9 Eingebettete Echtzeitsoftware[1]

Bisher wurden die Probleme, die bei der Entwicklung eingebetteter Systeme (= *Embedded Systems*) auftreten, im Software Engineering zu wenig beachtet. Die Softwareentwicklung für eingebettete Systeme konzentriert sich in der konventionellen Herangehensweise darauf, isolierte Kontroll- und Regelungsaufgaben zu behandeln und dabei die meist vorgegebenen und limitierten Hardware-Ressourcen bei Garantie der Echtzeitanforderungen bestmöglich auszunutzen. Das lässt wenig Freiraum für Abstraktionen und Software-technische Konzepte, die in anderen Anwendungsbereichen zur besseren Beherrschbarkeit der Komplexität, zur Rationalisierung des Entwicklungsprozesses und zur Verbesserung der Software-Qualität geführt haben.

Wegen des verbesserten Preis-/Leistungsverhältnisses bei Hardware, dem wachsenden Markt und den steigenden Anforderungen an die Funktionalität eingebetteter Software ist es angebracht, dass man sich damit auseinander setzt, mit welchen Techniken und Methoden man die Probleme, denen man bei der Entwicklung eingebetteter Echtzeitsysteme begegnet, am besten lösen kann.

Da das Gebiet der Echtzeitsysteme ein komplexes und umfangreiches Spezialgebiet ist, kann es im Rahmen dieses Buches nicht annähernd erschöpfend behandelt werden. Es ist jedoch das Anliegen der Verfasser, den Leser auf grundsätzliche Probleme, mit denen man bei der Entwicklung von Echtzeitsoftware konfrontiert ist, aufmerksam zu machen und einen Eindruck davon zu vermitteln, wie man diese Probleme lösen kann.

In diesem Kapitel wird zunächst auf grundlegende Aspekte von Echtzeitsoftware eingegangen. Anhand eines Fallbeispiels wird danach skizziert, mit welchen Methoden und Werkzeugen der Übergang von einer primär Hardware-zentrierten Entwicklung von (eingebetteten) Echtzeitsystemen zu einer Software-zentrierten Entwicklung erfolgen kann. Das Kapitel ergänzt somit die in Kapitel 4 eingeführten Sprachkonstrukte um Konstrukte zur Unterstützung der Realisierung von eingebetteter Echtzeitsoftware und zeigt, wie Modularisierung (siehe dazu Kapitel 6) im Kontext von Echtzeitsoftware erfolgen kann.

9.1 Grundlegende Konzepte und Voraussetzungen

Ein eingebettetes System ist ein Computersystem, das in der Regel über Sensoren und Aktuatoren in ein anderes System eingebettet ist (vergleiche auch Simon, 1999): Eingebettete Systeme verarbeiten und interpretieren Signale unterschiedlicher Medientypen und beeinflussen oder kontrollieren dadurch das System, in dem sie eingebettet sind.

[1] Zusammen mit Christoph Kirsch (Computational Systems Group, Fachbereich Informatik, Universität Salzburg) und Josef Templ (Software Templ OEG, Linz und Software Research Group, Fachbereich Informatik, Universität Salzburg); unterstützt von der FIT-IT Embedded Systems Initiative des österreichischen Bundesministeriums für Verkehr, Innovation und Technologie (bmvit) im Projekt MoDECS (807144).

Eingebettete Systeme finden wir beispielsweise in Mobiltelefonen, bei Antiblockiersystemen in Fahrzeugen oder in Satellitensteuerungen. Die Schlüsselkomponenten von eingebetteten Systemen sind in zunehmendem Maße nicht mehr die Hardwarekomponenten, sondern die erforderlichen Software-Komponenten.

9.1.1 Charakteristika von Echtzeitsoftware

Viele eingebettete Systeme sind so genannte Echtzeitsysteme. Das heißt, es kommt nicht nur darauf an, dass die von ihnen ausgeführten Operationen und berechneten Ergebnisse korrekt sind, sondern pünktlich zu einem bestimmten Zeitpunkt (Deadline) erfolgen beziehungsweise zur Verfügung stehen. Echtzeitsysteme, die noch akzeptabel funktionieren, obwohl eine Deadline geringfügig überschritten wurde, werden oft als „weiche" Echtzeitsysteme bezeichnet. Meist wird bei Nichteinhalten der Deadlines bei weichen Echtzeitsystemen die Qualität der Ergebnisse und der erbrachten Dienste schlechter, aber die Funktionalität wird noch bereitgestellt. Ein Beispiel ist die Dekompression von Videodaten, die über ein Netzwerk geladen werden. Je langsamer die Dekompression erfolgt, desto schlechter ist die Qualität des angezeigten Videos. In diesem Zusammenhang wurde, um beispielweise den Grad der Anforderungserfüllung festzulegen oder festzustellen, der Begriff „*Quality of Service*" eingeführt.

Bei so genannten „harten" Echtzeitsystemen müssen die Deadlines unter allen Umständen eingehalten werden, da sonst gefährliche Situationen oder gar Katastrophen die Folge sein können. Beispiele für harte Echtzeitsysteme sind Flugzeugsteuerungen, Motormanagement, Bremsassistent oder der Auslöser eines Airbags.

Bei der Entwicklung von harten Echtzeitsystemen ist man mit viel schwieriger zu lösenden Problemen konfrontiert, als bei der Entwicklung von weichen Echtzeitsystemen. Ein hartes Echtzeitsystem muss unter allen möglichen Ausnahme- und Fehlerbedingungen die Deadlines einhalten. Die Umgebung, in der ein hartes Echtzeitsystem eingesetzt wird, gibt vor, welches Zeitverhalten verlangt wird und welcher Grad an Parallelität beziehungsweise Verteilung auf Seite der Software zu beherrschen ist. Beispielsweise definieren die Charakteristika eines Verbrennungsmotors die Anforderungen an die zugehörige Echtzeitsoftware, also welche parallel ablaufenden chemischen und mechanischen Prozesse im Motor beobachtet werden müssen, in welchen Zeitabständen, zum Beispiel vorgegeben durch die Motordrehzahl, die Messungen durch Sensoren und die Regelungsimpulse durch Aktuatoren erfolgen müssen.

Die Echtzeitsoftware muss innerhalb fix vorgegebener Zeitlimits Berechnungen durchführen, um die Stellwerte an die Aktuatoren zu liefern, so dass das erwünschte Verhalten des Motors erreicht wird. Die Anforderung, verschiedene Prozesse in der realen Welt parallel, das heißt, gleichzeitig zu beobachten und zu regeln führt dazu, dass auch die dafür eingesetzte Software imstande sein muss, Prozesse gleichzeitig, das heißt parallel auszuführen.

Im Gegensatz zu einem parallel arbeitenden Programm ist ein Echtzeitprogramm nur dann korrekt, wenn nicht nur die gewünschten Ergebnisse (Ausgabeobjekte) bezogen

auf die Eingabeobjekte korrekt erzeugt werden, sondern die Ein- und Ausgabeobjekte auch zum gewünschten Zeitpunkt oder innerhalb eines gewünschten Zeitraumes zur Verfügung stehen. Diese Anforderung hat fundamentale Auswirkungen auf den gesamten Entwicklungsprozess von Echtzeitprogrammen. Viele der herkömmlichen Techniken aus dem Gebiet der parallelen Programmierung (siehe dazu zum Beispiel Lea, 1999) lassen sich nur mit großem Aufwand in der Echtzeitprogrammierung verwenden.

Ein wesentliches Problem stellt das nichtdeterministische Verhalten von Software-Prozessen wie zum Beispiel von Java-Threads dar. Wenn man einem System von Java-Threads eine bestimmte Folge von Eingaben zu exakt festgelegten Zeitpunkten zur Verfügung stellt, lässt sich im Allgemeinen das Verhalten des Systems, also die Ausgaben, deren Reihenfolge und die Zeitpunkte, zu denen die Ausgaben erfolgen, nicht reproduzieren. Bei einer Wiederholung kann das System trotz exakt gleicher Eingaben ein anderes Ausgabeverhalten zeigen. Das liegt daran, dass die Ausführungsreihenfolge und Ausführungsdauer von Java-Threads von vielen Faktoren, wie dem verwendeten Scheduling-Verfahren und der CPU-Geschwindigkeit und CPU-Last, abhängen. In vielen Anwendungen ist diese Art von Nichtdeterminismus nicht problematisch. Bei harten Echtzeitsystemen stellt das nichtdeterministische Verhalten jedoch ein ernsthaftes, in den meisten Fällen nicht akzeptables Problem dar, weil man sich auf das korrekte Verhalten des Systems nicht mehr verlassen kann. Im Folgenden werden wir uns darauf konzentrieren, wie man diese Problematik in den Griff bekommt.

9.1.2 Logische versus tatsächliche Ausführungszeiten von Echtzeitsoftware

Zur Illustration der Vorgehensweise bei der Entwicklung von Echtzeitsoftware beschreiben wir in diesem Abschnitt die Implementierung von Echtzeitsoftware-Komponenten für einen Modellhubschrauber mithilfe der so genannten Timing Description Language (TDL; Templ, 2004), die aus der Programmiersprache Giotto (Henzinger et al., 2001) hervorgegangen ist. Die Semantik von Giotto und damit von TDL basiert auf einem Programmiermodell, das sich zur Realisierung harter Echtzeitsysteme, wie zum Beispiel Regelungssysteme, eignet.

Eine typische Aufgabe eines Regelungssystems ist, parallel ablaufende Software-Prozesse auszuführen, die zum Beispiel periodisch Sensoren abfragen, dann Funktionen basierend auf den Sensorwerten berechnen und abschließend Aktuatoren mit den berechneten Funktionswerten ansteuern. In Giotto/TDL gibt es zu diesem Zweck so genannte Tasks. Ein *Giotto/TDL-Task* ruft periodisch eine Funktion auf, die aus den Eingabewerten und dem Taskzustand neue Ausgabewerte und einen neuen Taskzustand liefert. Ein Giotto/TDL-Task ist ein Software-Prozess, der periodisch immer dasselbe sequentielle Programm, das die Taskfunktion berechnet, ausführt.

Die entscheidende Abstraktion von Giotto/TDL ist, für die Berechnungsdauer der Taskfunktion eine logische Ausführungszeit, *logical execution time* (LET), statt der tatsächlichen Ausführungszeit, anzunehmen (siehe Abbildung 9.1). In Giotto/TDL entspricht

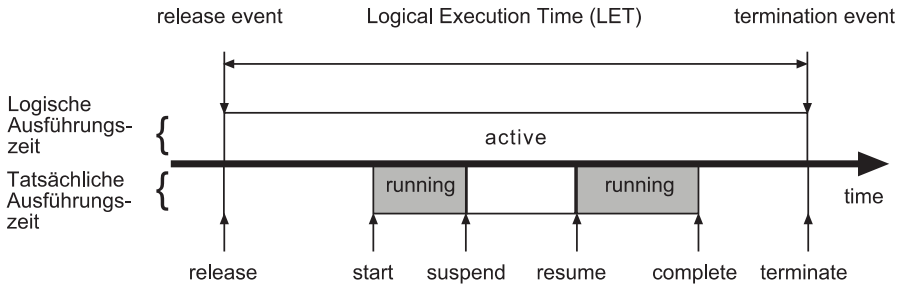

Abbildung 9.1: Logische versus tatsächliche Ausführungszeiten einer Giotto/TDL Task

die LET eines Tasks der Periode des Tasks. Ein 100-Hz-Task hat zum Beispiel eine Periode von 10 ms und damit eine LET von 10ms. Die LET des Tasks besagt, dass der Task für die Ausführung der Taskfunktion vom Lesen der Eingabewerte, *release event*, bis zur Bereitstellung der Ausgabewerte, *termination event*, exakt 10 ms benötigt, unabhängig davon, wie schnell der ausführende Prozessor ist oder wie die Taskfunktion implementiert ist. Das illustriert der als „Logische Ausführungszeit“ bezeichnete Teil oberhalb der Zeitachse. Der untere, als „Tatsächliche Ausführungszeit“ bezeichnete Teil zeigt exemplarisch, wie ein Task auf einer konkreten Plattform, also auf einem Prozessor oder mehreren Prozessoren und einem bestimmten Betriebssystem ausgeführt wird.

In Abbildung 9.1 wird ausgedrückt, dass die Ausführung des Tasks nicht unmittelbar nach dem Release-Event startet. Der eigentliche Start der Ausführung ist mit „start“ markiert. Die Task-Ausführung wird einmal unterbrochen („suspend“) und wird bei „resume“ wieder fortgesetzt, bis „complete“ eintritt, etwas vor dem Termination-Event, das durch die LET festgelegt ist.

Während der Ausführung des Tasks innerhalb seiner LET kann der Task weder mit Sensoren und Aktuatoren noch mit anderen Tasks kommunizieren, selbst wenn der Task die Berechnung seiner Taskfunktion nach dem Beginn seiner LET startet oder vor dem Ablauf seiner LET beendet oder während der LET unterbricht. Zur Übersetzung eines Giotto/TDL-Programms muss vom Giotto/TDL-Compiler überprüft werden können, ob alle Tasks tatsächlich innerhalb ihrer LETs auf einer gegebenen Plattform von einem oder mehreren Prozessoren ausgeführt werden können. Die dazu nötige Information wird dem Compiler zum Beispiel in Form der so genannten *Worst Case Execution Time (WCET)*-Angaben für jede Taskfunktion zur Verfügung gestellt. Ein Giotto/TDL-Programm, für das diese Eigenschaft gilt, bezeichnen wir als *time-safe* bezüglich der gegebenen Plattform. Ein Giotto/TDL-Programm, das die Eigenschaft *time-safe* hat, ist auch deterministisch.

Ein time-safe Giotto/TDL-Programm wird bezüglich derselben zeitlich festgelegten Folge von Eingabewerten immer dieselbe zeitlich festgelegte Folge von Ausgabewerten liefern. Wir sagen auch, dass das *logische* Verhalten des Programms immer gleich bleibt. Das geht sogar so weit, dass dasselbe Giotto/TDL-Programm selbst auf verschiedenen Plattformen sich logisch immer gleich verhalten wird, solange das Programm für diese Plattformen time-safe ist.

Dieser Determinismus von *time-safe* Giotto/TDL-Programmen hat große Vorteile bei der Entwicklung von harten Regelungssystemen und sollte den Nachteil, nämlich die Wartezeit für Tasks, die vor dem Ablauf ihrer LET fertig werden, aufwiegen. Zum Beispiel können mehrere Giotto/TDL-Programme parallel ausgeführt werden, ohne dass das logische Verhalten eines Programms das logische Verhalten der anderen Programme verändert, vorausgesetzt, die parallele Komposition der Programme ist wieder time-safe. Diese Eigenschaft ermöglicht unter anderem auch die modulare Entwicklung und die Wiederverwendbarkeit von Giotto/TDL-Programmen.

In dem nun folgenden Fallbeispiel werden diese Vorteile demonstriert und es wird versucht, den potenziellen Nachteil der LET-Abstraktion auszuräumen.

9.2 Fallbeispiel: Echtzeitsoftware eines autonom fliegenden Helikopters

Die Anwendung von TDL wird anhand eines komplexen Regelungssystems eines autonom fliegenden Modellhubschraubers[2] illustriert. Das Beispiel wurde gewählt, weil es sich sehr gut eignet, um das Charakteristische der Echtzeitsoftware-Entwicklung sowie die Vorteile des Einsatzes der dafür geeigneten Programmiersprache TDL aufzuzeigen.

Die Hardware des Regelungssystems besteht aus einem Board mit einem 200 MHz StrongARM-Prozessor (siehe zum Beispiel die Web-Referenz zu StrongARM, 2004) und entsprechenden Ein-/Ausgabe-Anschlüssen. Die Software des Regelungssystems ist in Oberon implementiert. Als Betriebssystem wird das von Niklaus Wirth eigens für den Helikopter entwickelte Echtzeitbetriebssystem HelyOS (siehe Sanvido, 1999) verwendet. Das Regelungssystem, bestehend aus Hardware, HelyOS und der Autopilot-Software, wurde OLGA genannt, als Akronym für Oberon Language Goes Airborne (siehe Sanvido, 1999).

Das betrachtete System besteht aus einem Modellhubschrauber, dem OLGA-Regelungssystem und einer Bodenstation. Abbildung 9.2 zeigt den Helikopter. Abbildung 9.3 zeigt die Systemstruktur.

Mit Hilfe einer Bodenstation wird ein Flugmuster definiert. Von der Bodenstation werden Kommandos an den Helikopter gefunkt, und der Flug wird überwacht. Ein mit der Bodenstation verbundenes GPS wird verwendet, um das GPS im Helikopter zu ergänzen, damit eine höhere Genauigkeit der Positionsdaten erreicht werden kann. Da die Bodenstation keine Relevanz für das Fallbeispiel hat, wird darauf nicht näher eingegangen.

---

[2] Die Soft- und Hardware wurden an der ETH Zürich entwickelt. Im Rahmen eines Kooperationsprojektes der University of California, Berkeley, mit der ETH Zürich haben wir den für das Zeit- und Kommunikationsverhalten der Echtzeitsoftware verantwortlichen Code durch Code ersetzt, der aus einem TDL-Programm erzeugt wurde.

Abbildung 9.2: Der Modellhubschrauber mit OLGA (Quelle: Henzinger et al., 2003)

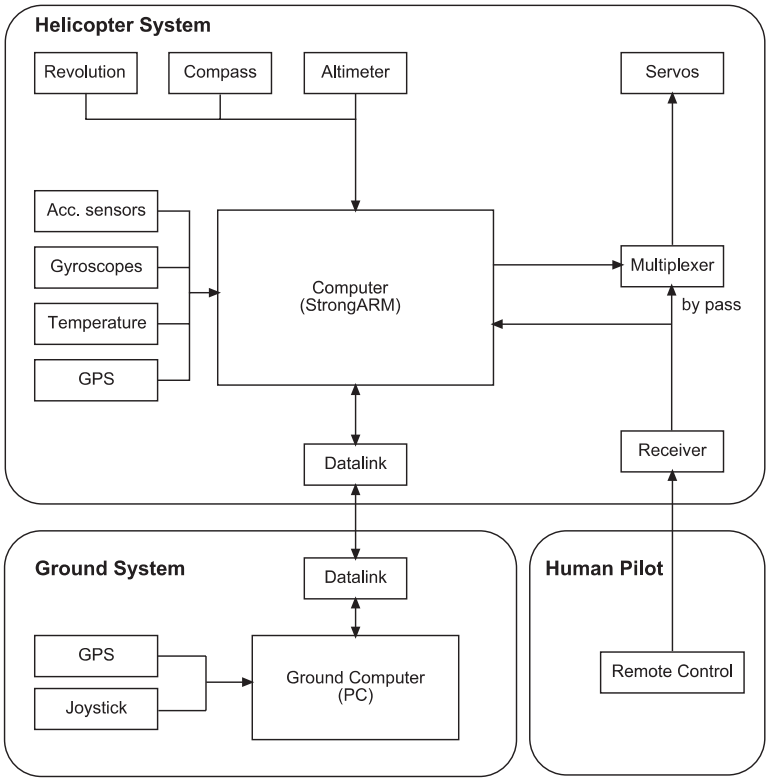

Abbildung 9.3: Die Systemstruktur des Helikopter-Systems und seine Verbindungen zur Bodenstation und zum Piloten

Alle Sensoren, Aktuatoren und Rechnerkomponenten, die zur Steuerung des Helikopters benötigt werden, sind *im* beziehungsweise *am* Helikopter. Die Sensoren im Helikopter sind ein GPS-Empfänger, ein Kompass, zwei Drehzahlmesser, die die Drehzahl der Rotorblätter messen, ein Laser-Höhenmesser, drei Beschleunigungsmesser, drei Gyroskope und ein Temperatursensor. Als Aktuatoren dienen sechs Servo-Motoren, die die Stellung der Rotorblätter (Hauptrotor und Heckrotor) und das Gas für den 2-Takt-Verbrennungsmotor steuern.

Das Regelungssystem OLGA erzeugt die pulsmodulierten Kommandos für die Servo-Motoren. Start und Landung des Helikopters werden von einem Piloten mittels Fernsteuerung durchgeführt. Der Übergang zum autonomen Flug erfolgt typischerweise kurz nach dem Abheben. Um einen sanften Übergang zu gewährleisten, verfolgt OLGA die Steuerkommandos des Piloten und übernimmt beim Umschalten die Einstellungen der Aktuatoren. Im Notfall kann sich der Pilot einschalten, indem OLGA von ihm deaktiviert wird.

Die Komplexität des Regelungssystems ergibt sich vor allem aus der großen Anzahl von Sensoren und Aktuatoren, die parallel zu handhaben sind, aus der schwierigen Aufgabe, den Flug des Helikopter so zu steuern, dass es zu keinem Absturz kommt, und den diversen Rahmenbedingungen, wie zum Beispiel Stromverbrauch, limitierte Rechner-Ressourcen und Beherrschung beziehungsweise Vermeidung von Vibrationen. Da der Helikopter ein teures Gerät ist und, wenn er außer Kontrolle gerät, Gefahr im Verzug ist, ist ein Ansatz zur Systementwicklung, der primär auf Versuch und Irrtum beruht, nicht angebracht.

An das Regelungssystem werden Echtzeitanforderungen gestellt, die unter allen Umständen einzuhalten sind. In OLGA wird die Regelung mit einer Frequenz von 40 Hz (alle 25 ms) und die Verarbeitung von Sensorwerten mit 200 Hz (alle 5 ms) periodisch ausgeführt.

9.2.1 Das OLGA-System

Das OLGA-System beruht auf der Einteilung in sechs Betriebszustände (Modi): Init, Idle, Motor, TakeOff, ControllOff, und ControlOn (siehe Abbildung 9.4). In jedem Modus sind verschiedene periodisch ausgeführte Funktionen, also TDL Tasks, aktiv. In jedem Modus, mit Ausnahme von ControlOn, steuert der Pilot die Stellung der Rotorblätter. Die Drehzahl der Rotorblätter wird von OLGA bestimmt. Die drei Modi Init, Idle, und Motor werden benötigt, um die Initialisierung richtig zu handhaben.

Im Modus Motor wird die Drehzahl des Hauptrotors von 0 auf 300 Umdrehungen pro Minute erhöht. Bei dieser Drehzahl bleibt der Helikopter noch sicher am Boden. Durch ein entsprechendes Kommando von der Bodenstation erfolgt der Übergang in den Modus TakeOff. Nach erfolgtem Abheben ist der Modus ControlOff aktiv. In diesem Modus dreht sich der Rotor mit 1200 Umdrehungen pro Minute und der Pilot steuert die Stellung der Rotorblätter. Der Pilot kann nun in den ControlOn-Modus umschalten, in dem dann der Helikopter die vorgegebenen Flugmuster autonom fliegt.

Nachfolgend betrachten wir lediglich die beiden Modi ControlOff und ControlOn, da eine genaue Beschreibung aller Modi den Rahmen des Buches sprengen würde. Das Fallbeispiel dient dazu, dem Leser beispielhaft einen Eindruck darüber zu geben, mit welchen Problemen man bei der Entwicklung von Echtzeitsystemen konfrontiert ist und wie man sie lösen kann.

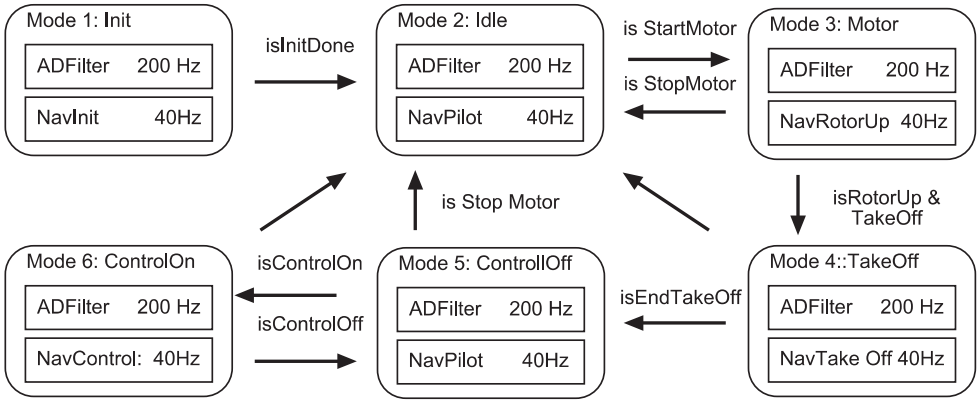

Abbildung 9.4: Die Betriebsmodi von OLGA

Im Modus ControlOff werden die Kommandos des Piloten per Funk an den Helikopter übertragen und an die Servo-Motoren weitergeleitet. Im diesem Modus sind zwei Tasks aktiv, der Task ADFilter (200 Hz) und der Task NavPilot (40 Hz). Der ADFilter-Task decodiert Sensorwerte und führt eine Vorverarbeitung durch. Der NavPilot-Task beobachtet die Position und Geschwindigkeit des Helikopters, indem die vorverarbeiteten Daten der ADFilter-Task verwendet werden. Außerdem werden die Steuerungskommandos vom Piloten an die Servo-Motoren weitergegeben. Wenn der Pilot einen bestimmten Knopf der Fernsteuerung drückt, wird vom ControlOff- in den ControlOn-Modus gewechselt. In diesem Modus ist unverändert der ADFilter-Task enthalten. Statt der NavPilot-Task ist der NavControl-Task (40 Hz) enthalten. Der NavControl-Task verfolgt nicht nur die Position und Geschwindigkeit des Helikopters, sondern stabilisiert auch den Helikopter. Vom ControlOn-Modus wird in den ControlOff-Modus gewechselt, wenn der Pilot wiederum den entsprechenden Knopf der Fernsteuerung drückt.

9.2.2 Die TDL-Spezifikation des Zeit- und Kommunikationsverhaltens

In TDL wird das Zeitverhalten der Tasks spezifiziert, also mit welcher Frequenz sie ausgeführt werden, und wie die Tasks untereinander sowie mit Sensoren und Aktuatoren kommunizieren. Die TDL basiert auf der Logical Execution Time (LET)-Annahme für Tasks, die von Giotto eingeführt wurde und besagt, dass die Ausführung einer Task logisch immer gleich lange dauert und die in einer Task berechneten Ergebniswerte am Ende der Ausführungsdauer weitergegeben werden.

Durch die TDL wurde die Syntax von Giotto vereinfacht und insbesondere das Modul-Konstrukt für Echtzeitsoftware eingeführt. Mit einem TDL-Modul kann ein Regelungssystem wie OLGA repräsentiert werden. Ein TDL-Modul enthält dann die Beschreibung der benötigten Sensoren, Aktuatoren, Tasks und Modi. Ein Modus definiert eine Reihe von Aktivitäten: Task-Aufrufe, Aktualisierungen von Aktuatoren und Wechsel zwischen Modi. Ein TDL-Modul verhält sich so, dass zu jedem Zeitpunkt genau ein Modus aktiv sein kann. Die Bedingungen zum Wechsel zwischen den Modi werden periodisch geprüft. Die Frequenz der Prüfung kann geändert werden. Nachfolgend wird ein Ausschnitt des TDL-Programms gezeigt, das das Zeit- und Kommunikationsverhalten des OLGA-Regelungssystems beschreibt.

```
module OLGA {

  type //opaque types
    GPS; Laser; Compass; RPM; Servo; Analog; DataPool;

  sensor
    GPS gps uses GPSGet;
    Laser laser uses LaserGet;
    Compass compass uses CompassGet;
    RPM rpm uses RotorGet;
    Servo pilot uses ServoGet;
    Analog accelerometers uses AccGet;
    Analog gyroscopes uses GyrosGet;
    Analog temperature uses TempGet;
    boolean startswitch uses StartSwitchGet;
    boolean stopswitch uses StopSwitchGet;

  actuator
    Servo servos uses ServoPut;
    DataPool datapool uses DataPoolPut;

  task ADFilter [wcet=3ms] {
    input Analog accs; Analog gyros; Analog temp;
    output Analog filter uses FilterInit;
    uses ADFilterImplementation(accs, gyros, temp, filter);
  }

  task NavPilot [wcet=5ms] {
    input GPS gps; Laser laser; Compass compass; Analog filter;
      RPM rpm; Servo pilot;
    output Servo control uses ServoInit;
      DataPool data uses DataInit;
    uses NavPilotImplementation(
      gps,laser, compass, filter, rpm, pilot, control, data);
  }

  task NavControl [wcet=10ms] {
    input GPS gps; Laser laser; Compass compass; Analog filter;
      RPM rpm; Servo pilot;
    output Servo control uses ServoInit;
      DataPool data uses DataInit;
    uses NavControlImplementation(
      gps,laser, compass, filter, rpm, pilot, control, data);
  }
```

```
  // other tasks ...

  start mode Init [period=25ms] {
    //...
  }

  mode ControlOff [period=25ms] {
    task
      [freq=5] ADFilter(accelerometers, gyroscopes, temperature);
                          // period of ADFilter = 25ms/5 = 5ms
      [freq=1] NavPilot(gps, laser, compass, ADFilter.filter, rpm, pilot);
                          // period of NavPilot = 25ms/1 = 25ms
    actuator
      [freq=1] servos := NavPilot.control;
                          // actuator update frequency = 1s/25ms*1 = 40Hz
      [freq=1] datapool := NavPilot.data;
                          // actuator update frequency = 1s/25ms*1 = 40Hz
    mode
      [freq=1] if isControlOn(startswitch) then ControlOn;
                          // mode switch frequency = 1s/25ms*1 = 40Hz
  }

  mode ControlOn [period=25ms] {
    task
      [freq=5] ADFilter(accelerometers, gyroscopes, temperature);
      [freq=1] NavControl(gps, laser, compass, ADFilter.filter, rpm, pilot);
    actuator
      [freq=1] servos := NavControl.control;
      [freq=1] datapool := NavControl.data;
    mode
      [freq=1] if isControlOff(stopswitch) then ControlOff;
  }

  // other modes ...
}
```

Tasks sind jene Einheiten, die Funktionalität bereitstellen. Typischerweise werden von Tasks Regelungsgesetze zur Berechnung der Ausgaben verwendet. Tasks werden je nach Betriebszustand periodisch mit einer bestimmten Frequenz aufgerufen. Sie geben Ergebnisse an Aktuatoren oder andere Tasks über Output-Ports weiter. Sie erhalten die Eingaben von Sensor-Ports oder von Output-Ports anderer Tasks. Damit beschreibt ein TDL-Programm die Echtzeitinteraktionen von Komponenten mit der realen Welt sowie die Echtzeitinteraktionen der Komponenten untereinander. Die Funktionalität einer Task kann mit jeder beliebigen Programmiersprache, wie zum Beispiel Oberon im Fall von OLGA, implementiert werden.

Die Task *ADFilter* definiert drei Eingabeports: *accs*, *gyros* und *temp*. Der Ausgabeport *filter* ist eine Sammlung von gefilterten Eingabewerten und wird mit dem Ergebnis der externen Funktion *FilterInit* initialisiert. Zur Durchführung der Datenfilterung wird die externe Prozedur *ADFilterImplementation* mit den entsprechenden Ports als Parameter aufgerufen. Die Abarbeitung dieser Prozedur darf nicht länger als drei Millisekunden dauern. In TDL sind Ausgabeports als Referenzparameter realisiert. Sie können also zur Ein- und Ausgabe von Werten verwendet werden.

In analoger Form sind die Tasks *NavPilot* und *NavControl* spezifiziert. Sie ermitteln jeweils für die manuelle und automatische Steuerung die von den Servo-Motoren benötigten Stellwerte. Zusätzlich werden Flugdaten in einem Daten-Pool für spätere Auswertungen gesammelt.

Das Zusammenwirken der Tasks wird für jeden Betriebszustand (Modus) in einer Mode-Deklarationen festgelegt. Der Startzustand ist der Modus *Init*, der für diverse Initialisierungsschritte eingeführt wurde. Von diesem Modus aus gelangt man durch Mode-Switches (nicht im Quelltext gezeigt) zu den Nachfolgezuständen gemäß Abbildung 9.4. Zwei Modi sind detailliert angegeben: *ControlOff* repräsentiert den Zustand der manuellen Steuerung und *ControlOn* den autonomen Flugbetrieb. Beide Modi umfassen periodisch ausgeführte Aktivitäten, die sich alle 25 Millisekunden wiederholen.

Der Modus *ControlOff* enthält zwei Tasks, die mit einer Frequenz von 5 bzw 1 ausgeführt werden. Das bedeutete, dass *ADFilter* innerhalb einer Periode von 25 ms 5 mal, also alle 5 ms, ausgeführt wird. *NavPilot* hingegen wir nur einmal alle 25 ms ausgeführt. Die Task *NavPilot* benutzt den Ausgabeport *ADFilter.filter* als Eingabe, verwendet aber auf Grund der unterschiedlichen Frequenzen nur jeden fünften Wert. Die Zwischenergebnisse von *ADFilter.filter* dienen der Verbesserung der Sensorwerte durch Bildung von Durchschnitt und ähnlichen Mitteln der Signalaufbereitung.

Alle 25 ms werden außerdem die Aktuatoren *servos* und *datapool* aktualisiert, was jeweils zu einem Aufruf der externen Prozeduren *ServoPut* bzw. *DataPoolPut* führt.

Einmal alle 25 ms wird ferner geprüft, ob ein Wechsel der Betriebsart in den Modus *ControlOn* erfolgen soll.

Der Modus *ControlOn* ist analog zum Modus *ControlOff* definiert; es werden keine neuen Konzepte eingeführt. Deshalb wird auf eine weitere Erläuterung verzichtet.

9.2.3 Die Übersetzung der TDL-Spezifikation in ein ausführbares Programm

Eine TDL-Spezifikation wird durch einen TDL-Compiler in eine binäre Form transformiert, den so genannten E-Code (Embedded Machine Code). Die Sprache TDL ist durch eine attributierte Grammatik (siehe Kapitel 7) beschrieben und der Scanner und Parser sind mittels Coco/R (Mössenböck, 2004) implementiert. Die Abarbeitung des E-Codes erfolgt durch eine virtuelle Maschine, die Embedded-Machine (E-Machine), durch Interpretation des E-Codes und damit der Ausführung der benötigten Aktivitäten zum jeweils richtigen Zeitpunkt. Abbildung 9.5 zeigt die logische Ausführung des Moduls OLGA im Zustand ControlOn.

Der dargestellte Programmablauf entspricht der logischen Ausführungszeit (LET) der beteiligten Tasks. Zum Zeitpunkt 0 werden die Ausgabeports initialisiert und die Tasks ADFilter und NavControl mit Eingabewerten versehen und logisch gestartet. Nach 5 ms ist ADFilter fertig und liefert einen neuen Wert des Ausgabeports filter. NavControl ist nach 25 ms fertig und liefert neue Ausgabewerte control und data, die den jeweiligen

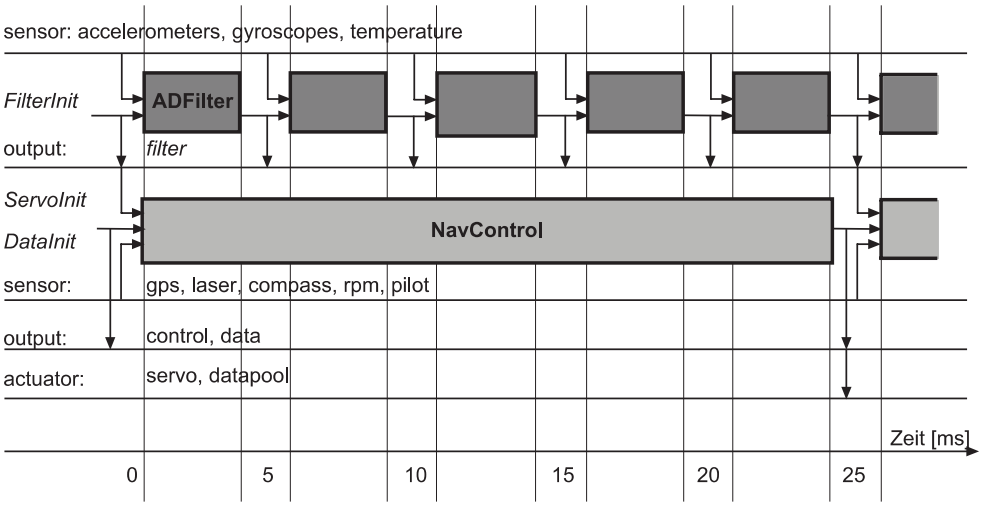

Abbildung 9.5: Die Ausführungslogik des TDL-Programms im Modus ControlOn

Aktuatoren servo und datapool übermitttelt werden. Man beachte, dass zwischen den einzelnen Task-Aktivierungen keine Zeit vergeht. Für die Aktivierungen von Tasks einschließlich der Parameterübergabe, Aktuator-Aktualisierungen und Tests auf Mode-Switches wird im Giotto/TDL-Programmiermodell logisch keine Zeit benötigt.

Zu welchem Zeitpunkt nun die CPU tatsächlich an welchen Tasks arbeitet, ist im TDL-Modell nicht spezifiziert, sondern eine Eigenschaft der Implementierung. Mögliche Strategien zur Festlegung der physischen Abarbeitung der Tasks sind zum Beispiel „earliest deadline first“ oder „rate monotonic“ Scheduling, auf die hier aber nicht näher eingegangen werden kann (siehe zum Beispiel in Buttazzo, 1997). In obigem Beispiel führen beide zur gleichen Abarbeitungsstrategie, die darin besteht, dass jeweils abwechselnd 3 ms ADFilter und 2 ms NavControl aktiv sind. Damit kann unter Annahme der Einhaltung der WCET (*Worst Case Execution Time*) der beiden Tasks garantiert werden, dass die Anforderungen an das Zeitverhalten erfüllt werden.

9.2.4 Simulation eines TDL-basierten Regelungssystems

Zusätzlich zur TDL-Spezifikation muss im Zuge der Systementwicklung auch die Funktionalität der Tasks implementiert werden. In dem hier gewählten Fall des Helikopters liegen die Regelungsalgorithmen bereits in Oberon vor.

Wenn zum Zwecke der prototypischen Erprobung das Regelungssystem simuliert werden soll, bevor es zusammen mit der tatsächlichen Hardware erprobt wird, kann man dazu spezielle Werkzeuge heranziehen. Ein Beispiel für ein solches Wekzeug ist Simulink von der Firma Mathworks (siehe dazu die Web-Referenz zu Mathworks). Abbildung 9.6 zeigt das Zusammenspiel der Simulink-Werkzeugkomponenten mit den TDL-Komponenten im Zuge des Modellierungsprozesses für ein Regelungssystem.

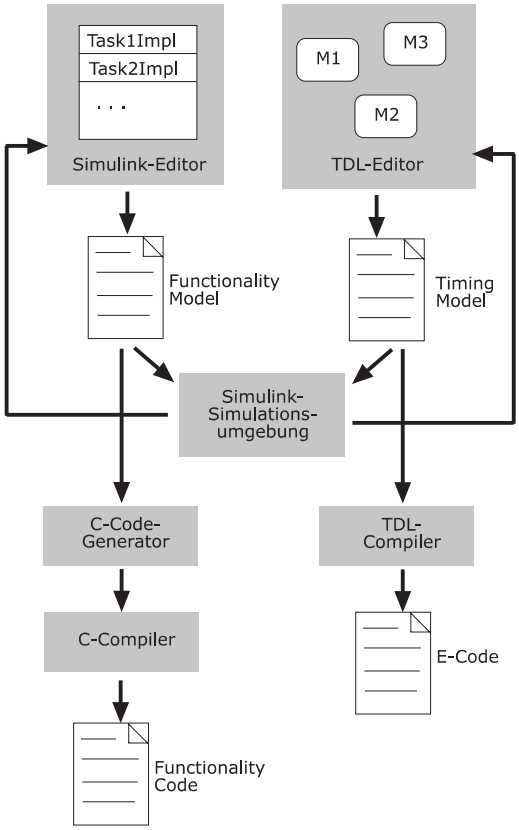

Abbildung 9.6: Modellierung eines Regelungssystems mit Simulink und TDL-Werkzeugen

Mit einem TDL-Editor wird das Zeit- und Kommunikationsverhalten spezifiziert (Timing Model). Statt einer textbasierten Spezifikation, wie sie im Beispiel verwendet wurde, kann auch ein visuell-interaktiver Editor zur Spezifikation verwendet werden. Mit Hilfe von Simulink werden die Tasks modelliert, das heißt, es wird ein so genanntes Functionality Model erstellt. Um die Simulationsumgebung von Simulink verwenden zu können, muss das Timing Model in ein entsprechendes Simulink-Modell transformiert werden. Auch das macht man in der Praxis unter Zuhilfenahme von Werkzeugen. Ein Beispiel dafür ist das so genannte T/S-Translator-Werkzeug (siehe dazu Pree und Stieglbauer, 2004).

Die Aktivitäten der Spezifikation und Simulation werden so lange durchgeführt, bis die Simulationsergebnisse zufriedenstellend sind. Dann wird, wiederum mit entsprechenden Werkzeugen (ein Beispiel dafür ist der Real-Time-Workshop Embedded Coder der Firma Mathworks, siehe dazu die Web-Referenz MathWorks, 2004) aus dem Functionality Model zum Beispiel C-Code zu dessen Realisierung generiert. Der C-Compiler und der TDL-Compiler erzeugen dann die Teile, die zusammen den ausführbaren Code bilden.

Der E-Code (siehe dazu Henzinger et al., 2001) wird durch die E-Machine, eine virtuelle Maschine (siehe Kapitel 6) mit einer Interpreter-Architektur, abgearbeitet. Die E-Machine ruft zu den entsprechenden Zeitpunkten die Funktionen auf, die die TDL-Tasks implementieren.

9.3 Zusammenfassung

TDL ist eine spezielle Sprache, die bei der Entwicklung von Echtzeitsoftware, wie zum Beispiel Regelungssystemen, eingesetzt werden kann. TDL ist also eine domänenspezifische Sprache (Domain-Specific-Language), die wie andere domänenspezifische Sprachen als Werkzeug zur Unterstützung der Entwicklung spezieller Software eingesetzt werden kann.

Ein wesentlicher Aspekt ist, dass in TDL von der konkreten Hardware- und Betriebssystem-Plattform abstrahiert wird, indem eine logische Ausführungszeit (Logical Execution Time; LET) definiert wird. Der von einem der Autoren erfolgreich durchgeführte Reengineering-Prozess für die OLGA-Autopilot-Echtzeitsoftware zeigte die Nützlichkeit dieser Abstraktion in der Praxis.

Das Fallbeispiel veranschaulicht, wie man durch geeignete Abstraktionen für einen Anwendungsbereich, hier für die parallele, verteilte Programmierung unter harten Echtzeitanforderungen, die Komplexität meistern kann. Die Parallelität und die Echtzeitanforderungen an die erforderlichen Programme werden in TDL beschrieben. Ein Compiler prüft, ob die Zeitvorgaben für eine bestimmte Plattform einhaltbar sind, und erzeugt, falls das gegeben ist, den ausführbaren Code für eine bestimmte (gegebenenfalls verteilte) Plattform. Da der Code, der für das Zeitverhalten verantwortlich ist, generiert, das heißt automatisch erzeugt wird, können die Zeitanforderungen, wie zum Beispiel die Perioden der einzelnen Tasks, ohne großen Aufwand und ohne Fehlerrisiko modifiziert werden. Das deterministische Zeitverhalten von TDL-Programmen ist die Voraussetzung für deren deterministische Komposition.

Die im Fallbeispiel skizzierte Vorgehensweise zusammen mit den eingesetzten Werkzeugen führt im Ergebnis zu einer besseren Beherrschbarkeit der Komplexität, zur Rationalisierung des Entwicklungsprozesses und zur Verbesserung der Softwarequalität bei der Entwicklung eingebetteter Echzeitsoftware.