5 Softwareentwurf eingebetteter Systeme

Wie wir bereits in Kapitel 4 gesehen haben, besteht die Entwicklung eingebetteter Systeme nicht nur aus deren Programmierung, sondern erfordert darüber hinaus eine systematische Vorgehensweise, die sich in aufeinander folgende Entwicklungsphasen unterteilen lässt. In der Phase des Softwareentwurfs (also vor der Programmierung) muss sich der Entwickler Spezifikationstechniken bedienen, die eine abstraktere, aber trotzdem möglichst eindeutige und vollständige formale oder semiformale Beschreibung des Systems als Ergebnis haben. Ausgewählte Vertreter dieser visuellen Beschreibungsmöglichkeiten werden wir in diesem Kapitel betrachten. Dabei sei auch erwähnt, dass die Grenzen zwischen dem Entwurf und der Programmierung eingebetteter Systeme zunehmend verschwimmen. So hätte man prinzipiell auch die Beschreibung von Esterel und Giotto in dieses Kapitel mit aufnehmen können. Insgesamt besteht hier langfristig die Bestrebung, eingebettete Systeme überhaupt nicht mehr programmieren zu müssen, sondern ganz auf abstraktere Beschreibungsmöglichkeiten zurückgreifen zu können.

Eingebettete Systeme werden in der Regel nicht von HW- oder SW-Spezialisten sondern von Experten des jeweiligen Fachgebiets entwickelt (z. B. Regelungstechnik). Die verwendeten Formalismen zur Beschreibung des Verhaltens der Funktion sind Domänenspezifisch optimiert (z. B. Zustandsautomaten, Differenzen- oder Differentialgleichungen). Darüber hinaus enthalten diese Systeme immer auch zusätzliche Funktionalitäten, wie Kommunikation oder Diagnose, die mit gänzlich anderen Formalismen adäquat beschrieben werden. Diese heterogenen Beschreibungen sind Ausgangspunkt des Hardware/Software-Codesigns. Nach der Lektüre dieses Kapitels sollte der Leser die wichtigsten Modellierungstechniken zur Entwicklung eingebetteter Systeme kennen und voneinander abgrenzen können.

5.1 Modellierung eingebetteter Systeme

Klassifikation

Die zur Analyse und Modellierung eingebetteter Systeme verwendeten Sprachen können auf mehrere Arten klassifiziert werden:

- Modellierung diskreter und/oder kontinuierlicher Information,
- Zustands-, Aktivitäts- und Struktur-orientierte Verfahren und deren Mischformen,
- mit oder ohne Modellierung zeitabhängiger Informationen.

Viele eingebettete Systeme v. a. in der Automobiltechnik zeichnen sich durch einen *hybriden* Charakter aus: Sie steuern bzw. regeln Vorgänge, in denen diskrete und kontinuierliche Anteile in Wechselwirkung stehen.

Folgende Sprachen werden zurzeit auf breiter, größtenteils industrieller Basis zur Modellierung und Simulation von Software für eingebettete Systeme verwendet. Nachstehende Liste erhebt keinen Anspruch auf Vollständigkeit; vgl. (Broy et al., 1998):

Beschreibungs-techniken

- Statecharts
- Hybrid Statecharts
- ROOM
- VHDL-A
- MSC bzw. EET
- Structured Analysis / SA-RT
- UML
- SDL
- OMT
- Entity-Relationship-Diagramme
- Timed Automata
- Hybride Automaten
- Temporale Logiken (z. B. CTL, LTL)
- mechatronisches Datenmodell
- u.v.m.

5.2 Formale Methoden

Gerade weil eingebettete Systeme zunehmend in sicherheitskritischen Bereichen eingesetzt werden und dadurch die Qualitätssicherung (vgl. Kapitel 6) von besonderer Bedeutung ist, leistet die theoretische Informatik durch Bereitstellung formaler Grundlagen zur *Spezifikation, Verifikation und Codeerzeugung* einen entscheidenden Beitrag zur effektiven Entwicklung eingebetteter Systeme, die damit ein Paradebeispiel für ein interdisziplinäres Forschungs-, Entwicklungs- und Arbeitsfeld der Informatik darstellen (Broy et al., 1998).

Gerade in dem Bereich der *Verifikation* existieren schon viele formale Verfahren und Werkzeuge. Diese werden bisher jedoch kaum im industriellen Softwareentwicklungsprozess für Steuergeräte eingesetzt. Hier ist auch der Aspekt der Skalierbarkeit entscheidend: Beispielsweise entsteht beim Model Checking oft das Problem, dass die zu verifizierenden Systeme, insbesondere beim Auftreten von nebenläufigen Vorgängen, zu komplex für die Bearbeitung sind. In diesem Fall führen Ansätze zur kompositionellen Verifikation und/oder Abstraktionsverfahren zu einer nennenswerten Verbesserung.

In vielen Fällen mangelt es jedoch an der *werkzeugorientierten Umsetzung* in softwaretechnischen Methoden und Vorgehensmodellen für die Praxis: Während formale Methoden für etablierte visuelle Formalismen wie etwa Statecharts mittlerweile in Form von Add-Ons zur Verifikation oder automatischen Test-Generierung für Design-Tools wie Statemate (Firma iLogix) Einzug in die industrielle Praxis gehalten haben, sind formale Methoden für den UML Bereich (UML = Unified Modeling Language) weiterhin ein Gebiet aktiver Forschung.

Um den *Herstellungsprozess für Steuergeräte* bzw. deren Steuerungssoftware möglichst flexibel, kostengünstig und schnell zu gestalten, braucht es Möglichkeiten zur Codegenerierung, d. h. zur automatischen Erzeugung von Code aus einer (formalen) Spezifikation. Hier existieren zwar bereits Lösungen in Form von Methoden und teilweise sogar unterstützenden Werkzeugen, jedoch ergibt sich hier immer noch das Problem, dass der resultierende, automatisch generierte Code zu umfangreich ist (ca. Faktor 2) und/oder eine zu langsame Ausführungszeit besitzt.

5.3 Statecharts

Historie, Prinzip

Statecharts, eine graphische Spezifikationssprache für komplexe, zustandsbasierte Systeme, wurden bereits 1987 von *David Harel* als eine Erweiterung sequentieller Automaten (diese bestehen aus Zuständen und Transitionen) eingeführt. Statecharts kombinieren diese Automaten mit Konzepten für Nebenläufigkeit, Kommunikation und hierarchischer Dekomposition. Sie sind heute Basis vieler Werkzeuge im industriellen Einsatz zur Modellierung eingebetteter Systeme. Statecharts beschreiben das zustandsbasierte Ablaufmodell einer Softwarekomponente.

Von Statecharts existieren heute zahlreiche Varianten mit teilweise sehr unterschiedlicher Syntax und vor allem Semantik. In der *UML* (Unified Modeling Language) werden Statecharts häufig auch eingesetzt, um den Lebenszyklus eines Objekts, also die Abhängigkeiten zwischen Aufrufen von Objektmethoden und dem Zustand des Objekts zu beschreiben. Ein Statechart kann darüber hinaus auch dazu verwendet werden, um das Ablaufmodell einer komplexen Methode darzustellen.

Im Gegensatz zu Sequenzdiagrammen zeigt ein Statechart keine Interaktion zwischen Objekten, sondern modelliert gleichzeitig den Zustand und das Verhalten eines Objekts. Dabei können Statecharts das Verhalten vollständig beschreiben, während Sequenzdiagramme meist exemplarisch sind. Leider wird dadurch die Statechart-Notation komplexer und benötigt einigen Lernaufwand.

Transitionen

Die *Transitionen* (Zustandsübergänge, Pfeile, Kanten) sind im Allgemeinen mit einem Ereignis e bzw. dem Namen einer Methode beschriftet, welche(s) die Transition schaltet. Zusätzlich kann ggf. eine Bedingung b angegeben werden, unter der die Transition erfolgt. Darüber hinaus kann jede Transition mit einer Aktion a attributiert sein, die beim Schalten der Transition ausgeführt wird. Die Schreibweise für die Beschriftung der Transitionen in den meisten Statechartsvarianten lautet damit wie folgt:

„e [b] / a“.

Nebenläufigkeit, Kommunikation, Komposition, Dekomposition

Um die Spezifikation praktisch relevanter Systeme zu ermöglichen, können die Automaten in Statecharts parallel komponiert oder hierarchisch dekomponiert werden. Statecharts erweitern das Modell der sequentiellen Automaten um die Konzepte der *Nebenläufigkeit* durch Komposition, der *Kommunikation* (zwischen Automaten) sowie der *hierarchischen Dekomposition.* Durch das Konzept der

Nebenläufigkeit bzw. Parallelkomposition lässt sich das Verhalten verteilter eingebetteter Systeme beschreiben.

Mit Hilfe der hierarchischen Dekomposition können Gruppen von Zuständen zusammengefasst werden. Bei der *Parallelkomposition* ist die wesentliche Grundannahme, dass zwei auf diese Weise zusammengesetzte sequentielle Automaten im Gleichtakt in Bezug auf einen gemeinsamen Taktgeber ihre Zustandsübergänge vornehmen. Ohne weiteres Zutun interagieren die auf solche Weise komponierten Automaten zunächst überhaupt nicht. Beim Entwurf eingebetteter Software mit Statecharts kann jedoch zusätzlich angegeben werden, dass diese Mealy Maschinen das gegenseitige Verhalten durch den Austausch von Nachrichten wechselseitig beeinflussen können.

Informelle Semantik

Informell lautet die Semantik dann wie folgt: Liegt ein passendes Eingabeereignis im aktuellen Systemschritt bzw. -takt (vgl. e_1 in Abbildung 5.1) an und ist die angegebene Zusatzbedingung (vgl. b_1 in Abbildung 5.1) logisch wahr, so wird der Zustandsübergang vollzogen und gleichzeitig die spezifizierte Aktion (vgl. a_1 in Abbildung 5.1) ausgeführt. Enthält a_1 ein Ereignis, welches das zweite parallel komponierte Statechart im aktuellen Zustand beeinflussen kann, so vollzieht auch dieses alle hierdurch ermöglichten Zustandsübergänge (in Abbildung 5.1 vollzieht sich daher auch der Zustandswechsel von z_2 zu z_3 – vorausgesetzt dass zusätzlich auch die Bedingung b_2 wahr ist). Die (formale) Semantik zahlreicher Statechartsvarianten unterscheidet sich im Übrigen meist genau darin, welches zeitliche Verhalten im Detail diesem kombinierten Verhalten zweier auf diese Art komponierten Statecharts zugrunde liegt.

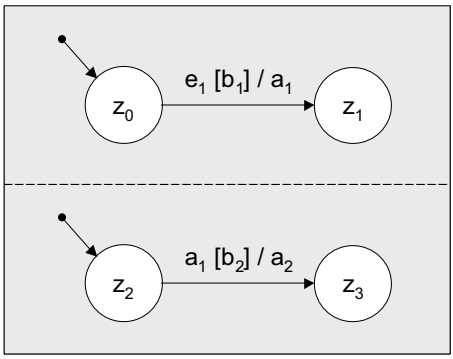


Abb. 5.1: Parallelkomposition bei Statecharts

Parallel-komposition

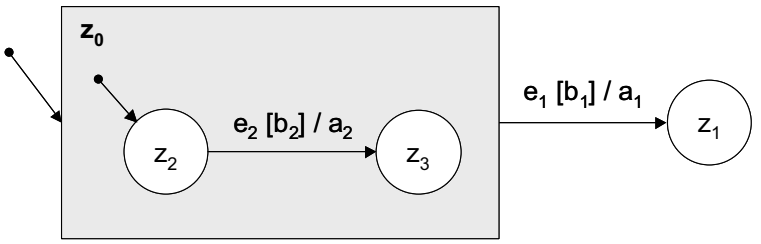
Die Parallelkomposition ermöglicht eine wesentlich kompaktere Darstellung komplexer Zustandsräume als es mit einem einzigen sequentiellen Automaten möglich wäre. Der Zustandsraum einer Parallelkomposition ist das algebraische Produkt der Zustandsräume beider im Rahmen der Komposition verbundenen Automaten, wodurch die Größe der Spezifikation anstelle eines quadratischen Wachstums bezogen auf die Anzahl der Zustände lediglich linear anwächst.

Hierarchische Dekomposition

Reaktive, eingebettete Systeme nehmen meist komplexe Systemzustände ein, so dass einzelne Zustände eines Automaten nicht für deren realistische Abstraktion geeignet sind und somit ausgefeiltere Ansätze gebraucht werden. Neben der Parallelkomposition erlauben daher Statecharts eine weitere Spezifikationstechnik, welche diesem Umstand Rechnung trägt und einzelne Zustände eines Automaten durch *hierarchische Dekomposition* in Subzustände aufteilen lässt: Das Statechart, welches das Systemverhalten in einem spezifischen Zustand in größerem Detail beschreibt, wird auf graphischer Ebene einfach als Subkomponente innerhalb dieses Zustands, der damit zu einem sogenannten „Superzustand“ wird, gezeichnet.

Das Verhalten eines hierarchisch dekomponierten Statecharts ist vergleichbar mit dem eines Prozedur- bzw. Funktionsaufrufs bei einer imperativen Programmiersprache wie etwa Pascal oder C. Ein Automat auf einer hierarchisch höheren Ebene ruft im Sinne dieses Vergleichs „Subroutinen“ auf, nämlich das innerhalb des Zustands modellierte Statechart und zwar immer in genau den Systemtakten, wenn der Superzustand selbst aktiv ist, d. h. sich der Ablauf der Systemreaktion in genau diesem Zustand befindet.

Abb. 5.2: Hierarchische Dekomposition bei Statecharts



UML

In der *UML* (Unified Modeling Language) werden Statecharts häufig auch eingesetzt, um den Lebenszyklus eines Objekts, also die Abhängigkeiten zwischen Aufrufen von Objektmethoden und dem Zustand des Objekts zu beschreiben. Ein Statechart kann darüber

hinaus auch dazu verwendet werden, um das Ablaufmodell einer komplexen Methode darzustellen.

Im Gegensatz zu Sequenzdiagrammen zeigt ein Statechart keine Interaktion zwischen Objekten, sondern modelliert gleichzeitig den Zustand und das Verhalten eines Objekts. Dabei können Statecharts das Verhalten vollständig beschreiben, während Sequenzdiagramme meist exemplarisch sind. Leider wird dadurch die Statechart-Notation komplexer und benötigt einigen Lernaufwand.

5.4 Die Unified Modeling Language (UML)

Aufgrund der stetig wachsenden Komplexität von Softwaresystemen verfolgt man in den letzten Jahren zunehmend das Ziel, den Entwurf und die Programmierung solcher Systeme mit Hilfe objektorientierter (OO) Ansätze zu vereinfachen. Eine Möglichkeit hierfür ist die Unified Modeling Language (UML), eine Sprache mit verschiedenen, meist graphischen Beschreibungsmitteln zur Spezifikation, Konstruktion, Visualisierung und Dokumentation von Modellen für Softwaresysteme.

Die UML stellt aktuell einen Standard dar und ist wegen ihrer Struktur- und Verhaltensbeschreibungsmittel und zahlreicher Notationselemente sehr umfangreich.

Historie

Die Entwicklung objektorientierter Ansätze geht in die achtziger Jahre zurück. Die wichtigsten damals verbreiteten dieser OO-Ansätze waren:

- Booch Methode (Grady Booch),
- OOSE (Ivar Jacobsen) und
- OMT (Jim Rumbaugh).

Zwischen diesen Ansätzen gab es jeweils Unterschiede hinsichtlich der verwendeten Notation, so dass ein einheitlicher Standard nicht möglich war, zumal dieser von den jeweiligen Verfechtern der Ansätze auch abgelehnt wurde. Im Oktober 1994 verließ Jim Rumbaugh General Electric und wechselte zur Firma Rational (mittlerweile IBM), wo bereits Grady Booch tätig war. Zusammen gaben sie im Oktober 1995 ihre Veröffentlichung über die Unified Method Version 0.8 heraus und erklärten den „Methodenkrieg“ damit für beendet: „The methods war is over – we won“. Im Herbst 1995 kaufte Rational auch die Firma Objectory, für die der dritte im Bunde, Ivar Jacobsen tätig war. Booch, Jacobsen und Rumbaugh

werden seitdem auch die „drei Amigos“ genannt. Die Arbeiten an der UML begannen 1996, und eine Version 0.9 wurde im Juni 1996 publiziert. Die Unified Modeling Language Version 1.0 wurde im Januar 1997 veröffentlicht und bei der Object Management Group (OMG) als Standardisierungsvorschlag eingereicht. Als die drei Amigos im September 1997 die Version 1.1 einreichten, wurden alle Gegenvorschläge zurückgezogen, da deren Konzepte bereits von dieser UML-Version berücksichtigt wurden. Im November 1997 wurde schließlich die UML 1.1 von der OMG als Standard verabschiedet. (Balzert, 2001). Aktuell ist die UML in der Version 2.0 verfügbar.

Sprache, nicht Methode

Die UML ist eine Modellierungs*sprache*, keine Methode und enthält darum keinerlei Beschreibung für Prozesse. Ihre hauptsächliche Anwendung besteht in der Modellierung von Softwaresystemen und, wenn möglich, anschließender Codegenerierung. Das zu entwerfende System wird in der UML mittels der Diagramme definiert und festgelegt; anschließend wird ggf. ein Codegerüst generiert, das die Basis für die darauf folgende Programmierung darstellt (Oestereich, 1997).

UML 2.0

Die UML ist so konzipiert, dass sie unabhängig von der Anwendungsdomäne funktioniert. Dennoch enthält die UML 2.0 neue Aspekte, welche die Entwicklung von Echtzeitsystemen unterstützen:

- Modellierung interner Strukturen,
- Timing Diagramme (neu),
- bessere Verhaltensmodellierung bei Aktivitätsdiagrammen, Sequenzdiagrammen und Zustandsdiagrammen durch Strukturierungsmöglichkeiten komplexer Diagramme sowie Festlegung formaler(er) Semantik und
- ausführbare Modelle.

Änderungen von UML 1.5 auf UML 2.0

Insgesamt haben sich bei der UML 2.0 im Vergleich zu ihrer Vorgängerversion 1.5 folgende Änderungen ergeben:

- Marginale Änderungen
  - Klassen-Diagramm
  - Use Case-Diagramm
  - Objektdiagramm
- Kleine Änderungen:
  - Verteilungsdiagramm

- Paketdiagramm
- Tiefgreifende Änderungen:
  - Aktivitätsdiagramm
  - Sequenzdiagramm
  - Zustandsautomat
- Vollständig neu sind:
  - Kompositionsstrukturdiagramm
  - Interaktionsübersichtsdiagramm
  - Zeitdiagramm
  - Kommunikationsdiagramm

In der Version 2.0 ist die UML auch in den Interessensbereich von Entwicklern eingebetteter Systeme gerückt – und dies nicht nur aufgrund der immer komplexer werdenden Systeme. Die gegenüber den Vorläuferversionen formalere Spezifikation der UML 2.0 stellt die Ausführbarkeit der mit der UML beschriebenen Modelle sicher. Eine Überprüfung von Systemeigenschaften ohne Verlassen der Modellierungsebene ist somit möglich. Ferner ermöglicht die UML 2.0 exaktere Systembeschreibungen, die eine möglichst präzise Nachbildung der (Kunden-)Anforderungen ermöglichen.

Insgesamt ist vor dem Hintergrund des zurzeit bei eingebetteten Systemen immer weiter verbreiteten Einsatzes objektorientierter Sprachen wie C++ oder Java davon auszugehen, dass die UML nun endgültig auch bei der Entwicklung von Embedded Systems Fuß fassen wird. Immer mehr Modellierungswerkzeuge aus diesem Bereich setzen die UML 2.0 ein. Eine Modellierungstechnik wie etwa die Zeitdiagramme sind in der Lage, eine Verbindung zwischen der Entwicklung zeitkritischer Software einerseits und Model-Driven-Development andererseits herzustellen.

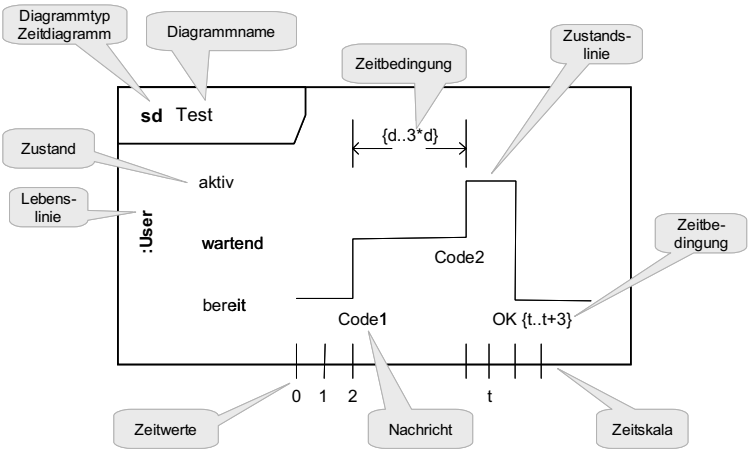
Die Neuauflage der UML mit ihrer Version 2.0 hat einige interessante Aspekte für Entwickler von eingebetteten Systemen bzw. Echtzeitsystemen zu bieten, die im Folgenden exemplarisch anhand der Zeitdiagramme (auch: Timing-Diagramme von engl. timing diagrams) dargestellt werden soll. Eine detaillierte Einführung in die UML kann im Rahmen dieses Buches nicht geschehen. Der interessierte Leser sei beispielsweise auf (Balzert, 2001), (Grässle et al., 2003) und (Oestereich, 2004) verwiesen. Bewährte Konzepte aus „Real-Time Object-Oriented Modeling" (ROOM, vgl. Abschnitt 5.5), einem der bekanntesten Ansätze zur Modellierung von Echtzeitsystemen wurden in die UML 2.0 aufgenommen.

Timing-Diagramme:

Timing-Diagramme

Timing-Diagramme ermöglichen die Modellierung von zeitgenauen Zustandsübergängen. Letztgenannte Beschreibungstechnik wird schon lange erfolgreich von Ingenieuren aus dem Bereich der Elektronik bzw. Elektrotechnik verwendet und wurde nun in die UML 2.0 als eigene Diagrammform adaptiert aufgenommen (siehe auch Abbildung 5.3).

Abb. 5.3: Beispiel Timing Diagramm



Sequenz-Diagramme eignen sich besonders gut, um die Reihenfolge von Methodenaufrufen darzustellen. Sie sind jedoch nicht geeignet, um den exakten zeitlichen Ablauf von Aktionen darzustellen. Um den Zeitaspekt besser darzustellen, wurden die aus *Real-Time UML: Developing Efficient Objects for Embedded Systems*, Addison-Wesley, 1998 bekannten Timing-Diagramme in die UML 2.0 mit aufgenommen.

Ein Timing-Diagramm beschreibt die zeitlichen Bedingungen (Zeitanforderungen, Zeitgrenzen, zeitliche Abhängigkeiten) von Zustandswechseln mehrerer Objekte. Zeitdiagramme finden ihre Verwendung vor allem in der graphischen und dennoch formalen Beschreibung von Echtzeitsystemen.

Ein Zeitdiagramm besteht aus zwei Achsen. Auf der Abszisse verläuft von links nach rechts die Zeit linear. Auf der Ordinate werden die zu betrachtenden Objekte und ihre (diskreten) Objektzustände dargestellt. Mit Hilfe von Linien kann der Zusammenhang verschiedener Zustandswechsel im zeitlichen Verlauf spezifiziert werden. Sowohl signifikante Zeitpunkte also auch Zeiträume können auf der Abszisse benannt werden.

Rational Unified Process (RUP)

Die UML stellt eine Ansammlung von Beschreibungstechniken dar. Sie selbst stellt kein Vorgehensmodell zur Softwareentwicklung zur Verfügung. Dies kann jedoch ergänzend der eigens für die UML entwickelte „Rational Unified Process“ (kurz: RUP) leisten. Das RUP-Modell wurde 1998 durch die Firma Rational entwickelt. Es bietet einen Anwendungsfall-zentrierten Zugang zur Softwareentwicklung. Es bildet das Rahmenwerk für den gesamten Entwicklungsprozess. Aktivitäten, Messkriterien und Ergebnisse werden definiert und gesteuert. Die Sprache des RUP ist die UML; Artefakte sind die Grundlage des Systems. Das Projekt wird durch das RUP-Modell dokumentiert.

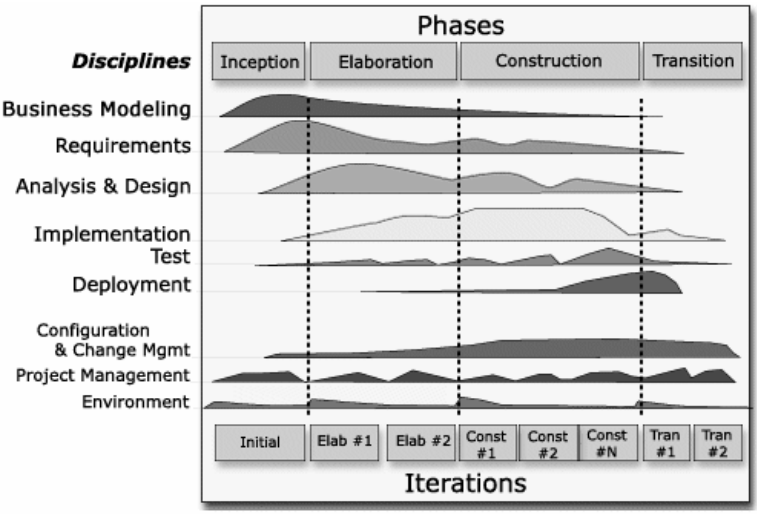


Abb. 5.4: RUP (Quelle: IBM, vormals Rational)

Bei dem Modell wurde darauf geachtet, sich an „Best Practices“ zu orientierten. Es vereint eine iterative Softwareentwicklung und das Anforderungsmanagement, es verwendet Komponenten-basierte Architekturen. Die Software wird visuell modelliert. Die Softwarequalität wird geprüft und das Änderungsmanagement unterliegt einer Kontrolle. Der RUP definiert Workflows für folgende neun Kernaufgaben (engl. disciplines), siehe Abbildung 5.4:

9 RUP Kernaufgaben

- Business Modeling (Geschäftsprozessmodellierung)
- Requirements (Anforderungsanalyse)
- Analysis and Design (Analyse und Design)
- Implementation (Implementierung)
- Test

- Deployment (Softwareverteilung)
- Configuration and Change Management (Konfigurations- und Änderungsmanagement)
- Project Management (Projektmanagement)
- Environment (Umgebung)

4 RUP Phasen

Die vier dazugehörigen Phasen definieren sich wie folgt (siehe auch Abbildung 5.4):

- **Phase 1 (Inception, Konzept)** spezifiziert die Endproduktversion und die wesentlichen Geschäftsvorfälle. Der Umfang des Projekts sowie Kosten und Risiken werden prognostiziert. Ergebnis von Phase 1 ist der „Life Cycle Objective Milestone".
- **Phase 2 (Elaboration, Entwurf)** spezifiziert die Produkteigenschaften, die Architektur wird entworfen, notwendige Aktivitäten und Ressourcen werden geplant. Diese Phase endet mit dem „Life Cycle Architecture Milestone".
- In **Phase 3 (Contruction, Implementierung)** wird das Produkt erstellt und die Architektur entwickelt. Am Ende dieser Phase ist das Produkt fertig und der „Initial Operational Capability Milestone" abgeschlossen.
- **Phase 4 (Transition, Produktübergabe)** beinhaltet die Übergabe des Produkts an die Benutzer, sowie die Überprüfung des Qualitätslevels. Der „Release Milestone" ist damit gesetzt.

Innerhalb jeder Phase erfolgt die Aufteilung der Aufgaben in kleine Schritte (Iterationen), deren Workflow ähnlich wie beim Wasserfallmodell abgearbeitet werden. Es stehen jederzeit Planungshilfen, Leitlinien, Checklisten und Best Practices zur Verfügung.

Mario Jeckle† schrieb zum RUP: „Die konsequente und komplette Nutzung von RUP macht erst bei Teams mit über 10 Personen Sinn. Es müssen über 30 Rollen besetzt werden und über 100 verschiedene Artefakttypen erzeugt, dokumentiert und verwaltet werden." Artefakttypen sind Arbeitsergebnistypen, also Informationseinheiten, die während des Projektes entstehen und gebraucht werden; sie stellen sowohl Ein- als auch Ausgabe von Aktivitäten dar.

Beim RUP-Modell gibt es weitestgehend keine Kontrolle für die gesamte Qualitätsverantwortung. Stattdessen ist jeder Mitarbeiter für die Qualität seiner Arbeitsergebnisse selbst verantwortlich. Erreicht

wird dies durch zwei verschiedene Rollen innerhalb jeder Phase in der der Mitarbeiter entweder für die Qualitätsplanung (dieser Phase) oder für die Qualitätssicherung zuständig ist.

5.5 Der Ansatz ROOM

Architektur- und Detailschicht

ROOM (Real-time Object Oriented Modeling) wurde von Objectime für den Entwurf von Echtzeitsystemen entwickelt. Es teilt sich in eine Architekturschicht sowie eine Detailschicht auf. In der Architekturschicht wird ein Grobentwurf des Gesamtsystems mit Hilfe von graphischen Darstellungsmitteln erstellt. In der Detailschicht werden die Funktionen mit herkömmlichen Programmiersprachen wie C/C++ und Java beschrieben. Besonders hervorzuheben ist die formale semantische Fundierung von ROOM. Auf diese Weise sind bereits Grobentwürfe ausführbar bzw. testbar (Selic, 1994).

5.5.1 Softwarewerkzeuge und Umgebung

ROOM ist insbesondere für die Software-gestützte Anwendung entwickelt worden; denn nur so können seine Stärken in vollem Umfang genutzt werden. Eine dieser Stärken ist die automatische Codegenerierung aus einem Entwurf in ROOM heraus. Es kann jederzeit ein lauffähiger Code erzeugt werden. Dadurch werden Änderungen nur am Entwurf in der Architekturschicht und in der Detailschicht durchgeführt und nicht im Code. Ein „Architekturverfall“, wie er oft bei der Entwicklung mit UML vorkommen kann, wird damit vermieden.

Echtzeitsysteme

Echtzeitsysteme werden meist mit Spezialhardware betrieben. Da es oft nicht möglich ist, die Software direkt auf dem Zielsystem zu entwickeln, gibt es bei ROOM-Werkzeugen die Möglichkeit, das Zielsystem auf einem Entwicklungsrechner (Arbeitsplatzrechner) zu emulieren. Dadurch kann bereits im Vorfeld erkannt werden, ob das Zusammenspiel zwischen Hard- und Software reibungslos funktioniert.

Neben der Funktionalität kann mit der Emulation auch das Echtzeitverhalten überprüft werden. Durch die systematische Abbildung der Modelle auf Programmcode ist die Ausführungszeit für jedes Konstrukt nachvollziehbar. Auf diese Weise kann schon

sehr früh erkannt werden, ob die Hardware genügend Rechenleistung besitzt, um alle Echtzeitanforderungen zu erfüllen. Jedoch können auf diese Weise nicht alle prinzipiell auftretenden Ablaufmöglichkeiten des Kontrollflusses abgedeckt werden. Da die Detailschicht in einer herkömmlichen Programmiersprache implementiert wird, ist es nicht ohne weiteres möglich, Rückschlüsse auf deren Laufzeit zu ziehen. Um diese Hürde zu überwinden, kann bei der Simulation die voraussichtliche reale Laufzeit dieser Programmteile manuell angegeben werden. Neben der Emulation kann das Laufzeitverhalten des Entwurfs auch analytisch bestimmt werden. Dadurch kann der denkbar schlechteste Fall (engl. worst case) gefunden werden.

Ein sehr mächtiges Tool das diese Eigenschaften unterstützt ist „Rational Rose Real Time“ das von IBM (früher: Rational) vertrieben wird.

Wie bereits erwähnt, findet beim Entwicklungsprozess mit ROOM der Softwareentwurf auf einem Arbeitsplatzrechner statt und nicht auf dem Zielsystem selbst. Um eine bestmögliche Portabilität vom Entwicklungsrechner auf das Zielsystem zu gewährleisten, wird sowohl bei der Simulation als auch auf dem Zielsystem die ROOM Virtual Machine verwendet. Sie besitzt eine zur Java Virtual Machine vergleichbare Funktionsweise, erlaubt aber zusätzlich an ihr vorbei den direkten Zugriff auf die Hardware.

5.5.2 Einführung

Aktoren

Aktoren sind vergleichbar mit Klassen in anderen objektorientierten Programmiersprachen. Sie stellen eine vollständige Programmeinheit dar, d. h. sie können unabhängig von anderen Aktoren ausgeführt und getestet werden. Im Programm werden sie nebenläufig ausgeführt; jeder Aktor bildet dabei eine eigene Task. Aus diesem Grunde wird beim verwendeten Betriebssystem die Multitasking-Fähigkeit vorausgesetzt. Auf die innere Struktur kann von außen nicht direkt zugegriffen werden, sondern lediglich über ihre Schnittstellen. Dadurch wird eine Datenkapselung realisiert. Aktoren können hierarchisch dekomponiert werden. Sie bestehen dann aus mehreren Subaktoren (Selic, 1994).

Protokolle, Ports, Nachrichten

Da Aktoren nicht direkt miteinander kommunizieren können, werden hierfür spezielle Schnittstellen, sogenannte *Ports*, benötigt. Über Ports werden bidirektionale Nachrichten ausgetauscht, die aus einem Signal sowie optionalen Parametern bestehen. Über einen Port dürfen nur Nachrichten bestimmter Typen gesendet bzw.

empfangen werden. Diese Nachrichtenregeln werden in *Protokollen* festgelegt. Ein Protokoll besteht wiederum aus einer Menge von Signalen, deren Parametern und der jeweiligen Kommunikationsrichtung (ein- oder ausgehend). Um zwei Ports zu verbinden, muss ein Port konjugiert werden, d. h. die Nachrichtenrichtung wird invertiert. Ein Protokoll muss dann nur einem der beiden Ports zugewiesen werden, da dem anderen Port automatisch das konjugierte Protokoll zugewiesen wird. Ist ein Aktor in Subaktoren dekomponiert und soll ein Port einer der Subaktoren nach außen weitergegeben werden, so können normale Ports nicht benutzt werden, da der empfangende Port die Nachricht nicht nach außen weiterleiten würde. Eine Lösung bieten hier die sogenannten „Relay Ports“, die eine Nachricht empfangen und weiterleiten (Selic, 1994).

ROOM Charts

In ROOM wird das Verhalten eines Programms durch endliche Automaten beschrieben. Die graphische Darstellung dieser Automaten bezeichnet man als ROOM Chart. Die Zustandswechsel werden durch sogenannte Trigger ausgelöst. Ein Trigger besteht aus einem Signal und einem Port. Kommt nun an diesem Port das angegebene Signal an wenn der Trigger aktiv ist, findet ein Zustandsübergang statt. Bei jedem Zustandsübergang kann zusätzlich eine Aktion angegeben werden, die beim Zustandsübergang ausgeführt wird. Bevor ROOM ein neues Signal verarbeitet, führt es vorher die Aktion vollständig aus. Da es bei den parallel laufenden Aktoren vorkommen kann, dass zwei Signale zum fast selben Zeitpunkt kommen hat jeder Port eine Warteschlange, welche die Signale solange puffert, bis sie abgearbeitet werden können. Das bei Echtzeitsystemen oft Signale vorkommen, die sofort ausführt werden müssen (z. B. der Notstopp einer Maschine), gibt es die Möglichkeit, den Signalen eine statische Priorität zuzuordnen. Dadurch können Signale mit hoher Priorität solche mit einer niedrigeren in der Warteschlange überholen.

Wie bei Aktoren ist es auch bei Zuständen möglich, sie hierarchisch zu dekomponieren. Befindet man sich aber gerade bei der Abarbeitung eines Subzustands und ein Trigger aus der nächst höheren Kompositionsebene wird ausgelöst, so wird die Abarbeitung des Subzustands abgebrochen, und der Zustandsübergang der höheren Ebene wird ausgeführt. Dies geschieht aber nur dann, wenn der gleiche Trigger nicht auch bei beim Subzustand einen Zustandsübergang auslöst. Mit bedingten Übergängen ist es möglich, den Zielzustand eines Zustandsübergangs von zusätzlichen Bedingungen abhängig zu machen (Selic, 1994).

Vererbung

Um eine bessere Wiederverwendbarkeit der Konstrukte zu ermöglichen, unterstützt ROOM die Vererbung für die Sprachelemente Aktoren, ROOM Charts sowie Protokolle. Allerdings ist nur die Einfachvererbung möglich; ansonsten können alle Attribute vererbt, verändert oder gelöscht werden.

Dynamischer Entwurf

Die bisher vorgestellten Elemente erlauben nur den statischen Entwurf. In ROOM ist es aber möglich Aktoren erst zur Laufzeit zu erzeugen. Dazu dienen optionale Instanzen als Platzhalter für dynamische Aktoren, die statisch festlegen, welche Aktorklasse verwendet werden soll. Durch eine Kombination aus austauschbaren und optionalen Instanzen wird eine Schnittstelle für die zur Laufzeit erzeugten Objekte geschaffen. Dies ist möglich da ROOM auch „Dispatching“ unterstützt.

Schichten

Um ROOM für sehr umfangreiche Entwürfe übersichtlich zu halten, kann man den Entwurf in verschiedene Schichten unterteilen. Die Kommunikation zwischen zwei Schichten wird mit speziellen Aktoren realisiert. Die sendende Schicht benutzt einen sogenannten Service Provision Point (SPP) und die empfangende einen Service Access Point (SAP). Die Verbindung beider Schichten erfolgt dann implizit. Auf diese Weise kann jeder Aktor den Service Point seiner Schicht wie einen ganz normalen Port nutzen (Selic, 1994).

5.5.3 Echtzeitfähigkeit

Wie sein Name bereits verrät, wurde ROOM für den Entwurf von Echtzeitsystemen entwickelt. Inwieweit ROOM tatsächlich den Anforderungen zur Modellierung von Echtzeitsystemen gerecht wird, soll im Folgenden kurz diskutiert werden:

Rechtzeitigkeit

Rechtzeitigkeit: Bei Echtzeitsystemen spielt die Rechtzeitigkeit eine besonders große Rolle. Jedoch bietet ROOM keine Möglichkeit Laufzeiten und Reaktionszeiten zu modellieren. Allerdings ist es möglich, die Worst Case Execution Time zu berechnen, um den für ein Zielsystem spezifischen Nachweis der Rechtzeitigkeit zu führen.

Zuverlässigkeit

Zuverlässigkeit: Wie bereits erwähnt, wird mit Hilfe der automatischen Codegenerierung in ROOM ein Architekturverfall vermieden, die Wiederverwendung von Aktoren und Schnittstellen ermöglicht und damit letztendlich auch die Wartbarkeit der Software erhöht. Dies trägt erheblich zur Zuverlässigkeit des entwickelten Systems bei. Weiterhin wirken sich umfangreiche Möglichkeiten für

Test und Analyse in den ROOM Werkzeuge positiv auf die Zuverlässigkeit aus.

Verteiltheit

Verteiltheit: Durch Protokolle und Verarbeitung von Signalen durch Automaten lassen sich verteilte Systeme in ROOM gut darstellen.

Nebenläufigkeit

Nebenläufigkeit: Aktoren laufen in ROOM nebenläufig ab, Nebenläufigkeit wird demnach unterstützt.

Komponierbarkeit

Komplexität: Aktoren und ROOM Charts können hierarchisch dekomponiert werden. Dadurch kann ein komplexer Softwareentwurf in mehrere kleinere zerlegt werden, die dann einzeln im Detail entworfen werden.

Dynamik

Dynamik: In ROOM können Aktoren dynamisch zur Laufzeit erstellt werden. Durch optionale Instanzen wird zeigen an wo zur Laufzeit Aktoren entstehen können. Dadurch kann die Dynamik bereits im Entwurf dargestellt werden.

Insgesamt kann festgehalten werden, dass die Sprache ROOM zur Entwicklung von Echtzeitsystemen nur bedingt geeignet ist, da sie es nicht erlaubt, Echtzeitanforderungen in Form der Rechtzeitigkeit (siehe Kapitel 3) zu modellieren. Die verfügbaren Tools schließen aber diese Lücke größtenteils, da mit ihrer Hilfe Echtzeitanforderungen teils simulativ, teils analytisch überprüft werden können.

5.6 Hardware/Software-Codesign

Die meisten modernen eingebetteten Systeme bestehen aus kooperierenden Hardware- *und* Softwareteilen. Dabei wird die Gesamtaufgabe der Systementwicklung in Teilaufgaben zerlegt, die dann entweder von Hardware- oder Softwarekomponenten bearbeitet werden. So lassen sich Teilaufgaben an die harte Echtzeitanforderungen geknüpft werden, oft nur mit Hilfe von (teuerer) Spezialhardware realisieren. Andererseits sollen Entwurfskosten bzw. -zeit sowie die Systemkomplexität eingebetteter Systeme so gering wie möglich gehalten werden.

Time-to-Market

Vor dem Hintergrund stetig schrumpfender Produktlebenszyklen eingebetteter Systeme ist dies dringend erforderlich. So gibt es beispielsweise in der Telekommunikationsindustrie Produkte, die einen Lebenszyklus von weniger als zwei Jahren besitzen. Das zwingt die Industrie, die Produkte so schnell und kostengünstig wie möglich auf den Markt zu bringen. Dadurch steigen die Time-to-Market-Anforderungen, die zu bewältigen sind, erheblich an.

Der Entwurfsprozess für solche gemischte Hardware/Software-Systeme stellt uns vor neue Herausforderungen, da zum einen die Komplexität der zu entwerfenden Systeme sehr hoch ist und zum anderen die zyklische Abhängigkeit von Hardware- und Softwarekomponenten aufgelöst werden muss. Der Einsatz von rechnergestützten Entwurfswerkzeugen ist hier notwendig.

Cosimulation

Stand der Technik ist hier lediglich die sogenannte *Cosimulation.* Bei ihr wird für eine vorgegebene Hardware die „passende" Software entwickelt und anschließend mit Ansätzen des Debugging sowie der gemischten Simulation von Hard- und Software überprüft, ob die Entwicklungsziele weitestgehend eingehalten werden konnten. Bei dieser Vorgehensweise werden Hard- bzw. Software getrennt spezifiziert. Schon zu Beginn des Entwicklungsprozesses ist es erforderlich zu entscheiden, welche Teile des Produkts in Hardware und welche in Software realisiert werden sollen. Treten zu einem späteren Zeitpunkt Unstimmigkeiten auf, können diese meist nur durch umfangreiche Änderungen in vielen Entwicklungsschritten behoben werden. Wünschenswert ist hier ein Entwicklungsmodell, welches die dedizierte und schrittweise Verbesserung von Entwicklungsschritten zulässt, sowie eine integrierte Entwicklung des gesamten eingebetteten Systems, die alle Anforderungen (hinsichtlich Funktionalität, Zeitbedingungen, Sicherheit etc.) erfüllt.

HW/SW-Codesign

Den vorgenannten Herausforderungen kann mittels Hardware/Software-Codesign entgegen getreten werden. Unter *Hardware/Software Codesign* (kurz: HW/SW-Codesign) versteht man den gemeinsamen Entwurf von Hardware- und Softwarekomponenten eines Systems.

Bei diesem Ansatz wird erst zu einem späten Zeitpunkt in der Entwicklungsphase entschieden, welche Teilbereiche eines Produktes in Hardware und welche in Software realisiert werden. Beide Teilbereiche werden bis zu diesem Zeitpunkt gemeinsam und gleichzeitig entwickelt. Ziel dieser späten Entscheidung ist die Optimierung einer Nutzen-Kosten-Zielfunktion, in welcher Anforderungen wie harte Echtzeitbedingungen genauso ihren Niederschlag finden können wie Entwicklungskosten.

Die Entwicklung solcher Systeme wirft jedoch eine Reihe neuartiger Herausforderungen des Entwurfs auf, insbesondere

Herausforderungen

- die Frage der Auswahl von Hardware- und Softwarekomponenten,
- die Partitionierung einer Spezifikation in Hard- und Software, d. h. die Abbildung von Teilen der Spezifikation auf Hardware bzw. auf Software,
- die automatische Synthese von Schnittstellen- und Kommunikationsstrukturen und
- die Verifikation und Cosimulation.

Allgemein lässt sich der Prozess des HW/SW-Codesign technologieunabhängig in folgende 3 Schritte unterteilen:

Schritte des HW/SW-Codesign

- **Modellierung:** Konzeption und kontinuierliche, möglichst schrittweise Verfeinerung der Spezifikation bis zum integrierten HW/SW-Modell
- **Validierung:** Überprüfung auf Übereinstimmung zwischen Systemanforderungen und -verhalten.
- **Implementierung:** Physikalische Realisierung der Hardware und Codeerzeugung der Software.

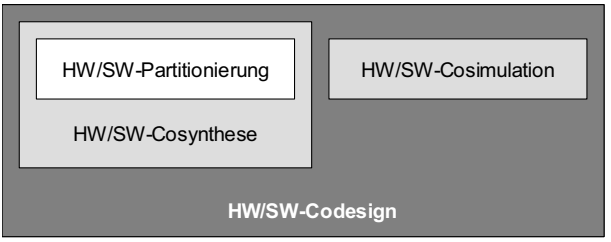


Abb. 5.5: Einordnung HW/SW-Codesign

Abbildung 5.5 zeigt eine Einordnung der Fachbegriffe. Das Kriterium bei der *Hardware/Software-Partitionierung* ist das folgende: Standardhardware auf Basis programmierbarer Mikrocontroller ist im Vergleich zu Spezialhardware billiger, rechnet aber weniger schnell. Aufgrund der immer höheren Anforderungen an die Verarbeitungsgeschwindigkeit und –vielfältigkeit haben daher die Entwickler eingebetteter Systeme immer häufiger die Entscheidung zwischen einer sehr komplexen Spezialhardware oder Standardhardware mit großem Softwareanteil

zu treffen. In diesem Zusammenhang spricht man auch vom *Hardware/Software-Tradeoff*.

Ausgewählte Ansätze des HW/SW-Codesign

Zwar hat sich bis heute noch keine Standardlösung für das HW/SW-Codesign durchgesetzt, wohl gibt es aber zahlreiche, größtenteils sehr fundierte Ansätze hierfür, von denen wir im Folgenden exemplarisch einige wenige aufgreifen werden:

- **COSYMA** Cosynthesis for Embedded Architectures: Technische Universität Braunschweig, siehe (Ernst et al., 1993).
- **VULCAN:** Stanford University, siehe (Gupta, DeMicheli, 1993).
- **SpecSyn:** University of California, Irvine, siehe (Gajski et al., 1998).
- **POLIS:** University of California, Berkeley, siehe (Balarin et al., 1997).

Partitionierung

Bei diesen Ansätzen des HW/SW-Codesigns kommen Verfahren aus dem Compilerbau gleichermaßen zum Einsatz wie solche aus der Hardwaresynthese. Von zentralem Interesse in aktuellen Forschungsarbeiten sind hier immer wieder verschiedene Aspekte *der Partitionierung.* Alternative Zielfunktionen der Partitionierung können prinzipiell die folgenden sein:

- *Verbesserung der Gesamtperformanz* einer Applikation oder einer Gruppe von Applikationen. Die wesentliche Aktion der Partitionierung besteht hierbei in einem sogenannten „Profiling“ mit typischen Datensätzen zur Abschätzung des Systemverhaltens.
- *Einhaltung harter Echtzeitanforderungen* für dedizierte Operationen. Eine weitverbreitete Strategie ist hierbei die Abbildung der gesamten Operation in Hardware, während der „Rest“ möglichst in Software bleibt, um die Systemkosten möglichst gering zu halten.
- *Einhaltung dedizierter Implementierungsgrößen*, da auch physikalische Größen der Hardware, auf die abgebildet wird, Rahmenbedingungen darstellen können. In diesem Fall wird diese Abbildung so gestaltet werden, dass jede Funktionalität in einem physikalischen Baustein integriert wird.

Schritte der Partitionierung

Die Partitionierung wird dabei in der Regel in Form der nachstehenden drei Schritte durchgeführt:

- **Allokation:** Aus gegebener HW-Bibliothek (z. B. Standardprozessoren, digitale Signalprozessoren, Mikrocontroller, Applikationsspezifische Integrierte Schaltungen, Speicher, Busse) benötigte Komponenten auswählen bzw. Synthese der Komponenten
- **Partitionierung** der funktionalen Objekte im engeren Sinne (nur Aufteilung)
- Zuordnung (**Mapping**, **Technologieauswahl**) der partitionierten Funktionalität zu HW-Komponenten

Alternativen der Partitionierung

Für die Partitionierung (im weiteren Sinne) gibt es die folgenden beiden grundsätzlichen Alternativen:

- **Strukturelle Partitionierung** (System wird erst implementiert, dann partitioniert)
- **Funktionale Partitionierung** (System wird erst partitioniert, dann implementiert)

Aufgrund ihrer algorithmischen Komplexität kann die Partitionierung meist nicht vollständig Computer-gestützt erfolgen, sondern erfordert vielmehr zusätzlich die Kreativität des Entwicklers.

Zwei aktuelle Ansätze zur funktionalen Partitionierung sind COSYMA und VULCAN (siehe auch oben). Wie Tabelle 5.1 zeigt, unterscheiden sie sich teilweise erheblich.

Umfänge des HW/SW-Codesign

Ein zukünftiger, idealtypischer Lösungsansatz für HW/SW-Codesign müsste eine Entwicklungsumgebung mit -verfahren nebst -methodik für folgende Umfänge vorsehen:

- Systemmodellierung funktionaler und nicht-funktionaler Aspekte
- Möglichkeit der Technologievorgabe
- Automatische sowie teilautomatische HW/SW-Partitionierung
- Hierarchische Analyse
- Schrittweise Verfeinerung
- Automatische HW/SW-Cosynthese und Codegenerierung

Tab. 5.1: COSYMA und VULCAN im Vergleich

	COSYMA	VULCAN
Sprache	C^x: C erweitert um Prozesskonzept und Interprozesskommunikation	HardwareC: C erweitert um Prozesskonzept und Interprozesskommunikation
Echtzeitanforderungen	Spezifikation von min. bzw. max. Laufzeiten zwischen Labels im Programm	Spezifikation von min. bzw. max Laufzeiten sowie Datenraten
Strategie	SW-orientiert: Soviel Funktionalität wie möglich (aus Kostengründen) in SW implementieren, aufwändige Programmblöcke (z. B. Schleifen) auf HW verlagern, anwendungsspezifische HW nur dort, wo andernfalls harte Echtzeitanforderungen verletzt würden.	HW-orientiert: Soviel Funktionalität wie möglich (aus Zeitgründen) in HW implementieren, SW nur dort, wo harte Echtzeitanforderungen nicht verletzt werden.
Datenaustausch	über gemeinsamen Speicher	mittels Sende- und Empfangoperationen
Zielarchitektur	1 Standardprozessor, 1 Coprozessor (ASIC), Kopplung über 1 gemeinsamen Speicher	1 Standardprozessor, mehrere ASICs, Kopplung über 1 globalen Bus
Abstraktionsebene	Programmblöcke	Programmblöcke und Operationen
Optimierungsverfahren	Simulated Annealing	Greedy Algorithmus

Eine umfangreiche kommentierte Bibliographie zu HW/SW-Codesign ist in (Buchenrieder, 1995) zu finden. Ferner beleuchten (De Micheli, 1996), (Kumar, 1995) und (Rosenblit, 1995) das Thema aus unterschiedlichen Blickwinkeln. Darüber hinaus sei dem interessierten Leser für aktuelle Detailinformationen die Lektüre der Tagungsbände (engl. proceedings) der internationalen Workshops bzw. Symposen „on Hardware/Software Co-Design“ (kurz: CODES) zu raten die bis 2002 zehnmal stattgefunden haben und seit 2003 in Kombination mit dem Thema der Systemsynthese als „International Conference on Hardware/Software Codesign and System Synthesis“ (www.codesign-symposium.org) weitergeführt werden.

5.7 Die MARMOT-Methode

Komponenten

Der am Fraunhofer-Institut für Experimentelles Software Engineering (IESE) entwickelte MARMOT (Method for Component-Based Real-Time Object-Oriented Development and Testing) Ansatz verfolgt die Idee, Systeme als Kombination verschiedener schon existierender Bauteile bzw. Komponenten zu betrachten (vgl. www.marmot-project.de). Diese Bauteile haben sich bereits in anderen Applikationen bewährt und müssen evtl. nur noch an das neue System angepasst werden. Solche Komponenten müssen nicht zwingend selbst entwickelt worden sein und können somit auch von Drittanbietern stammen.

Völlig problemlos gestaltet sich ein derartiges Zusammenbauen natürlich nicht. Objektorientierte Sprachen sind compilergebunden. Somit sind Objekte zur Laufzeit in größere Systeme eingebunden und stehen nicht zur Wiederverwendung zur Verfügung. Ein weiteres Problem ergibt sich oft aus der Vernachlässigung nichtfunktionaler Eigenschaften (Speicherplatzbedarf, Sicherheit, Performanz, Zeitverhalten). Effektive und systematische Wiederverwendung hängt jedoch von der Komponentenqualität *und* ihren nichtfunktionalen Eigenschaften ab.

MARMOT unterstützt diesen Entwicklungsansatz unter Beachtung der genannten Herausforderungen. MARMOT basiert auf der KobrA-Methode (siehe unten), erweitert diese und unterstützt gezielt die Entwicklung eingebetteter, sicherheitskritischer Echtzeit-Systeme. Solche Systeme, die aus Hard- und Software Bausteinen bestehen, werden mittels MARMOT durchgehend und systematisch beschrieben. Dabei gibt es keine Trennung der Hard- und Softwarekomponenten. Beide werden einheitlich betrachtet und beschrieben. Je nach Komponententyp (Hard- oder Software) werden bestimmte Konzepte und Dokumente zur Verfügung gestellt.

Wichtig bei der Entwicklung neuer Applikationen sind die einzelnen Komponenten. Durch deren einheitliche Entwicklungsschritte ergibt sich eine Unabhängigkeit von der Größe und Komplexität des Systems. Durch verschiedene Sichtweisen auf die einzelnen Komponenten („was“ versus „wie“) wird schon während der Entwicklung auf deren funktionale bzw. nichtfunktionale Anforderungen eingegangen. Die bereits eben erwähnten Probleme werden somit explizit berücksichtigt. Jede Komponente wird als eigenes System betrachtet, jedes System wiederum als Komponente. Komponenten werden durch ihre Spezifikation und ihre Realisierung beschrieben. Die Spezifikation beschreibt dabei die

Funktion einer Komponente, die Realisierung hingegen deren Umsetzung.

In beiden sind UML-Modelle sowie Bau- und Schaltpläne enthalten. Dies verdeutlicht nochmals den Gedanken der verzahnten Entwicklung von Hard- und Software. In einem rekursiven Ansatz werden die einzelnen Komponenten soweit verfeinert bzw. spezifiziert, bis eine bereits existierende Lösung gefunden wird. Diese kann dann als Element eingegliedert werden. Gibt es eine solche Lösung noch nicht, wird die Komponente entwickelt und kann für weitere Projekte verwendet werden. MARMOT Projekte sind hierarchisch aufgebaut, d. h. sie werden als Summe einzelner Komponenten betrachtet. Komponenten ordnen sich in das Gesamtsystem ein, können aber völlig isoliert betrachtet und entwickelt werden.

Eine derartige Entwicklungsstrategie bringt eine hohe Änderbarkeit mit sich. Somit kann schnell auf geänderte Verhältnisse oder neue Anforderungen reagiert werden. Dies gilt natürlich sowohl für die Software als auch für die Hardware. Durch das Prinzip der Wiederverwendung und der auf Sichten basierten Komponentenentwicklung wird der Qualitätssicherung schon früh im Projekt Rechnung getragen. Für das fertige Produkt und daraus hervorgehenden Weiterentwicklungen und Varianten ergibt sich eine deutliche Qualitätssteigerung. Das Baukastenprinzip mit der Entwicklung und Verwendung eigener Elemente oder kommerzieller Produkte Dritter führt zu geringen Entwicklungskosten und steigert den Grad der Wiederverwendung. Die kombinierte Entwicklung von Hard- und Software verkürzen die Zeit bis zur Marktreife des Produktes.

KobrA Methode

Die *KobrA*-Methode (Komponentenbasierte Anwendungsentwicklung) entstand im Rahmen des *KobrA* – Projekts, das vom Bundesministerium für Bildung und Forschung ins Leben gerufen wurde. Beteiligt waren die Softlab GmbH aus München als Projektleiter, die Psipenta GmbH aus Berlin und die GMD FIRST aus Berlin. Die „Komponentenbasierte Anwendungsentwicklung“ basiert auf dem *Product Line Software Engineering*, besser bekannt als PULSE, des Fraunhofer Institute *for Experimental Software Engineering* (IESE).

Prozessteile

Die Entwicklung der Systemfamilie basiert auf drei Prozessteilen:

Kontext-realisierung

(1) Innerhalb der Kontextrealisierung wird ein Unternehmensmodell mit Konzept- und Prozessdiagrammen zur Beschreibung der Geschäftsprozesse, ein Strukturmodell zur Beschreibung von

strukturellen Interaktionen mittels Klassendiagramm, ein Aktivitätsmodell bestehend aus Use Cases (Anwendungsfällen) und Aktivitätsdiagrammen, ein Interaktionsmodell bestehend aus Sequenz- und Kollaborationsdiagrammen sowie ein Entscheidungsmodell erstellt.

Application Engineering

(2) Nach der Kontextrealisierung findet das Framework Engineering für die Systemfamilie oder das Application Engineering für ein Familienmitglied statt. Das Framework enthält die Referenzarchitektur der Systemfamilie, die das Ableiten einer Applikation aus den gewünschten Komponenten erlaubt. Auch das Framework selbst stellt eine Komponente dar, wodurch KobrA als rekursiver Prozess bezeichnet wird, d. h. alles wird als Komponente behandelt und kann wiederum aus Komponenten zusammengesetzt sein.

Implementierung

(3) Im dritten Teil der KobrA-Methode findet die Implementierung der Komponenten statt. Zum einen können bereits existierende Komponenten genutzt oder, falls existierende Komponente den gegebenen Anforderungen nicht entsprechen, müssen entsprechende Komponenten neu implementiert werden.

Trennung von Produkten und Prozessen

Das zentrale Konzept von *KobrA* ist die Trennung von Produkten und Prozessen. Die Produkte stellen dabei die Ergebnisse der Entwicklung dar und werden unabhängig von den Prozessen und zeitlich davor definiert. Alle Produkte zielen auf die Beschreibung von individuellen Komponenten ab, in denen die Variabilität der Familie enthalten ist.

UML Use Cases

Die Phase des *Requirements Engineering* in KobrA basiert auf textuellen Beschreibungen und Use Case Diagrammen der UML, die Variationen über Stereotypen modellieren. Die Variationen der einzelnen Komponenten werden in der Tabelle vermerkt, indem für jede Variante der Familie die benötigten Merkmale eingetragen werden, was in KobrA als *Product Map* bezeichnet wird. Für die spätere Ableitung von Applikationen wird in KobrA noch ein *Decision Model* erarbeitet, in dem alle Variationspunkte enthalten sind, wobei hier eine Ähnlichkeit zu den Merkmalmodellen von Kang besteht, ohne dass die Merkmalmodelle von KobrA genutzt werden. Vor der Ableitung einer Variante muss für jeden Variationspunkt entschieden werden, ob er im System vorhanden ist. Darüber hinaus sind auch Parameter als Variationspunkte möglich, wobei diese für eine konkrete Applikation mit einem Wert belegt werden müssen. Im *Decision Model* sind eine Reihe von Fragen enthalten, die der Kunde zur Ableitung einer Applikation beantworten muss. Stellt der Kunde eine Anforderung, die nicht als Komponente in der Systemfamilie enthalten ist, muss diesem

Wunsch durch Integration einer neuen Komponente in die Familie nachgekommen werden. Dabei werden wenn möglich Ähnlichkeiten zu bereits existierenden Komponenten der Familie ausgenutzt. Die Interaktion der einzelnen erfragten Variabilitäten des *Decision Models*, als *Effect* bezeichnet, sind ebenfalls ein textueller Form im Modell hinterlegt und müssen vom Entwickler aufgelöst werden.

Zusammenfassung KobrA

KobrA stellt eine umfassende Methode zur Entwicklung von Systemfamilien dar. Das von KobrA genutzte *Decision Model* zur Erfassung und Verarbeitung von Variabilität in der *Requirements Engineering* Phase stellt eine gute Basis für die Ableitung eines Familienmitglieds dar, kann jedoch mit der Übersichtlichkeit und Verständlichkeit der Merkmalmodelle nicht konkurrieren. Das *Decision Model* kann nicht automatisiert verarbeitet werden, insbesondere müssen die enthaltenen Auswirkungen, sog. *Effects,* manuell durch den Entwickler überprüft werden.

5.8 Hybride Systeme und hybride Automaten

Einordnung

Hybride Systeme gehören zur Klasse der eingebetteten Echtzeitsysteme. Sie sind durch die nicht-triviale Verkopplung diskreter und kontinuierlicher Aspekte charakterisiert. Systeme werden dann als *hybrid* bezeichnet, wenn deren Eingabewerte nicht nur in digitaler Form vorliegen, sondern auch in analoger Form. Die analogen Werte bestehen dabei nicht aus einer Menge von definierten Werten, sondern sie verändern sich – wie in der Natur üblich – stufenlos. Beispiele für analoge Werte sind Temperatur, Neigungswinkel usw. Hybride Systeme kommen dann zum Einsatz, wenn mit der Umwelt interagiert wird. Sie sind durch die nicht-triviale Verkopplung diskreter und kontinuierlicher Aspekte charakterisiert. Viele eingebettete Systeme sind hybride Systeme, da sie neben kontinuierlich arbeitenden Komponenten (Schnittstellen zur Umwelt) auch welche, die auf diskreter Basis (Steuerungen) funktionieren, beinhalten.

5.8.1 Einleitung

Beispiele

Beispiele für hybride Systeme sind Systeme wie das Antiblockiersystem (ABS), bei dem abhängig vom Rollen oder Gleiten des Reifens unterschiedlich von der Software eingegriffen

werden muss. Darüber hinaus spielen hybride Systeme angesichts der Flexibilität, die Software-basierte Steuerungen bieten und angesichts der sinkenden Hardwarepreise eine immer wichtigere Rolle in Forschung und Industrie. Einsatzgebiete für hybride Systeme sind z. B. Automatisierungstechnik, Fahrassistenzsysteme im Automobilbereich, Flugleitsysteme und Robotik. Einige Arbeiten in diesem Gebiet beschäftigen sich etwa mit Greifprozessen von Roboterhänden oder fußballspielenden Robotern.

Funktionen eines eingebetteten Systems wie etwa das Ein- und Ausschalten von Motoren oder Öffnen und Schließen von Ventilen zählen zu den ereignisdiskreten dynamischen Systemen. Bei der Untersuchung ereignisdiskreter dynamischer Systeme zeigte sich, dass bei ihnen ganz andere Probleme als bei kontinuierlichen Systemen auftreten. Die Ursache hierfür liegt in der Begrenztheit der Werte, die diskrete Signale und Systeme annehmen können.

In technischen Systemen interagieren die ereignisdiskreten Teile mit kontinuierlichen dynamischen Prozessen, indem sie z. B. Stelleingriffe ausführen oder kontinuierliche Prozesse starten oder beenden. Die durch dieses Zusammenspiel entstehenden Systeme bezeichnet man als hybride dynamische Systeme.

Signalfluss in Systemen

Signalfluss in Systemen:
Im System können die Informationen (Ein-/Ausgangsignale) auf drei verschiedene Arten fließen (siehe auch Abbildung 5.5):

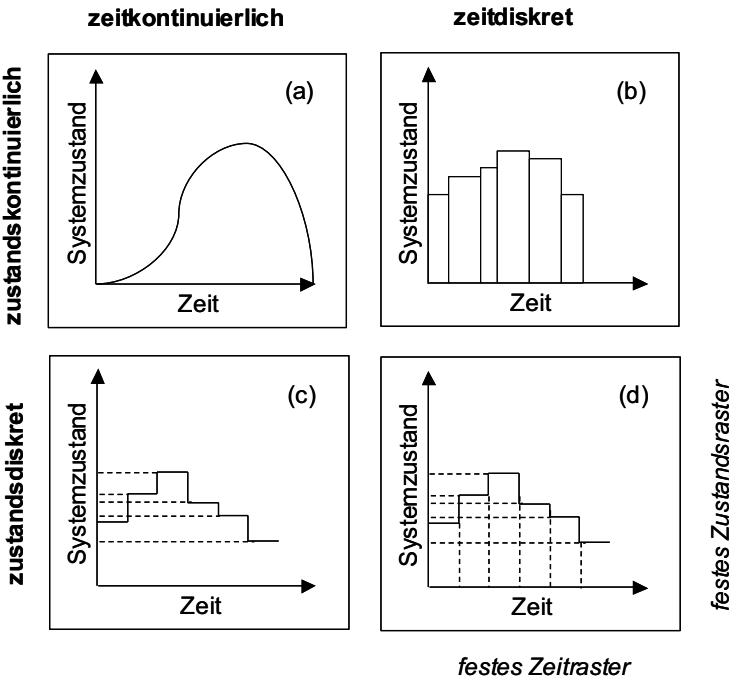
- **Kontinuierlich:** Informationen fließen in einem ununterbrochenen Strom (analog). Die meisten Prozesse aus der Natur, Physik und Technik sind kontinuierlicher Natur. Sie können durch Differentialgleichungen beschrieben werden.
- **Diskret:** Informationen fließen in (un-)regelmäßigen Abständen(digital). Die Elektronik und damit auch Computer arbeiten mit diskreten Informationen.
- **Hybrid:** Kombination aus kontinuierlichen und diskreten Informationsflüssen.

Die ersten beiden Arten werden von Wissenschaft und Industrie in weiten Teilen gut beherrscht. Die dritte Variante, die hybriden Informationsflüsse, stellt derzeit noch große Herausforderungen an die Informatik.

Kontinuierliche Signale können durch Abtastung in diskrete (quasi kontinuierliche) und diskrete Signale durch Interpolation in kontinuierliche umgewandelt werden. Neben diesen quasi kontinuierlichen, zeitdiskreten Signalen spielen ereignisdiskrete,

asynchron auftretende binäre Signale eine große Rolle, so beispielsweise in Steuerungen.

Abb. 5.5: Klassifikation Signalfluss



a) Kontinuierliche Systeme
b) Systeme diskreter Ereignisse
c) Diskrete Systeme
d) Ganzzahlige zeitdiskrete Systeme

Hybride Systeme haben häufig Regelungsaufgaben zu erfüllen. Standardmethoden der Regelungstechnik beschäftigen sich mit linearen Systemen. Lineare Systeme reagieren auf eine Linearkombination der Eingaben mit einer Linearkombination der Ausgaben. Hybride Systeme sind meist unstetig (Stauner, 2001). Es bedarf daher angepasster Ansätze für deren Spezifikation.

5.8.2 Spezifikation hybrider Systeme

Graphische Formalismen

Im Folgenden wollen wir, ohne Anspruch auf Vollständigkeit, einige Formalismen zur graphischen Spezifikation von hybriden Systemen kurz angeben.

Hybride Petri-Netze

Hybride Petri-Netze (David, 1997), (DiFebbraro, 2001):
In hybriden Petri-Netzen sind Token nicht mehr unteilbar. Sie können kontinuierlich fließen. Die Fließgeschwindigkeit der Token wird an speziellen „analogen" Knoten textuell durch Differentialgleichungen angegeben. Eine umfangreiche Online-Bibliographie zu hybriden Petri-Netzen kann unter bode.diee.unica.it/~hpn/ nachgeschlagen werden.

Hybride Statecharts

Hybride Statecharts (Kesten, Pnueli, 1992):
Sie stellen eine Erweiterung von Statecharts dar. Die Modellierung von kontinuierlicher Zeit geschieht über die Definition von Zeitintervallen für Transitionen. Jeder Transition ist ein Zeitintervall zugewiesen, welches obere und untere Grenzen für diese Transition festlegt. Zustände können mit Differentialgleichungen versehen werden, die kontinuierliche Änderungen beschreiben, solange das System in diesem Zustand ist. Der Vorteil dieses Ansatzes liegt in der Möglichkeit der hierarchischen Strukturierung.

HyCharts

HyCharts (Grosu, Stauner, 2002):
Sie sind eine Erweiterung des Modells der hybriden Statecharts. In HyCharts ist es möglich, auch die hierarchische Zerlegung eines Systems in einzelnen Teilen darzustellen.

Hybride Automaten

Hybride Automaten (Henzinger, 1996):
Sie sind eine Erweiterung von endlichen Automaten. Anders als diese enthalten sie sowohl diskrete als auch kontinuierliche Anteile. Der Zustand eines hybriden Automaten setzt sich aus „Ort" (engl. location) und „Variablenbelegung" (engl. valuation) zusammen. Solange der Automat in einem Ort ist, verändern sich die Variablen anhand vorgegebener Differentialgleichungen (DGL). Die Modellierung kontinuierlicher Änderungen geschieht in hybriden Automaten durch DGL in den Zuständen des Automaten. Diskrete Zustandsänderungen werden mit Hilfe von Transition vollzogen. Bei Benutzung einer Transition sind Zuweisungen an Variablen möglich.

Um dem Leser einen Eindruck von der Komplexität und den Herausforderungen der Modellierung hybrider Systeme zu verschaffen, wollen wir im Folgenden hybride Automaten ansatzweise analysieren.

Definition (hybrider Automat)

Definition (hybrider Automat) nach (Henzinger, 1996):
Ein hybrider Automat A ist definiert durch ein 10-Tupel (X, V, inv, init, flow, E, upd, jump, L, sync) =: A wobei:

- **X:** Endliche, geordnete Menge x_1, x_2, ..., x_n von Variablen aus den reellen Zahlen
- **V:** Endliche Menge von Orten (engl. locations). Bei hybriden Automaten sind Zustand und Ort nicht gleichbedeutend, wie das bei endlichen Automaten der Fall ist. Hier ist der Zustand durch das Tupel (v,s), bestehend aus einem Ort v und einer Belegung s charakterisiert.
- **inv:** Abbildung, die jedem Ort ein Prädikat als Invariante zuweist. In einem Ort v sind nur Belegungen s aus inv(v) erlaubt.
- **init:** Abbildung, die jedem Ort ein Prädikat als Anfangsbedingung (engl. initial condition) zuweist
- **flow:** Abbildung, die jedem Ort ein Prädikat (engl. flow condition) über $X \cup X' := \{x_1, x_2, ..., x_n, x'_1, x'_2, ..., x'_n\}$ zuweist. Sie beschreibt die Veränderung einer Variablen während sich der hybride Automat in einem Ort befindet. Hier sind nur DGL ersten Grades erlaubt. Damit wird das modellierbare Systemverhalten auf kontinuierliche Veränderungen beschränkt.
- **E:** Menge von Transitionen (engl. edges) zwischen den Orten. Die Übergänge werden jeweils durch einen Anfangs- und End-Ort dargestellt. Es sind auch mehrere Übergänge zwischen zwei Orten erlaubt.
- **upd:** Abbildung, die jedem Übergang eine Menge derjenigen Variablen zuweist, denen neue Werte zugewiesen werden (engl. update set). Bei einem Übergang können sich nur die Variablen verändern, die in der Update-Menge sind.
- **jump:** Abbildung, die jeder Transition ein Prädikat zuweist, welches erfüllt sein muss, damit die Transition gefeuert wird (engl. jump condition)
- **L:** Menge der Labels für die Transitionen. Sie stellt die endliche Menge von Synchronisationsbezeichnungen dar. Bei parallelen Automaten werden Übergänge mit gleicher Bezeichnung, d. h. mit gleichem Label gleichzeitig ausgeführt.

- **sync:** Abbildung, die jedem Übergang ein Label zuordnet. Jedem Übergang wird damit eine Synchronisationsbezeichnung zugeordnet.

Beispiel (Hybrider Automat) nach (Henzinger, 1996):
Nachstehende Abbildung 5.6 zeigt einen hybriden Automaten. Hierbei handelt es sich um die Steuerung eines Termostates zur Temperaturregelung eines Raumes. Bei einer angenommenen Anfangstemperatur von 20 Grad (Celsius) ist die Heizung zunächst aus. Im Zustand „Aus" fällt die Raumtemperatur aufgrund der gegebenen Umwelteinflüsse kontinuierlich linear um 0,1 Grad pro Zeiteinheit. Fällt sie unter 19 Grad, so findet ein diskreter Zustandsübergang von „Aus" nach „An" statt, d. h. die Steuerung des Termostates schaltet die Heizung ein. Da auch bei eingeschaltetem Zustand der Heizung noch die selben Umweltbedingungen gelten wie im ausgeschalteten, kühlt die Umwelt den Raum immer noch um 0,1 Grad pro Zeiteinheit. Diesem Kühlungseffekt wirkt nun aber eine „Heizleistung" von 5 Grad entgegen. Der vorgegebene hybride Automat in Abbildung 5.6 besitzt keine Labelmenge „L" und damit folglich auch keine Synchronisationsabbildung „sync"; diese wären nur bei Synchronisation mit anderen, parallel gekoppelten, hybriden Automaten erforderlich.

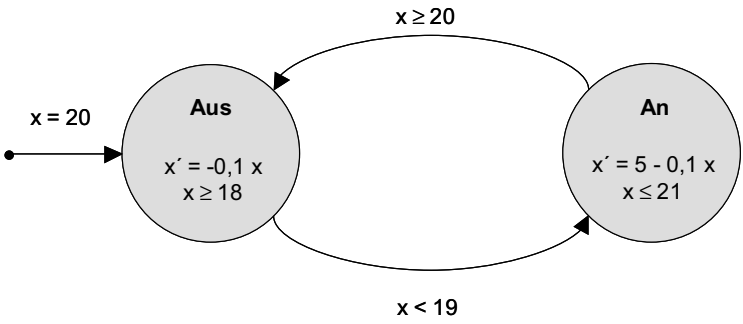


Abb. 5.6: Hybrider Automat (Termostat),vgl. (Henzinger, 1996)

Formale Verifikation, Model Checking

Die *formale Verifikation* von mit Hilfe hybrider Automaten spezifizierter hybrider Systeme ist durch Einsatz von Model Checking prinzipiell möglich. Hier kann beispielsweise das Softwarewerkzeug „Hytech" (Henzinger, 1996) Anwendung finden. Es erlaubt den Nachweis von sicherheitsrelevanten Eigenschaften, die insbesondere dadurch gekennzeichnet sind, dass sie nicht oder nur bei bestimmten Bedingungen eintreten sollen. Beispiel: „Alle Ampeln einer Kreuzung grün".

Entscheidbarkeit der Erreichbarkeit

Die Möglichkeit der formalen Verifikation von Sicherheitseigenschaften ist dabei jedoch stark eingeschränkt, da bereits die *Erreichbarkeit* von Systemzuständen nur für bestimmte Klassen hybrider Automaten entscheidbar ist. So ist zwar beispielsweise das Erreichbarkeitsproblem für die Klasse der sogenannten „Zeit-Automaten“ sowie für die Klasse der „einfachen Multirate-Zeit-Automaten“ entscheidbar, im Allgemeinen nicht aber für die Klassen der „linearen hybriden Automaten“ oder der „einfachen hybriden Automaten“. In keinem Fall entscheidbar ist das Erreichbarkeitsproblem ferner für „2-rate Zeit-Automaten“ sowie „einfache Integrator-Automaten“. Die vorgenannten Klassen hybrider Automaten unterscheiden sich im Wesentlichen durch die Form der Prädikate in „inv“, „init“, „flow“ und „jump“. Eine exakte Definition ist (Henzinger, 1996) zu entnehmen.

Die Verifikation eingebetteter Systeme ist wegen des erforderlichen Aufwands für die formale Modellierung zum einen, vor allem aber durch den exponentiell wachsenden kombinatorischen Aufwand schwierig. Für mittelgroße Systeme ist dies jedoch unter Einsatz von Model Checking bereits gut durchführbar. Die Verifikation hybrider Systeme kann ein noch viel komplexeres Problem darstellen. Es können Grenzzyklen, Gleitbewegungen, evtl. sogar chaotische Dynamik auftreten. Zurzeit erscheint der Ansatz der kompositionalen Analyse am erfolgversprechendsten; dabei werden gewisse Eigenschaften auf Teilsystemebene formal nachgewiesen und daraus das Verhalten des gesamten Systems gefolgert.

Vorteile

Hybride Automaten besitzen folgende *Vorteile:*

- Diskrete und kontinuierliche Anteile des Systems sind leicht modellierbar.
- Einfache Systeme sind einfach zu verstehen.
- Die formale Verifikation ist teilweise automatisierbar.

Nachteile

Sie weisen jedoch andererseits auch nachstehende *Nachteile* auf:

- Keine Hierarchie; komplexe Systeme werden ggf. unübersichtlich.
- Verifikationsalgorithmen sind sehr rechen- und speicherintensiv.
- Die Terminierung der Verifikationsalgorithmen ist bei vielen Klassen von hybriden Automaten nicht garantiert.
- Die Verifikation einzelner Module ist nicht möglich; nur das Gesamtsystem kann verifiziert werden.

5.9 Zusammenfassung

Die Entwicklung der Software eingebetteter Systeme ist eine komplexe Aufgabe, die derzeit noch nicht in allen Teilen zufriedenstellend beherrscht wird. Um die korrekte Funktionsweise solcher Systeme und kurze Entwicklungszeiten zu gewährleisten, ist es erforderlich, den Systementwickler in allen Phasen der Entwicklung möglichst gut methodisch und technisch durch ein passendes Werkzeug zu unterstützen.

Der Mensch kann aufgrund seiner visuellen Prägung Spezifikationen in Form von Diagrammen tendenziell besser verstehen und umsetzen als dies bei Beschreibungen in Textform der Fall ist. Aus diesem Grunde haben sich mehr und mehr graphische Beschreibungstechniken zum Entwurf von Softwaresystemen durchgesetzt. Einige hiervon (Statecharts, UML, ROOM usw.) haben wir in diesem Kapitel kennen gelernt.

Die getrennte Entwicklung von Hardware und Software führte in der Vergangenheit dazu, dass schon bei der Spezifikation und dem Entwurf berücksichtigt werden musste, welche Teile des Entwurfs letztendlich in Hardware und welche in Software realisiert werden sollen. Diese frühe Entscheidung war oft nur sub-optimal. Wir haben an dieser Stelle den Ansatz des HW/SW-Codesigns diskutiert, der hier Abhilfe schafft.

Hybride Systeme gehören zur Klasse der Echtzeitsysteme. Sie sind durch die nicht-triviale Verkopplung diskreter und kontinuierlicher Aspekte charakterisiert. Ziel des vorangegangen Abschnitts war es, dem Leser einen Einblick in das Feld der hybriden Systemem zu geben und einige wichtige Ansätze kurz vorzustellen.