MIL-STD-1553 [8] Buscontroller verbunden, der über einen Bus in Verbindung mit Geräten wie Global Positioning System oder Flight Test Instrumentation steht. Dieser Controller agiert als Busmaster und ist damit für die Arbitrierung des Busses verantwortlich. Die Kommunikation zwischen Buscontroller und Prozessor geschieht über einen gemeinsamen Speicherbereich, auf den sowohl Prozessor als auch Buscontroller zugreifen können.

3.2 FCS Software

3.2.1 Scheduler und Dispatcher

Bei sicherheitskritischer Software ist es unabdingbar, dass das zeitliche Verhalten der Software exakt vorhersagbar ist. Aus diesem Grund besteht der Scheduler, der im FCS Verwendung findet, aus zwei Teilen: einem preemtiven und einem nicht-preemtiven. Im nicht-preemtiven Teil werden Tasks, die durch eine statische Tabelle in der Software definiert werden, vom Dispatcher sequentiell abgearbeitet. Der Dispatcher besitzt nicht die Möglichkeit, die vordefinierte Reihenfolge der Tasks zu ändern. Diese Tasks werden im Regelfall nicht unterbrochen und sind für den Dispatcher durch die Attribute Funktionspointer, frühester Startzeitpunkt und maximale Laufzeit definiert. Die Planung der Taskliste, die definiert in welcher Reihenfolge die Tasks abgearbeitet werden, geschieht offline und nicht während der Laufzeit des FCS. Der Dispatcher startet einen Task aus der Liste, sobald der Startzeitpunkt

erreicht wurde. Überschreitet ein Task seine maximale Laufzeit, heißt das, dass ein Fehler aufgetreten ist. Dieser Fall führt direkt zum Stilllegen der kompletten Lane.

Die Tasks werden dabei wie folgt definiert:

Listing 2: Ausschnitt aus der Minor-Frame-Definition

```
const MinorFrameRomType
        MinorFrameRomTable[MAX_NO_MINOR_FRAMES+1] =
  /* Frame 0                                    */
  /* Adr, SyncOffset [us], Deadline [us] */
  {{
   {ILDL_FcsGroup_Update,        100,    21},
   {IO_Discretes_Receive,        121,    27},
   {GBL_LaneId_UpdateId,         148,    19},
   ....
   ....
   /* Last entry */
   {NULL,                          0,     0},
  }},

  /* Last entry */
  {{
   {NULL, 0, 0},  /* Last entry */
  }},
};
```

Aus dieser Tabelle entnimmt der Dispatcher alle Informationen zur zeitlichen Ablaufsteuerung der Tasks. Der erste Parameter eines Eintrags, die Adresse, zeigt auf den aufzurufenden Task im Speicher. Der zweite Parameter, der Sync-Offset, gibt den frühesten Zeitpunkt an, zu dem der Task vom Dispatcher gestartet werden darf. Dieser Zeitpunkt bezieht sich auf den Start des Minor Frames. Der Parameter Deadline beschreibt die maximale Rechenzeit, die ein Task verbrauchen darf. Ein NULL-Element markiert das Ende eines Minor-Frames.

Die Abarbeitung dieser Tasks geschieht in einem Fenster fester zeitlicher Länge, dem so genannten Minor Frame. Sollte nach Ausführung der nichtpreemtiven Tasks noch Rechenzeit innerhalb dieses Frames zur Verfügung stehen, wird diese gleichmäßig auf die preemtiven Tasks verteilt. Das Scheduling für preemtive Tasks geschieht also dynamisch während der Laufzeit. Beliebig viele Minor Frames, die die gleiche Länge besitzen müssen, bilden das Major Frame. Das Ende eines Major Frames wird durch ein Minor Frame gekennzeichnet, das aus einem einzigen NULL-Element besteht. Innerhalb des Major Frames werden alle Minor Frames nacheinander abgearbeitet. Man hat so die Möglichkeit, Tasks mit verschiedenen Frequenzen laufen zu lassen. Diese Funktionalität wird derzeit jedoch nicht genutzt.

Auf allen Prozessoren des FCC ist dieser Scheduler implementiert. Der

Start der jeweiligen Frames sind synchronisiert, das heißt es startet auf allen Prozessoren das selbe Frame zum gleichen Zeitpunkt. Zusätzlich werden im Scheduler Laufzeitdaten, wie maximal benötigte Rechenzeit der Tasks, erfasst und aufgezeichnet.

Aufgrund dieser Voraussetzungen wird ersichtlich, dass die Zeitplanerstellung der nicht-unterbrechbaren Tasks nicht zwangsläufig auf dem FCS berechnet werden muss, sondern auch außerhalb dieses Systems geschehen kann. Dieses Vorgehen hat den Vorteil, dass auf dem Zielsystem für die Planung keine Rechenzeit verloren geht. Obwohl die Systemzeit eine kontinuierliche Größe ist, sind sämtliche Zeiten quantisiert in μS angegeben.

3.2.2 Busnachrichten

Da der an die Control-Computer angeschlossene MIL-STD-1553 Buscontroller als Busmaster konfiguriert ist, muss auch für die Busnachrichten eine zeitliche Abfolge definiert werden. Da funktionale Abhängigkeiten zwischen Busnachrichten und Tasks bestehen, ist auch die Planung der Arbitrierung des Busses Gegenstand des Schedule Planners. Sämtliche Konfigurationsdaten für die Busnachrichten werden dem Buscontroller bei Systemstart in Form einer Liste übergeben, die ähnliche Informationen enthält wie auch die Tabelle für Tasks. Eine Busnachricht ist aus Sicht des Buscontrollers durch den Namen der Nachricht, deren Länge und den Abstand zur nächsten Nachricht definiert. Auch diese Liste wird periodisch durch den Buscontroller innerhalb eines Minor Frames abgearbeitet, das die gleiche Länge wie ein Task-Minorframe besitzt und ebenfalls zu einem gemeinsamen Zeitpunkt startet.

3.2.3 Constraints

Aufgrund der Physik des Flight Control Computers und softwareseitiger Abhängigkeiten zwischen Tasks entstehen Bedingungen (Constraints), deren Einhaltung durch den Schedule Planner sichergestellt werden müssen. Im folgenden werden die verschiedenen Arten von Abhängigkeiten und deren Entstehungsgrundlage genauer erörtert.

"Must end before start" (MEBS): Diese Bedingung ergibt sich, wenn Task 2 Daten benötigt, die innerhalb eines anderen Tasks 1 berechnet werden. Damit diese Bedingung erfüllt ist, muss Task 1 beendet sein, bevor Task 2 startet. Diese Bedingung kann sich auch auf Busnachrichten oder auf beliebige Kombinationen von Tasks und Busmessages beziehen, also allgemein auf planbare Elemente, im folgenden *Items* genannt.

Seien I_m und I_n Items, $t_{end}(I_m)$ und $t_{start}(I_n)$ End- und Startzeitpunkt von I_m und I_n, dann gilt

$$MEBS(I_m \rightarrow I_n) \text{ ist erfüllt, wenn gilt: } t_{end}(I_m) \leq t_{start}(I_n) \qquad (1)$$

"Offsetted" (OFS): Dieses Constraint legt den exakten Abstand zweier Items fest.

Seien I_m und I_n Items, $t_{start}(I_m)$ und $t_{start}(I_n)$ die Startzeitpunkte von I_m und I_n und t_{ofs} der zeitliche Versatz zweier Items, dann gilt

$$OFS(I_m \rightarrow I_n, t_{ofs}) \text{ ist erfüllt, wenn gilt: } t_{start}(I_m) + t_{ofs} = t_{start}(I_n) \quad (2)$$

"Must not overlap" (MNO): Aufgrund des verwendeten hardwareseitigen Systemdesigns kann es vorkommen, dass beim Übertragen einer Busnachricht über den gemeinsamen Speicher zwischen Buscontroller und Prozessor inkonsistente Daten gelesen werden. Dies ist genau dann der Fall, wenn das Auslesen der Nachricht genau während der Übertragung derselben auf dem Bus stattfindet oder wenn der Prozessor genau zu dem Zeitpunkt Daten in den gemeinsamen Speicher schreibt, zu dem der Buscontroller bereits am Senden dieser Nachricht ist.

Sei T_m ein Task und M_n eine Busnachricht, $[t_{start}(M_m), t_{end}(M_m)]$ das Intervall, in dem T_m rechnet, dann gilt

$$\begin{aligned} &MNO(T_m \leftrightarrow M_n) \text{ ist erfüllt, wenn gilt:} \\ &(t_{start}(T_m) \notin [t_{start}(M_m), t_{end}(M_m)]) \wedge \\ &(t_{end}(T_m) \notin [t_{start}(M_m), t_{end}(M_m)]) \wedge \\ &(t_{start}(M_m) \notin [t_{start}(T_m), t_{end}(T_m)]) \wedge \\ &(t_{end}(M_m) \notin [t_{start}(T_m), t_{end}(T_m)]) \end{aligned} \qquad (3)$$

"Connected by FIFO" (CBF): Die Verbindung aller Prozessoren per HSSL sowohl über CCDL als auch ILDL führt zu der Forderung nach dieser Constraint. Damit lassen sich zwei mögliche Fehler im Betrieb verhindern:

- Da jedem Prozessor für das Versenden von Daten per HSSL nur ein einziger FIFO-Puffer begrenzter Größe zur Verfügung steht, muss sichergestellt werden, dass dieser unter keinen Umständen überläuft. Unterliegen zwei Tasks dieser Bedingung, kann beim Generieren des Schedules die Auslastung des FIFO-Puffers berechnet und ein Überlauf verhindert werden.

- Die Verwendung dieser Constraint ermöglicht es dem Schedule-Planner zu berechnen, ob die zu übertragenden Daten schon komplett übermittelt wurden und dem Zieltask bereits zur Verfügung stehen. Wäre dies

nicht der Fall, würde der Zieltask mit alten oder inkonsistenten Daten arbeiten.

Die Größe der zu übertragenden Nachricht wird als *messagesize* bezeichnet und gibt die Anzahl der zu übertragenden Worte an. Die Daten werden mit der Geschwindigkeit *throughput* übertragen. Aus Sicherheitsgründen wird vorausgesetzt, dass der FIFO nicht nur genug freien Platz für die zu übertragenden Nachrichten bereitstellt, sondern vor der Übertragung komplett leer ist.

Sei T_m und T_n Tasks, dann gilt

$$\begin{aligned} &CBF(T_m \rightarrow T_n, messagesize) \text{ ist erfüllt, wenn gilt:} \\ &(t_{end}(T_m) \leq t_{start}(T_n)) \wedge \\ &(\text{FIFO ist leer in } [t_{start}(T_m), t_{end}(T_m)]) \wedge \\ &(t_{start}(T_n) \geq t_{end}(T_m) + throughput * messagesize) \end{aligned} \tag{4}$$

"Fixed" (FIXED): Mit Hilfe dieser Bedingung lässt sich der exakte Startzeitpunkt eines Tasks oder einer Busnachricht festlegen. Die Interfacespezifikation eines über den Bus angeschlossenen Gerätes des Demonstrators macht dieses Constraint notwendig.

Sei I_m ein Task oder eine Busnachricht, dann gilt

$$FIXED(I_m, t_{fixed}) \text{ ist erfüllt, wenn gilt: } t_{end}(I_m) = t_{fixed} \tag{5}$$

4 Konzept

4.1 Übersicht

Das folgende Schema zeigt die entwickelten Programme und Dokumente, die für Erstellung und Analyse von Schedules notwendig sind. Bei blau hinterlegten Objekten handelt es sich um Dokumente oder Programme, die im Rahmen dieser Arbeit entstanden sind. Dabei symbolisieren Ovale Programme, bei Rechtecken handelt es sich um Dokumente, wie beispielsweise Quellcode-Dateien oder Skripten.

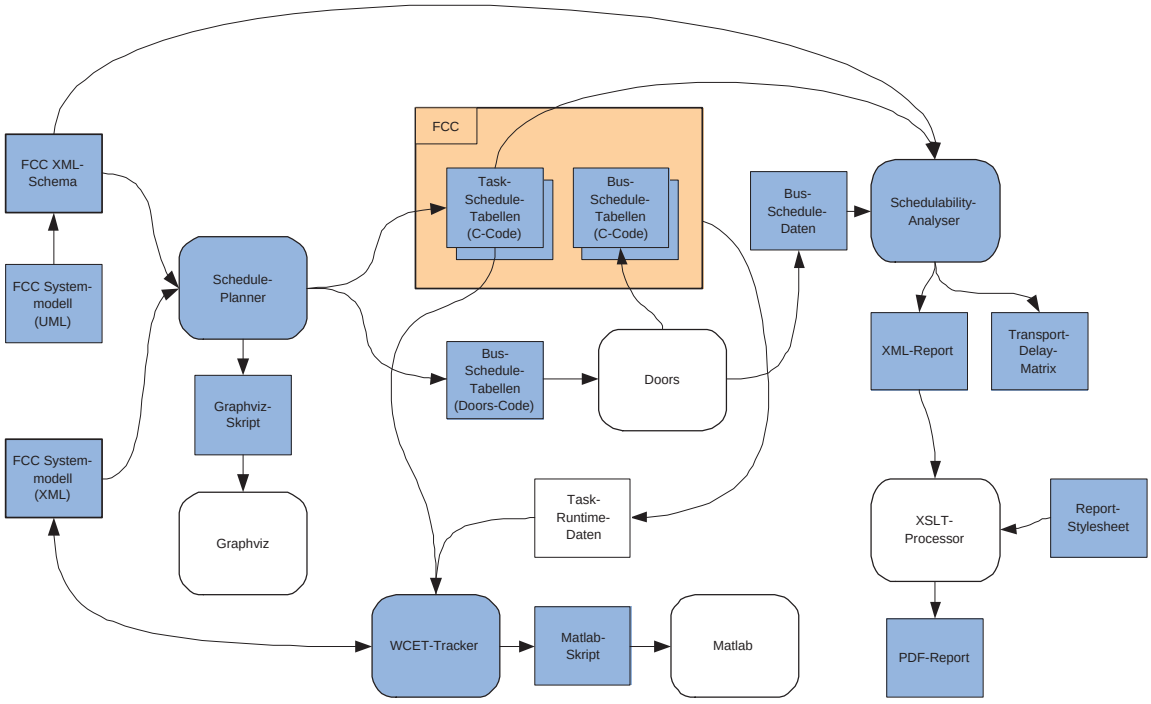

Abbildung 2: Toolchain

Ausgangspunkt des Prozesses ist die Beschreibung des Systems mittels eines UML-Klassendiagramms. Das daraus entwickelte XML-Schema ([34],[33]) legt die Form der konkreten XML-Beschreibung des FCCs fest. Das XML-Schema und das XML-Systemmodell sind Eingaben für den Schedule Planner, der mit Hilfe des XML-Schemas die Syntax des XML-Systemmodells validiert. Ist dieser Test erfolgreich, generiert der Schedule-Planner mehrere Ausgaben:

- Ein Graphviz-Skript: Die Verwendung von XML als Beschreibungsform für das System bietet Vorteile gegenüber eines proprietären Formats, da XML die Erstellung menschen- und maschinenlesbarer Do-

kumente in Form einer Baumstruktur standardisiert. Ab einer gewissen Größe sind jedoch solche Dokumente vom Menschen nur noch schwer zu überschauen. Deswegen ist es sinnvoll, die im Systemmodell des FCC enthaltenen Informationen auch grafisch darzustellen. Der Schedule-Planner erzeugt aus diesem Grund ein Skript, das als Eingabe für das Programm Graphviz dient. Bei Graphviz handelt es sich um eine Open-Source-Software, die mittels einer einfachen Skriptsprache namens "dot" aus strukturellen Daten Graphen generiert([18], [19]).

- Task-Schedule-Tabellen: Der Planner erzeugt direkt C-Code, der den erstellten Plan repräsentiert. Diese Dateien enthalten im wesentlichen eine Tabelle in einer Form wie in Listing 1 beschrieben. Der generierte Code wird zusammen mit der restlichen FCC-Software im Build-Prozess übersetzt, gelinkt und auf den FCC geladen.

- Bus-Schedule-Tabellen: Prinzipiell wäre es auch im Fall der Busnachrichten möglich, direkt Code für den FCC zu erzeugen. Jedoch wurde hier ein anderer Weg gewählt. Da über den verwendeten Bus mehrere Geräte unterschiedlicher Domänen kommunizieren, die zudem von verschiedenen Arbeitsgruppen betreut werden, ist es notwendig, dass alle Mitarbeiter über eine zentrale Schnittstelle Anforderungen und Eigenschaften des Busses bearbeiten können. Diese Aufgabe erfüllt Doors, eine kommerzielle Requirements-Management-Software. Der Schedule-Planner erzeugt ein Skript für Doors, das die berechneten Daten in das Requirements-Management einpflegt. Die eigentliche Codegenerierung wird dann aus Doors heraus gestartet.

Der Schedulability-Analyzer hat die Aufgabe, die vom Schedule Planner erzeugten Pläne auf Korrektheit zu überprüfen. Als Eingaben dafür benötigt er das XML-Systemmodell, in dem der FCC inklusive sämtlicher Bedingungen und Abhängigkeiten zwischen Tasks und Busnachrichten beschrieben ist, die generierten Task-Schedule-Tabellen und Informationen über Busse und Busnachrichten, die aus Doors heraus exportiert werden. Der Analyzer prüft daraufhin, ob alle modellierten Constraints erfüllt sind und erzeugt einen Report, in dem die gewonnenen Ergebnisse zusammengefasst sind. Dieser XML-Report wird mittels eines XML-Stylesheets in ein PDF-Dokument transformiert, das als Dokumentation und als Ausgangspunkt für einen Qualifizierungsprozess des Schedules dienen kann.

Einer der wichtigsten Parameter eines Tasks, den der Schedule Planner für die Berechnung des Plans benötigt, ist die maximale Laufzeit ("Worst Case Execution Time", WCET) des Tasks. Sie gibt unter Berücksichtigung aller Verzögerungen wie Interrupts, Waitstates oder Hardware die längste Zeit an, während der ein Task ausgeführt wird. Da sich diese WCET im Allgemeinen nicht automatisch ermitteln lässt, wird diese Zeit mit der Un-

terstützung eines Werkzeuges, dem WCET-Tracker, empirisch ermittelt. Der Tracker ermittelt die Laufzeitdaten der Tasks, die während des Betriebs des FCC aufgezeichnet wurden. Verglichen werden diese mit den Laufzeiten, die der Planner dem Systemmodell entnommen und für die Planung verwendet hat. Wurden für einen Task größere als die im Modell angegebenen Laufzeiten gemessen, werden diese angepasst und ein neuer Plan wird generiert.

4.2 Systemmodell

Der folgende Graph beschreibt den FCC und die darin enthaltenen Constraints zwischen Tasks und Messages als UML-Klassendiagramm.

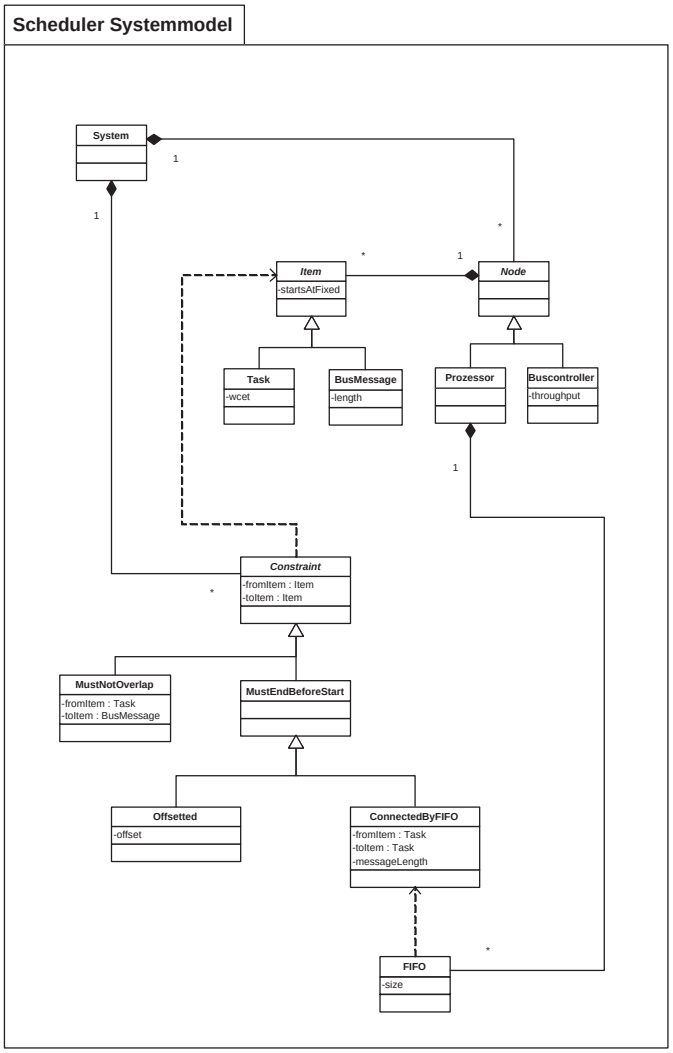

Abbildung 3: UML-Systemmodell

Das System besteht unter anderem aus einer beliebigen Anzahl von Knoten, den "Nodes". Eine abstrakte Node ist die Generalisierung eines Prozessors oder eines Buscontrollers, also einer Einheit, die Ressourcen zur Verfügung stellt und für die folglich ein Schedule zu erstellen ist. Ein Prozessor kann zusätzlich noch einen oder mehrere FIFO-Puffer besitzen, die die über HSSL übertragenen Daten zwischenspeichern. Der Datendurchsatz des Buscontrollers, der mit dem Attribut *throughput* angegeben wird, be-

schreibt, wie viele Datenelemente pro Zeiteinheit auf dem Bus übertragen werden.

Ein Knoten setzt sich aus Items zusammen, deren Ausprägung entweder ein Task oder eine Busnachricht sein kann. Die maximale Laufzeit eines Tasks wird mittels des Attributes *wcet* angegeben, analog dazu definiert das Attribut *length* die Anzahl der zu übertragenden Worte einer Busnachricht. Da der Durchsatz des Buscontrollers bekannt ist, lässt sich mit Hilfe der Länge die Zeit berechnen, während der der Bus durch die Nachricht blockiert wird. Das Attribut *startsAtFixed* modelliert die Constraint "FIXED". Ist dieser Wert gesetzt, wird der Schedule Planner versuchen, den entsprechenden Task oder die Busnachricht zeitlich zu dem angegebenen Wert einzuplanen.

Ein weiterer Bestandteil der Systembeschreibung sind die Constraints. Mit ihnen werden die durch Hard- und Softwarestruktur festgelegten Bedingungen und Abhängigkeiten, wie in Kapitel 3.2.1 erläutert, definiert. Eine Constraint bezieht sich dabei immer auf zwei Items "fromItem" und "toItem". Handelt es sich um eine gerichtete Constraint, wie beispielsweise *MEBS(fromItem, toItem)*, beschreibt *fromItem* das Item, von dem *toItem* abhängt. Konkret würde das in diesem Fall bedeuten, dass *fromItem* beendet sein muss, bevor *toItem* starten darf.

Da die Aufgabenstellung fordert, dass der FCC mittels einer XML-Beschreibung modelliert wird, ist es sinnvoll, für diese Datei eine Schema-Definition zu erstellen. Dieses Schema ("XML Schema Definition", XSD) definiert die XML-Dokumentenstruktur wiederum in der Form eines XML-Dokumentes. Die folgende Abbildung beschreibt diese XSD und wurde mit Hilfe des Programms "XMLSpy" ([2]) erstellt.

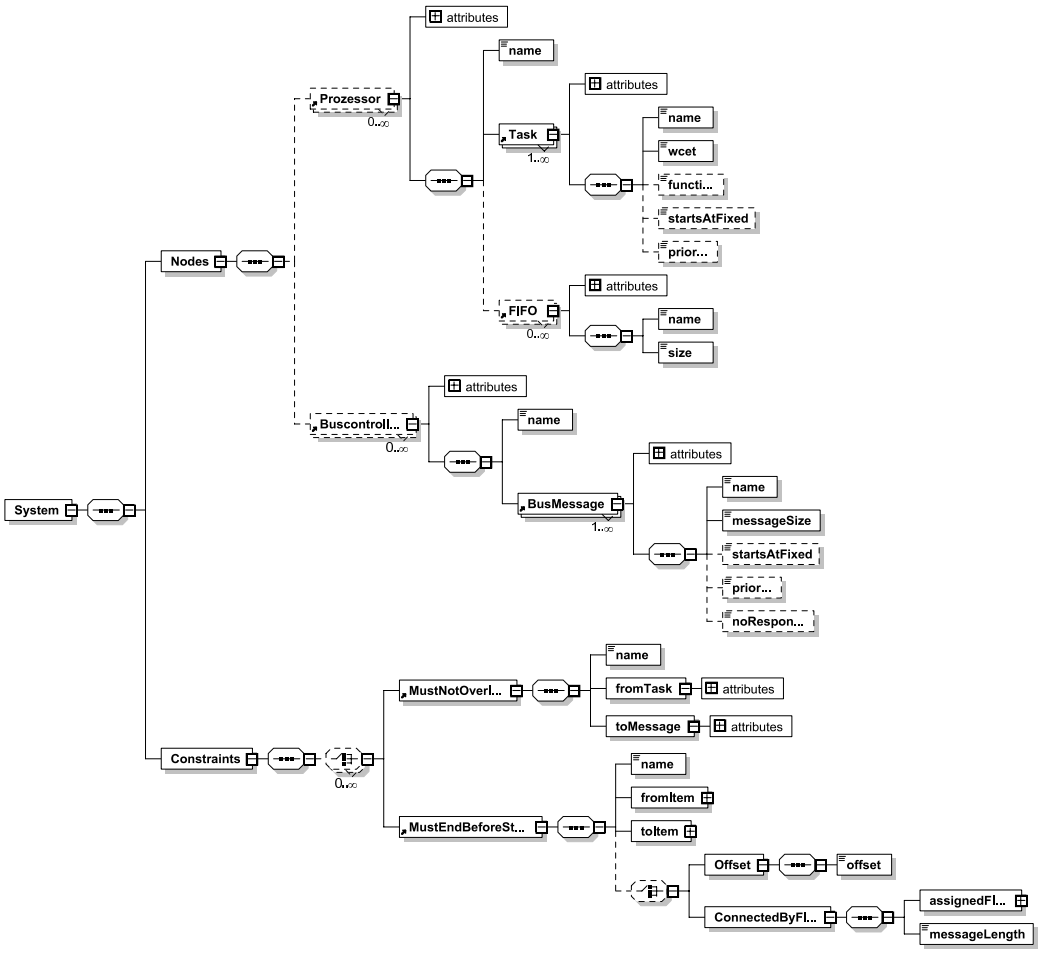

Abbildung 4: XML-Schema des Systemmodells

Mit Hilfe der XML-Schema Definitionen können neue XML-Elemente, deren Attribute und Kindelemente definiert werden. Dies soll hier exemplarisch am Beispiel des Elementes "Task" erläutert werden. Das folgende XML-Fragment definiert ein neues Element mit dem Namen *Task*:

Listing 3: XSD-Definition von *Task*

```
<xs:element name="Task">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="name"
                type="xs:string"/>

            <xs:element name="wcet"
                type="xs:nonNegativeInteger"/>

            <xs:element name="startsAtFixed"
                type="xs:nonNegativeInteger" minOccurs="0"/>

        </xs:sequence>
        <xs:attribute name="TaskId"
            type="xs:ID" use="required"/>
    </xs:complexType>
</xs:element>
```

Der Definition des komplexen Elementes "Task" folgen die Kindelemente *name*, *wcet* und *startsAtFixed*, wobei die Angabe des Elementes *startsAtFixed* in der Instanziierung des Schemas optional ist. Diese Eigenschaft wird durch das Attribut *minOccurs="0"* beschrieben. *Task* besitzt wie alle anderen Elemente, auf die im XML-File referenziert wird, ein Attribut vom Typ "xs:ID". Dieser dokumentenweit eindeutige Identifier ermöglicht es, Verknüpfungen im XML-Dokument mittels des Schemas auf Korrektheit zu überprüfen. Eine mögliche Instanz dieser Typdefinition sieht folgendermaßen aus:

Listing 4: XML-Instanz von *Task*

```
<Task TaskId="CC_IO_Discretes_Receive">
    <name>IO_Discretes_Receive</name>
    <wcet>13</wcet>

    <!-- optional: -->
    <startsAtFixed>15000</startsAtFixed>
</Task>
```

Die Modellierung des Systems als UML-Klassendiagramm lässt sich jedoch nicht direkt auf eine XML-Schemadefinition abbilden, da in beiden Modellen nicht die selben Modellelemente zur Verfügung stehen. Die Generalisierung aus UML wurde in der XSD durch die Verwendung optionaler Elemente nachgebildet und Abhängigkeitsbeziehungen durch Schema-Identifier ersetzt.

Die Entwicklung einer XML-Schemadefinition hat einen weiteren, für die spätere Implementierung entscheidenden Vorteil: Aus XSD-Dateien lässt sich direkt Quellcode generieren, der zum Einlesen der XML-Daten in die Planner- und Analyzersoftware benötigt wird. Der zeitaufwändige Schritt des manuellen Erstellens geeigneter Parser-Klassen entfällt damit. Außerdem verbessert diese Methode der Codegenerierung die Wartbarkeit des Co-

des, da sehr schnell und flexibel auf Änderungen im Modell reagiert werden kann. Neben XMLSpy ist beispielsweise noch das Open-Source-Tool Castor ([32]) in der Lage, aus XSD-Dokumenten Java-Klassen zu erzeugen.

5 Implementierung

5.1 Modellierung und Visualisierung

Sämtliche Eingaben, die zur Erstellung eines Plans notwendig sind, erhält der Schedule-Planner aus der XML-Systembeschreibung. Deshalb ist es notwendig, diese Datei im ersten Schritt der Implementierung einzulesen. Als Programmiersprache wurde Java der Firma Sun Microsystems gewählt [31]. Dies hat mehrere Gründe: Java erzeugt plattformunabhängigen Bytecode, der in der Regel ohne Anpassungen auf verschiedenen Plattformen lauffähig ist. Zudem hat Java in den letzten Jahren einen hohen Reifegrad erreicht, der sich auch in punkto Performance und der Vielfalt frei verfügbarer Frameworks und Projekte bemerkbar macht. Als Entwicklungsumgebung kommt die ebenfalls frei verfügbare IDE Eclipse zum Einsatz.

Wie bereits erwähnt ist das XML-Modellierungstool XMLSpy in der Lage, Java-Code zum Übertragen der XML-Daten auf Java-Klassen zu generieren. Diese Funktionalität erspart es dem Entwickler, XML-Dateien "per Hand" auszuwerten. XMLSpy erzeugt bei diesem Prozess für jedes XML-Element eine Java-Klasse mit den zugehörigen Abhängigkeiten. Die Verwendung der autogenerierten Klassen gestaltet sich einfach:

Listing 5: Verwendung der autogenerierten Klassen

```
// create new root document
SystemmodelDoc doc = new SystemmodelDoc();

// load xml-tree from file
SystemType system = new SystemType(doc.load(xmlFilename));

// get nr. of nodes
int count= system.getNodesCount();
```

Zeile 1 des Codefragments erzeugt eine Instanz des Root-Dokumentes, Zeile 2 liest den XML-Baum ein und erstellt den Objekt-Tree. Bei *SystemType* handelt es sich um die von XMLSpy generierte Klasse, die das Wurzelelement des XML-Baumes repräsentiert. Solche Klassen, deren Hauptaufgabe es ist, Daten zur Weiterverarbeitung zur Verfügung zu stellen, bezeichnet man als Container-Klassen. Waren diese beiden Schritte erfolgreich, lässt sich nun wie in Zeile 3 gezeigt über Java-Methoden auf die Daten des XML-Files zugreifen.

Da sich die UML-Modellelemente nicht direkt mit XML-Elementen modellieren lassen, können Konstrukte wie abstrakte Klassen oder Vererbung

nicht durch den Codegenerator von XMLSpy nachgebildet werden. Da jedoch in der implementierten Software mit den Datenstrukturen und Klassen gearbeitet werden soll, die im UML-Entwurf des Systems erarbeitet wurden, ist eine Übertragung der Daten aus den von XMLSpy generierten Klassen in die aus dem UML-Diagramm abgeleiteten Klassen notwendig. Das folgende Schema verdeutlicht diesen Datenfluss:

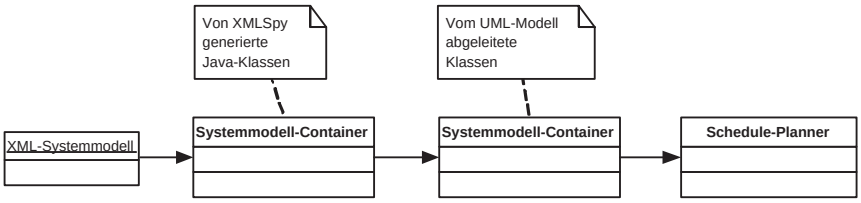

Abbildung 5: Datenfluss zwischen Containern

Nachdem die Daten in den internen Datenstrukturen zur Verfügung stehen, gilt es nun, das Systemmodell und insbesondere die darin enthaltenen Abhängigkeiten zwischen Tasks und Busnachrichten grafisch darzustellen. Ausgangspunkt dafür ist ein endlicher, gerichteter Graph, in dem Items durch Knoten und Abhängigkeiten zwischen Items durch Kanten symbolisiert werden. Da die Abhängigkeit "*Must not overlap*" in der Regel zwischen einem Task und mehreren dazu abhängigen Busnachrichten besteht und damit recht überschaubar bleibt, werden diese Anhängigkeiten der Übersichtlichkeit zuliebe nicht in den Graphen gezeichnet. Die praktische Umsetzung hat weiterhin gezeigt, dass es sinnvoll ist, für jede Node einen eigenen Graphen zu erzeugen. Die folgende Abbildung zeigt einen solchen Graphen:

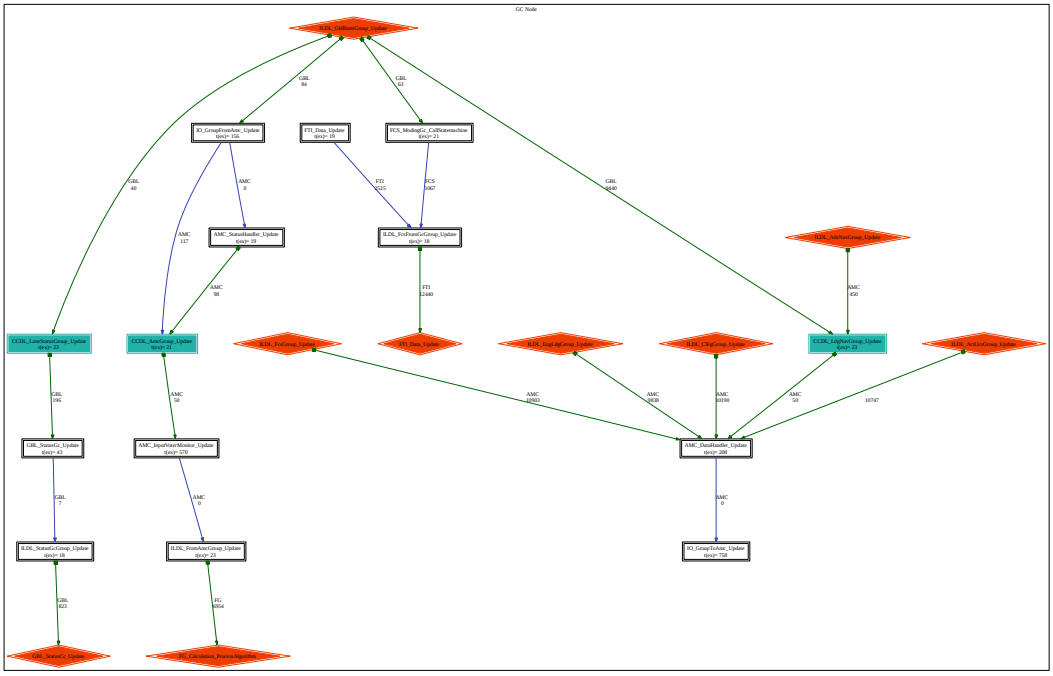

Abbildung 6: Constraint-Graph

Die rechteckigen Knoten des Graphen stehen dabei für Tasks, die auf dem selben Knoten laufen, auf den sich der Graph bezieht. In diesem Fall ist der Abhängigkeitsgraph des Guidance-Computers dargestellt. Rautenförmige, rot gefärbte Knoten verweisen auf Tasks anderer Knoten, in diesem Fall handelt es sich um Tasks des Control-Computers.

Bei den Kanten lassen sich zwei Typen unterscheiden: Blau eingefärbte Kanten stehen für die Anhängigkeiten "*Must end before start*" und "*Offsetted*", die abgesehen von dem zeitlichen Versatz von "*Offsetted*" die gleiche Bedeutung haben. Grün gefärbte Kanten symbolisieren die Abhängigkeit "*Connected by FIFO*".

Die automatische Visualisierung von Daten in Form eines Graphen erfordert die Verwendung komplexer Algorithmen, deren Erstellung nicht Bestandteil dieser Arbeit ist. Aus diesem Grund wurde auf das freie Tool "Graphviz" zurückgegriffen. Diese Software erstellt aus Strukturdaten eines Graphen dessen graphische Repräsentation. Eingabe für Graphviz ist ein Skript folgender Struktur:

```
digraph system_dependencies {

   subgraph cluster0 {
      label="GC Node, 20060214_101001";

      GC_CCDL_LdgNavGroup_Update [shape= box, peripheries= 2, label= "CCDL_LdgNavGroup_Update\nt(ex)= 23", style= filled];
      GC_FCS_ModingGc_CallStatemachine [shape= box, peripheries= 2, label= "FCS_ModingGc_CallStatemachine\nt(ex)= 21"];
      GC_FTI_Data_Update [shape= box, peripheries= 2, label= "FTI_Data_Update\nt(ex)= 21"];
      GC_IO_GroupFromAmc_Update [shape= box, peripheries= 2, label= "IO_GroupFromAmc_Update\nt(ex)= 156"];
      GC_ILDL_FcsFromGcGroup_Update [shape= box, peripheries= 2, label= "ILDL_FcsFromGcGroup_Update\nt(ex)= 19"];
      CC_ILDL_GblRtmsGroup_Update [shape= diamond, peripheries= 2, label= "ILDL_GblRtmsGroup_Update", style= filled];
      ...
   }

   GC_FCS_ModingGc_CallStatemachine -> GC_ILDL_FcsFromGcGroup_Update [label= "FCS\n1068", color= "0.650 0.700 0.700"];
   GC_FTI_Data_Update -> GC_ILDL_FcsFromGcGroup_Update [label= "FTI\n2524", color= "0.650 0.700 0.700"];
   CC_ILDL_GblRtmsGroup_Update -> GC_CCDL_LdgNavGroup_Update [label= "GBL\n9559", color= darkgreen , arrowtail= box];
   CC_ILDL_GblRtmsGroup_Update -> GC_IO_GroupFromAmc_Update [label= "GBL\n84", color= darkgreen , arrowtail= box];
   CC_ILDL_GblRtmsGroup_Update -> GC_FCS_ModingGc_CallStatemachine [label= "GBL\n63", color= darkgreen , arrowtail= box];
   ...
}
```

Graphviz ist in der Lage, sowohl gerichtete als auch ungerichtete Graphen zu zeichnen. In diesem Fall werden gerichtete Graphen zur Visualisierung des Systems verwendet, was durch das Schlüsselwort *digraph* definiert wird. In der Sektion *subgraph cluster0* werden die Knoten definiert, die aus dem Systemmodell entnommen wurden. Der letzte Abschnitt beschreibt die Kanten zwischen den Knoten, die zu den Subgraphen gehören. Die in eckigen Klammern angegebenen Attribute beschreiben Eigenschaften der Knoten und Kanten, wie beispielsweise Form oder Farbe.

Die Erzeugung dieses Skriptes ist Bestandteil des Schedule-Planners und wird intern mit Hilfe der Velocity-Template-Engine generiert. Velocity ist ein Open-Source-Projekt der Jakarta-Projektgruppe von Apache, das es ermöglicht, in "Schablonen" mit Hilfe einer einfachen Sprache auf Java-Objekte direkt zuzugreifen. Im Fall des Schedule Planners wird eine Instanz des UML-Systemmodells an den Velocity-Kontext übergeben. Die Velocity-Engine erstellt dann mit Hilfe des Templates, in dem der Rumpf des Graphviz-Skriptes und eine Iteration über Tasks und Constraints definiert sind, das fertige Skript. Dieses Architekturmuster realisiert das Model-View-Controller-Modell, bei dem im Programm die Einheiten Datenmodell, Präsentation und Programmsteuerung getrennt werden. Es ist so möglich, durch die Verwendung eines anderen Templates ohne Änderung des Programmcodes beispielsweise Excel-Tabellen oder HTML-Ausgaben zu erzeugen.

5.2 Schedule Planner

Der Kern des Schedule-Planners, der aus der Eingabe des Systemmodells einen Plan errechnet, verwendet Backtracking, um zu einer Lösung zu gelangen. Bevor jedoch näher auf Details eingegangen werden soll, müssen einige Hilfsklassen eingeführt werden, die für den Betrieb notwendig sind.

Items: Ergebnis des Planning-Algorithmus sind die Zeiten, zu denen ein bestimmter Task oder eine bestimmte Busnachricht eingeplant, also vom Dispatcher gestartet wird. Da es sich sowohl bei Tasks als auch bei Busnachrichten allgemein um Objekte handelt, die eine bestimmte Zeit eine bestimmte Ressource belegen, unterscheidet der Schedule-Planner in diesem Zusammenhang nicht mehr zwischen beiden. Zudem benötigt der Algorithmus weitere Variablen, die den Zustand eines Items während der Laufzeit des Backtrackings beschreiben, beispielsweise ob ein Item schon eingeplant wurde. Da diese Variablen nicht Bestandteil des UML sind, verwendet der Scheduling-Algorithmus eigene Container-Klassen für die Modellierung der Items. Items werden durch die Klasse *PlannerItem* beschrieben und beinhalten folgende Attribute:

Listing 6: Attribute von *PlannerItem*

```
public class PlannerItem {

    // derived from systemmodel
    private int id;
    private int nodeId;
    private int executionTime;
    private boolean fixed;

    // set by planner
    private boolean planned= false;
    private boolean ready= false;
    private int startTime;
    private int endTime;
}
```

Die Variable *id* ist ein algorithmusweit eindeutiger Identifier, mit dem sich ein bestimmtes Item direkt adressieren lässt. Simultan dazu verweist *nodeId* auf die dem Item zugehörige Node. *startTime* und *endTime* geben ab dem Zeitpunkt, ab dem das Item eingeplant wurde, Start- und Endzeit des Items an. Die Endzeit berechnet sich dabei durch

$$endTime = startTime + executionTime \tag{6}$$

Die *executionTime* wird aus dem Systemmodell abgeleitet. Falls es sich bei dem Item um einen Task handelt, wird die *executionTime* direkt aus der Worst Case Execution Time (WCET) übernommen. Die Berechnung der *executionTime* für Busnachrichten hängt von der Tatsache ab, ob eine Quittierung der entsprechenden Nachricht auf dem Bus vorgesehen ist (Parameter *noResponse* im Systemmodell). Die MIL-STD-1553-Spezifikation und Messungen ergeben folgenden Zusammenhang:

$$executionTime = (messageSize + 1) * 20 + 33 \text{ wenn } noResponse{=}true \tag{7}$$

$$executionTime = (messageSize+2)*20+14 \text{ wenn } noResponse=false \quad (8)$$

Ist bei einem Item das Attribut *fixed=true*, handelt es sich um ein Item, das laut Systemmodell und entsprechendem Constraint zu einem festen Zeitpunkt eingeplant werden muss. Die Attribute *planned* und *ready* werden vom Backtracking-Algorithmus genutzt, um zu erkennen, ob ein Item eingeplant wurde bzw. ob die logischen Bedingungen des Items bereits erfüllt sind.

Sämtliche *plannerItem*-Objekte, die der Algorithmus zur Berechnung des Plans verwendet, werden in der Klasse *plannerItemList* verwaltet.

Constraints: Analog zu dem Vorgehen bei *PlannerItem* werden auch die im Systemmodell beschriebenen Constraints behandelt.

Listing 7: Attribute von *PlannerConstraint*

```
public class PlannerConstraint {

    private int id;
    private int fromId;
    private int toId;
    private short type;

    private int offset;
    private int fifoId;
    private int msgLength;
}
```

Die Variable *id* gibt den Identifier der jeweiligen Constraint an, *fromId* und *toId* verweisen auf entsprechende Items der Constraint. Das Attribut *type* definiert die Art der Bedingung. Wie bei den Items wurde auch hier im Gegensatz zu dem UML-Systemmodell bewusst auf die Verwendung einer Klassenhierarchie zum Differenzieren der Constraints verzichtet. Bedingt durch die kritische Zeitkomplexität von Backtracking-Algorithmen ist die Zugriffsgeschwindigkeit auf die benötigten Daten ein entscheidender Faktor. Tests haben ergeben, dass diese Art von Containern einen Vorteil bezüglich der Laufzeit von Zugriffen auf Attribute bieten.

Ressourcen des FCC: Die Klasse *PlannerResource* modelliert für den Planning-Algorithmus Ressourcen wie Prozessoren und Buscontroller.

Listing 8: Attribute von *PlannerResource*

```
public class PlannerResource {

        private int nodeId;
        private int minorFrameTime;

        private int [] plannedItemIndex;
        private int plannedCount=0;

        private int timeLine=0;
        private int gapEnd;
}
```

Bereits geplante Items werden in dem Index-Feld *plannedItemIndex* gespeichert, *plannedCount* gibt die Anzahl der auf dieser Ressource geplanten Items an. Die folgende Grafik verdeutlicht die Bedeutung der Attribute.

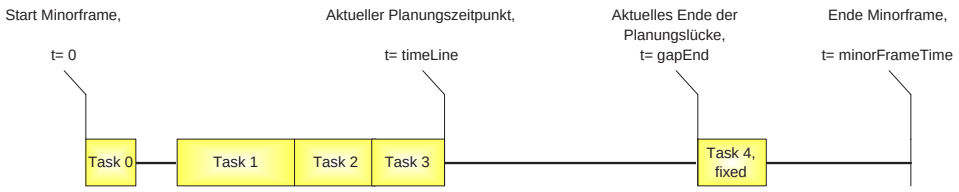

Abbildung 7: Planner-Ressource

Für die Planung steht ein Zeitraum zur Verfügung, der durch den Wert von *minorFrameTime* nach oben hin begrenzt wird. In der Regel werden Items fortlaufend eingeplant. Das Attribut *timeLine* gibt dabei den aktuellen Planungszeitpunkt an. Ausnahme bilden Items, die mit der Constraint "*FIXED*" oder "*OFFSETTED*" versehen sind. Items, die an einem explizit angegebenen Zeitpunkt starten müssen, werden vor dem Start des Backtracking-Algorithmus geplant, um die Tiefe des Suchbaums zu verringern. Dadurch kann auf der Zeitlinie der Ressource eine Lücke entstehen. Das Ende dieser Lücke wird durch *gapEnd* markiert.

Backtrack-Stack: Da für die Implementierung des Backtracking-Algorithmus nicht die rekursive Version implementiert wurde, besteht die Notwendigkeit, einen Stack (Stapel) zu verwenden. Ein Stapel kann im allgemeinen eine beliebige Anzahl von Objekten speichern und gibt diese in umgekehrter Reihenfolge wieder aus. Dazu werden die Methoden *push* (legt ein Objekt auf den Stapel) und *pop* (holt ein Objekt vom Stapel) verwendet.

Im Fall des Scheduling-Algorithmus entspricht ein Knoten im Suchbaum der Entscheidung, welches Item in diesem Schritt geplant wurde. Aus diesem Grund muss ein Stapel-Element folgende Informationen speichern:

- Das aktuell geplante Item, um die Scheduling-Entscheidung rückgängig machen zu können, falls sich herausstellt, dass der aktuelle Lösungsweg nicht mehr zu einer Gesamtlösung führen kann.
- Eine Liste mit alternativen Scheduling-Entscheidungen, also konkret eine Liste der Items, die im aktuellen Schritt bereit zum Einplanen sind.
- Eine Liste aller verwendeter Ressourcen

Mit Hilfe dieser Klassen ist es nun möglich, die Funktionsweise des Planning-Algorithmus zu beschreiben.

Bevor das Backtracking startet, werden zunächst mehrere nach verschiedenen Kriterien sortierte Felder berechnet. Sie dienen dem Zweck, schnell auf bestimmte Daten zugreifen zu können, ohne innerhalb des Backtracking-Algorithmus Suchoperationen über Constraint- oder Itemlisten ausführen zu müssen.

Listing 9: Kern des Backtracking-Algorithmus

```
while(solutionCount<desiredSolutions) {
   if (  ((btStack.size()-1)==itemsToBePlannedCount)
       &&(genNewSolution==false)) {
      // check if we reached a leaf.
      // this is a solution.

      solutionCount++;
      btStack.print();
      genNewSolution=true;

   } else {
      // this is not a leaf, there are still items to plan

      if (  (btStack.peek().hasReadyItems())
          &&(genNewSolution==false)) {
         // this node has untried children

         BacktrackStackElement topOfStack= btStack.peek();

         // clone resources for the current backtrack step
         PlannerResourceList plannerResourceList=
            new PlannerResourceList(topOfStack.getResources());

         // item to be planned
         PlannerItem currItem=
            selectPlanningItem(topOfStack, plannerResourceList);

         topOfStack.removeReadyItem(currItem);
         planItem(currItem, plannerResourceList);

         PlannerItem readyItems[]=
            calcReadyItems(plannerResourceList);

         if (readyItems.length==0) {
            // search for l1ready-items

            adjustTimelines(plannerResourceList);
            readyItems= calcReadyItems(plannerResourceList);
         }

         btStack.push(
            new BacktrackStackElement(currItem,
                                      readyItems,
                                      plannerResourceList));

      } else {
         // dead-end; pop the node off the stack
         BacktrackStackElement topOfStack = btStack.peek();
```

```
            if (topOfStack.getScheduledItem()==null) {
               // this is the root element
               return false;

            } else {
               unplanItem(topOfStack.getScheduledItem());
               btStack.pop();

               if (genNewSolution==true) {
                  genNewSolution=false;
               }
            }
         }
      }
   }
}
return true;
```

Der Algorithmus ist in der Lage, mehrere gültige Pläne zu berechnen, falls diese existieren. Die äußere *while*-Schleife implementiert diese Funktionalität. Der Algorithmus terminiert erst, wenn die vorher festgelegte Anzahl zu generierender Pläne erreicht wurde (`solutionCount<desiredSolutions`). Der Rumpf dieser Schleife wird bei jedem Backtracking-Schritt ausgeführt.

Die Bedingung, dass ein vollständiger Plan gefunden wurde, lässt sich mit Hilfe der Stackgröße formulieren: *itemsToBePlannedCount*, also die Anzahl der zu planenden Items, kann leicht aus dem Systemmodell bestimmt werden. Da im ersten Element des Stacks, dem Wurzel-Element, keine Scheduling-Entscheidung getroffen wurde, muss dieses von der aktuellen Stacktiefe abgezogen werden, um die tatsächliche Anzahl der aktuell geplanten Items zu erhalten. Sind nun beide Werte identisch, bedeutet dies, dass für jedes Item ein Eintrag im Stack existiert und somit jedes Item eingeplant wurde. Die Variable *genNewSolution* dient dem Algorithmus als Indikator, dass im vorangegangenen Schritt eine Gesamtlösung gefunden wurde und ein neuer Plan zu generieren ist.

Wurde in einer Iteration noch keine Gesamtlösung gefunden, kann dies zu mehreren Reaktionen führen.

Ist `(btStack.peek().hasReadyItems()==true)`, bedeutet dies, dass im aktuellen Entscheidungs-Knoten des Suchbaums noch unversuchte Lösungswege existieren. Ist dies der Fall, werden die Ressourcen dieses Knotens kopiert und ein Item aus der Liste der bereiten Items zum Planen ausgewählt.

Welches Item mit der Methode *selectPlanningItem* ausgewählt wird, entscheidet eine Heuristik, der folgende Erfahrungswerte zugrunde liegen:

- Mittels der Constraint "OFS" verbundene Items sind bei der Planung besonders kritisch, da die Planung des Ausgangs-Items der Constraint

auch direkt den Planungszeitpunkt des oder der Ziel-Items festlegt. Daher werden diese Item als erstes ausgewählt.

- Es ist möglich, bei der Modellierung des Systems an die Items Scheduling-Prioritäten zu vergeben, um eine manuelle Eingriffsmöglichkeit in die Planung zu haben. Werden Items mit solchen Prioritäten gefunden, werden diese in der Reihenfolge absteigender Prioritäten ausgewählt.

- Sind weder per "OFS" verbundene oder priorisierte Items in der Liste der bereiten Items, wird als nächstes das Item ausgewählt, von dem die meisten Constraints ausgehen. Ein Einplanen dieses Items erhöht somit die Wahrscheinlichkeit, dass im nächsten Schritt mehr Items bereit werden.

- Lässt sich aufgrund der obigen Kriterien keine Entscheidung fällen, werden zuerst die Busnachrichten gescheduled, da diese im Durchschnitt eine größere Laufzeit als Tasks haben.

Das ausgewählte Item wir daraufhin aus der Liste der bereiten Elemente entfernt und eingeplant. Da sich durch das Einplanen eines Items der Zustand der Ressourcen und das Evaluierungsergebnis der Constraints ändern können, muss die Liste der bereiten Items neu berechnet werden. Die Methode *calcReadyItems* versucht zunächst, zur Einplanung bereite Items zu finden, ohne auf einer der Ressourcen eine Lücke einplanen zu müssen. Eine solche Lücke würde bedeuten, dass auf der entsprechenden Ressource, beispielsweise einem Prozessor, Rechenzeit verschwendet wird, die den restlichen Tasks nicht mehr zur Verfügung steht. Je nach modelliertem Constraints kann es jedoch vorkommen, dass eine solche Lücke nicht zu vermeiden ist. Der Algorithmus überprüft dabei für alle Items, die noch nicht geplant wurden

- ob alle **ausgehenden** Constraints vom Typ "Offsetted" erfüllbar sind. Konkret wird berechnet, ob der Partner des aktuellen Items, der in diesem Constraint definiert wird, eingeplant werden kann. Ist dies nicht der Fall, ist das aktuelle Item nicht bereit. Ausgehende Constraints anderer Typen sind für die Betrachtung, ob ein Item bereit ist, nicht relevant.

- ob alle **eingehenden** Constraints erfüllbar sind. Hierbei muss zwischen den vier verschiedenen Constraint-Typen unterschieden werden. Bei "MEBS" und "CBF" muss das Partner-Item bereits eingeplant sein und terminiert haben. Ist diese Bedingung erfüllt, ist das Item bezüglich des entsprechenden Constraints einplanbar. Es kann jedoch noch sein, dass auf der entsprechenden Ressource die aktuell zur Planung benutzte Lücke zu klein oder der notwendige Planungszeitpunkt noch nicht erreicht ist. In diesem Fall wird das Item als "vorläufig nicht

bereit" markiert und bei der Anpassung der Ressourcen entsprechend berücksichtigt. Bei einem Constraint des Typs "CBF" muss zusätzlich noch zugewiesene FIFO frei sein, was bedeutet, dass sämtliche Elemente übertragen wurden.

Besitzt ein Item das Constraint "OFS" in eingehender Richtung, wird dieses generell als nicht bereit markiert, da diese Items geplant werden, wenn das Wurzel-Item, also das Item, von dem die "OFS"-Constraints ausgehen, geplant wird.

Bei Items, die mittels "MNO" verbunden sind, wird überprüft, ob der Partner entweder schon geplant und terminiert oder noch nicht eingeplant ist. Beide Fälle führen zum Bereitwerden des aktuellen Items. Auch in diesem Fall kann es vorkommen, dass Ressourcen angepasst werden müssen, um ein Bereitwerden zu ermöglichen.

Bleibt die Suche nach bereiten Items erfolglos, wird im nächsten Schritt versucht, auf den Ressourcen soviel ungenutzte Rechenzeit einzuplanen, dass wieder Items bereit werden können. Dies erledigt die Routine *adjustTimelines*. Darin wird zunächst untersucht, ob die aktuellen Lücken, die aktuell zur Planung von Items zur Verfügung stehen, noch groß genug sind, um mindestens einen der potentiell bereiten Tasks dort einzuplanen. Ist dies nicht der Fall, wird zur nächsten Lücke weitergeschaltet und damit die freie Rechenzeit der aktuellen Lücke als nicht planbar markiert und nicht weiter verwendet.

Der nächste Schritt, der bei dieser Methode durchgeführt wird, ist die Berechnung der minimalen Startzeiten aller potentiell bereiten Tasks. Dieser Schritt wird über alle Constraint-Typen und Knoten iteriert, so dass am Ende der Berechnung vier Zeitpunkte pro Knoten vorliegen. Wird der aktuelle Planungszeitpunkt jeder Ressource auf das jeweilige Maximum dieser vier Werte gesetzt, kann so sichergestellt werden, dass potentiell bereite Tasks beim nächsten Aufruf von *calcReadyItems* bereit werden.

Aus der so berechneten Liste der bereiten Items und der unter Umständen angepassten Liste aller Ressourcen wird nun ein neues Stack-Element erzeugt und auf den Backtracking-Stack gelegt. Mit dieser Aktion wird ein möglicher Zweig des Backtrackings, nämlich der Fall, dass die Gesamtlösung noch nicht gefunden wurde und noch Items in der Bereit-Liste liegen, beendet.

Für den Fall, dass der Backtracking-Algorithmus in einem Iterationsschritt eine leere Liste bereiter Tasks vorfindet, existieren zwei mögliche Ursachen: Entweder wurde der komplette Lösungsraum durchsucht, oder die Suche endete in einem Ast, der nicht zur Gesamtlösung führt und in dem keine weiteren Scheduling-Entscheidungen getroffen werden können. Der erste Fall wird durch die Tatsache erkannt, dass auf der obersten Position des Stacks ein NULL-Element liegt, das bei der Initialisierung des Stacks als

erstes Element dort abgelegt wurde. Daraufhin wird der Algorithmus mit der Rückgabe des Wertes *false* beendet.

Wurde der Lösungsraum noch nicht komplett durchsucht, wird die letzte Scheduling-Entscheidung, von der aus keine weiteren Items mehr eingeplant werden können, verworfen. Das Item, das im vorangegangenen Schritt geplant wurde, wird als ungeplant markiert und der letzte Scheduling-Schritt wird durch das Entfernen des obersten Stack-Elementes rückgängig gemacht. Da alle Zustände sämtlicher Ressourcen in jedem Stack-Element gespeichert sind, müssen diese hierbei nicht explizit modifiziert werden. Durch das Entfernen des obersten Elementes des Stacks wird automatisch der Ressourcen-Zustand der vorangegangenen Iteration wiederhergestellt.

Falls mehrere Lösungen erzeugt werden sollen, bereits eine Lösung gefunden wurde und damit die Variable genNewSolution==true ist, muss der Wert an dieser Stelle zurückgesetzt werden, damit der Algorithmus im nächsten Schritt wieder neue Lösungen evaluieren kann.

Wurde ein gültiger Plan gefunden, wird dieser in Form von C-Code und Doors-Skripten ausgegeben. Wie bei der Generierung der Graphviz-Skripte, deren Generierung von dem Erfolg des Planners unabhängig int, werden die Ausgaben mit Hilfe von Velocity-Templates erzeugt. Das folgende Beispiel verdeutlicht die Funktionsweise.

Listing 10: Funktionsweise von Velocity

```
public class VelocityExample {

    public static void main(String[] args) throws Exception
    {
        // get and initialize an engine
        VelocityEngine ve = new VelocityEngine();
        ve.init();

        // load the Template
        Template t= ve.getTemplate("velocityExample.vm");

        // create a context and add data
        VelocityContext context = new VelocityContext();
        TaskList taskList= new TaskList();
        context.put("taskList", taskList);

        // render the template into a StringWriter
        StringWriter writer= new StringWriter();
        t.merge(context, writer);

        // show it
        System.out.println(writer.toString());
    }
}
```

Nach dem Erzeugen und Initialisieren einer Velocity-Engine *ve* liest die Software das Template ein. Bei dem Rendern der Ausgabe, das durch die Methode *merge(...)* des Templates startet, wird der Inhalt des Template-Files in die Ausgabe überführt. Während dieses Vorganges wird Text, der in der Template-Datei steht, in die Ausgabe kopiert. Mit speziellen Befehlen innerhalb des Templates ist es möglich, beispielsweise Schleifen zu benutzen oder direkt auf Methoden von Klassen zuzugreifen, die vor dem Start der Engine mittels der Methode *context.put(...)* an Velocity übergeben wurden:

Listing 11: Velocity-Template

```
List of Tasks:

#foreach ($task in $taskList.getTaskList())
    $task.getName()
#end
```

Die Instanz der Klasse *TaskList* erlaubt durch die Methode *getTaskList()* Zugriff auf einen Vektor, der Elemente des Typs *Task* enthält. Die Schleife, die durch das Schlüsselwort *#foreach(Bedingung)* definiert wird, iteriert über alle Elemente des Vektors und greift über die Methode *task.getName()* direkt auf das Java-Objekt zu. Das Zeichen *$* weist dabei Velocity an, den folgenden Ausdruck zu evaluieren. Die Vorteile werden hier deutlich sichtbar:

- Die Syntax der Templates ist einfach und übersichtlich. Änderungen in der Ausgabe sind ohne Änderung des Codes möglich.
- Code und Templates sind getrennt, somit ist es möglich, beide voneinander unabhängig zu entwickeln und zu warten.
- Die Velocity-Engine kann in einfacher Weise auf alle öffentlichen Methoden zugreifen, die im Kontext zur Verfügung gestellt werden.

5.3 Schedulability Analyzer

Da die erzeugten Pläne in einem sicherheitskritischen Umfeld zum Einsatz kommen, ist es notwendig, größtmögliches Vertrauen in die Korrektheit solcher Schedules zu schaffen. Dies ist einer der Gründe, warum die in einem ersten Schritt generierten Pläne in einem zweiten Schritt nochmals verifiziert werden sollen. Diesem Zweck dient der Schedulability-Analyzer.

Software, die in einem sicherheitskritischem Umfeld zum Einsatz kommt, unterliegt besonderen Bestimmungen bezüglich der Zertifizierung. Für den Fluggerätebau kommt das Regelwerk DO-178B der RTCA zum Tragen. Die RTCA ("Radio Technical Commission for Aeronautics") existiert seit 1935 und besteht zwischenzeitlich aus über 250 Organisationen. Arbeitsgebiete der RTCA sind die Entwicklung und Herausgabe von Standards und Richtlinien für verschiedenste Bereiche der Luft- und Raumfahrt.

Der für diese Arbeit relevante Standart DO-178B ("Software Considerations in Airborne Systems and Equipment Certification") beschreibt notwendige Prozesse, Produkte und Aktivitäten, die für eine spätere Zertifizierung vorausgesetzt werden. Der Software-Level wird anhand der Auswirkungen eines möglichen Fehlers bestimmt und mittels der Stufen A bis F klassifiziert. Software-Level A bedeutet dabei, dass ein Fehlverhalten des betrachteten Systems zu katastrophalen Auswirkungen führen kann, in Systemen mit Software-Level F führen Fehler zu keinerlei Schäden an Personen oder Material.

Bei dem in dieser Arbeit erstellten Schedule-Planner handelt es sich um ein Werkzeug, das Code generiert und damit Prozesse und Aktivitäten, die normalerweise während der Entwicklung des Systems notwendig wären, ersetzt. Die Tatsache, dass der vom Schedule-Planner generierte Code nicht mehr von Hand verifiziert werden soll, führt zu der in der DO-178B beschriebenen Forderung der Tool-Qualification. Das bedeutet, dass die Entwicklung des Schedule-Planners den gleichen Richtlinien unterliegt, wie die restliche Software des FCC.

Der Standart DO-178B unterscheidet zwei Arten von Tools:

- Software development tools: In diese Kategorie fallen alle Tools, deren Ausgaben Teile flugkritischer Software sind. Es besteht also die Möglichkeit, dass aus einem Fehlverhalten des Tools direkt Fehler im Code des sicherheitskritischen Systems resultieren. Der Schedule-Planner ist nach dieser Definition ein Development Tool.

- Software verification tools: Diese Art von Tool kann nicht direkt Fehler in die Software einbringen, aber daran scheitern, vorhandene Fehler zu erkennen.

Für beide Kategorien von Tools gelten laut DO-178B unterschiedliche Anforderungen bezüglich des Aufwandes, der für eine Qualifizierung not-

wendig ist. Dieser ist für Verification-Tools geringer als für Development-Tools. Dies war ein weiterer Grund, neben dem Schedule-Planner einen Schedulability-Analyzer zu entwickeln. Daraus resultieren zwei Vorteile: Da es sich bei dem Schedulability-Analyzer um ein Verification-Tool handelt, ist der Qualifizierungsaufwand niedriger als bei einem Development-Tool. Außerdem ist der Code des Analyzers deutlich weniger komplex als der des Schedule-Planners, was wiederum zu einer Verringerung der Kosten einer Qualifizierung führt.

Der Schedulability-Analyzer benötigt für seine Aufgabe, die generierten Pläne gegen die im XML-Systemmodell definierten Bedingungen und Eigenschaften zu verifizieren, folgende Eingaben:

- Das XML-Systemmodell: Da dieses Modell sämtliche Ressourcen, Items und deren Abhängigkeiten zueinander beschreibt, ist es für den Schedulability -Analyzer die Grundlage der durchgeführten Testschritte.

- Pläne für Prozessoren: Der Schedule-Planner erzeugt für jeden Prozessor eine C-Codedatei, die den berechneten Plan auf dem Prozessor implementiert. Da diese Dateien direkt in die Software des FCC einfließen und Fehler an dieser Stelle fatale Auswirkungen haben könnten, werden sie vom Schedulability-Analyzer eingelesen und auf Korrektheit überprüft.

- Pläne für Buscontroller: Die Pläne für die im System vorhandenen Buscontroller werden nicht vom Schedule-Planner in Code umgesetzt, sondern nehmen den Weg über Skripte in das Requirements-Management-Tool Doors. Um eine Korrektheit der Pläne nachzuweisen, sind zwei Ansätze denkbar: Es ist entweder möglich, die vom Schedule-Planner generierten Skripte gegen das Systemmodell zu validieren oder die Daten über das Scheduling der Busse direkt aus Doors zu exportieren, und diese Daten für den Test zu verwenden. Letzter Weg hat den Vorteil, dass nicht nur die Korrektheit der Bus-Schedules nachgewiesen wird, sondern auch die Tatsache, dass die Pläne richtig in Doors integriert wurden.

Als Ausgabe soll der Analyzer zwei Dokumente generieren:

- Einen XML-Report: Um die Ergebnisse des Analyse-Vorgangs allgemein lesbar zu halten und die Weiterverarbeitung zu sichern, wurde auch an dieser Stelle das Format XML verwendet. Ein XML-Schema definiert die Struktur der Ausgabedatei und ermöglicht die automatische Codegenerierung via XMLSpy.

- Eine Transport-Delay-Matrix: Diese Matrix beschreibt die Differenz der Endzeiten sämtlicher Objekte zueinander. Anhand dieser Information lässt sich beispielsweise bestimmen, nach welcher Zeit das System

im schlechtesten Fall auf eine Änderung eines Sensorwertes reagiert. Würde in diesem Beispiel ein Sensorwert in Task T1 eingelesen, in T2 verarbeitet und durch T3 auf einen Ausgang gegeben werden, könnte man die Verzögerungszeit des Systems wie folgt berechnen:

Seien $MEBS(T_1 \rightarrow T_2)$ und $MEBS(T_2 \rightarrow T_3)$ im Systemmodell definiert, dann gilt

$$TransportDelay(T_1 \rightarrow T_3) = t_{end}(T_3) - t_{end}(T_1) \tag{9}$$

Der XML-Report beinhaltet neben der Auswertung der Verifizierung zudem noch Informationen über das Gesamtsystem, die teilweise aus dem Systemmodell abgeleitet werden und für Reviews nützlich sein können. Dazu zählen:

- Zusammenfassung aller Dateien, die Ein- oder Ausgabe der Analyse sind. So lässt sich auch im Nachhinein noch feststellen, welche Versionen miteinander verglichen wurden.
- Gesamtanzahl der Prozessoren, der Buscontroller, der Tasks, der Busnachrichten und der Constraints, aufgeschlüsselt nach Typen.
- Anzahl der Items pro Knoten, also Tasks pro Prozessor und Busnachrichten pro Buscontroller. Zudem die Summe der Laufzeiten aller Items eines Knotens, um die Auslastung auf dem Knoten abschätzen zu können.
- Auflistung aller Constraints und Items mitsamt ihrer Start- und Endzeiten, geordnet nach Knoten und Startzeiten.

Die Transport-Delay-Matrix wird wie folgt definiert:

Seien $I_1..I_m$ die im Systemmodell definierten Items, die in aufsteigender Reihenfolge ihrer Endzeiten sortiert sind, also $t_{end}(I_i) \leq t_{end}(I_{i+1})$

und sei

$$TDM : \{1, ..., m\} \times \{1, ..., m\} \rightarrow \mathbf{N}; (i, j) \longmapsto TDM(i, j) = a_{ij} \tag{10}$$

die Transport-Delay-Matrix, so gilt für deren Wert:

$$TDM = \begin{cases} t_{ex}(I_i) & \text{wenn } i = j; i = 1, ...m; j = 1, ..., m \\ t_{end}(I_j) - t_{end}(I_i) & \text{wenn } i < j; i = 1, ...m; j = 1, ..., m \\ t_{end}(I_j) - t_{end}(I_i) + t_{minorFrame} & \text{wenn } i > j; i = 1, ...m; j = 1, ..., m \end{cases} \tag{11}$$

Der Abstand zwischen I_i und I_j steht in der Matrix in der Zelle, die durch den Schnittpunkt von Zeile i mit Spalte j gebildet wird. Ist $i < j$,

wird Item I_i vor Item I_j beendet in einem Minor-Frame beendet. Ist $i > j$, wird Item I_j erst im nächsten Minorframe beendet. Die Diagonale gibt die Execution-Time der jeweiligen Items an.

Der Schedulability-Analyzer wird mittels einer XML-Datei konfiguriert. In dieser Datei stehen sämtliche Informationen, die für die Analyse notwendig sind, wie beispielsweise die Dateinamen der Ein- und Ausgabedateien. Das Schema dieser Konfigurationsdatei orientiert sich an dem von Java definierten Standart für Propertyfiles. Dies ermöglicht die Verwendung der Java-Klasse *java.util.Properties*.

Diese Datei wird beim Starten des Analyzers eingelesen und auf Plausibilität überprüft. Sind die Angaben korrekt, wird in einem zweiten Schritt das Systemmodell in den Speicher eingelesen. Der Busnachrichten-Schedule, der aus dem Requirements-Management exportiert wird, liegt auch in XML-Format vor und wird auf die selbe Weise geladen. Auch an dieser Stelle kommt dabei der autogenerierte Code zum Einsatz, der von XMLSpy erzeugt wurde.

Nächster Schritt ist das Erfassen der C-Codedateien, in denen die Schedules in Form einer Tabelle implementiert sind. Dabei stellt sich folgendes Problem: Es können auf verschiedenen Prozessoren Tasks laufen, die den exakt gleichen Tasknamen besitzen. Bei der Systemmodellierung wird dieses Problem durch die systemweit eindeutigen IDs gelöst, die jedes Item besitzt. In den Codedateien sind Informationen über die System-ID jedoch nicht mehr vorhanden, da die Abarbeitung der Tasks wie in Listing 2 beschrieben per Funktionspointer realisiert wurde. Dieser Funktionspointer entspricht dem Attribut *name* im Systemmodell und ist nicht eindeutig. Auch die Information, auf welchem Prozessor der Schedule überhaupt läuft, ließe sich aus den Informationen innerhalb der Quellcodedatei nur indirekt ableiten.

Aufgrund dieser Tatsachen werden bei der Generierung der Codedateien für Tasklisten die Dateien durch Ansi-C-kompatible Kommentare ergänzt, die eine Korrelation zwischen Task und Systemmodell ermöglichen. Der folgende Abschnitt zeigt ein solche Zeile:

Listing 12: Ausschnitt aus der Minor-Frame-Definition

```
{IO_Analogs_Receive, 987, 24},
    /* <Task index="15" taskId="CC_IO_Analogs_Receive"
    nodeId="CC Node" */
```

Das Attribut *index* verweist auf die Position innerhalb der Task-Tabelle. Der Wert von *taskId* ist gleich der systemweit eindeutigen ID, die auch bei der Modellierung zur Differenzierung der Items verwendet wird. *nodeId* ermöglicht die einfache Zuordnung des Tasks zu einem Knoten, wobei diese Information redundant ist, da sich aus *taskId* und dem Systemmodell bereits folgern lässt, auf welchem Knoten der Task implementiert ist.

Der Algorithmus, der die Quellcodedateien parst, versucht zuerst, den Anfang der Task-Tabelle zu finden. Da die Variable, unter der die Tabelle im Speicher abgelegt ist, in allen Dateien den gleichen Namen trägt, wird mit Hilfe eines regulären Ausdrucks nach diesem Konstrukt gesucht. Durch die Evaluierung der folgenden Klammern lässt sich feststellen, an welchem Punkt die Tabelle endet. Nachdem Leer- und Kommentarzeilen entfernt wurden, können die Tabelleneinträge ausgewertet werden. Da sich die bisherige Implementierung auf ein Minor-Frame beschränkt, wird dieser Umstand entsprechend überprüft.

Das Einlesen der Busnachrichten gestaltet sich deutlich einfacher, da diese Liste im XML-Format aus Doors exportiert wird. Sämtliche Informationen, die für die Analyse notwendig sind, werden in dem generierten Dokument zur Verfügung gestellt.

Sind alle geplanten Items erfasst, startet die eigentliche Analyse. Dabei werden verschiedene Punkte überprüft, die im Folgenden genauer erklärt werden.

- Execution-Times: In diesem Schritt werden die Execution-Times, die durch das Systemmodell festgelegt wurden, mit denen verglichen, die dann auf dem FCC durch Task- und Busnachrichentabelle den zeitlichen Ablauf der Items steuern.

- Geplante Items: Es wird überprüft, ob alle Items, die geplant wurden, auch tatsächlich im Modell existieren.

- Items des Modells: Alle im Modell definierten Items müssen auch in der auf dem FCC laufenden Software existieren.

- Überlappende Items: Durch einen Fehler des Schedule-Planners wäre es theoretisch denkbar, dass Items in einer Form falsch geplant wurden, dass sie sich gegenseitig überlappen. Dies hätte ein zeitlich nicht vorhersagbares Fehlverhalten des Systems zur Folge. Aus diesem Grund wird überprüft, dass sich keine zwei Items eines Knotens überlappen.

- Constraints: Hauptbestandteil der Analyse ist die Überprüfung der Constraints. Um eine korrekte Funktion des FCC zu gewährleisten muss sichergestellt sein, dass sämtliche Constraints erfüllt sind. Hierzu wird jede Constraint einzeln anhand der eingelesenen Daten verifiziert.

Falls eine der Überprüfungen negativ ausfällt, bricht der Analysevorgang mit einer entsprechenden Meldung ab. Verläuft der Test erfolgreich, wird der Schedulability-Report erzeugt, der die Ergebnisse zusammenfasst. Dabei wird sowohl ein XML- als auch ein PDF-Dokument generiert.

5.4 WCET Tracker

Damit ein Echtzeitsystem mit harten Echtzeitanforderungen stets korrekt funktioniert, muss sichergestellt sein, dass Deadlines unter keinen Umständen verletzt werden. Um dies sicherstellen zu können, ist es notwendig, die Laufzeit der Tasks im schlimmsten Fall (Worst Case Execution Time, oder kurz WCET) genau zu kennen.

Um die maximale Ausführungszeit zu bestimmen, müsste man folgendes Problem lösen: Gegeben ist ein Programm P. Gesucht ist ein weiteres Programm WCET, das die Laufzeit von P im schlimmsten Fall, also WCET(P), berechnet. Es lässt sich jedoch zeigen, dass ein solches Programm WCET nicht existieren kann.

Da alle jene Programme, die in eine Endlosschleife laufen, eine maximale Ausführungszeit ∞ haben, kann man das Programm WCET benutzen, um zu entscheiden, ob das zugrunde liegende Programm terminiert. Daher kann man WCET benutzen, um das Halteproblem zu lösen. Da das Halteproblem unentscheidbar ist, kann es das Programm WCET nicht geben.

Die maximale Ausführungszeit kann also im Allgemeinen nicht automatisch bestimmt werden. Weitere Faktoren erschweren die Bestimmung der WCET. Moderne Prozessoren besitzen zur Erhöhung der Rechenleistung Komponenten wie Pipelines und Caches. Diese Konstrukte machen die Laufzeitvorhersage komplex, da die Ausführungszeit einer Instruktion von der vorangegangenen Programmausführung abhängt und stark schwanken kann. Würde man Caches oder Pipelines zu Gunsten einer vereinfachten Laufzeitberechnung deaktivieren, könnte dies zu drastischen Einbußen der Performance führen. Zusätzlich führt umfangreiche und komplexe Software zu einer großen Anzahl möglicher Pfade, die durchlaufen werden und bei der Bestimmung der WCET berücksichtigt werden müssen.

Es existieren mehrere Möglichkeiten, die WCET zu analysieren. Zum einen besteht die Möglichkeit, den ausführbaren Code zu untersuchen([21]). Dies kann entweder manuell oder mit der Unterstützung eines Tools geschehen. Beispiele für solche Tools sind "aiT WCET Analyzer" ([1]) oder "RapiTime WCET Analyzer". Der Code wird dabei in der Regel in so genannte "Basic Blocks" unterteilt. Als Basic Block wird ein Codestück bezeichnet, das keine weiteren Verzweigungen enthält, in dem also die Befehle sequentiell abgearbeitet werden. Die genaue Kenntnis der Hardware vorausgesetzt, ist es möglich, die Ausführungszeit dieser Blöcke zu bestimmen. Komponenten wie Cache und Pipeline müssen dabei berücksichtigt beziehungsweise simuliert werden. Anhand des Kontrollgraphen, der aus dem ausführbaren Code rekonstruiert werden kann, ist die Bestimmung des längsten Pfades möglich, der die Berechnung der WCET ermöglicht.

Eine andere Möglichkeit, die WCET näherungsweise zu bestimmen, ist das Messen der Laufzeit. Obwohl es sehr einfach ist, Laufzeiten von Tasks zu

messen, besitzt diese Methode einen entscheidenden Nachteil im Vergleich zu einer statischen Analyse: Um die tatsächliche WCET zu messen ist es notwendig, das System mit allen denkbaren Eingaben in allen denkbaren Zuständen zu testen. Dies ist jedoch in der Praxis bei komplexen Systemen kaum möglich. Es bleibt also zu befürchten, dass das eigentliche Worst-Case Szenario nicht untersucht wurde und dadurch die gemessene WCET als zu niedrig angegeben wird.

Trotz dieser Nachteile wurde bei dieser Arbeit auf den Einsatz eines Tools verzichtet, das mittels einer statischen Analyse die Ausführungszeit der Tasks bestimmt. Dies liegt zum einen in der Tatsache begründet, dass ein solches Tool während der Erstellung der Arbeit noch nicht zur Verfügung stand, und zum anderen daran, dass trotz des Einsatzes eines Werkzeuges noch sehr viel Arbeit investiert werden muss, um damit Ergebnisse zu erzielen.

Um dennoch möglichst zuverlässige Laufzeitdaten zu erhalten, wurde der WCET-Tracker entworfen. Alle Laufzeitdaten sämtlicher Tasks, wie minimale Laufzeit, maximale Laufzeit und Deadlineverletzungen werden während der Ausführung auf dem FCC von der Systemsoftware erfasst und gespeichert. Dabei interessiert für die WCET-Analyse vor allem die maximale Laufzeit der Tasks.

Wird während eines Testlaufs des Systems eine Task-Laufzeit gemessen, die größer ist als die Laufzeit, die bei der Modellierung eines Tasks verwendet wurde, muss das Systemmodell angepasst und die Laufzeit des entsprechenden Tasks auf den neuen Wert gesetzt werden. Kommt es zu diesem Fall, ist es notwendig, die Schedules neu zu generieren. Diesen Arbeitsablauf unterstützt der WCET-Tracker.

Wie in Abbildung 2 zu sehen, benötigt der WCET-Tracker folgende Eingaben, um die Laufzeitdaten zu verarbeiten:

- Laufzeitdaten: Der FCC überträgt aus Instrumentierungs- und Analysegründen eine Vielzahl von Parametern auf den Bus, die von einem Band aufgezeichnet werden. Darunter befinden sich auch die Laufzeitdaten sämtlicher Tasks. Nach der Dekodierung des aufgezeichneten Datenstroms stehen diese Daten in Form einer Textdatei zur Verfügung. Darin aufgelistet sind die maximalen Laufzeiten der Tasks und die Version des Schedules, anhand dessen die Daten entstanden sind.

- Schedule: Da in der Aufzeichnung der Laufzeitdaten die Information über Tasknamen, also die Zuordnung von Tasknummer der RTMS-Table zu Taskname nicht mehr zur Verfügung steht, müssen diese Daten durch den entsprechenden Schedule zur Verfügung gestellt werden.

- Systemmodell: Die WCETs, mit denen die Pläne erzeugt wurden, sind im Systemmodell als Eigenschaften der Tasks angegeben. Da diese mit

den neuen, vom Band aufgezeichneten Daten verglichen und eventuell modifiziert werden sollen, muss dem WCET-Tracker das Systemmodell zur Verfügung stehen.

Der WCET-Tracker erzeugt nach der Analyse zwei Ausgaben. Ein autogeneriertes Matlab-Skript stellt mögliche Änderungen der Laufzeitdaten grafisch in Form eines Balkendiagramms dar. So ist sehr schnell ersichtlich, ob und wie sich die WCETs der Tasks geändert haben. Weiterhin wird ein neues Systemmodell generiert, in das die neuen Daten einfließen.

Wie bereits erwähnt, besteht bei dieser Methode der WCET-Bestimmung die Gefahr, nicht das Worst-Case Szenario auszuwerten und damit eine kleinere als die tatsächliche WCET zu benutzen. Aus diesem Grund wird die maximale Laufzeit bei der Generierung der Schedules mit einem Sicherheitsaufschlag belegt, um Fehler durch falsch bestimmte WCETs möglichst zu vermeiden. Die maximale Laufzeit, die für die Planung verwendet wird, errechnet sich durch

$$WCET_{planung} = (WCET_{gemessen} + 11) * 1.15 \tag{12}$$

Die gemessenen Zeiten werden mit dem Faktor (in diesem Fall 1.15) multipliziert. Zusätzlich wird die gemessene Laufzeit vor der Multiplikation mit einem festen Wert von 11μS beaufschlagt. Dies hat den Hintergrund, dass Messfehler bei Tasks mit kleinen Laufzeiten durch den Aufschlagsfaktor nur schwer aufgefangen werden können.

Der Programmablauf des WCET-Trackers ist dem des WCET-Analyzers sehr ähnlich. Da auch der Tracker mit dem selben XML-Systemmodell arbeitet, ist es möglich, auch an dieser Stelle den autogenerierten Code von XMLSpy zu verwenden. Dem Einlesen des Systemmodells folgt das Erfassen der Schedule-Tabellen in Form von C-Quellcodedateien. Auch an dieser Stelle kann auf den beim Schedulability-Analyzer entwickelten Code zum Parsen dieser Files zurückgegriffen werden.

Im nächsten Schritt werden die aufgezeichneten Laufzeitdaten eingelesen. Diese vom FCC über den Bus übertragenen und auf Band aufgezeichneten Daten werden von einer externen Software dekodiert und stehen daraufhin dem Tracker in folgender Form zur Verfügung:

Listing 13: Ausschnitt der Laufzeitdaten-Auswertung

```
BuildId: 20060208_194738
Task: 1 MaxRuntime: 5
Task: 2 MaxRuntime: 18
Task: 3 MaxRuntime: 9
Task: 4 MaxRuntime: 218
Task: 5 MaxRuntime: 125
Task: 6 MaxRuntime: 7
....
```

Ein wichtiger Punkt ist die Verfolgbarkeit der Schedules und der zugehörigen Laufzeitdaten. Anhand des Listings ist ersichtlich, dass die Daten keinerlei Informationen mehr über Tasknamen enthalten. Diese sind jedoch notwendig, um die maximalen Ausführungszeiten den entsprechenden Tasks zuordnen zu können. Aus diesem Grund gibt die erste Zeile der Datei, in der die Runtimes abgelegt sind, die eindeutige Identifikationsnummer des Schedules an, der für die Aufzeichnung der Laufzeitdaten verwendet wurde. Diese Nummer muss mit der des Schedules, der im zweiten Schritt eingelesen wurde, übereinstimmen.

Eine Identifikationsnummer wird bei jedem Generierungsvorgang eines Schedules neu erzeugt und setzt sich aus Datum und Uhrzeit zusammen. Da diese Nummer in Form zweier Variablen auch während der Ausführung der FCC-Software zur Verfügung steht, ist eine Übertragung über den Bus möglich. Das folgende Listung verdeutlicht die Implementierung:

Listing 14: Definition der Schedule-Identifikationsnummer

```
....
const GBL_UInt32Type RTMS_Table_Date = 0x20060223u;
const GBL_UInt32Type RTMS_Table_Timestamp = 0x170822u;
....
```

Durch den Vergleich dieser beiden Identifikationsnummern (BuildId der Laufzeitdatenauswertung und RTMS_Table_Date in Kombination mit RTMS_Table_Timestamp) kann der Tracker nun sicherstellen, dass die Laufzeitdaten mit Hilfe des angegebenen Schedules erstellt wurden. Diese Tatsache ermöglicht die eindeutige Zuordnung von Runtimedaten zu Tasks.

Nachdem sowohl Laufzeitdaten, Systemmodell und Schedules eingelesen sind, startet die Auswertung der Laufzeitdaten. Verglichen werden dabei die gemessenen Runtimes mit denen im Systemmodell definierten. Ist eine gemessene Laufzeit größer als die im Modell angegebene, bedeutet dies, dass im Modell eine unzureichend genaue WCET definiert ist, die durch den neuen Wert ersetzt werden muss. Der WCET-Tracker vergleicht in dieser Art und Weise die Laufzeiten sämtlicher Tasks und erstellt eine Kopie des Systemmodells, in das die neuen Daten einfließen. Dieses modifizierte Modell dient dann als Grundlage für den Schedule-Planner zur Generierung eines neuen Schedules. Um die Handhabung zu verbessern ist der WCET-Tracker in der Lage, in einem Durchlauf mehrere Laufzeitdaten-Files zu verarbeiten. Diese werden bei Programmstart als Parameter an den Tracker übergeben.

Zusätzlich zu dem modifizierten Systemmodell erzeugt der Tracker Matlab-Skripte, um Änderungen der Laufzeitdaten in Form einer Grafik darzustellen. Abbildung 8 zeigt einen solchen Graphen.

Auf der X-Achse werden die Tasks aufgetragen, die Höhe des Balkens

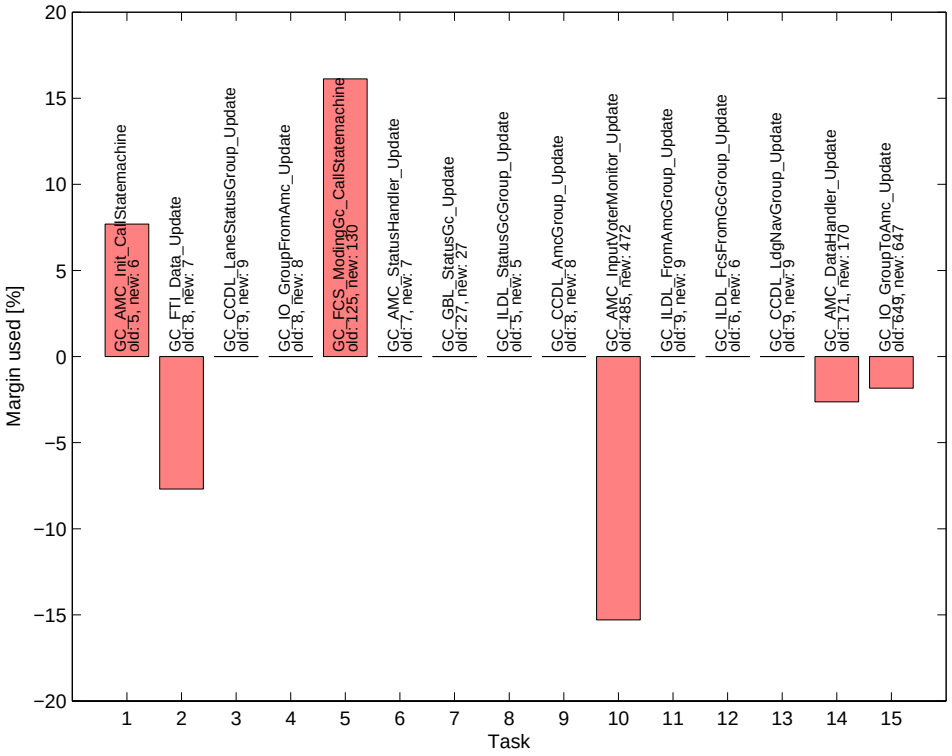

Abbildung 8: Matlab-Ausgabe der Laufzeitänderungen

wird durch

$$Margin = (WCET_{modell} + 11) * 1.15 - WCET_{modell}$$

$$MarginUsed[\%] = \frac{WCET_{gemessen} - WCET_{modell}}{Margin} * 100 \qquad (13)$$

bestimmt.

Der Wert sagt aus, wie viele Prozente des Sicherheitsaufschlags der maximalen Laufzeit des geplanten Tasks im Vergleich zu dem vermessenen Task benutzt wird. So ist es in einfacher Weise möglich zu bewerten, welche Laufzeiten sich geändert haben und ob Handlungsbedarf bezüglich einer Neuplanung besteht. Bleiben die Werte aller Balken unter 100%, treten aufgrund des Sicherheitsaufschlags keine Deadlineverletzungen auf, da sich die Laufzeit des Tasks immer noch im eingeplanten Bereich befindet. Des Weiteren sind dem Diagramm die konkreten Werte der maximalen Laufzeiten und die Namen aller Tasks zu entnehmen.

Das Matlab-Skript wird mit Hilfe der Velocity-Template-Engine erzeugt. Der folgende Ausschnitt zeigt die relevante Stelle, in dem die Taskdaten in

das Skript übertragen werden:

Listing 15: Ausschnitt Velocity-Skript

```
....
TaskData = struct( ...
    'Name',          {}, ...
    'wcet',          [], ...
    'newWcet',       []);

#foreach ($task in $exportData.getTaskData())
TaskData(end + 1).Name     = '$task.getTaskName()';
TaskData(end).wcet         = $task.getWcet();
TaskData(end).newWcet      = $task.getNewWcet();

#end
....
```

Die Struktur *TaskData*, in der Name und die alte und neue Laufzeit jedes Tasks definiert sind, stellt Matlab die benötigten Daten zur Verfügung. Gefüllt wird diese Struktur innerhalb der #foreach-Schleife, mit Hilfe derer Velocity über alle Tasks iteriert und die entsprechenden Daten in die Struktur schreibt.

6 Ergebnisse

Dieses Kapitel stellt die Ergebnisse vor, die mit der entwickelten Software erzielt wurden. Es werden sowohl Ergebnisse des realen Systems sowie theoretische Szenarien behandelt.

Sämtliche Tests wurden auf dem selben Rechner durchgeführt, der wie folgt spezifiziert ist:

CPU	Intel Pentium IV 2.8 GHz
Ram	512 Mb DDR
OS	Microsoft Windows XP SP2
Java HotSpot Client VM	1.5.0_04-b05

Tabelle 1: Testumgebung - Spezifikation

6.1 Test des FCC-Schedules

Der FCC, für den ein Schedule zu erstellen war, hat folgende Eigenschaften:

Prozessoren	3
Buscontroller	1
Tasks	82
Busnachrichten	45
Constraints gesamt	188
MEBS-Constraints	105
MNO-Constraints	46
OFS-Constraints	3
CBF-Constraints	34
FIXED-Constraints	4

Tabelle 2: FCC-Systemmodell Eigenschaften

Somit muss für eine Gesamtzahl von 127 Elementen ein Plan gefunden werden. Vier der Elemente besitzen das FIXED-Contraint, drei besitzen ein Constraint von Typ OFS. Es sind mindestens $127 - 3 - 4 = 120$ Scheduling-Entscheidungen notwendig, um einen vollständigen Plan zu errechnen. Dies ist genau dann der Fall, wenn der Schedule-Planner in der Lage ist, den gesuchten Plan ohne Backtracking zu errechnen. Die Anzahl der benötigten Scheduling-Entscheidungen, die zur Generierung eines Plans vom Schedule-Planner benötigt werden, ist gleichzeitig ein Maß für die Qualität der Heuristik. Eine schlechte Heuristik würde dazu führen, dass der Planner häufiger Wege im Lösungsbaum einschlägt, die nicht zu einer Gesamtlösung führen.

Folgende Tabelle fasst die Ergebnisse der Berechnung des Schedules für den FCC zusammen.

Pläne	Zeit (ms)	Scheduling-Entscheidungen
1	15	120

Tabelle 3: Laufzeit FCC-Schedule-Berechnung

Es wird deutlich, dass die erste Gesamtlösung, also ein Plan, der alle Constraints erfüllt, mit einem Minimum an benötigten Scheduling-Entscheidungen gefunden wird. Um die Performance des Schedulers zu testen, wird im nächsten Schritt mehr als nur ein Plan errechnet.

Pläne	Zeit (ms)	Scheduling-Entscheidungen	Pläne/Sekunde
50.000	4.281	263.950	11.697
100.000	6.890	526.500	14.513
500.000	35.344	2.914.408	14.146
1.000.000	71.196	5.896.167	14.045

Tabelle 4: Laufzeit FCC-Schedule-Berechnung für mehrere Pläne

Diese Ergebnisse zeigen die korrekte Funktion des Backtrackings, da nur in diesem Fall unterschiedliche, korrekte Pläne erzeugt werden können. Die Werte für die Berechnungszeit der Schedules beziehen sich auf die Generierung von Plänen für den FCC. Sie sind abhängig von dem verwendeten Systemmodell und lassen sich anhand dieses Beispiels nicht verallgemeinern. Es zeigt sich jedoch, dass für dieses gegebene Systemmodell die Berechnungszeit für einen Schedule unabhängig von der Anzahl der generierten Schedules ist. Aufgrund des verwendeten Backtracking-Algorithmus hängt der Speicherplatzbedarf nicht von der Laufzeit ab, sondern wird durch die Tiefe des Stacks begrenzt. Da im Speicher immer nur ein Ast des Suchbaums aufgebaut wird, beträgt der maximale Speicherplatzbedarf für n Tasks $n * c$ Bytes, c ist konstant.

Die Korrektheit der Pläne wird einerseits durch den Schedulability-Analyzer überprüft und durch Tests bestätigt. Hierzu wurde der Dispatcher des FCC um eine Funktion erweitert, der mögliche Überschreitungen von Deadlines der Tasks detektiert und aufzeichnet. Zusätzlich wird während der Laufzeit ausgewertet, ob beim Start eines Tasks, der den FIFO-Puffer des HSSL benutzt, dieser leer ist. Wäre dies nicht der Fall, würde das auf fehlerhaft berechnete Pläne und eine Verletzung der CBF-Constraints schließen lassen.

Zur Verifikation der Schedules standen Bänder zur Verfügung, auf denen Daten von über 300 Stunden Tests aufgezeichnet wurden. Die Auswertung ergab keinerlei Verletzungen von Constraints oder Deadlines.

6.2 Exemplarische Schedules

Dieser Anschnitt stellt die Ergebnisse des Schedule-Planners für exemplarische Systemmodelle und Constraints vor.

MEBS-Constraints: In diesem Beispiel sind drei Tasks wie folgt definiert:

Task	WCET [μS]	Knotted
T1	100	P1
T2	100	P1
T3	100	P1

Tabelle 5: Definition der Tasks (MEBS-Beispiel 1)

Die drei Tasks gleicher Länge laufen auf dem gleichen Prozessor. Die Tasks sind über zwei MEBS-Constraints verbunden.

Constraint	Typ	Task 1	Task 2
C1	MEBS	T1	T2
C2	MEBS	T2	T3

Tabelle 6: Definition der Constraints (MEBS-Beispiel 1)

Aus diesem Systemmodell errechnet der Scheduler folgenden Plan:

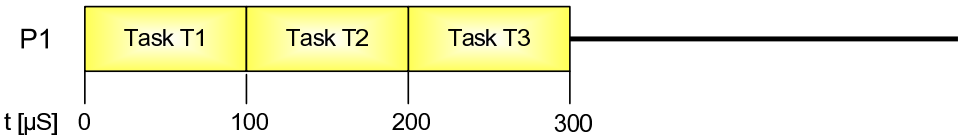

Abbildung 9: Schedule (MEBS-Beispiel 1)

Wie zu erwarten, plant der Scheduler die Tasks T1 T2 und T3 in dieser Reihenfolge auf dem Prozessor P1 ein. Verschiebt sich Task T2 auf einen anderen Prozessor T2, wie folgender Tabelle zu entnehmen ist,

Task	WCET [μS]	Knoten
T1	100	P1
T2	100	P2
T3	100	P1

Tabelle 7: Definition der Tasks (MEBS-Beispiel 2)

kommt es zu dem Fall, dass der Scheduler freie Zeit auf dem Prozessor einplanen muss:

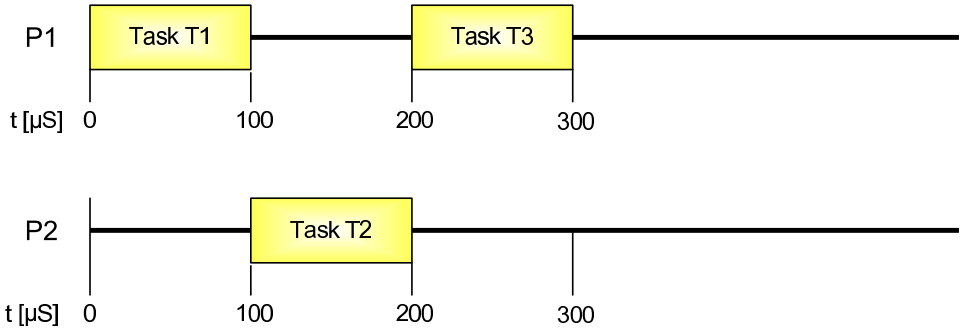

Abbildung 10: Schedule (MEBS-Beispiel 2)

Task T2 darf aufgrund der im Systemmodell definierten Constraint C1 erst ab einem Zeitpunkt eingeplant werden, an dem Task T1 beendet wurde. Gleiches gilt für die Tasks T2 und T3, die über Constraint C2 verbunden sind. Wie man leicht erkennt, ist der generierte Plan der optimale Plan bezüglich der Gesamtlaufzeit.

CBF-Constraints: Eine weitere Klasse von Constraints, deren Auswirkungen auf die generierten Pläne hier untersucht werden sollen, sind die CBF-Constraints. Wie in Kapitel 3.2.3 beschrieben, muss zur Erfüllung einer solchen Constraint sichergestellt sein, dass der verwendete FIFO-Puffer beim Start der Übertragung leer und die Daten beim Start des Zieltasks über-tragen wurden. Da der Schedule-Planner nicht feststellen kann, an welcher Stelle in einem Task der Sendevorgang startet, wird angenommen, dass dies zum letztmöglichen Zeitpunkt, nämlich am Ende des Tasks geschieht.

Für das Beispiel gelten folgende Tabellen, die Tasks und Constraints definieren. Des Weiteren wird davon ausgegangen, dass die Übertragung eines Wortes einer Nachricht 10μS in Anspruch nimmt.

Task	WCET [μS]	Knoten
T1	100	P1
T2	100	P1

Tabelle 8: Definition der Tasks (CBF-Beispiel 1)

Constraint	Typ	Task 1	Task 2	Parameter
C1	CBF	T1	T2	messageSize=12 via FIFO 1

Tabelle 9: Definition der Constraints (CBF-Beispiel 1)

Tasks T1 und T2 sind über die CBF-Constraint C1 verbunden. T1 sendet Daten über den FIFO-Puffer "FIFO 1", die in Task T2 benötigt werden.

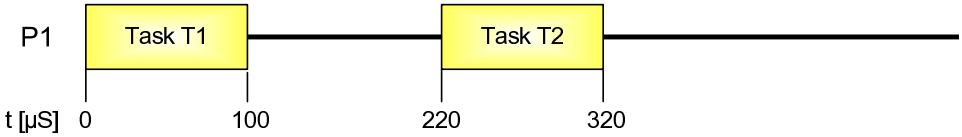

Abbildung 11: Schedule (CBF-Beispiel 1)

Da die Übertragung eines Nachrichtenworts 10μS in Anspruch nimmt, dauert es $messageSize * 10\mu S = 120\mu S$, bis die komplette Nachricht übertragen wurde. Dies wird in Abbildung 11 deutlich. Task T2 startet erst zu dem Zeitpunkt $t = 220\mu S$, nachdem alle Datenworte übertragen wurden.

Der Scheduling-Algorithmus ist in der Lage, Lücken, die durch die Verwendung von CBF-Constraints entstehen, zu füllen. Folgendes Systemmodell demonstriert diese Fähigkeit:

Task	WCET [μS]	Knoten
T1	100	P1
T2	100	P1
T3	100	P1

Tabelle 10: Definition der Tasks (CBF-Beispiel 2)

Der Unterscheid zu CBF-Beispiel 1 besteht in der Tatsache, dass dem Systemmodell ein zusätzlicher Task T3 hinzugefügt wurde, der keine Abhängigkeiten zu anderen Tasks besitzt. Die Definition der Constraints entspricht also der aus CBF-Beispiel 1.

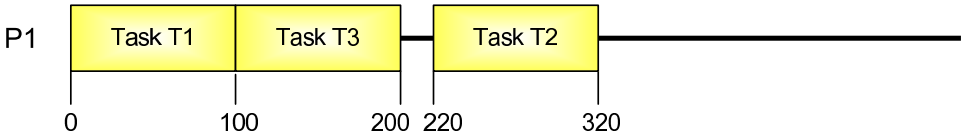

Abbildung 12: Schedule (CBF-Beispiel 2)

Abbildung 12 zeigt den Schedule, der durch den Schedule-Planner aus dem Systemmodell generiert wird. Wie zu erwarten ist, wird Task T3 zwischen Task T1 und Task T2 eingeplant und füllt die entstandene Lücke. Dies ist der optimale Plan bezüglich der benötigten Laufzeit. Zu Beginn des Planning-Vorgangs sind nur Tasks T1 und T3 bereit, da Task T2 von T1 abhängt. Würde T3 zuerst eingeplant werden, könnte die durch die Constraint C1 notwendige Lücke zwischen Task T1 und T2 nicht mehr genutzt werden und der Schedule würde sich um 100μS verlängern.

MNO-Constraints: Abschließend demonstriert das folgende Beispiel das Verhalten des Schedule-Planners im Falle von per "MNO"-verbundenen Items.

Task	WCET [μS]	Knoten
T1	100	P1
T2	100	P1

Tabelle 11: Definition der Tasks (MNO-Beispiel 1)

Da Constraints des Typs "MNO" eine Beziehung zwischen einem Task und einer Busnachricht herstellen, werden folgende Busnachrichten definiert, die dem Buscontroller B1 zugeordnet werden.

Message	Execution Time [μS]	Buscontroller
M1	80	B1
M2	80	B1

Tabelle 12: Definition der Busnachrichten (MNO-Beispiel 1)

Constraint	Typ	Task 1	Task 2
C1	MNO	T1	M1
C2	MNO	T2	M2

Tabelle 13: Definition der Constraints (MNO-Beispiel 1)

Die Constraints sagen aus, dass sich Task T1 zeitlich nicht mit Busnachricht M1 überlappen darf. Gleiches gilt für Task T2 und Busnachricht M2. Der Schedule-Planner liefert folgendes Ergebnis:

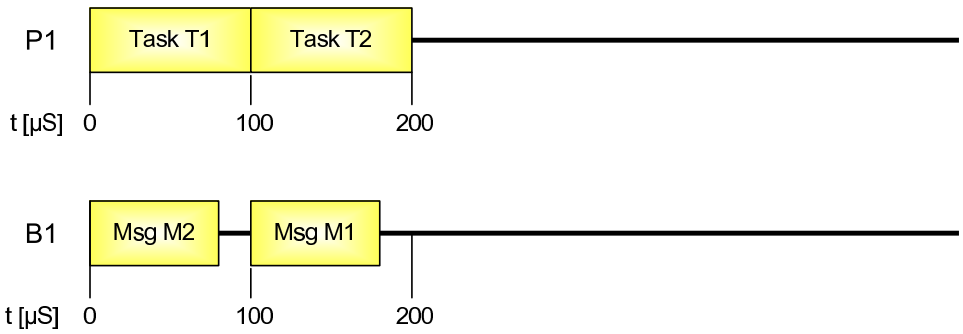

Abbildung 13: Schedule (MNO-Beispiel 1)

Beide Constraints C1 und C2 sind erfüllt. Ein Plan, in dem Tasks T1 und T2 beziehungsweise Busnachrichten M1 und M2 in umgekehrter Reihenfolge geplant sind, wäre bezüglich der Gesamtlaufzeit ebenfalls optimal.

7 Schlussbetrachtung

7.1 Zusammenfassung

Ziel dieser Diplomarbeit war das Design und die Implementierung eines Schedule-Planners für einen FCC. Zunächst wurden die Problembeschreibung, die Grundlagen über Scheduling und die dafür notwendigen Methoden wie Backtracking zusammengefasst, die für das Verständnis der weiteren Schritte notwendig waren.

Grundlage sämtlicher Entwicklungen ist das entwickelte Systemmodell. Nach der Identifizierung von Objekten und deren Beziehungen wurde für die erste Modellierung des Systems UML verwendet, eine standardisierte Sprache zur Beschreibung von Software und Systemen. Dieses Modell war Grundlage für die Abbildung des Systems auf ein XML-Modell. Dieser Schritt war sinnvoll, da durch die Verwendung eines XML-Schemas die Regeln für den Aufbau eines solchen Modells genau definiert werden. Ein XML-Dokument ist sowohl für Menschen als auch für Maschinen leicht lesbar und sichert die Interoperabilität der Daten. Ein weiterer Vorteil bei der Verwendung eines XML-Schemas liegt in der Möglichkeit, aus einem Schema mit Hilfe eines Tools automatisch Code zu generieren, der für die Weiterverarbeitung in der entwickelten Software notwendig ist. Aufgrund dieses Schemas wurde der FCC instanziiert.

Die Implementierung wurde in drei Teilprobleme partitioniert. Der erste Teil umfasst den Schedule-Planner. Diese Komponente deckt zwei Anforderungen ab. Aus dem instanziierten und in den Planner eingelesenen Systemmmodell werden Graphen generiert, die das System in anschaulicher Weise visualisieren. Die entwickelte Software errechnet Skripte, die für jeden Knoten des realen Systems die Objekte und deren Abhängigkeiten beschreiben. Das Open-Source-Tool Graphviz transformiert diese Skripte in gerichtete Graphen.

Hauptaufgabe des Schedule-Planners ist die Berechnung der Schedules. Hierzu wurde ein iterativer Backtracking-Algorithmus entwickelt, um auf Basis des Systemmodells zu einer Scheduling-Lösung zu gelangen, in der alle Constraints erfüllt sind. Um den Planner bezüglich der Performance zu optimieren, wurde eine auf den FCC und dessen Constraints angepasste Heuristik konzipiert. Damit sich der Schedule-Planner ohne die Notwendigkeit einer manueller Bearbeitung in den Software-Prozess des FCC einfügen kann, wird aus den erzeugten Plänen direkt lauffähiger Quellcode generiert, der sich nahtlos in die Codebasis des FCC integrieren lässt.

Das zweite Teilproblem adressiert die Verifikation der generierten Pläne. Da diese das Verhalten der Software eines sicherheitskritischen Systems bestimmen, wurde ein weiteres Tool entwickelt, das die Schedules anhand des

Systemmodells untersucht. Mittels dieses Schedulability-Analyzers werden alle Constraints und sämtliche Randbedingungen auf Erfüllung überprüft. Die Separierung dieser Funktionalität vom eigentlichen Planner war sinnvoll, da Software, die automatisch Code generiert und in sicherheitskritischen Systemen in der Luftfahrt zum Einsatz kommt, zertifiziert werden muss und sich die Zertifizierung für eine Analyse-Tool aufgrund der geringeren Komplexität einfacher gestaltet als für eine Planning-Tool. Die Ergebnisse der Analyse werden in einem XML-Report zusammengefasst, der mit Hilfe eines XML-Stylesheets in ein PDF-Dokument überführt wird. Eine zusätzlich erzeugte Transport-Delay-Matrix ermöglicht die Bewertung von Laufzeiten innerhalb des FCCs.

Die dritte Teilkomponente, die im Rahmen dieser Arbeit entstand, vereinfacht die Konsolidierung der gemessenen maximalen Laufzeit, der WCET, jedes einzelnen Tasks. Da dieser Wert entscheidenden Einfluss auf den Planungsvorgang hat, ist die Auswertung einer möglichst hohen Anzahl von aufgezeichneten Daten notwendig. Diese Aufgabe übernimmt der WCET-Tracker. Die entwickelte Implementierung ist in der Lage, eine beliebige Anzahl von Laufzeitdaten einzulesen, auszuwerten und das Systemmodell, in dem die WCET der Tasks definiert werden, automatisch anzupassen. Um anfallende Änderungen verfolgen zu können, wurde ein Skript generiert, mit dessen Hilfe Matlab die Veränderungen graphisch darstellt.

Um die Korrektheit und den praktischen Nutzen der erstellten Software zu zeigen, wurden zahlreiche Tests durchgeführt. Die Auswertung der umfangreichen Testdaten führte zu dem Schluss, dass die errechneten Pläne keinerlei Fehler aufwiesen. Durch den Einsatz des Schedule-Planners und der unterstützenden Tools wurde das Ziel erreicht, die Softwareentwickler beim Erstellen der Schedules weitgehend zu entlasten.

7.2 Ausblick

Obwohl der implementierte Schedule-Planner sowie die unterstützenden Tools wie Schedulability-Analyzer und WCET-Tracker ihre Funktion sowie ihren praktischen Nutzen in zahlreichen Tests an der konkreten Hardware bewiesen haben, besteht durchaus noch Potential für zukünftige Erweiterungen, die die Handhabung verbessern könnten.

Ein wesentlicher Punkt, der sich bei der Modellierung des Systems herausstellte, betrifft die Identifizierung der Constraints am Beispiel des FCC. Die Constraints werden in der aktuellen Situation per Hand identifiziert, was ein umfangreiches Wissen über das komplette System erfordert. Dieses Wissen besitzen nur wenige Entwickler, und die Praxis zeigte, dass eine vollständige Identifizierung aller Constraints ein langwieriger und zeitaufwändiger Prozess ist. Auch besteht die Gefahr, existierende Constraints zu übersehen oder fehlerhafte zu modellieren. Um dieser Situation entgegenzuwirken wäre es denkbar, durch Analyse der Systemsoftware des FCCs zumindest einen Großteil der Constraints automatisch aus dem Code zu extrahieren. So wäre es zumindest möglich, Constraints, bei denen Tasks über das Lesen bzw. Schreiben von Variablen gekoppelt sind, automatisch zu erkennen. Auch CBF-Constraints ließen sich mittels dieser Methode ausfindig machen, da das Schreiben und Lesen des FIFOs über autogenerierten Code und wohldefinierte Methoden erfolgt und sich die relevanten Codestellen somit leichter identifizieren lassen.

Ein weiterer Punkt, der betrachtet werden muss, ist die Methode der WCET-Bestimmung. Im aktuellen System werden die maximalen Laufzeiten der Tasks empirisch bestimmt. Das Vertrauen in die gemessenen Werte ist sehr hoch, da Laufzeitdaten von circa 300 Stunden Tests vorliegen und ausgewertet wurden. Trotzdem ist die Korrektheit dieser Werte dadurch keineswegs bewiesen. Absolute Sicherheit bezüglich der WCET erreicht man nur durch eine statische Codeanalyse, die mit Hilfe eines Tools oder manuell durchgeführt werden muss. Eine solche Analyse würde auch die Möglichkeit eröffnen, den Sicherheitsaufschlag, der vom Planner zu den gemessenen WCET-Werten hinzuaddiert wird, zu verringern oder komplett zu eliminieren. Damit stünde auf dem FCC mehr Rechenzeit für Tasks zur Verfügung.

Des Weiteren ist der Schedule-Planner derzeit auf ein einziges Minor-Frame beschränkt. Für den FCC, der die praktische Grundlage dieser Arbeit darstellt, ist dies ausreichend und gewünscht. Es sind jedoch andere Szenarien denkbar, in denen sich die Verwendung mehrerer Minor-Frames zur Erfüllung eines gültigen Schedules nicht vermeiden lässt. Diese Tatsache wurde sowohl bei der Modellierung als auch der Implementierung des

Systems berücksichtigt, jedoch sind noch Erweiterungen erforderlich, um Schedules erstellen zu können, die sich über mehrere Frames erstrecken.

Letztendlich steht noch die Zertifizierung des Schedulability-Analyzers aus. Dieser sehr zeitaufwändige Vorgang ist erforderlich, wenn autogenerierte Software wie die generierten Schedules ohne manuelle Reviews in sicherheitskritische Systeme wie das des FCCs einfließen. Der zeitliche Rahmen dieser Arbeit hätte solch einen Schritt jedoch nicht zugelassen. Der entwickelte Schedulability-Analyzer ist als Prototyp für einen zertifizierbaren Analyzer zu sehen, und wurde hinsichtlich der Kriterien, die dem Standart DO-178B definiert sind, optimiert.

Tabellenverzeichnis

1 Testumgebung - Spezifikation 47
2 FCC-Systemmodell Eigenschaften 47
3 Laufzeit FCC-Schedule-Berechnung 48
4 Laufzeit FCC-Schedule-Berechnung für mehrere Pläne 48
5 Definition der Tasks (MEBS-Beispiel 1) 49
6 Definition der Constraints (MEBS-Beispiel 1) 49
7 Definition der Tasks (MEBS-Beispiel 2) 50
8 Definition der Tasks (CBF-Beispiel 1) 51
9 Definition der Constraints (CBF-Beispiel 1) 51
10 Definition der Tasks (CBF-Beispiel 2) 52
11 Definition der Tasks (MNO-Beispiel 1) 53
12 Definition der Busnachrichten (MNO-Beispiel 1) 53
13 Definition der Constraints (MNO-Beispiel 1) 53

Abbildungsverzeichnis

1 Aufbau des FCC 9
2 Toolchain 15
3 UML-Systemmodell 18
4 XML-Schema des Systemmodells 20
5 Datenfluss zwischen Containern 23
6 Constraint-Graph 24
7 Planner-Ressource 28
8 Matlab-Ausgabe der Laufzeitänderungen 45
9 Schedule (MEBS-Beispiel 1) 49
10 Schedule (MEBS-Beispiel 2) 50
11 Schedule (CBF-Beispiel 1) 51
12 Schedule (CBF-Beispiel 2) 52
13 Schedule (MNO-Beispiel 1) 54

Glossar

CCDL	(Cross Channel Datalink) - HSSL, der verschiedene Lanes eines FCC verbindet	10
EADS	(European Aeronautic Defence and Space Company) - EADS ist ein weltweit führendes Unternehmen der Luft- und Raumfahrt, im Verteidigungsgeschäft und den dazugehörigen Dienstleistungen.[14]	2
FCC	(Flight Control Computer) - Hardware zur Steuerung eines Fluggerätes, Teil eines Flight Control Systems	2
FCS	(Flight Control System) - Gesamtsystem (Software und Hardware) zur Steuerung eines Fluggerätes	2
FIFO	(First in First out) - Verfahren der Speicherung, bei denen diejenigen Elemente, die zuerst gespeichert wurden, auch zuerst wieder aus dem Speicher entnommen werden[36]	10
HSSL	(High Speed Serial Link) - Serielles Bussystem innerhalb des FCC	10
ILDL	(Intra Lane Data Link) - HSSL, der mehrere Prozessoren einer Lane verbindet	10
Java	Objektorientierte Programmiersprache und als solche ein eingetragenes Warenzeichen der Firma Sun Microsystems. Sie ist eine Komponente der Java-Technologie.[36]	2
UAD	(Unmanned Aircraft Demonstrator) - Simulationssystem eines unbemannten Flugzeugs	2
WCET	(Worst Case Execution Time) - Maximale Ausführungszeit eines Tasks oder Programms	16

Literatur

[1] AbsInt Angewandte Informatik GmbH. aiT WCET Analyzer, March 2006. www.absint.de.

[2] Altova, Inc. Altova XMLSpy Datasheet, March 2006. www.altova.com.

[3] Barták, Roman. Constraint Programming: In Pursuit of the Holy Grail.

[4] Bate, Iain John. *Scheduling and Timing Analysis for Safety Critical Read-Time Systems*. PhD thesis, Department of Computer Science - University of York, November 1998.

[5] Chapin, Steve J., und Weissman, Jon B. DISTRIBUTED AND MULTIPROCESSOR SCHEDULING, 2001.

[6] Clements, Paul C. . A Survey of Architecture Description Languages, March 1996.

[7] Committee 167 of RTCA. Software Considerations in Airborne Systems and Equipment Certification (RTCA/DO-178B), 1992.

[8] Condor Engineering, Inc. MIL-STD-1553 Tutorial, 1993.

[9] Cucu, Liliana, and Sorel, Yves . Non-preemtive multiprocessor scheduling of strict periodic systems with precendence constraints, November 2004.

[10] Dechter, Rina, und Frost, Daniel . Backtracking algorithms for constraint satisfaction problems - a total survey, April 1998.

[11] Di Gaspero, L. *Local Search Techniques for Scheduling Problems: Algorithms and Software Tools*. PhD thesis, Universita degli Studi di Udine - Dipartimento di Matematica e Informatica, December 2002.

[12] EADS. Diploma Thesis Problem Description, September 2005.

[13] Eclipse Foundation. eclipse.org, March 2004. www.eclipse.org.

[14] European Aeronautic Defence and Space Company. eads.com, March 2006. www.eads.com.

[15] Fohler, Gerhard. Analyzing a Pre Run Time Scheduling Algorithm and Precedence Graphs, June 1995.

[16] Fohler, Gerhard, and Koza, Christian. Heuristic Scheduling for Distributed Hard Real-Time Systems, 1990.

[17] Fromherz, Markus P.J. Constraint-based Scheduling, Juni 2001.

[18] Gansner, Emden, and Koutsofios, Eleftherios, and North, Stephen . Drawing graphs with dot, 2002. `www.graphviz.org`.

[19] Gansner, Emden R., and Koutsofios, Eleftherios, and North, Stephen C., and Vo, Kiem-Phong. A Technique for Drawing Directed Graphs, 1990.

[20] Goossens, Michel, und Mittelbach, Frank, and Samarin, Alexander. *Der LATEX Begleiter*. Addison-Wesley, 2000.

[21] Heckmann, Reinhold, und Ferdinand, Christian. Worst-Case Execution Time Prediction by Static Program Analysis, 2004.

[22] IEEE World Congress on Computational Intelligence. *Dynamic Scheduling of Computer Tasks Using Genetic Algorithms* (July 1994). Proceedings of the First IEEE Conference on Evolutionary Computation.

[23] Jones, James C. GUIDELINES FOR THE QUALIFICATION OF SOFTWARE TOOLS USING RTCA/DO-178B, 2001.

[24] Kurtulus, MÜmin. Multiprocessor Task Scheduling, April 1999.

[25] Matuszek, David. Programming Languages and Techniques II, 2005.

[26] Nehmer, JÜrgen, und Sturm, Peter. *Systemsoftware*. Dpunkt Verlag, März 2001.

[27] Richter, Kai, und Ziegenbein, Dirk, und Jersak, Marek und Ernst, Rolf. Model Composition for Scheduling Analysis in Platform Design, June 2002. e-mail: {richter, ziegenbein, jersak, ernst}@ids.ing.tu-bs.de.

[28] Schemel, Holger. Verwendung effizienter Operations-Research-Algorithmen im Rahmen von Constraint Programming zur Modellierung von Scheduling-Problemen, April 1996.

[29] Schärer, S. *Design and Implementation of an EF FCC (Eurofighter Flight Control Computer) Schedule Planner (EFSP)*. PhD thesis, Universität Kaiserslautern, März 2004.

[30] Sha, Lui, and Sathaye, Shirish S. . Distributed Real-Time System Design: Theoretical Concepts and Applications. Tech. rep., Software Engineering Institute - Carngegie Mellon University, Pittsburgh, Pennsylvania 15213, March 1993.

[31] Sun Microsystems, Inc. JavaTM 5 Platform, Standard Edition, v 1.5.2 API Specification, 2003. java.sun.com/j2se/1.5.0/docs/api/.

[32] The Castor Project. Castor, 2006. www.castor.org.

[33] W3C. Extensible Markup Language (XML), November 2003. http://www.w3.org/XML/.

[34] W3C. XML Schema, November 2003. http://www.w3.org/XML/Schema.

[35] Whitley, L.D., and Howe, A.E., and Rana, S., and Watson, J.P., and Barbulescu, L. Comparing Heuristic Search Methods and Genetic Algorithms for Warehouse Scheduling, 1998. e-mail: {whitly, howe, rana, warsonj, laura}@cs.colostate.edu.

[36] Wikipedia. Diverse Artikel, März 2006. de.wikipedia.org.

[37] Ziegenbein, Dirk, and Ernst, Rolf, and Richter, Kai and Teich, J., and Thiele, L. Combining Multiple Models of Computation for Scheduling and Allocation, March 1998.