4 Programmierung eingebetteter Systeme

Motivation

Die Programmierung von Software für eingebettete Systeme in Assembler war lange Zeit möglich, da die Größe der Software meist nur einige 100 Bytes umfasste. Das rapide Anwachsen der Softwareumfänge, vor allem aufgrund der stark anwachsenden Leistungsfähigkeit neuer Mikrokontroller führt dazu, dass eine kostengünstige Programmierung in Assembler nicht länger erfolgen kann. Zudem ist die geschätzte Leistung eines Programmierers in Codezeilen pro Tag bei Anwendungssoftware ca. zehnfach höher als bei Systemsoftware. Daran wird deutlich, dass heutige Softwarewerkzeuge immer noch nicht auf den Entwurf eingebetteter Systeme zugeschnitten sind.

Wie bereits diskutiert, sind viele eingebettete Systeme Echtzeitsysteme. Bei ihnen kommt es nicht nur darauf an, dass die berechneten Ergebnisse logisch korrekt sind, sondern pünktlich zu einem bestimmten Zeitpunkt, einer sogenannten Deadline, zur Verfügung stehen. Echtzeitsysteme, die noch akzeptabel funktionieren, obwohl eine Deadline geringfügig versäumt wurde, sind weiche Echtzeitsysteme. 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 (Pree et al., 2003) ist die Dekompression von Videodaten, die über ein Netzwerk geladen werden. Je langsamer die Dekompression erfolgt, desto schlechter ist die Qualität des angezeigten Videos. Bei sogenannten harten Echtzeitsystemen müssen hingegen alle Deadlines unter allen Umständen eingehalten werden, da sonst möglicherweise eine Katastrophe eintreten könnte. Beispiele für harte Echtzeitsysteme sind Flugzeugsteuerungen, oder im Automobilbereich das Motormanagement im Verbrennungsmotor, Bremsassistent oder der Auslöser eines Airbags.

Die Entwicklung von harten Echtzeitsystemen unterscheidet sich grundlegend von der von weichen Echtzeitsystemen. Ein hartes Echtzeitsystem muss unter allen möglichen Ausnahme- und Fehler-

bedingungen die zeitlichen Restriktionen einhalten. Es ist die Systemumgebung, in der ein hartes Echtzeitsystem eingesetzt wird, die vorgibt, welches exakte Zeitverhalten verlangt wird und welcher Grad an Parallelität beziehungsweise Verteilung auf Seite der Software zu beherrschen ist. Beispielsweise definieren die Charakteristika eines Motors die Anforderungen an die zugehörige Steuerungssoftware, also welche parallel ablaufenden chemischen und mechanischen Prozesse im Motor beobachtet werden müssen, in welchen Zeitabständen, z. B. vorgegeben durch die Motordrehzahl, die Messungen durch Sensoren und die Steuerungsimpulse durch Aktoren erfolgen müssen. Die Steuerungssoftware muss innerhalb der Zeitvorgaben die Berechnungen durchführen, um die richtigen Stellwerte an die Aktoren zu liefern, so dass das gewünschte Verhalten des Motors erzielbar ist. (Pree et al., 2003).

Ein derzeit besonders interessantes Forschungsthema im Bereich harter Echtzeitsysteme ist die automatische Übereinstimmung der Zeitachse der Umgebung (also der realen Welt) mit der Zeitachse des Rechners, auf dem die Software läuft. In der realen Welt ist Zeit kontinuierlich, also reellwertig, während digitale Rechner mit einem diskreten Zeitbegriff arbeiten. Da Rechnerplattformen für eingebettete Systeme zunehmend als verteilte Systeme eingesetzt werden, ist es mühsam und fehleranfällig, die Übereinstimmung der beiden Zeitachsen manuell herzustellen (Broy und Pree, 2003).

Genau diese Vorgehensweise muss aber in Ermangelung besserer Ansätze heutzutage in der Entwicklungspraxis eingebetteter Echtzeitsysteme verwendet werden. Stattdessen wäre es aber wünschenswert, eine automatische Übereinstimmung der Zeitachsen sicherzustellen. Eine tiefgehende formale Betrachtung dieser Problemstellung kann die Grundlage dafür liefern, das ungewollt nichtdeterministische zeitliche Verhalten heutiger Systeme zu bereinigen. Deterministisch bedeutet in diesem Kontext, dass das Ausgabeverhalten der Aktoren bei gegebenem Input an den Sensoren vorhersagbar ist. Das stellt eine Voraussetzung für die formale Verifikation und Implementierbarkeit der Software dar. Hier ist nach (Broy und Pree, 2003) die Entwicklung neuartiger Sprachen und Konzepte erforderlich, „die sich ausschließlich darauf konzentrieren, das gewünschte Zeitverhalten von eingebetteten Systemen auf verteilten Plattformen durch eine automatische Codeerzeugung zu gewährleisten“. Beispiele für brauchbare, aber sehr unterschiedliche Sprachansätze in diese Richtung sind Ansätze wie Esterel, Lustre, Signal und Giotto.

Kapitelübersicht

Dieses Kapitel soll zunächst eine Diskussionsbasis schaffen, inwieweit gängige Programmiersprachen wie C/C++ und Java zur Entwicklung eingebetteter Systeme geeignet sind. Der Leser soll

dann überzeugt werden, dass für die Programmierung eingebetteter Systeme im Vergleich zu gängigen Praktiken völlig neuartige Ansätze erforderlich sind. Im Folgenden wollen wir daher alternative Programmiersprachen zur Entwicklung ggf. nebenläufiger, eingebetteter Echtzeitsysteme überblicksartig betrachten. Danach werden die Sprachen Esterel und Giotto als zwei sehr unterschiedliche Vertreter für mögliche Lösungsansätze kurz vorgestellt. Eine Vermittlung tiefergehender Programmierkenntnisse ist dagegen weder möglich noch nötig.

4.1 Der Einsatz von C/C++ für eingebettete Systeme

Assembler und C im Zusammenspiel

Im Jahr 2000 wurden laut empirischen Erhebungen (Lewis, 2001) noch etwa 80 Prozent der eingebetteten Systeme in C entwickelt. Aus Performance-Gründen wird jedoch C immer noch in folgender Weise in Kombination mit Assemblersprache verwendet:

- Assemblersprache wird zur Realisierung echtzeitkritischer Anforderungen verwendet,
- C wird nach Assembler übersetzt und dann von Hand optimiert.

Gründe für die Verbreitung von C

Gründe für die immer noch weite Verbreitung von C auf diesem Gebiet sind:

- Assemblersprache ist häufig nicht mehr möglich (siehe oben), und C kommt der maschinennahen Programmierung am nächsten,
- Entwicklungszeit und -zyklen werden immer kürzer,
- die Komplexität der eingebetteten Systeme steigt,
- eingebettete Systeme erfüllen mehr und mehr sicherheitskritische Aufgaben,
- leistungsfähigere Hardware ermöglicht Hochsprachen, dennoch liefert die Programmierung in Maschinensprache schnellere und kürzere Programme (bis zu Faktor 10).

Die Anzahl der eingebetteten Systeme, die nicht mehr in C oder Assembler programmiert werden nimmt jedoch stetig zu. Dies ist u. a. mit der steigenden Systemkomplexität begründet. Durch den

Einsatz objektorientierter Hochsprachen in eingebetteten Systemen soll die gestiegene Komplexität kompensiert werden. Objektorientierte Hochsprachen ermöglichen ferner die einfachere Wiederverwendung von Code sowie eine komplett objektorientierte, teilweise automatisierte Entwicklung von der Analyse bis hin zur Realisierung. Auch um die Softwarequalität bei gleichzeitig möglichst geringen Entwicklungskosten zu steigern, werden immer mehr objektorientierte Programmiersprachen wie C++ oder auch Java verwendet. C++ ist wegen der vielen C-Sprachelemente und der damit verbundenen leichteren Eingewöhnung der Entwickler sehr populär. Embedded C++, kurz EC++, soll die Entwicklung erleichtern und zugleich die Kompatibilität zu gängigen Standards sicherstellen. Aber auch Java gewinnt nicht zuletzt wegen der großen Entwicklergemeinde und den verschiedenen direkt für eingebettete Systeme zugeschnittenen Varianten, an Bedeutung.

Sowohl Embedded C++ als auch die Java 2 Micro Edition (J2ME) stellen objektorientierte Programmiersprachen zur Verfügung, die hinsichtlich ihres Sprachumfangs im Vergleich zu C++ bzw. Java eingeschränkt wurden. So wurde z. B. bei EC++ die Mehrfachvererbung bzw. bei der CLDC (Connected Limited Device Configuration) der J2ME die Gleitkommaarithmetik entfernt. Ebenso sind viele Standard-APIs meist nicht vollständig verwendbar.

4.2 Embedded C++

Sprachumfang von EC++

Im Vergleich zu C enthält C++ einige Sprachumfänge, die den Einsatz von C++ zur Programmierung eingebetteter Systeme behindern, so z. B. die Ausnahmebehandlung mittels Exceptions sowie das Konzept der Mehrfachvererbung, die selbst dann schon Code Overhead generieren, wenn die Konzepte im konkreten Programm gar nicht verwendet werden. Aber auch Templates oder die Verwendung mancher Bibliotheksfunktionen wie etwa für Ein- und Ausgabe von Daten kann bei weitem mehr Code erzeugen, als dies vor dem Hintergrund der restriktiven Speicherplatzanforderungen eingebetteter Systeme vertreten werden könnte. Diesen Nachteilen von C++ tritt die Entwicklung von Embedded C++ (EC++) entgegen.

Ziel von EC++

Ziel von EC++ ist es, einen offenen Standard für die Entwicklung kommerzieller eingebetteter Systeme zu definieren (Plauger, 1999). Entwicklern von eingebetteten Systemen wird eine Untermenge von C++ geboten, die für C-Programmierer leicht verständlich und damit nutzbar ist. Diese Untermenge ist aufwärtskompatibel zur vollen

Version von Standard C++ und partizipiert daher an zahlreichen Vorteilen von C++. Insbesondere existiert ein sogenannter Styleguide, der die Programmierung eingebetteter Systeme mit Embedded C++ erleichtert. Embedded C++ wurde auf die speziellen Bedürfnisse bei der Programmierung eingebetteter Systeme konfektioniert. Die Entwicklung des Standards wurde durch das sogenannte „Embedded C++ Technical Comittee“ vorangetrieben. Diesem 1995 in Japan gegründeten Komitee gehören namhafte, vor allem japanische Unternehmen an. Unter www.caravan.net/ec2plus/ ist der Internetauftritt des Komitees zu finden.

Rahmenbedingungen von EC++

Folgende Rahmenbedingungen für die Auswahl einer geeigneten Untermenge von C++ hat dieses Komitee zur Definition von EC++ definiert:

- Die Spezifikation sollte so klein wie möglich sein, ohne die objektorientierten Elemente auszuschließen.
- Es sollten solche Funktionen und Spezifikationen vermieden werden, die nicht die Erfordernisse der Entwicklung eingebetteter Systeme erfüllen. Die drei Haupterfordernisse eingebetteter Systeme sind nach Ansicht des Komitees:
  - Die Vermeidung von exzessivem Speicherverbrauch
  - Keine unvorhersagbaren Reaktionen zu generieren
  - Die Lauffähigkeit des Codes im ROM zu garantieren
- Nichtstandard-Erweiterungen von C++ dürfen nicht in die Untermenge einfließen.
- Zielplattform der mit EC++ programmierten Anwendungen sind 32 Bit RISC MCUs (RISC = Reduced Instruction Set Computer; MCU = Micro Controller Units). Für Anwendungen mit 4 bzw. 8 Bit MCUs stellen nach Meinung des Komitees die Sprache C sowie Assemblersprachen eine ausreichende Umgebung dar.

Embedded C++ ist daher keine neue Sprachspezifikation die eine Konkurrenz zum existierenden C++ Standard darstellen würde, sondern vielmehr eine – bis auf eine Ausnahme – Untermenge von C++. Die Untermengen-Bedingung von EC++ zu C++ wird im folgenden Punkt aufgeweicht: Die speziellen EC++ Datentypen “float_complex” und “double_complex” besitzen kein Pendant im offiziellen C++ Standard.

4.2.1 Einschränkung: Das Schlüsselwort „mutable“

Objekte einer als „const“ deklarierten Klasse werden bei ihrer Realisierung in eingebetteten Systemen üblicherweise im ROM (Read Only Memory) abgelegt. Klassenelemente die als „mutable“ deklariert wurden, können auch dann geändert werden, wenn das Objekt selbst „const“ ist. Folglich können Klassen, welche ein „mutable“-Element besitzen, nicht länger im ROM gespeichert werden. Das technische Komitee hat deshalb entschieden, das „mutable“-Schlüsselwort nicht in die EC++ Spezifikation aufzunehmen.

4.2.2 Einschränkung: Ausnahmebehandlung

Ausnahmebehandlungen (engl. exception handling) sind zur kontrollierten Behandlung von Laufzeitfehlern von Vorteil, bergen jedoch einige Fallstricke für die Programmierer. So erweist es sich zunächst als schwierig, die Zeit, die zwischen dem Auftreten der Ausnahme und ihrer Weitergabe an die Behandlungsroutine vergeht, abzuschätzen. Wenn die Ablaufkontrolle vom Auftrittspunkt zur Behandlungsroutine übergeht, werden die Destruktoren aller Objekte aufgerufen, die seit Betreten des „try“-Blockes automatisch erzeugt wurden. Daher ist die Zeit, die zur Zerstörung der automatisch erzeugten Objekte benötigt wird, nur schwer abzuschätzen. Ausnahme-Mechanismen erfordern Compiler-generierte Datenstrukturen sowie eine Laufzeitunterstützung. Dies kann zu einem unerwarteten Anwachsen der Programmgröße führen. Der genaue Speicherverbrauch zur Abwicklung der Ausnahmebehandlung kann daher nicht exakt bestimmt werden.

Für Entwickler von eingebetteten Systemen ist es wichtig, die Ausführungszeit ihrer Programme zu kennen. Ebenso muss die erzeugte Programmgröße so klein wie möglich sein. Deshalb können die oben genannten Nachteile nicht ignoriert werden.

Aus diesen Gründen hat sich das technische Komitee dazu entschieden, die Ausnahmebehandlung nicht in den EC++ Standard aufzunehmen. Ausnahmebibliotheken werden folglich in der Standard EC++ Bibliothek nicht unterstützt.

4.2.3 Typidentifikation zur Laufzeit

RTTI

Um die Typidentifikation zur Laufzeit (engl. Run-Time-Type Identification, kurz: RTTI) zu unterstützen, wird ebenfalls Code Overhead erzeugt, da Typinformationen für polymorphe Klassen benötigt werden. Der Compiler erzeugt diese Informationen automatisch – selbst dann, wenn ein Programm RTTI gar nicht benutzt. Das technische Komitee hat deshalb RTTI nicht in die EC++ Spezifikation aufgenommen.

4.2.4 Namenskonflikte

Namespaces

Aufgrund der vergleichsweise geringen Dimensionierung des Arbeitsspeichers typischer Zielprozessoren von EC++ Programmen ist der Codeumfang dieser Programme stark limitiert. Aus diesem Grund treten Namenskonflikte selten auf und können ggf. durch die Verwendung statischer Klassenelemente vermieden werden. Die Funktionalität dedizierter Namensräume (engl. namespaces) ist daher nicht relevant für die EC++ Spezifikation und wurde deshalb nicht in sie aufgenommen.

4.2.5 Templates

Templates werden zum Erzeugen generischer Funktionen und Klassen verwendet, erzeugen jedoch wiederum Code Overhead. Da es durch die unachtsame Verwendung von Templates zu regelrechten „Code Explosionen“ kommen kann, hat sich das technische Komitee dazu entschlossen, sie nicht in die EC++ Spezifikation mit aufzunehmen.

4.2.6 Mehrfachvererbung und virtuelle Vererbung

Programme mit Mehrfachvererbung tendieren dazu, weniger lesbar, weniger wiederverwendbar und schwer wartbar zu sein. In den EC++ Standard wurde Mehrfachvererbung deshalb nicht aufgenommen. Da die virtuelle Vererbung (Spezifikationsvererbung,

Schnittstellenvererbung, engl. interface inheritance) nur in Kombination mit der Mehrfachvererbung vorkommt, ist auch sie kein Teil des Standards.

4.2.7 Bibliotheken

STL

Da Templates in EC++ aus vorgenannten Gründen keine Beachtung gefunden haben, werden auch Bibliotheken wie etwa die STL (Standard Template Library), welche Templates benutzen, nicht unterstützt. Einige aus der STL bekannten Klassen werden allerdings als Nicht-Template-Klassen unterstützt. Dazu zählen die Klassen „string“, „complex“, „ios“, „streambuf“, „istream“ und „ostream“; sie sind im EC++ Standard enthalten.

Die Bibliotheken für die Typen „wchar_t“ oder „long double“ werden im Bereich eingebetteter Systeme sehr selten benutzt. Ihre Aufnahme in den EC++ Standard wurde deshalb vom Komitee abgelehnt.

Dateioperationen

Bibliotheken für Dateioperationen sind abhängig vom Betriebssystem und werden daher nicht unterstützt.

Internationalisierung

Internationalisierungs-Bibliotheken benötigen einerseits viel Speicher und sind andererseits für die meisten Embedded Anwendungen nutzlos. Deshalb sind sie nicht Teil der EC++ Spezifikation.

4.2.8 EC++ Styleguide

Teil A

Der Teil A „Migrating from C Language to C++ Language“ des Styleguides beschäftigt sich mit Regeln zur Migration von C nach Embedded C++. Dieser Teil soll zum einen C-Programmierern den Umstieg von C nach EC++ erleichtern, zum anderen das Portieren bereits vorhandener C-Programme durch die dort dargestellten Migrationsrichtlinien beschleunigen.

Teil B

In Teil B „Guidelines for Code Size“ des Styleguides werden Hinweise gegeben, welche Sprachmittel man einsetzen sollte, damit der durch den Compiler generierte Bytecode möglichst klein ist.

Teil C

Teil C „Guidelines for Speed” erläutert Programmiertechniken, die eingesetzt werden sollten, um Code mit möglichst hoher Laufzeit- und Speichereffizienz zu generieren.

Teil D („Guidelines for ROMable Code“) zeigt, welche Programmkonstrukte verwendet werden dürfen, um Objekte im ROM ablegen zu können.

Teil D

4.3 Der Einsatz von Java für eingebettete Systeme

Java für die Programmierung eingebetteter Systeme einzusetzen mag zunächst als ein Widerspruch erscheinen, da es sich bei Java um eine Plattform-unabhängige, sehr aufwändige, objektorientierte Sprache handelt, die zunächst ihre Anwendung im Internetbereich fand und daher meist interpretiert wird. Hierfür kommt die Java Virtual Machine (JVM) zum Einsatz, eine registerlose und damit vergleichsweise langsame, virtuelle Stack-Maschine. Insgesamt wird eine Performanz von nur ca. 5–10% eines Native Compilers erreicht.

Geschwindigkeitseinbußen

Weitere Geschwindigkeitseinbußen gibt es bei Java u. a. auch wegen folgender Eigenschaften:

- Keine Pointer
- Viele Laufzeitüberprüfungen
- Sicherheitsüberprüfung des Bytecodes
- Synchronisation
- Symbolische Namensauflösung
- Langsame Array-Initialisierung
- Automatische Speicherverwaltung (Garbage Collection)

Nachteile

Weitere Nachteile für die Verwendung von Standard-Java im Bereich eingebetteter Systeme sind:

- Es bietet für den Umgang mit begrenzten Ressourcen wenig Unterstützung.
- Es verbraucht viel Speicherplatz (allein die Laufzeitumgebung benötigt schon 26 MByte).
- Hardwarezugriffe sind kaum möglich.
- Mannigfaltige, evtl. sogar unbekannte Ein- bzw. Ausgabeschnittstellen müssten angeboten werden.
- Es ist nicht echtzeitfähig.

Hindernisse für Echtzeitanwendungen

Im Umfeld von Echtzeitsystemen wurde man auf Vorteile von Java wie Sicherheit, Portabilität und die Unterstützung von Multithreading aufmerksam. Folgende Punkte sind aber hinderlich für den Einsatz von Java in Echtzeitanwendungen:

- **Schlechte Performance:** Java ist eine interpretierte Sprache; u. a. entstehen auch Performanceeinbussen bedingt durch die Virtuelle Maschine
- **Nichtdeterministisches Verhalten:** Java bietet keine garantierten Antwortzeiten. Zusätzlich besitzt Java einen Garbage Collection Mechanismus, der in undefinierten Zeitabständen unreferenzierte Objekte (wieder) freigibt. Dieser Mechanismus unterbricht die Programmausführung in unvorhersagbaren Zeitabständen.
- **Scheduling und Synchronisation:** Die Semantik der Synchronisation und des Schedulers in Java genügt den Anforderungen von Echtzeitsystemen nicht. In Echtzeitsystem muss bekannt sein, wenn/wann ein Task gestartet und beendet wird.
- **Speicherverwaltung:** In Java ist kein direkter Zugriff auf den Speicher möglich, alle Objekte werden auf dem Heap gespeichert und die Garbage Collection ist zuständig für das Abbauen von nicht mehr gebrauchten Objekten. Daher hat man keine Möglichkeit, den Lebenszyklus von Objekten zu kontrollieren.

Historie

Dennoch wurde Java ursprünglich für den Einsatz in eingebetteten Systemen entwickelt. 1991 wurde bei Sun von Patrick Naughton, Mike Sheridan und James Gosling das „Green Project“ gestartet. Ziel des Projekts war das Aufspüren der nächsten Trendwelle der Computertechnik. Aufgrund ihrer Vermutung, dass digitale Konsumgeräte und Computer immer mehr zusammenwachsen würden, wurde StarSeven (*7) entwickelt. Diese Multimedia-Fernbedienung für Unterhaltungselektronik war in der Lage, ein umfangreiches Spektrum an Plattformen zu steuern. Die Grundlage dafür bildete eine Programmiersprache namens „Oak“, die später in Java umbenannt wurde.

Obwohl Java also ursprünglich für eingebettete Systeme entwickelt wurde, setzte es sich vor allem wegen seiner flexiblen Verwendung auf unterschiedlichsten Plattformen, Netzwerkanbindungen und dem Konzept der Objektorientierung schnell auf Desktops, Servern und als Applets im Internet durch. Dennoch ist Java für eingebettete Systeme durch speziell für die Anforderungen dieser Systeme ausgerichtete Versionen interessant. Sie ermöglichen es, viele Java Vorteile auch auf *mobilen Systemen* mit geringen Ressourcen zu nutzen.

4.3.1 Java 1

In Java1 existieren zur Anwendungsentwicklung für eingebettete Systeme zwei Plattformen: Personal Java für leistungsstarke und Embedded Java für schwächere Systeme. Allerdings werden beide von Sun im Rahmen des „Sun End of Life (EOL) Process“ nicht mehr weiterentwickelt da neuere Produkte (J2ME) existieren. Online-Dokumentationen sind im Internet unter folgenden Adressen zu finden:

java.sun.com/products/personaljava
java.sun.com/products/imp/overview.html

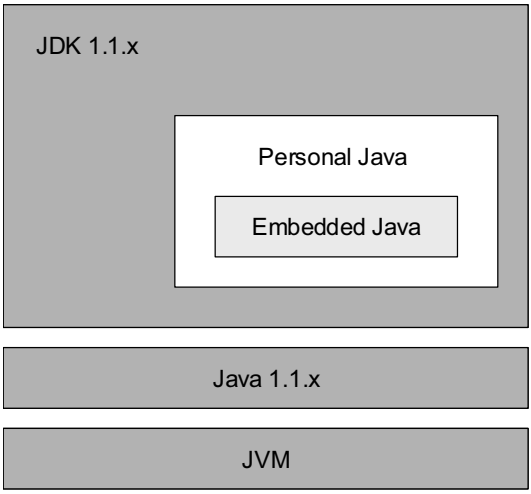


Abb. 4.1: Aufbau von Java 1

4.3.1.1 Personal Java

GUIs, Netzwerke

Diese Java Plattform ermöglicht die Entwicklung von Anwendungen mit graphischer Benutzerschnittstelle (java.awt) und Netzwerkverbindungen. Die Personal Java API (Application Programming Interface) ist von der JDK 1.1 API abgeleitet und nur um wenige neue APIs ergänzt. Sie umfasst eine virtuelle Maschine (z. B. Standard VM) und eine optimierte Version der Java Klassenbibliothek. Um Ressourcen auf den Zielgeräten zu sparen, ist Personal Java auf drei Ebenen konfigurierbar. Es ist möglich auf Paket-, Klassen- und Methodenebene genau die benötigten Elemente auszuwählen. Das frei verfügbare Entwicklungswerkzeug JavaCheck von Sun ermög-

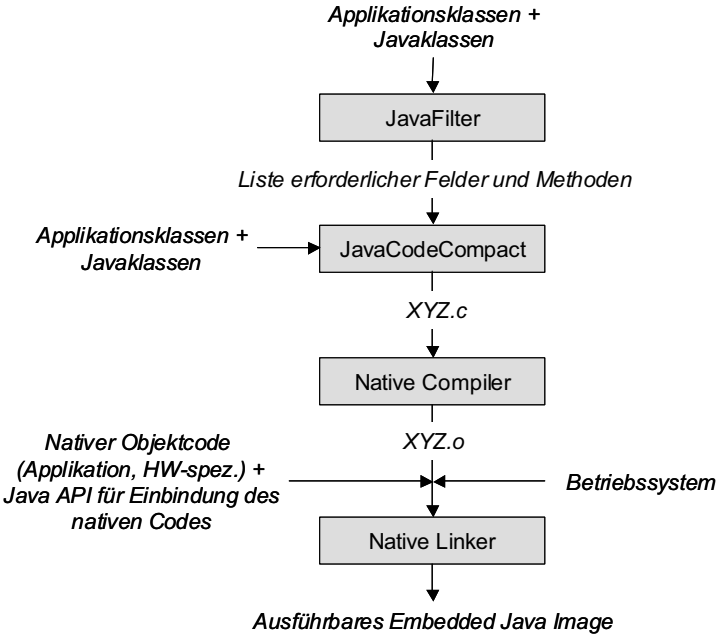
licht eine Überprüfung, ob eine Anwendung bzw. ein Applet auf einer Personal Java Plattform lauffähig ist.

4.3.1.2 Embedded Java

Beschränkte Ressourcen

Embedded Java ist für Geräte gedacht, deren Ressourcen für Personal Java nicht ausreichen. So werden z. B. keine Core APIs benötigt, sondern man kann je nach Anwendungsgebiet konfigurieren welche APIs verwendet werden sollen. Damit kann man fast alle APIs des JDK 1.1.7 bei minimaler Applikationsgröße verwenden. Allerdings ist der Entwicklungsprozess mit den von Sun zur Verfügung gestellten Werkzeugen aufwändiger als bei Personal Java, vgl. Abbildung 4.2.

Abb. 4.2: Entwicklungsprozess mit Embedded Java, Java 1



Optimierung

Nach der Entwicklung der Anwendung in Java wird mit dem JavaFilter Werkzeug eine Liste mit allen verwendeten Feldern, Klassen und Methoden erstellt. Diese Liste wird durch JavaCodeCompact weiter optimiert (beispielsweise werden nicht verwendete Codeteile entfernt) und in Datenstrukturen der Sprache C umgewandelt. Diese können dann mit einem C-Compiler für das jeweilige System in Objekt-Dateien übersetzt werden. Mit

JavaDataCompact ist es möglich, benötigte Daten wie Bilder oder HTML-Dateien umzuwandeln, um diese dann zu den Objekt-Dateien zu verlinken, falls auf dem Zielgerät kein Dateisystem verfügbar ist. Die so entstandenen Embedded Java Images können auf dem jeweiligen Gerät ausgeführt werden.

4.3.2 Java 2 (J2ME)

J2EE, J2SE, J2ME

Mit Java2 wurde die Java Technologie in drei Teilbereiche aufgeteilt: Java 2 Enterprise Edition (J2EE), Java 2 Standard Edition (J2SE) und Java 2 Micro Edition (J2ME). In letzterer greift Sun Personal Java und Embedded Java auf, um Anwendungen dieser Plattformen kompatibel zu halten. Jedoch wurden nicht monolithische Plattformen geschaffen, die dann umfangreich je nach Anwendung mühsam konfiguriert werden müssten, sondern modulare Konfigurationen und Profile. Diese bieten dann für verschiedene Anwendungsbereiche definierte APIs.

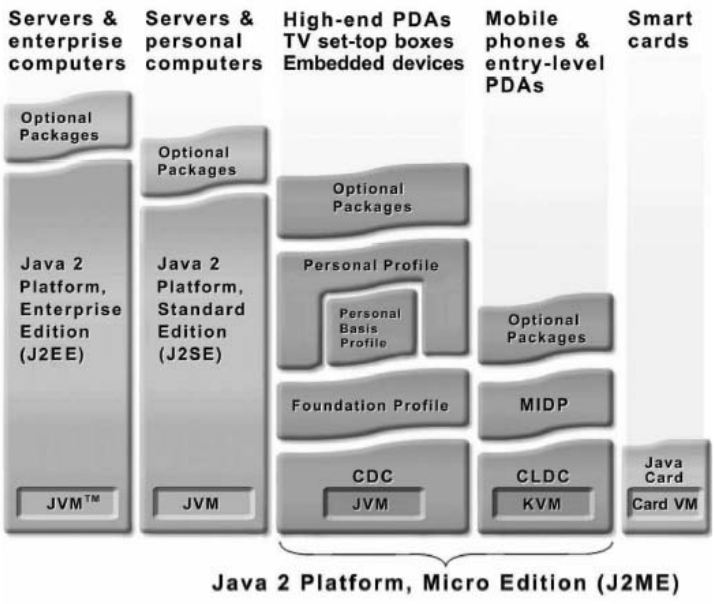


Abb. 4.3: Java 2 Architektur (Quelle: Sun)

Konfigurationen, Profile

Hierbei beschreiben *Konfigurationen* die Grundfunktionen der jeweiligen virtuellen Maschine (VM) für eine Klasse von Geräten mit ähnlicher Leistungsfähigkeit; vgl. Abbildung 4.3. Eine Konfiguration umfasst eine VM, Klassenbibliotheken und (minimale) APIs, die auf die jeweiligen Geräte zugeschnitten sind und stellt damit eine funktionstaugliche Laufzeitumgebung dar. *Profile* setzen auf den Konfigurationen auf und stellen weitere APIs für bestimmte Klassen von Zielgeräten zur Verfügung. Sie können ineinander verschachtelt sein oder aufeinander aufbauen. Allerdings können auch optionale Pakete wie z. B. das „RMI Optional Package“ oder das „JDBC Optional Package“ (beide für CDC) eingebunden werden.

4.3.2.1 Connected Device Configuration (CDC)

Die Connected Device Configuration umfasst eine voll funktionsfähige VM der Java 2 Plattform – die CDC HotSpot Implementation oder auch kurz CVM genannt – sowie eine kleine Anzahl von Klassenbibliotheken und APIs. Erst mit einem darauf aufbauenden Profil ist es möglich, eine Applikation zu entwickeln. Für diese Profile gibt es zusätzliche optionale Pakete, die weitere spezielle Funktionen bereitstellen. Die CDC kommt häufig in Geräten mit einem 32 Bit Prozessor zum Einsatz. Sie benötigt ca. 2 MByte RAM und in etwa 2,5 MByte ROM für die Java Umgebung. Zurzeit existieren die folgenden drei Profile für die CDC.

Foundation Profile

(1) Foundation Profile:
Das Foundation Profile setzt auf dem CDC auf. Es dient als Basis für weitere Profile und stellt deshalb keine Klassen für das Benutzerinterface zur Verfügung. Es dient im Wesentlichen dazu, die Pakete des CDC konzeptuell verfügbar zu machen. Es umfasst Socket-Klassen und ermöglicht Internalization und Localization. Außerdem enthält es Klassen aus den Paketen java.lang, java.lang.ref, java.lang.reflect, java.net, java.security, java.text, java.util, java.util.jar und javax.microedition.io. Eine graphische Benutzeroberfläche (GUI) wird nicht unterstützt. Meist wird dieses Profil nicht unmittelbar selbst verwendet, sondern dient lediglich als Grundlage für weitere Profile.

Personal Basis Profile

(2) Personal Basis Profile:
Dieses Profil baut auf dem Foundation Profile auf und erweitert dieses um die sogenannten „lightweight“ (leichtgewichtigen, schlanken) Komponenten aus dem Paket java.awt. Dies bedeutet, dass eine einzige Instanz von java.awt.Frame erlaubt ist, um

Komponenten aufzunehmen. Im Gegensatz dazu sind „heavyweight“ GUI-Komponenten wie etwa java.awt.Button oder java.awt.Panel nicht vorhanden. Außerdem stehen javax.microedition.xlet sowie javax.microedition.xlet.ixc für xlet-Anwendungen zur Verfügung. Xlets sind von den Java TV APIs abgeleitete Anwendungen, deren Lebenszyklus genau wie der von Applets von der Software, die sie aufgerufen hat, kontrolliert wird.

Personal Profile

(3) Personal Profile:
Dieses Profil schließt das Personal Basis Profile als Submenge (damit folglich auch das Foundation Profile) ein und beinhaltet im Gegensatz zum Personal Basis Pofile das Paket java.awt komplett. Außerdem können Anwendungen, die unter Personal Java (siehe Java1) entwickelt wurden, in dieses Profil überführt werden. Es eignet sich insbesondere für die Softwareentwicklung für PDAs (Personal Digital Assistants).

4.3.2.2 Connected Limited Device Configuration (CLDC)

Mobiltelefon, PDA

Diese Konfiguration wurde speziell für Geräte mit wenig Speicher (zwischen 128 und 512 KByte) und 16 bzw. 32 Bit Prozessoren wie beispielsweise Mobiltelefone, Pager oder kleine PDAs entwickelt. Sie stellt eine Untermenge des CDC (siehe oben) dar. CLDC Programme sind folglich auch auf CDC Plattformen lauffähig, falls die erforderlichen Profile zur Verfügung stehen.

KVM

Die CLDC umfasst eine VM, oft handelt es sich hierbei um die KVM, sowie grundlegende Klassenbibliotheken. Das „K“ der KVM ist hierbei als Abkürzung für „Kilobyte“ zu interpretieren, weil der Speicherplatzbedarf dieser virtuellen Maschine sehr gering ist. Bereits ab 70 KByte sind ausreichend. Datentypen wie „float“ oder „double“ werden nicht unterstützt, da die Prozessoren der Zielplattformen oft keine Gleitkomma-Arithmetik zur Verfügung stellen. Anstelle von java.net steht das Generic Connection Framework zur Verfügung. Mit ihm ist es Herstellern eingebetteter Systeme möglich, Geräte auf die KVM zu portieren und weitere Verbindungsprotokolle wie z. B. Bluetooth oder Irda zu unterstützen. Alle Klassen-Dateien müssen, bevor sie auf der KVM ausgeführt werden können, durch einen Pre-Verifier auf dem Desktop geprüft werden. Neben einigen optionalen Paketen für die CLDC wie beispielsweise die Wireless Messaging API (WMA) oder die Mobile Media API (MMAPI) gibt es zurzeit die beiden folgenden Profile.

MIDP

Mobile Information Device Profile (MIDP):
Dieses Profil ist für mobile Geräte wie Mobiltelefone oder Pager ausgelegt und enthält Pakete für den Zugriff auf persistenten Speicher sowie zur Erstellung von Benutzerschnittstellen. Da letztere geräteunabhängig in Form sogenannter MIDP-Programme (oft auch kurz MIDlets) beschrieben werden und kein konkretes, gerätespezifisches Layout festgelegt wird, können Darstellungsunterschiede auf den Zielgeräten die Folge sein. So können beispielsweise Displayauflösung oder Farbtiefe variieren. Detailinformationen zum MIDP sind im Internet unter java.sun.com/products/midp/overview.html zu finden.

IMP

Information Module Profile (IMP):
Dieses Profil enthält eine Submenge des MIDP. Es wurde im Rahmen des Java Community Prozesses (JCP) im Wesentlichen von den Firmen Siemens und Nokia entwickelt. Die zugehörigen APIs beinhalten Klassen zur Erstellung von „IMLets“ (IMP-Programme), zur Kommunikation und zur Ein-/Ausgabe. Damit ist z. B. durch die unterstützte Untermenge des HTTP (sowohl TCP/IP als auch nicht IP basiert) ein Zugriff auf WAP- (Wireless Access Protocol) oder iMode- (Interactive Media on Demand) Informationen möglich. Das MIP ist vor allem für Geräte, die über keine graphische Benutzeroberfläche verfügen bzw. nur vergleichsweise kleine Displays bieten gedacht (Beispiele: Notrufsäulen, Parkuhren, Alarmsysteme, Ferndiagnose- und Fernwartungssysteme). Weiterführende Informationen können unter java.sun.com/products/imp/overview.html detailliert nachgelesen werden.

4.3.3 JavaCard

Smart Cards

Smart Cards sind ein fester Bestandteil unseres Lebens geworden. Sie werden z. B. als SIM-Karten (SIM = Subscriber Identity Module) in Mobiltelefonen oder als Zugangskarten für gesicherte Bereiche eingesetzt. Eine JavaCard ist eine Smart Card auf der Java-Programme laufen können. JavaCards unterliegen demnach der Standardisierung Smart Cards nach der ISO/IEC 7816 Norm.

ISO/IEC 7816

Diese Norm legt die physikalischen Eigenschaften von Chipkarten und Protokollen für die Kommunikation mit dem Kartenleser fest. Er besteht aus den Teilen ISO/IEC 7816-1 bis ISO/IEC 7816-10. Dort sind unter anderem Anordnung und Größe der Kontakte

(ISO/IEC 7816-2) oder Material und Format der Karte (ISO/IEC 7816-2) standardisiert.

Software

Bei der auf der Karte laufenden Software unterliegen die Hersteller allerdings nur sehr wenigen Einschränkungen. Dies führt neben anderen Nachteilen zu einer großen Vielfalt an Karten, die meist untereinander so gut wie nicht kompatibel sind. Die Entwicklungskosten sind aufgrund nicht portabler Implementierungen vergleichsweise hoch; meist ist eine Anwendung sogar nur auf einer bestimmter Hardware-Version einer Karte lauffähig. Dies birgt den weiteren Nachteil, dass Entwickler sehr detaillierte Kenntnisse über die verwendete Hardware besitzen müssen. Oft ist bereits ein Update der Software im Nachhinein schon nicht mehr möglich.

JavaCard

Die JavaCard Technologie bietet hier viele Vorteile bei der Entwicklung und dem Einsatz von Smart Cards.

Smart Card Typen

Man unterschiedet im Wesentlichen zwei Gruppen von Smart Cards. Karten die nur zum Speichern von Daten verwendet werden (sogenannte Memory Chip Cards) und Karten mit eigenem Mikroprozessor. Erstgenannte verfügen nur über eine sehr eingeschränkte Menge vordefinierter Funktionen, die sich im Wesentlichen auf das Speichern und Verarbeiten der Daten beschränken. Mikroprozessorkarten beinhalten eine CPU sowie mehrere Arten von Speicher: ROM (Read-Only Memory) für Betriebssystem und Programme, EEPROM (Electrically Erasable Programmable Read-Only Memory) für Daten und Programme und RAM (Random Access Memory) als Arbeitsspeicher. Die Größe des Speichers reicht von wenigen Bytes bis hin zu mehreren MByte und wird meist genauso wie die Leistungsfähigkeit der CPU nur durch finanzielle Vorgaben eingeschränkt. Da Smart Cards über keine eigene Stromversorgung verfügen, beziehen sie die benötigte Energie von den Lesegeräten. Die Kommunikation kann hierbei sowohl drahtlos als auch über Kontakte erfolgen.

Hardware Minimalanforderungen

Die Minimalanforderungen von JavaCard an die Hardware sind mindestens 16 KByte ROM, 8 KByte EEPROM sowie 256 KByte RAM (Quelle: www.javawold.com). Die Architektur dieser Karten kann man wie in Abbildung 4.4 (aus www.javaworld.com) dargestellt in fünf Schichten einteilen.

Architektur

Wie Abbildung 4.4 zeigt, setzt die JavaCard VM auf einer speziellen CPU und deren Betriebssystem auf und abstrahiert von diesen, so dass der Zugriff der Applets auf die Hardware nur über die VM möglich ist. Die Aktivierungsdauer der VM ist mit jener der Smart Card selbst identisch. Falls die Smart Card nicht mit Strom versorgt wird, wird die VM nur temporär angehalten. Das JavaCard

Framework und andere (optionale) Klassen (sogenannte Industry Add-on Classes) stellen den Applets zusätzliche Funktionen zur

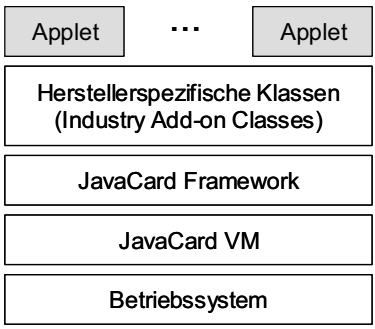


Abb. 4.4: Architektur JavaCard 2.0

Verfügung. Das JavaCard Framework ab 2.0 beinhaltet im Wesentlichen:

- javacard.framework: definiert Klassen wie Applet, PIN, System, Util und APDU (Application Protocol Data Units)
- javacardx.framework: objektorientiertes Design für ein ISO/IEC 7816-4 kompatibles Dateisystem. Es unterstützt „Elementary Files“ (EF), „Dedicated Files“ (DF) und dateiorientierte ADPUs.
- javacardx.crypto und javacard.cryptoEnc: zur Verschlüsselung von Daten

JavaCard Applets

JavaCard Applets:
JavaCard Applets sind Instanzen von Klassen, die von der Klasse javacard.framework.Applet abgeleitet wurden. Da mehrere Applikationen auf eine Smart Card geladen werden können, ist jedes Applet eindeutig durch seine *AID* (Application Identifier, ISO/IEC 7816-5) gekennzeichnet. Die vier Methoden *install*, *select*, *deselect* und *process* stellen die Schnittstelle von JavaCard Applets dar. Aus Sicherheitsgründen steht für Applets eine Sandbox zur Verfügung. Daten können je nach Bedarf in flüchtigen Objekten im RAM oder dauerhaft im EEPROM gespeichert werden.

Für die Kommunikation zwischen der Smart Card und der Lesestation (engl. Card Acceptance Device, kurz CAD) wird ein Protokoll benötigt. Smart Card und CAD verwenden zu diesem Zweck APDUs (Application Protocol Data Units). Hier wird grundsätzlich zwischen Befehlen (Command APDU) und Antworten (Response APDU) unterschieden. Dabei übernimmt die Smart Card die Rolle des passiven Kommunikationspartners. Sie wartet auf den Empfang

einer APDU vom CAD, verarbeitet diese und sendet dem CAD eine Antwort zurück.

Der Byte-Code, der nach seiner Übersetzung aus Java-Code im „Class File Format“ vorliegt, ist in diesem Format allerdings nicht direkt auf die Karte ladbar. Dazu muss der Byte-Code zunächst in das sogenannte CAP-Format konvertiert werden. Neben der Übersetzung nimmt der Konverter auch eine digitale Signierung der Datei vor. Dies verhindert das Umgehen von Sicherheitsüberprüfungen der VM durch manipulierte Applets.

Da viele JavaCard-Hersteller auf proprietäre CAP-Formate zurückgreifen, ist eine Kartenapplikation nach der Überführung in das CAP-Format eines Kartenherstellers auf Karten anderer Hersteller nicht mehr ladbar.

Zusammenfassung JavaCard

Zusammenfassung:
Bei herkömmlichen Smart Cards sind Betriebssystem und Anwendungen nicht klar getrennt. Daher sind die Chipkarten der unterschiedlichen Hersteller untereinander nicht kompatibel. Mit Hilfe von JavaCard entwickelte Anwendungen sind Plattformunabhängig; sie sind auf jeder Smart Card, die über eine VM verfügt, lauffähig. Dank der Multiapplikationsfähigkeit beschränkt nur der Speicherplatz auf der Karte die Anzahl der installierten Programme. Applikationen können, anders als bei herkömmlichen Karten, nachträglich installiert und auf den neuesten Stand gebracht bzw. auch ggf. wieder deinstalliert werden.

Allerdings können JavaCard Anwendungen, was die Geschwindigkeit und Leistungsfähigkeit der Hardware betrifft, nicht mit herkömmlichen Smart Cards Applikationen konkurrieren. Dies macht sich vor allem bei den für Smart Cards relevanten, teilweise rechenintensiven kryptographischen Algorithmen bemerkbar. Vor dem Hintergrund der stetig anwachsenden Rechenleistung der Chipkarten-Prozessoren, ist hier zukünftig jedoch mit einer Entschärfung der Situation zu rechnen. Allerdings schränkt derzeit das Herstellerspezifische CAP-Formart die Portabilität der Anwendungen auf Byte Code Ebene immer noch ein.

4.3.4 Echtzeiterweiterungen für Java

Eine Echtzeitanwendung unterscheidet sich von einer herkömmlichen durch die zusätzlichen Anforderungen an ihr zeitlichen Verhaltens. Dies bedeutet, dass Abläufe zeitlichen Einschränkungen bzgl. ihres Start- und Endzeitpunktes unterliegen. Da bei der Ent-

wicklung von Java auf solche Aspekte nur sehr wenig Rücksicht genommen wurde, eignet es sich (ohne zusätzliche Erweiterungen) nicht für Echtzeitanwendungen. Oft werden deshalb zeitkritische Komponenten immer noch in C oder C++ geschrieben.

Echtzeit-anforderungen

Wie bereits diskutiert, unterscheidet man aufgrund ihres Einsatzgebietes und der zeitlichen Anforderungen zwischen „harten Echtzeitanwendungen“ und „weichen Echtzeitanwendungen“. Bei ersteren muss a priori analysiert und nachgewiesen werden, dass alle Echtzeitanforderungen eingehalten werden können. Dies ist einerseits sehr aufwändig und erfordert andererseits detaillierte Kenntnisse über die verwendete Hardware (z. B. Cachestrategien, Pipelining usw.). Bei weichen Echtzeitanwendungen genügt es in den meisten Fällen, das zeitliche Verhalten empirisch zu bestimmen. Sämtliche bis dato bekannten Java-Echtzeitvarianten eignen sich lediglich für weiche Echtzeitanwendungen. Derzeit wird Java im Bereich der eingebetteten Systeme daher überwiegend in Multimediakomponenten bzw. mobilen Geräten (Mobiltelefone, PDAs usw.) eingesetzt. Hier findet vor allem die KVM (Kilobyte Virtual Machine) der J2ME (Java 2 Micro Edition) Anwendung (siehe oben).

Ansätze zum Einsatz von Java für Echtzeitsysteme

Mittlerweile gibt es bereits mehrere Ansätze, Java für den Einsatz in echtzeitkritischen eingebetteten Systemen aufzubereiten. Verbesserungsmöglichkeiten, die hierfür in Betracht kommen sind:

- Entwicklung einer Echtzeitspezifikation für Java – *RTSJ* (real-time specification for Java) definiert Standards für echtzeitfähiges Java (siehe www.rtj.org)
- Der Einsatz spezieller, leistungsoptimierter, *echtzeitfähiger JVMs* (Beispiel: Die JamaicaVM des AJACS Projektkonsortiums mit Beteiligung der Universität Karlsruhe (TH), siehe www.ajacs.de)
- Verwendung sogenannter *Just-in-Time Compiler,* die zur Laufzeit jeweils genau den Teil des Java Byte Codse in die Maschinensprache des jeweiligen Zielrechners übersetzen, der gerade gebraucht wird
- Verwendung des *JNI* (Java Native Interface) zur Einbindung sogenannter nativer Methoden (etwa aus C/C++); derzeit aber noch leistungsschwach

Um die Unzulänglichkeiten von Java im Zusammenhang mit der Echtzeit-Verarbeitung zu lösen, fand sich im Juni 1998 unter der Federführung des NIST (National Institute of Standards and Technology) eine Gruppe zusammen, die Anforderungen für ein

echtzeitfähiges Java erarbeiten sollte. An dieser Gruppe waren insgesamt 37 Firmen beteiligt. Da die gemeinschaftliche Umsetzung der erarbeiteten Ergebnisse misslang, bildeten sich zwei Gruppen, die jeweils ihre eigenen Vorstellungen in die Spezifikation einbringen wollten.

RTSJ, Real-Time Core Erweiterungen

Eine Gruppe wurde von Lizenznehmern der Firma Sun gebildet, die andere bestand aus Herstellern, die unabhängig bleiben wollten. Sun lieferte ihren Beitrag mit der „Real-Time Specification for JAVA“, kurz RTSJ. Die unabhängigen Hersteller lieferten ihren Vorschlag mit den „Real-Time Core Extensions for the JAVA Platform“.

Beide Vorschläge basieren auf der Definition von Objekten, die nicht der globalen Speicherplatzverwaltung (Heap, Garbage Collection) unterliegen und darauf arbeitenden echtzeitfähigen Routinen, die nicht von der Garbage Collection beeinflusst werden und diese bei Bedarf unterbrechen können. Beide unterscheiden sich im Wesentlichen in der Form, wie garantiert wird, dass Echtzeitroutinen keine Heap-Objekte manipulieren können und darin, wie der direkte Zugriff auf Hardware erfolgt.

4.3.4.1 Real-Time Core Erweiterung

Die Real-Time Core Erweiterung wurde im September 2000 vom sogenannten J-Consortium in der Version 1.0.14 veröffentlicht (siehe www.j-consortium.org). Sie enthält zwei verschiedene APIs, zum einen die sogenannte Baseline Java API für Nicht-Echtzeit-Threads in Java und zum anderen die Real-Time Core API für Echtzeit-Tasks. Ein weiterer, auf dem NIST-Dokument basierender Ansatz, ist die Basic Real-Time Java Specification, die eine sehr einfache Lösung darstellt und als Alternative oder Ergänzung zum Real-Time Core gedacht ist.

4.3.4.2 Real Time Specification for Java (RTSJ)

Die RTSJ wurde im Rahmen des Java Community Process (JCP) Ende 2001 veröffentlicht (siehe www.rtj.org). Im Folgenden gehen wir auf einige Aspekte der RTSJ genauer ein. Zur Definition der RTSJ wurde zunächst ein Anforderungskatalog erstellt, der die Anforderungen an eine Echtzeiterweiterung für Java spezifizierte. Diese Anforderungen sind im Einzelnen:

Anforderungen

- RTSJ soll keine Spezifikationen beinhalten, welche die Anwendung auf spezielle Java-Umgebungen beschränken.
- RTSJ soll abwärtskompatibel sein. Nicht-Echtzeit Java-Programme sollen uneingeschränkt auch unter RTSJ-Implementierung ausführbar sein.
- Die Portabilität von Java muss weiterhin möglich sein.
- RTSJ soll den momentanen Stand der Echtzeittechnik abdecken, aber auch für zukünftige Entwicklungen offen sein.
- RTSJ soll als erste Priorität deterministische Ausführung ermöglichen.
- Es sollen in Java keine syntaktischen Erweiterungen, wie z. B. neue Schlüsselwörter, eingeführt werden.

Lösungen

Dieser Anforderungskatalog wurde wie folgt umgesetzt:

- **Threads und Scheduling:** Die RTSJ fordert mindestens 28 Prioritätsstufen für Echtzeit-Threads und weitere 10 Stufen für „normale“ Threads. Sollten mehrere Threads die gleiche Priorität haben, werden diese nach dem FIFO Prinzip abgearbeitet. Es stehen NoHeapRealtimeThreads und RealTimeThreads zur Verfügung, wobei RealTimeThreads nur eine kleinere Priorität als der Garbage Collector haben können und sich somit im Gegensatz zu den NoHeapRealtimeThreads nur für „weiche“ Echtzeitanforderungen eignen. Zusätzlich können Scheduler dynamisch geladen werden, wenn das Interface Schedulable implementiert ist. Neben dem minimalen Priority-Scheduler können Implementierungen der Spezifikationen weitere willkürliche Scheduling-Algorithmen beinhalten.
- **Speicher:** Da, wie bereits erwähnt, die Garbage Collection Probleme bei der Synchronisation und dem Scheduling mit sich bringt, wurden neue Speicherbereiche außerhalb des Heaps (und damit auch außerhalb der Garbage Collection) definiert: der sogenannte *Scoped Memory* existiert erst nach dem er betreten (durch Thread oder „enter()“) wurde und dann nur solange Threads mit einer Referenz darauf existieren. Danach wird er wieder freigegeben. Objekte, die im sogenannten *Immortal Memory* erzeugt wurden, haben eine mit der Applikation identische Lebensdauer. Für diesen Speicher wird keinerlei Garbage Collection durchgeführt. Beim *Heap Memory* handelt es sich um den „normalen“ Speicher mit Garbage Collection. *Physical Memory* und *Raw Memory* sind Speicher mit verschiedenen Ansprechzeiten bzw. für Speicherzugriffe auf Byte-Ebene.

- **Synchronisation:** Falls mehrere Threads gleicher Priorität rechenbereit sind, werden diese nach dem FIFO Prinzip abgearbeitet. Die RTSJ fordert hierfür die Einrichtung einer Warteschlange für jedes Prioritätslevel. Falls ein Thread während der Ausführung blockiert wird, wandert er in der Prioritätswarteschlange wieder nach hinten, muss sich also neu „anstellen". Ferner wird mindestens die Implementierung des Priority Inheritance Protocol gefordert. Hier wird zu einem bestimmten Zeitpunkt die Priorität eines gesperrten Threads angehoben, um die Ausführung zu beschleunigen. Um eine nicht blockierende Kommunikation zwischen Threads (sowohl NoHeapRealTimeThreads als auch RealTimeThreads) zu gewährleisten, führt die RTSJ die sogenannten „free queues" ein.
- **Zeit und Timer**: Um relative oder absolute Zeit mit einer Genauigkeit im Nanosekundenbereich realisieren zu können, existieren spezielle Klassen.
- **Asynchrone Ereignis Behandlung:** Um die bestimmte zeitliche Ausführung einer Aktion nach einem Ereignis (sowohl selbst programmierte als auch externe wie z. B. Interrupts vom Betriebssystem) zu ermöglichen, wurden die Klassen AsyncEvent und AsyncEventHandler eingeführt. Hierbei repräsentiert AsyncEvent das zu überwachende Ereignis und der AsyncEventHandler enthält den nach Eintritt des Ereignisses auszuführenden Code.

Implementierungen

Bisher existieren nur wenige Implementierungen der RTSJ, die sich darüber hinaus größtenteils noch in der Entwicklung befinden. Neben RI, der Referenzimplementierung der RTSJ von TimeSys gibt es JRate, eine Open Source Lösung mit GNU Java Comiler sowie AJile, eine Implementierung der RTSJ auf Basis der CLDC 1.0 für aJ-80 und aJ-100 Chips.

JamaicaVM

Die JamaicaVM stellt eine der verbreitetsten Implementierungen der RTSJ dar. Ihre Erfinder, Teilnehmer des Projekt-Konsortiums AJACS (Applying Java to Automotive Control Systems, siehe www.ajacs.org bzw. www.ajacs.de), mittlerweile in einem Spinn-Off der TH Karlsruhe organisiert (ww.aiacs.com), bieten neben der VM auch spezielle Entwicklungswerkzeuge an (z. B. ein Plugin für Eclipse). Von der JamaicaVM werden mehrere Plattformen, darunter auch Echtzeitbetriebssysteme wie VxWorks, QNX oder verschiedene Linuxvarianten, unterstützt.

4.4 Synchrone Sprachen

Anwendungsgebiete

Synchrone Sprachen wie Lustre, Esterel, Signal usw. dienen zur Spezifizierung, Modellierung, Validierung und Implementierung von eingebetteten Systemen. Da ihre mathematische Grundlage formale Beweise ermöglicht, können damit Systeme mit hohen Sicherheitsanforderungen entwickelt werden.

Synchrone Sprachen haben enormes Interesse bei vielen führenden Firmen, die automatische Steuerungen für sicherheitskritische Systeme entwickeln, geweckt. So wurde z. B. Lustre von Schneider Electric zur Entwicklung von Steuerungen für Atomkraftwerke und von Aerospatiale zur Entwicklung von Steuerungen für den neuen Airbus verwendet. Esterel wurde erfolgreich von Dassault Aviation zur Entwicklung von Flugzeug-Steuerungen für den Rafale-Fighter eingesetzt und Signal von Snecma zur Entwicklung von Flugzeugmotoren benutzt. Weiterhin interessieren sich ST Microelectronics, Texas Instrument, Motorola und Intel für Esterel in Bezug auf Chip-Design mithilfe dieser Sprache (vgl. www-sop.inria.fr/cma/slap/). Der Hauptvorteil, den diese Firmen hervorheben, ist die formale semantische Fundierung der synchronen Sprachen; sie erlaubt eine formale Verifikation.

Synchrone Sprachen zeichnen sich dadurch aus, dass ihr Ein-/Ausgabeverhalten unabhängig von tatsächlichen Ausführungszeiten und -reihenfolgen beschrieben wird. Modellierungswerkzeuge, welche teilweise eine synchrone Semantik implementieren, dienen ebenfalls dazu, eine bestimmte Funktionalität auf einer höheren Abstraktionsebene zu beschreiben als dies mit klassischen imperativen (C/C++, Java) oder funktionalen Sprachen möglich ist. Beide Konzepte haben zunehmende Bedeutung für den Entwurf reaktiver Systeme, welche – oft sicherheitskritische – Steuerungsaufgaben wie z. B. die Steuerung von Aufzügen oder Flugzeugen übernehmen.

Historie

Das wesentliche Konzept dieser Sprachen ist, dass die meisten Anweisungen keine Zeit verbrauchen und ein Zeitverbrauch explizit zu programmieren ist. Synchrone Sprachen entstanden Anfang der 1980er Jahre basierend auf dem Modell der „perfekten Synchronie“. Anstoß war die Erkenntnis, dass herkömmliche Technologien nur noch bedingt für die Entwicklung von eingebetteten Systemen geeignet waren (Berry, 1998). Sie konnten in der Regelungstechnik schnell Fuß fassen, da sie sich nicht stark von den dort implizit verwendeten Modellen unterschieden. Synchrone Sprachen fassten in den 1990-ern auch Fuß im Hardware-Entwurf, nachdem man große Ähnlichkeiten des synchronen Modells mit den Modellen des

Schaltungsentwurfs entdeckte. Seit dem Advent synchroner Sprachen wächst das Interesse der Industrie an ihnen kontinuierlich.

Synchrone Sprachen werden auch heute noch ständig weiterentwickelt und daran wird sich aller Voraussicht nach auch in absehbarer Zukunft nichts ändern. Die Entwicklung der synchronen Sprachen bedient sich vielerlei Techniken aus verschiedensten Bereichen der Informatik (Berry, 1998): Regelungstechnik, Automatentheorie, binäre Entscheidungsbäume, konstruktive Logik sind nur einige davon.

Esterel

Esterel ist eine imperative Sprache, welche derzeit eine industrielle Verbreitung erfährt. In der Industrie werden jedoch hauptsächlich graphische synchrone Sprachen wie synchrone Varianten von Statecharts (μ-Charts) verwendet.

Perfekte Synchronie

Synchrone Sprachen verwenden das sogenannte *„perfekt synchrone"* Kommunikationsprinzip. Es unterstellt auf der Ebene der Spezifikation zunächst, dass eine Reaktion, also eine Transition von einem Systemzustand zu einem anderen, ebenso wie eine virtuelle Kommunikation gar keine Zeit verbraucht. Zeit vergeht lediglich in stabilen Systemzuständen.

Die Annahme der Rechtzeitigkeit hat mehrere Vorteile, die wir in Abschnitt 4.5.2 eingehend diskutieren werden. Bei den sogenannten „synchronen Sprachen" wie etwa Esterel wurde dieses Konzept bereits auf der Ebene der Programmierung eingebetteter Systeme erfolgreich umgesetzt.

Modellierungstechniken

Darüber hinaus müssen diese Konzepte von der Ebene der Programmiersprachen auf die Ebene der graphischen Modellierungstechniken erweitert werden. Hierzu gibt es ebenfalls schon Lösungen für die Modellierung zustandsbasierter Systeme, z. B. für die an Statecharts angelehnte Techniken SyncCharts Argos oder μ-Charts (Scholz, 1998), (Scholz, 2001). Letztere gibt dem Entwickler methodische Hilfestellungen bei der Verwendung von Statecharts, um ein eingebettetes System von der ersten, gänzlich architekturabhängigen Idee bis hin zur ggf. verteilten Implementierung zu konstruieren und dann zu partitionieren.

4.5 Ereignisbasierter Ansatz am Beispiel von Esterel

Esterel ist eine textuelle, perfekt synchrone Sprache die für die Programmierung reaktiver Systeme auf der Ebene des Systementwurfs (engl. system design) entwickelt wurde. Nach einem knappen historischen Einblick wird die Syntax der Sprache

beschrieben. Esterel besitzt eine mathematisch definierte und damit formale Semantik, die in diesem Rahmen nur in Ansätzen und lediglich informell erläutert werden kann.

4.5.1 Historie

Esterel wurde in einem gemeinsamen Projekt von der Ecole Nationale Supérieure des Mines de Paris (ENSMP) und vom Institut National de Recherche en Informatique et Automatique (INRIA) entwickelt. Grund hierfür war, dass sich „herkömmliche" Programmiersprachen für die Ansteuerung eines Roboters, den J.-P. Marmorat und J.-P. Rigault bauten, nicht geeignet waren, da sich die Struktur des reaktiven Systems „Roboter" nicht als Programm wiedergeben ließ. Das Esterel ist ursprünglich eine Bergkette zwischen Cannes und St. Raphaël, der Name erinnerte die Entwickler an „Echtzeit" (frz. temps réel). Gérard Berry vom INRIA entwickelte später eine formale Semantik für die Sprache, basierend auf der Hypothese der perfekten Synchronie. Mittlerweile gibt es von der Firma Esterel Technologies Inc. (www.esterel-technologies.com) eine industrielle Unterstützung, so dass bereits zahlreiche Firmen aus der Luftfahrt- und Automobilindustrie teilweise in Esterel entwickeln.

4.5.2 Hypothese der perfekten Synchronie

Hypothese

Synchrone Sprachen verwenden das sogenannte *„perfekt synchrone"* Kommunikationsprinzip (Berry, 1998). Es handelt sich hierbei um eine Hypothese, die auf der Ebene der Systembeschreibung zunächst unterstellt, dass eine Reaktion, also eine Transition von einem Systemzustand zu einem anderen ebenso wie eine virtuelle Kommunikation gar keine Zeit verbraucht. Ein System arbeitet also perfekt synchron, genau dann wenn. es ohne Zeitverzögerung auf externe Ereignisse reagiert. Ausgaben erscheinen somit zum gleichen Zeitpunkt wie die zugehörigen Eingaben. Zeit vergeht lediglich in stabilen Systemzuständen.

„Keine Zeit zu verbrauchen" darf hier aber keinesfalls missinterpretiert werden; es handelt sich dabei nur um eine Abstraktion der Realität, die annimmt, dass die tatsächliche Zeitspanne, die das später realisierte System verbrauchen wird, um seinen Zustand zu

verändern, kürzer sein wird, als die Zeit, die zwischen dem Eintreffen aufeinanderfolgender Signale der Sensorik vergeht. In der Praxis ist die Aussage „ohne Zeitverzögerung“ daher eher durch den Begriff der „Rechtzeitigkeit“ zu ersetzen. Berechnungen müssen somit abgeschlossen sein, bevor die nächste Berechnungsanforderung eintrifft. Die Zeitpunkte der Interaktionen müssen dabei nicht periodisch sein. Ein synchrones System kann sich synchron gegenüber seiner Umgebung verhalten, wenn es seine Berechnungen für eine Reaktion immer abgeschlossen hat, bevor neue Ereignisse eintreffen. Wenn dies sichergestellt werden kann, so ist gleichzeitig die Gültigkeit der Synchronitätshypothese gegeben oder anders ausgedrückt: Das komplexe Vibrationsmodell kann in das einfache Newtonmodell abstrahiert werden (Gunzert, 2003). Dazu müssen jedoch folgende Zeiten in Erfahrung gebracht werden:

1. Die minimale Zeit zwischen zwei Ereignissen (die aufeinander folgen können) sowie
2. die maximale Ausführungszeit der Reaktion.

Falls (2) größer als (1) ist, kann versucht werden durch Modifikation des Programms (z. B. Struktur, Übersetzungsoptionen usw.) oder der Laufzeitumgebung (Prozessor, Speicher usw.) die Bedingung für die Gültigkeit der Synchronitätshypothese trotzdem zu erfüllen (Gunzert, 2003).

Vorteile der Hypothese

Diese Annahme hat mehrere Vorteile. Die Reaktionszeiten sind in der Phase des Entwurfs noch unabhängig von der konkreten, ggf. verteilten Implementierung. Künstliche, vielleicht sogar falsche, zusätzliche Verzögerungszeiten werden nicht eingeplant; und schließlich kann auf diese Weise jede Reaktion in beliebig viele Sub-Reaktionen zerlegt werden, ohne das zeitliche Verhalten der Spezifikation zu beeinflussen. Bei den synchronen Sprachen wie etwa Esterel wurde dieses Konzept bereits auf der Ebene der Programmierung eingebetteter Systeme erfolgreich umgesetzt.

Die perfekte Synchronie unterstellt also ein ideales reaktives System, bei dem jede Reaktion durch eine quasi unendlich schnelle Maschine ausgeführt wird. Zeit vergeht nur, wenn das System einen stabilen Zustand erreicht hat. Zustandsübergänge verbrauchen dagegen keine Zeit.

Das synchrone Modell macht es einfacher, korrekte Systeme zu entwickeln und aufzubauen (Gunzert, 2003). Es verbirgt zeitliche Details und vereinfacht die Synchronisierung von Teilen des Systems. Die Funktionsweise eines Systems ist einfacher zu spezifizieren und zu verstehen, da das Verhalten vereinfacht wird.

Darüber hinaus wird durch diese Technik weniger Kontrolle über das Verhalten von einzelnen Komponenten eines Systems benötigt. Ihre genaue Geschwindigkeit spielt keine Rolle solange sie über einer bestimmten Schwelle liegt.

Die Synchronitätshypothese vereinfacht das modellierte zeitliche Verhalten und ermöglicht den synchronen Sprachen präzise definierte Semantiken (Gunzert, 2003). Dadurch ist es möglich, sowohl ausführbaren Code aus den Spezifikationen zu generieren als auch formale Methoden zur Verifikation einzusetzen. Synchrone Sprachen sind somit gleichzeitig Spezifikations- und Programmiersprachen.

In der Realität hat man häufig genauere Vorstellungen, in welchen Zeitabständen externe Ereignisse kommen können (z. B. welche Datenrate das System verarbeiten können muss). Dies erlaubt eine bessere Optimierung der zu synthetisierenden Systemkomponenten. In so einem Fall bietet sich ein zeitbehaftetes Modell an.

Alternative Lösungsansätze zur Modellierung der Zeitspanne zwischen Ein- und Ausgabe

Bei der Mehrzahl von reaktiven Systemen, insbesondere bei Echtzeitsystemen, ist es andererseits für deren tatsächliche Implementierung sehr wichtig zu wissen, wie viel Zeit zwischen Ein- und Ausgabe vergeht (Huizing und Gerth, 1991). Auf der Ebene der Beschreibung solcher Systeme sind zunächst mehrere Ansätze zur Lösung dieser Problematik denkbar:

1. Zunächst könnte man für jeden Systemschritt die exakte Zeit angeben, die hierfür seitens der Implementierung nötig sein wird. So eine Abschätzung ist aber zu aufwändig und zwingt den Designer der Software schon zu Beginn seiner Entwicklungsarbeiten, alle Vorgänge zeitlich exakt zu quantifizieren. Wollen wir davon abstrahieren, so müssen alternative Lösungswege gefunden werden.
2. Eine erste Näherung wäre, jeder Systemreaktion eine konstante endliche Zeitdauer zuzuschreiben. Obwohl dieser Ansatz schon sehr viel einfacher als der erste ist, ist er immer noch nicht abstrakt genug und besitzt darüber hinaus weitere Nachteile: In der Praxis stellt die Angabe einer fixen Zeitvorgabe zur Reaktion in der Regel eine obere Grenze für die Reaktionszeit der Implementierung dar, was dazu führen kann, dass einzelne Vorgänge ggf. künstlich verzögert werden müssen. Darüber hinaus kann eine Reaktion mit einer festen oberen Zeitschranke nicht weiter durch eine Folge von (Teil-)Reaktionen verfeinert, also konkretisiert werden. Schrittweise Konkretisierungen von Systembeschreibungen sind aber ein probates Mittel zur

konstruktiven Gewährleistung einer hohen Softwarequalität und sind daher im Zuge einer methodisch ordentlichen und fehlerfreien Softwareentwicklung wünschenswert.

3. Eine nächste Lösungsalternative könnte darum sein, jeder Reaktion eine beliebige, unterschiedliche (also nichtkonstante) positive Zeitdauer zuzuordnen. Dieser Ansatz wäre mit Sicherheit abstrakt genug, weil Reaktionszeiten beliebig ausgetauscht und Reaktionen damit verfeinert werden können. Allerdings lässt er durch dieses hohe Maß an Abstraktion so viel Freiraum (genauer: Nicht-Determinismus), dass es nahezu unmöglich ist, interessante Systemeigenschaften in dieser Entwicklungsphase nachzuweisen.
4. Es bleibt also nur die Lösung, jeder Systemreaktion die Zeitdauer 0 Zeiteinheiten zuzuschreiben und damit die Annahme zu verbinden, die Dauer einer Systemreaktion des implementierten Systems sei stets schneller als die Rate der eingehenden Ereignisse oder Signale. Eine vergleichbare Annahme wird bei der Entwicklung von Mikrocomputern, also von Hardware, übrigens schon seit langem verwendet: Hier geht man stillschweigend davon aus, dass viele komplexe arithmetische und logische Operationen von der arithmetisch-logischen Einheit (ALU) der CPU (Central Processing Unit) ausgeführt werden können, bevor das nächste Mal ein Taktsignal, beispielsweise mit einer Frequenz von 4 GHz, erzeugt wird. In diesem Fall bleiben der ALU nur vier Nanosekunden zur Reaktion. Dass diese Zeitspanne immer eingehalten werden kann, wird von den Prozessorherstellern stets akribisch nachgewiesen, bevor ein neuer Prozessor auf dem Markt kommt.

Vorteile der perfekten Synchronie

Diesen letzten Ansatz bezeichnet man wie eingangs schon erwähnt als „perfekte Synchronie“. Er besitzt mehrere Vorteile:

- Reaktionszeiten sind bereits in der frühen und damit noch abstrakten Phase der Systementwicklung bekannt.
- Reaktionszeiten hängen nicht von der konkreten Implementierung ab.
- Reaktionszeiten sind so kurz wie möglich, da künstliche Verzögerungen zur Überbrückung von zeitlichen Obergrenzen nicht nötig sind.

- Jede Systemreaktion kann durch eine Folge von Reaktionen verfeinert, also konkretisiert werden, da die folgende zeitliche Gleichung immer gilt: 0 + 0 = 0.

Esterel

Esterel basiert auf der Hypothese, dass Signalaustausch und elementare Berechnungen keine Zeit benötigen, so dass Systemausgaben mit ihren Eingaben synchron ablaufen („perfect synchrony hypthesis"). Diese „perfekte Synchronisation" macht nebenläufiges, deterministisches Verhalten in Esterel möglich. Weitere Hauptmerkmale von Esterel sind umfangreiche Konstrukte zur Zeitdarstellung und die Möglichkeit zur Modularisierung.

4.5.3 Determinismus

Ein Programm in Esterel verarbeitet Ströme (potentiell unendlich lange Sequenzen) von Ereignissen. Ereignisse dienen zur internen und externen Kommunikation, wobei ein *Ereignis* aus mehreren Elementarereignissen (z. B. Signalen) bestehen kann, die nicht unterbrechbar sind. In Esterel wird auf jedes Eingabeereignis durch Ändern des internen Zustandes sofort reagiert und diese Reaktion läuft perfekt synchron mit der Eingabe ab. Die verarbeitende Einheit ist genügend schnell, so dass während der internen Berechnung keine neue Eingabe auftritt. Dieses interne Berechnungsintervall wird *„Moment"* oder *„Augenblick"* (engl. instant) genannt und ist unendlich kurz (engl. instantaneous).

Esterel nimmt an, dass reaktive Systeme deterministisch sind. Beim Nichtdeterminismus handelt es sich um ein ebenso vielschichtiges wie komplexes Themengebiet, dessen ausführliche Behandlung mit einem Anspruch auf eine nur annähernde Vollständigkeit den Rahmen dieses Dokuments bei weitem sprengen würde. Auf informeller Ebene können wir ein System dann als nichtdeterministisch betrachten, wenn es auf ein gegebenes Eingabeereignis (engl. input event) mehr als ein (eindeutiges) Ausgabeereignis (engl. output event) produzieren kann. Es wählt sozusagen eine von mehreren möglichen Systemantworten aus, ohne dass ein Beobachter des Systems von außen die Möglichkeit besäße, diese Antwort vorauszusagen.

Nehmen wir beispielsweise an, ein Aufzug befände sich im dritten Stockwerk eines Gebäudes und Personen im ersten und fünften Stock forderten ihn per Knopfdruck gleichzeitig an. Falls das Verhalten des für diese Funktionalität zuständigen Softwaremoduls

nichtdeterministisch beschrieben wäre, könnte sich der Aufzug für eine der beiden Möglichkeiten entscheiden (ohne dass vorher jemand wüsste wie). Wie man ebenfalls anhand dieses Beispiels sieht, hat Nichtdeterminismus auch nichts mit Wahrscheinlichkeitstheorie zu tun, sondern kann als allgemeinerer Begriff betrachtet werden.

Für Esterel gilt: Alle Anweisungen bzw. Konstrukte sind (vom Compiler) garantiert deterministisch. Diese Aussage muss im weiteren Verlauf nochmals genauer betrachtet werden: Mit Hilfe der Parallelkomposition zweier ursprünglich deterministischer Anweisungen kann der Entwickler ungewollt nichtdeterministisches Verhalten einführen. Um dies zu vermeiden, sind ausgefeilte Algorithmen erforderlich, die ein fertiges Programm auf Nichtdeterminismen untersuchen und nur deterministische Programme für die Übersetzung freigeben. Da es sich bei Esterel um eine Sprache handelt, die in der Regel direkt in einen ausführbaren Code übersetzt wird, ist diese Analyse unbedingt erforderlich. Schließlich kann auf der Ebene der realen Hardware Nichtdeterminismus nicht realisiert werden.

4.5.4 Allgemeines

Mit Hilfe von Esterel können im Wesentlichen Zustandsmaschinen (Mealy Maschinen, sequentielle Automaten) beschrieben werden, bei denen die Zustandswechsel durch externe Events (englisch für Ereignisse, daher der Name „ereignisbasierte Sprache“) ausgelöst werden und wiederum zum Aussenden von Events führen können. Diese Events am Eingang eines Modules werden *nicht* gespeichert, sondern führen zur sofortigen Aktivierung des Moduls zwecks eines Zustandswechsels. Danach sind alle Events am Eingang verschwunden, auch diejenigen, die im aktuellen Zustandswechsel nicht abgefragt wurden.

Esterel-Prinzip „Write Things Once“ (WTO)

Esterel verfolgt das *„Write Things Once“ (WTO) Prinzip:* Die Sprache erlaubt eine Wiederholung von Programmcode durch Verschachtelung syntaktischer Strukturen. Mealy Maschinen dagegen erlauben dies z. B. nicht und erfordern statt dessen eine Wiederholung von Codesequenzen. Das WTO Prinzip wird in den meisten „herkömmlichen“ Programmiersprachen ebenfalls verwendet, z. B. wird dort durch Schleifen vermieden, den Schleifenrumpf mehrfach zu schreiben. Es führt zu kompakten Programmen, die weniger fehlerträchtig und damit besser wartbar sind.

Orthogonalität

Ein zweites in Esterel umgesetztes Paradigma ist das der *Orthogonalität:* Jedes Programmkonstrukt kann beliebig in andere Konstrukte verschachtelt werden.

4.5.5 Parallelität

In Esterel gibt es einen Operator für die Parallelkomposition von Programmen: Seien P1 und P2 zwei Esterel-Programme, dann ist auch P1 || P2 ein Esterel-Programm und zwar mit der folgenden Charakteristik:

Eigenschaften

- Alle Eingaben, die von der Systemumgebung empfangen werden, sind gleichermaßen für P1 und P2 verfügbar.
- Alle Ausgaben, die von P1 (bzw. P2) generiert werden, sind im gleichen Augenblick (engl. „instantanteously", „in the same instant") für P2 (bzw. P1) sichtbar (= perfekte Synchronie).
- P1 und P2 werden nebenläufig ausgeführt und ihre Parallelkomposition P1 || P2 terminiert, wenn beide, P1 und P2, terminieren.
- P1 und P2 teilen sich keine gemeinsamen Variablen.

Broadcasting

Da es weder gemeinsame Variablen noch Konstrukte für den gezielten Austausch von Werten gibt, muss die Kommunikation zwischen P1 und P2 auf andere Weise erreicht werden. Es handelt sich hierbei um das Prinzip des „Broadcasting" (ein Sender, viele Empfänger; vgl. Funk und Fernsehen), bei dem aufgrund der Annahme der perfekten Synchronie übertragene Werte sofort den Empfängern zur Verfügung stehen. Man spricht hier darum auch vom „Instantaneous Broadcasting". Beim Broadcasting erfolgt die Kommunikation nicht gerichtet, d. h. ein dedizierter Empfänger kann nicht spezifiziert werden.

Ein Esterel-Programm besteht im Wesentlichen aus Deklarationen und Instruktionen.

4.5.6 Deklarationen

Esterel besitzt nur sehr wenige Datentypen und kann daher auch nicht als vollständige Programmiersprache bezeichnet werden.

Vielmehr müssen benötigte Datentypen, Konstanten, Signale sowie darauf aufbauende Funktionen und Prozeduren wie etwa das Lesen von Eingaben und die Produktion von Ausgaben in einer sogenannten „Host Language“, in diesem Fall die Sprache C oder Ada, implementiert und dann in Esterel importiert werden. In Esterel selbst wird lediglich die Prozesskontrolle programmiert. Fertige Esterel-Programme werden dann nach C zurückübersetzt und sind auf Mikrocontrollern usw. lauffähig.

Module

Ein Esterel-Programm kann aus einem oder mehreren *Modulen* bestehen. Ein Modul beginnt mit dem Schlüsselwort module, gefolgt vom Modulnamen und einem Doppelpunkt. Das Konstrukt wird mit dem Schlüsselwort „end module“ (in älteren Versionen lediglich durch einen Punkt) abgeschlossen. Dazwischen stehen der Deklarations- und der Instruktionsteil. Zeilenweise Kommentare beginnen mit „%“ und enden mit dem Zeilenabschluss:

```
module ErstesEsterelBeispiel :
       % Deklarationen
       % Instruktionen
end module
```

Basisdatentypen

Esterel kennt drei vordefinierte *Basisdatentypen,* nämlich „integer“, „boolean“ und „string“. Komplexere Datenstrukturen wie Arrays oder Records müssen in Esterel vom Benutzer deklariert und extern in C oder Ada definiert werden.

Konstanten

Konstanten werden in Esterel folgendermaßen deklariert:

```
constant MIN, MAX: integer;
```

Jeder Konstanten bzw. jeder Gruppe von Konstanten muss ein Typ zugewiesen werden. Im obigen Beispiel handelt es sich hierbei um den Basistyp „integer“. Die Initialisierung muss dann extern in der Hostsprache (C oder Ada) geschehen, da eine Zuweisung von Konstanten im Esterel-Modul nicht erlaubt ist.

Funktions-deklarationen

Beispiele für die *Funktionsdeklaration* in Esterel sind:

```
function TOP(): integer,
         FAKULTAET (integer): integer,
         MULT (float, float): float;
```

Wie man unschwer erkennen kann, können Funktionen keine, eine oder mehrere Argumente besitzen. Die Definition der Funktion findet dann in der Hostsprache statt. Die Funktionsdeklaration in

Esterel bestimmt den Typ und die Anzahl der Eingabeargumente sowie den Typ des Ergebnisses. Nach dem Funktionsnamen folgt die Liste der Argument-Typen, nach dem Doppelpunkt der Ergebnistyp. Eine Funktion liefert immer einen Wert als Ergebnis. Dabei ist die Angabe des Resultat-Typs obligatorisch, da Funktionen nicht überladen (engl. overloaded) werden dürfen.

Prozedur-deklarationen

Prozedurdeklarationen sind Funktionsdeklarationen sehr ähnlich. Allerdings finden sich in Prozedurdeklarationen statt Argument- und Resultat-Typen Listen mit Referenz- und Wertangaben. Eine Prozedur gibt kein Ergebnis zurück, sondern ändert einige der Referenzparameter, also Variablen, die durch die Prozedur gelesen und verändert werden können (engl. call-by-reference). Die Wertparameter (engl. call-by-value) können wie die Argumente der Funktion nur gelesen werden. Die Definition von Prozeduren findet dann wiederum in der Hostsprache statt.

Signale

Signale dienen der Kommunikation per Broadcasting, die als Input-, Output- oder Input-Output-Signale deklariert werden können. Input-Signale dürfen nur in das umschließende Modul eingelesen, Output-Signale nur vom Modul ausgegeben werden und Input-Output-Signale dürfen beides. Signale müssen in der Hostsprache durch Prozeduren definiert sein.

Reine Signale

Drei Arten von Signalen werden unterschieden. *Reine Signale* (engl. pure signals) übertragen keine Daten und tragen keinen Wert; sie dienen ausschließlich der Synchronisation und benötigen daher auch keine Angabe von Datentypen:

```
input A, B, C;
```

Einfache Signale

Einfache Signale (engl. simple signals) übermitteln einen Wert. Sie brauchen deshalb neben dem Signalnamen (wie etwa A, B oder C oben) zusätzlich eine Typangabe:

```
input A(integer), B(integer), C(integer);
output D(string);
inputoutput E(boolean);
```

Mehrfache Signale

Bei *mehrfachen Signalen* werden die Werte von simultanen Sendern durch eine sogenannte Kombinationsfunktion zusammengeschlossen, die binär, assoziativ und kommutativ sein muss. Nach dem Schlüsselwort „combine“ wird der Typ der Eingabesignale, nach dem Schlüsselwort „with“ der Name der Kombinationsfunktion angegeben. Ein Beispiel wird durch

```
output VALUE (combine FLOAT with FLOAT_ADD),
function FLOAT_ADD (FLOAT, FLOAT): FLOAT;
```

gezeigt. Obige Kombinationsfunktion „FLOAT_ADD“ muss, wie geschehen, als Funktion deklariert werden.

Signale besitzen zu jedem Zeitpunkt einen eindeutigen Status, nämlich „present“ oder „absent“, der zu jedem Zeitpunkt aufs Neue bestimmt werden muss, da er nicht gespeichert wird. Gespeichert wird lediglich ggf. der Signalwert.

Variablen

Variablen besitzen dagegen keinen Status, sondern nur einen Wert eines beliebigen Typs (s.o.), der gespeichert wird. Variablen können während eines Zeitpunkts ihren Wert mehrmals ändern.

Sensoren

Darüber hinaus kennt Esterel noch *Sensoren,* die vom zugehörigen Modul nicht ausgeschickt, sondern nur empfangen werden können. Die einzige Operation, die auf einem Sensor zulässig ist, ist die Abfrage seines Wertes durch den „?“-Operator. Da Sensoren im Folgenden eine untergeordnete Rolle spielen, wollen wir hier nicht im Detail auf sie eingehen.

4.5.7 Instruktionen

Im Instruktionsteil eines Esterel-Programmes werden Anweisungen durch folgende Konstrukte aufgebaut:

Konstrukte

- Deklaration von lokalen Signalen und Variablen
- elementare Anweisungen wie Zuweisungen und Prozeduraufrufe
- Verzweigungen und Schleifen
- Eingabe, Ausgabe und Testen von Signalen
- zeitliche Anweisungen
- Ausnahmebehandlung

Syntax der Anweisungen

Gültige *Anweisungen* in Esterel sind:

- **nothing** (leere Anweisung)
- **pause** (Zeitverbrauch)
- **emit** x(τ) (Signalemission)
- x := τ (Variablenzuweisung)
- **present** σ **then** S1 **else** S2 end (Fallunterscheidung)
- S1 || S2 (Parallelkomposition)
- S1; S2 (Sequentielle Komp.)

- **loop** S **end** (Schleife)
- **trap** t in S **end** (async. Prozessabbruch)
- **exit** t (async. Prozessabbruch)
- **abort** S **when** σ (synchr. Prozessabbruch)
- **suspend** S **when** σ (Prozesssuspension)
- **signal** x : α in S **end** (lokale Signale)
- **var** x : α **in** S **end** (lokale Variable)

Hierbei bezeichnet σ ein Signal, τ eine gültige Typangabe und α eine beliebige Anweisung. Durch die Klammerung von Anweisung mittels eckiger Klammern kann die Bearbeitungsreihenfolge modifiziert werden.

Auf informeller Ebene kann diesen Basis-Anweisungen folgende Bedeutung zugewiesen werden (wir verzichten hier auf das Studium der formalen Semantik von Esterel):

Informelle Semantik der Anweisungen

- **nothing** terminiert sofort und emittiert nichts; es ist notwendig, da es keine leere Instruktion in Esterel gibt
- **pause** verbraucht genau eine Zeiteinheit; emittiert nichts
- **halt** verbraucht die restliche Zeit; emittiert nichts
- **emit** x(τ) setzt den Status von x auf „present“ und ändert evtl. den Wert von x auf τ (ohne Zeitverbrauch)
- x := τ ändert den Wert von x auf τ (ohne Zeitverbrauch)
- **present** σ **then** S1 **else** S2 end testet auf Präsenz des Signals σ. Im positiven Fall wird sofort S1, sonst S2 ausgeführt. Insgesamt besitzt die Anweisung ein Verhalten wie S1 *oder* S2.
- S1; S2 führt S1 aus und startet sofort nach dessen Terminierung S2.
- S1 ‖ S2 startet die synchrone nebenläufige Ausführung von S1 und S2 und terminiert, wenn S1 *und* S2 terminieren.
- **loop** S **end** startet S und startet S erneut, falls S terminiert (S muss hier Zeit > 0 verbrauchen, ansonsten ergibt das Konstrukt eine Endlosschleife in Nullzeit).
- **abort** S **when** σ startet S unabhängig von der initialen Präsenz von σ. Falls während der Ausführung von S σ präsent wird, wird S abgebrochen und Signale in S zu diesem Zeitpunkt nicht mehr emittiert (starker Abbruch, engl. strong preemption).

- **suspend** S **when** σ startet S unabhängig von der initialen Präsenz von σ. Falls während der Ausführung von S σ gilt, wird S unterbrochen, wenn σ nicht mehr präsent ist, wird die Ausführung von S fortgesetzt, Signale in S zu diesem Zeitpunkt aber nicht mehr emittiert (vgl. **abort**).

Durch die Verschachtelung verschiedener „abort"-Anweisungen können nebenläufige Kontrollfäden (engl. control threads) mit statischen Prioritäten versehen werden.

Neben den oben genannten Basis-Anweisungen gibt es in Esterel erweiterte Anweisungen wie etwa „halt", „await" und „sustain", die aber durch Kombination von Basiskonstrukten auf rein syntaktischer Ebene erzeugt werden können und darum als „syntaktischer Zucker", also als syntaktischer Abkürzungsmechanismus betrachtet werden können. Wir gehen auf sie daher im Folgenden nicht näher ein.

4.5.8 Beispiel: Die sogenannte ABRO-Spezifikation

Eine Ausgabe O soll immer dann erfolgen, wenn zwei Eingaben A und B eingetroffen sind. Dieses Verhalten soll mit dem Reset-Signal R zurückgesetzt werden. Der vergleichsweise simple Esterel-Code hierfür lautet:

ABRO-Beispiel

```
module ABRO:
       input A, B, R;
       output O;
       loop
              [ await A || await B ];
              emit O
       each R
end module
```

4.5.9 Semantik

Die formale Semantik zur Verhaltensbeschreibung von Esterel-Programmen von Berry (Berry, 1998) definiert die Sprache durch *Zustandsübergänge*. Eine Erweiterung ist die *Ausführungssemantik*, ebenfalls von Berry (Berry, 1998), die durch eine terminierende Folge von elementaren Mikroschritten angegeben wird. Auf dieser

Ausführungssemantik beruht auch der Compiler von *Esterel V3* (Version 3). Dadurch wird eine Spezifikation zuerst in endliche Zustandsautomaten und anschließend in eine klassische Sprache wie C oder Ada übersetzt. Der *Esterel V4* Compiler basiert auf der sogenannten *elektrischen Semantik* (Berry, 1998), die ein Programm erst in Schaltbilder mit Verbindungen und logischen Ebenen übersetzt. Anschließend kann das Programm. in einer Standardsprache oder als endlicher Automat ausgegeben werden. Ausführungssemantik und elektrische Semantik unterscheiden sich v. a. in ihrem Umgang mit Kausalitätsproblemen.

4.5.10 Kausalitätsprobleme

Dynamische Analyse

Durch die perfekte Synchronie können sogenannte Kausalitätsprobleme entstehen, d. h. dass Signale wie etwa in „present S then emit S end“ zyklisch voneinander abhängen. In einer *dynamischen Analyse* werden alle möglichen Folgeanweisungen berechnet. Dadurch wird festgestellt, ob es zu einem Deadlock (Verklemmung) kommt. Dabei werden auch Relationen zwischen Eingabesignalen berücksichtigt. Der Compiler V3 arbeitet nach diesem Modell.

Statische Analyse

Im Compiler V4 wird dagegen eine *statische Analyse* durchgeführt; dabei wird die Korrektheit unabhängig von möglichen Folgeanweisungen unter der Annahme bestimmter Bedingungen geprüft. Diese statische Überprüfung ist zwar schneller als die dynamische, kann aber auch dazu führen, dass ein eigentlich korrektes, will heißen deterministisches, Programm abgelehnt wird. Die elektrische Semantik übersetzt ein Esterel-Programm auf Gatterebenen in digitale Schaltbilder (Gatter) mit Verbindungen. Da Gatter durch boolesche Ausdrücke und Schaltnetze durch boolesche Gleichungen definiert sind, können in periodischen Abständen diese booleschen Gleichungen, die auch sämtliche Eingaben umfassen, gelöst und der neue Wert jeder elektrischen Verbindung berechnet werden. Dieses Vorgehen beruht auf der „Haltemengen-Theorie“. Dabei wird an jeder Stelle im Programm, an der auf weitere Ereignisse gewartet wird, die Anweisung „halt“ als Kontrollpunkt eingeführt. Dadurch kann ein Zustand durch die Menge seiner Kontrollpositionen, also Haltemengen, angegeben werden, was einen deterministischen Code zur Folge hat.

4.5.10.1 Logische Korrektheit

Fester Bestandteil aller Semantiken ist die logische Korrektheit. Ein logisch korrektes Esterel-Programm ist für jedes mögliche Eingabesignal reaktiv und deterministisch. Mit den gegebenen syntaktischen Konstrukten und dem Prinzip des Broadcastings ist es aber leicht möglich, syntaktisch korrekte, aber semantisch sinnlose Programme zu schreiben, die nicht reaktiv und/oder nicht deterministisch sind. Solche Programme sollen vom Compiler abgelehnt werden. Die logische Korrektheit kann durch umfangreiche Fallunterscheidungen der möglichen Eingabesignale geprüft werden.

Um Fallunterscheidungen der möglichen Eingabesignale durchzuführen, müssen wir zuerst festlegen, welchen Status Signale standardmäßig haben. Der Default-Status wurde auf „absent“ festgelegt. Wann ein Signal „present“ sein kann, wird formal durch das *Kohärenz-Prinzip* ausgedrückt.

Definition (Kohärenz)

Definition (Kohärenz):
Ein Ausgangssignal bzw. ein lokales Signal x ist zu einem Zeitpunkt präsent genau dann, wenn zu diesem Zeitpunkt die Anweisung „emit x“ im Sichtbarkeitsbereich von x ausgeführt wird.

Das Kohärenzprinzip verbietet also die Annahme, dass ein Signal „present“ ist, wenn zu diesem Zeitpunkt keine Emission dieses Signals stattfindet. Im Allgemeinen kann der Status eines Signals selbstverständlich vom Status eines anderen Signals abhängen. Beispielsweise hängt in „present y then emit x end“ x kausal von y ab. Die Kausalitätsrelation kann bei synchronen Sprachen durch Einsatz des Broadcasting Mechanismus Zyklen aufweisen, so z. B. in

```
signal x in
     present x then emit x end
end signal
```

Alternativen zur Behandlung solcher Kausalitätszyklen sind:

- *Alle zulassen:* Da Deadlock und Nichtdeterminismus ähnlich wie bei Statecharts möglich sind, wird diese Variante bei Esterel nicht verwendet.
- *Alle ablehnen, nur azyklische Programme zulassen:* Stellt eine zu starke Einschränkung dar. Es würden viele logisch korrekte Programme abgelehnt werden.

- *Zulassen, wenn Programm logisch korrekt:* Wird bei der logische Transitionssemantik in Esterel verwendet. Bei der Analyse des Verhaltens der Programme müssen die Status von Signalen zuerst angenommen und dann im weiteren Verlauf geprüft werden, ob die getroffenen Annahmen richtig waren. Die Prüfung der logischen Korrektheit ist algorithmisch jedoch sehr aufwändig.
- *Zulassen, wenn Programm logisch korrekt und dies einfach zu prüfen ist:* Wird bei der konstruktiven Semantik und damit in der Praxis verwendet.

Beispiele: Esterel

Im Folgenden werden zwei Beispiele logisch inkorrekter Esterel-Programme angegeben, bevor der Abschnitt mit einem korrekten Programm schließt.

Beispiel 1:
Nachstehender Programmtext stellt ein nichtreaktives Esterel-Programm dar, da keine logische Annahme über die Reaktion dieses Programmes gemacht werden kann. Die Annahme o sei „absent" bestätigt sich nicht, da dann im Else-Zweig eine Emission stattfindet (Verletzung des Kohärenzprinzips). Die Annhame o sei „present" bestätigt sich ebenfalls nicht, da dann keine Emission stattfindet (Verletzung des Kohärenzprinzips).

```
module P1:
  output o;
  present o else emit o end
end module
```

Beispiel 2:
Im Folgenden handelt es sich um ein nichtdeterministisches Esterel-Programm, weil zwei gültige Annahmen über die Reaktion dieses Programms existieren. Die Annahme o sei „absent" bestätigt sich, da dann keine Emission stattfindet. Die Annahme o sei „present" bestätigt sich gleichermaßen nicht, da dann eine Emission stattfindet.

```
module P2:
  output O;
  present O then emit O end
end module
```

Beispiel 3:
Das folgende Beispiel zeigt ein Esterel-Programm, das möglicherweise für den Leser auf den ersten Blick etwas sonderbar anmuten mag, tatsächlich aber logisch korrekt ist:

```
module P3:
  output o1, o2;
  present o1 then emit o1 end
  ||
  present o1 then
    present o2 else emit o2 end
  end
end module
```

Der erste Zweig ist die Kopie unseres nichtdeterministischen Beispiels 2. Der zweite Zweig ist die Kopie unseres nichtreaktiven Beispiels 1 eingeschlossen in ein „present“ Statement. Überraschenderweise ist dieses Programm reaktiv und deterministisch. Die einzige gültige Annahme ist o1 absent und o2 absent, welche sich dann bestätigt. Alle anderen Annahmen verletzten das Kohärenzprinzip.

4.5.10.2 Konstruktive Semantik

Beispiel 3 aus dem letzen Abschnitt, ein sonderbares, aber logisch korrektes Programm zeigt, dass eine unnatürliche Informationsverarbeitung in logisch korrekten Programmen möglich ist:

- Es werden Annahmen getroffen, die sich später selbst bestätigen müssen.
- Die logische Korrektheit ist in diesem Beispiel nur gegeben, weil keine andere Annahme sich bestätigt.

Bewertung der logischen Korrektheit:
Aus folgenden Gründen eignet sich die logische Korrektheit nicht als Basis für die Sprache:

- Die Prüfung eines Esterel-Programmes auf Reaktivität und Determinismus braucht exponentielle Zeit (NP-vollständiges Problem).
- Der Ansatz der logischen Korrektheit ist intuitiv für den Programmierer nicht nachvollziehbar, da der Kontrollfluss nicht

dem Zeitfluss entspricht. Da aber Esterel als eine imperative Sprache konzipiert wurde, soll auch in „present s then p end“ der Status von s nicht davon abhängen, was in p gemacht wird. Oder in anderen Worten: Bei der Ausführung dieses Statements möchte der Programmierer, dass zuerst a ausgewertet werden soll und dann in Abhängigkeit davon p. In „present s then p end“ wird durch „then“ eine kausale Reihenfolge ausgedrückt.

Dieser Missstand wird bei der *konstruktiven Semantik* ausgeschlossen. Sie berücksichtigt die Kausalität der Informationsverarbeitung, also einen gerichteten Informations- und Auswertungsfluss und lässt sich mit polynomiellen Aufwand effizient nachweisen. Dabei wird jedoch das Zurückweisen einiger logisch korrekter Programme in Kauf genommen.

Bei der konstruktiven Semantik werden keine Annahmen über den Status von Signalen gemacht, sondern das Programm und der Status der Signale werden einer Analyse unterzogen. Dabei wird anhand des bisherigen Programmablaufs berechnet ob ein Signal gesetzt werden muss oder nicht gesetzt werden kann. Aus diesen Informationen wird der weitere Programmablauf berechnet. Ein Programm wird abgelehnt, falls eine weitere Analyse aufgrund unbekannter Zustände von Signalen nicht fortgesetzt werden kann. Für weitere Informationen wird der Buchentwurf von Gérard Berry mit dem Titel „The Constructive Semantics of Pure Esterel“, im Internet unter www-sop.inria.fr/esterel.org/home.htm beziehbar (Stand: Dezember 2004), empfohlen.

4.5.11 Codegenerierung und Werkzeuge

Bei der Codegenerierung wird die gesamte Kommunikation zwischen einzelnen Prozessen oder Modulen vom Compiler aufgelöst. Das bedeutet, dass ein paralleles, aus vielen Esterel-Modulen bestehendes Programm in ein sequenzielles Programm umgewandelt wird. Die Verwaltung von Prozessen, Scheduling und Interprozesskommunikation wird zur Übersetzungszeit durchgeführt. Folglich sind dafür keine Aktionen zur Laufzeit mehr notwendig. Es kann zum großen Teil auf Echtzeitbetriebssysteme verzichtet werden, was wiederum zu einer Speicherplatzersparnis führt. Folgende Optionen stehen bei der Codegenerierung zur Verfügung:

Optionen bei der Code-generierung

- Erzeugung von *Automatencode* (explizite Automaten-Darstellung): Beim Automatencode werden alle Zustände und Transitionen des Zustandsautomaten explizit numeriert. Dieser Code ist sehr schnell, kann jedoch bei komplexen Systemen sehr groß werden. Im ungünstigsten Fall besteht ein exponentielles Verhältnis zur Spezifikationsgröße (Zustandsexplosion). In der Praxis stellt sich jedoch heraus, dass die generierten Automaten in vielen Fällen bereits minimal sind.
- Erzeugung von *Gleichungscode* (implizite Automaten-Darstellung): Der Gleichungscode ist eine implizite Darstellung eines Zustandsautomaten durch einen Satz boolescher Gleichungen. Hier wird die Reaktion auf Ereignisse erst zur Laufzeit durch Lösen des Gleichungssystems berechnet. Die Größe des Gleichungscodes wächst nur linear mit der Größe der Spezifikation. Die Ausführungszeiten sind dafür etwas länger.

Zielsprachen

Folgende Zielsprachen stehen zur Verfügung:

- C
- C++
- (Structural) VHDL – nur mit Compiler V7
- (Structural) Verilog – nur mit Compiler V7

Die letzten beiden Übersetzungsmöglichkeiten ermöglichen *Hardware/Software Codesign* (siehe Abschnitt 5.6) mit Esterel.

Werkzeug Esterel Studio

Esterel Studio ist ein kommerziell vertriebenes Tool, das von der graphischen Spezifikation über Simulation bis zur Verifikation von Programmen alles für eine professionelle Soft-/Hardware Entwicklung in Esterel anbietet. Esterel Technologies entwickelte die zurzeit (Dezember 2004) öffentlich verfügbare V5.91 Spezifikation von Esterel weiter und ergänzte sie um datenverarbeitende Konstrukte. Bei Esterel Studio bekommt man die Auswahl mit welchem Compiler man arbeiten möchte, mit dem V5.91 Compiler oder der Inhouse-Weiterentwicklung, dem V7 Compiler. Praktische Erfahrungen mit Studentengruppen (in 2004 und 2005) weisen darauf hin, dass das Tool noch etwas instabil ist. Dennoch kann nach Rücksprache mit den Entwicklern davon ausgegangen werden, dass dies in Kürze abgestellt sein wird.

Weiterführende Literatur

Im Internet finden sich ferner unter www.esterel-technologies.com weitere Informationen sowie freie Software (z. B. Compiler), Demos und White Papers zu Esterel.

4.6 Synchrone Datenflusssprachen am Beispiel von Lustre

Charakteristik

Lustre ist eine deklarative, synchrone, datenflussorientierte Sprache zur Entwicklung reaktiver Systeme. Es handelt sich um eine deklarative Sprache, da jeder Ausdruck in einem Programm stets durch ein Gleichungssystem repräsentiert ist, das durch Programmvariablen beschrieben wird. Dieser Ansatz wurde von technischen Formalismen, wie Differentialgleichungssystemen oder synchronen Anwendernetzen (Blockdiagrammen) inspiriert. Programme in Lustre können sowohl auf rein textueller Ebene als auch graphisch entwickelt werden. In Lustre repräsentiert jede Variable und jeder Ausdruck einen Datenfluss.

Historie

Die Entwicklung von Lustre begann im Jahr 1984. Seitdem wurde Lustre besonders auf den Gebieten der Compilation, Verifikation und des automatischen Testens kontinuierlich weiterentwickelt (siehe www-verimag.imag.fr).

Datenfluss

Im Gegensatz zu der imperativen Sprache Esterel handelt es sich bei Lustre und auch Signal um Datenfluss-orientierte Sprachen. Hierbei beschränken sich diese Sprachen auf jene Datenflusssysteme, welche mit fest definierter, d. h. limitierter Speichergröße implementiert werden können. In Lustre können alle syntaktischen, textbasierten Sprachmittel auch in Form graphischer Datenflussdiagramme repräsentiert werden. Lustre wird auf diese Weise zu einer für Softwareingenieure leicht verwendbaren Sprache und rückt damit in deren Interessensbereich. Da ihre Semantik im Wesentlichen auf temporaler Logik (siehe Abschnitt 6.4.3) basiert, bietet sie andererseits eine formale Basis, die für automatische Codegenerierung und Sicherung der Softwarequalität (Verifikation) genutzt werden kann. Vorgenannte Kausalitätsprobleme, wie sie etwa in Esterel auftauchen, würden in Lustre als zyklische Datenflussdefinitionen modelliert. In diesen Fällen würde eine (Datenfluss-) Variable von sich selbst abhängig sein.

Aufbau

Lustre stellt Datenoperatoren, beispielsweise Addition, Subtraktion oder Multiplikation zur Verfügung. Diese Operatoren arbeiten auf Operanden, welche alle mit einem gemeinsamen Takt (engl. clock) durch das Datenflussnetzwerk „gepumpt“ werden – kurzum: Alle Operanden besitzen den selben Takt.

4.6.1 Datenfluss und Clocks

Basic Clock

Jeder Datenfluss in Lustre wird durch eine unendliche Sequenz von Werten eines bestimmten Typs repräsentiert, welche durch eine Clock in Sequenzschritte unterteilt wird. Dadurch entsteht eine Taktung des Datenflusses. Jedes (Unter-)Programm besitzt ein zyklisches Verhalten. Ein derartiger Zyklus definiert eine Sequenz von Zeitpunkten, die sogenannte „Basic Clock“ (BC), in denen das Programm aktiv ist. Nur zu diesen Zeitpunkten finden Aktionen statt. Die Länge der Zeitabstände zwischen den Aktionen wird durch die Basic Clock festgelegt. Somit ergibt sich eine Sequenz von Zeitpunkten an denen Eingabedaten aus dem Eingabedatenfluss bearbeitet werden.

Die Basic Clock ist die feinste (diskrete) Zeitrasterung des Systems. Teilprogramme können ihre eigenen Clocks besitzen, allerdings stets nur mit gröberen Zeitrasterungen, also größeren diskreten Zeitintervallen als die Basic Clock. Unterprogramme werden in Lustre als sogenannte „Nodes“ bezeichnet. Insgesamt kann es in einem Lustre-Programm darum mehr als einen einzigen Takt geben. Alle Takte können aber stets auf die Basis Clock herunter „gesampelt“ (engl. down sampled) werden. Dadurch entstehen andere, langsamere Clocks. Im Programm geschieht dies durch Kombination der Basic Clock mit einem booleschen Datenfluss (BDF), welcher den gewünschten Sample definiert; vgl. Tabelle 4.1.

Tab. 4.1: Entstehung einer neuen Clock aus der Basic Clock und einem boolschen Datenfluss

BC	1	2	3	4	5	6
BDF	true	false	true	true	false	true
Ergebnis	1		2	3		4

Wie die Tabelle 4.1 zeigt, muss dieses Clock-Konzept nicht an die physikalische Zeit gebunden sein; Intervallschritte einer Clock müssen folglich nicht äquidistant sein. Soll ein physikalischer Zeitbegriff im Programm Einzug halten, kann als Eingabe ein boolescher Datenfluss benutzt werden, der beim Auftreten eines beispielsweise „millisecond“ Signals den Wert true liefert (Halbwachs, 1991).

4.6.2 Variablen, Konstanten und Gleichungen

Variablen

Eine Variable besteht aus einer unendlichen Sequenz von Werten eines Typs. Dabei werden von Lustre die Basistypen „boolean“, „int“ (Integer), „real“ für Gleitkommazahlen und ein Tupelkonstruktor für karthesische Produkte unterstützt. Komplexe Datentypen können bei Bedarf ähnlich wie in Esterel von anderen Hostsprachen importiert werden und als abstrakte Typen benutzt werden.

Gleichungen

Eingabevariablen werden bei der Schnittstellendeklaration mit ihrem Typ deklariert. Alle anderen Variablen dürfen nur einmal in Form einer Gleichung definiert werden. Die Gleichung „X = E;“ (E sei hier ein beliebiger Ausdruck) definiert X als identisch zu E. Dies hat zur Konsequenz, dass X dieselbe Sequenz von Werten, den gleichen Typ und die gleiche Clock wie E besitzt. Die Variable kann nach ihrer Definition nur noch gelesen, jedoch nicht mehr verändert werden. Dadurch wird ein wichtiges Prinzip der Sprache realisiert, das Substitutionsprinzip (X kann überall im Programm durch E ersetzt werden und umgekehrt). Folglich können die Gleichungen auch in beliebiger Reihenfolge geschrieben werden, ohne das Programmverhalten zu beeinflussen.

Konstanten

Konstanten können sowohl aus den Lustre-eigenen, als auch den portierten Datentypen bestehen. Ihre Sequenz von Werten ist dann konstant; ihnen liegt die Basic Clock zu Grunde. In Lustre gibt es zwei Arten von Operatoren: Datenoperatoren und temporale Operatoren.

4.6.3 Operatoren und Programmstruktur

Daten-operatoren

Lustre stellt nachstehende *Datenoperatoren* zur Verfügung:

- *arithmetische:* +, -, *, /, div, mod
- *boolsche:* and, or, not
- *relationale:* =, <, <=, >, >=
- *konditionale:* if-then-else
- *importierte Funktionen*

Diese Operatoren dürfen aber nur bei solchen Variablen angewendet werden, die dieselbe Clock besitzen.

Neben den vorgenannten Datenoperatoren stellt die Sprache vier *temporale Operatoren* zur Verfügung:

Temporale Operatoren

- **Pre** (Previous): Mit dem „Previous"-Operator kann auf den Wert einer Variablen zur Zeit der vorherigen Clock zugegriffen werden. Dieser Operator liefert den Wert eines Ausdrucks, den er bei der vorherigen Clock hatte. Sei E ein Ausdruck mit (e_1, e_2, ..., e_n, ...) als Sequenz von Werten, so liefert pre(E) die Sequenz (nil, e_1, e_1,.., e_{n-1},..).
  Vorsicht: Bei der Verwendung des pre-Operators entsteht für den ersten Wert der Sequenz ein undefinierter Ausdruck, kurz „nil" (als Kurzform für „not in list").
- **Fby** bzw. **->** (Followed by): Der „Followed by"-Operator dient zur Initialisierung von Variablen, um den Wert „nil" zu vermeiden, der beim Zugriff auf den Vorgänger der ersten Variablen einer Sequenz vorliegt (vgl. „Pre"). Seien E und F Ausdrücke mit der Wertesequenz (e_1, e_2, ..., e_n, ...) und (f_1, f_2, ..., f_n, ...), dann ergibt E->F die Wertesequenz (e_1, f_2, f_3, ..., f_n, ...). Mit anderen Worten setzt sich E->F aus dem ersten Wert von E gefolgt vom zweiten bis n-ten Wert von F zusammen.
- **When** (Sample): Dieser Operator passt einen Ausdruck an eine langsamere Clock an. Seien E ein Ausdruck und B ein boolscher Ausdruck mit derselben Clock, dann ist „E when B" eine Sequenz von Werten aus E an der Stelle, an der B true war mit der Clock, die durch die true werte von B bestimmt wird.
  Vorsicht: Alle Werte von E an der Stelle an der B false ist, werden weggeworfen. Benutzen Sie diesen Operator nur, wenn sichergestellt ist, dass keine wichtigen Werte verloren gehen.
- **Current** (Interpolation): Der „Current"-Operator ist das Pendant zum „When"-Operator und besitzt die entgegengesetzte Wirkung. Er passt einen Ausdruck mit einer langsameren Clock an eine schnellere an. Dies wird durch Interpolation realisiert. Sei E ein Ausdruck mit einer langsameren Clock und B ein boolscher Ausdruck der die schnellere Clock darstellt, dann hat current E dieselbe clock wie B. Die „leeren" Werte in der Sequenz werden mit dem Wert der vorherigen Clock aufgefüllt.

Während die ersten beiden temporalen Operatoren auf (Takt-) synchronen Datenflusselementen operieren, welche den gleichen Takt besitzen, werden die letzten beiden verwendet, um Datenflüsse unterschiedlicher Taktfrequenzen anzupassen.

Beispiel: Lustre

Beispiel (Lustre):

```
Node Zaehler (a, b: int; reset: bool)
returns (c: int);
Let
c = a -> if reset then a
         else pre(c) + b;
Tel
```

Zur Strukturierung des Programms dienen Knoten (engl. nodes). Sie sind Unterprogramme, welche die Funktionalität kapseln und wiederverwendbar gestalten. Eine Knotendeklaration besteht aus einem Knotennamen und einem Eingabe- und Ausgabeparameter mit ihren Datentypen. Im Knotenrumpf befinden sich die lokalen Variablendeklarationen, die Gleichungen und die Assertions (Halbwachs, 1991).

4.6.4 Assertions (Zusicherungen)

Um den Compiler bei der Codeoptimierung zu unterstützen, besteht die Möglichkeit, Assertions anzugeben. Durch sie kann dem Compiler mitgeteilt werden, dass bestimmte Inputereignisse nicht vorkommen, beispielsweise dass zwei Variablen nie den gleichen Wert besitzen können oder eine Variable nicht zweimal hintereinander den gleichen Wert annehmen kann. Assertions sind boolsche Ausdrücke, die immer wahr sind.

Sollte es z. B. der Fall sein, dass zwei Eingabevariablen A und B niemals gleich sein sollen, so schreibt man:

```
Assert not (A and B);
```

Assertions nehmen bei der Verifikation von Lustre-Programmen eine zentrale Rolle ein (Halbwachs, 1991).

4.6.5 Compilation

Überprüfungen

Neben den üblichen Überprüfungen eines Compilers muss der Lustre-Compiler zusätzlich folgende Eigenschaften überprüfen:

- Ist jede lokale Variable sowie jede Outputvariable durch genau eine Gleichung definiert? (Gebot)
- Kommen rekursive Knotenaufrufe vor? (Verbot)
- Werden alle Operationen nur auf Operanden mit erlaubten Clocks angewandt? (Gebot)
- Arbeitet jeder Datenoperator mit mehr als einem Operanden auf Operanden mit gleicher Clock? (Gebot)
- Werden uninitialisierte Ausdrücke verwendet? (Verbot) Es darf bei Clocks, Ausgabevariablen und Assertions keine undefinierten Werte geben, wie sie bei der Benutzung des „Pre"-Operators entstehen
- Kommen im Code zyklische Definitionen (Deadlocks) vor? (Verbot) Jeder Zyklus sollte mindestens einen „Pre"-Operator enthalten.

Die Vermeidung von Deadlocks verdient besondere Aufmerksamkeit, die an dieser Stelle durch ein Beispiel untermauert werden soll:

```
X = X+1;                  // nicht erlaubt
X = pre(X)+1;             // erlaubt
```

Der Compiler lässt keine strukturellen Deadlocks zu, auch wenn diese wie im nachstehenden Beispiel nie eintreten können:

```
X = if C then Y else Z;
Y = if C then Z else X;
```

Nach diesen Überprüfungen erzeugt der Compiler einen erweiterten endlichen Automaten. Dieser liegt dann, wie in Esterel in Form einer Datei im sogenannten OC-Format vor. Mit entsprechenden Codegeneratoren kann dann Code in Sprachen wie C, ADA oder Le-Lisp erzeugt werden.

Werkzeuge

Für Lustre stehen mehrere kostenlose Tools zur Verfügung, die zur Spezifikation, Modellierung und Verifikation benutzt werden können. Ihre kommerzielle Verwendung ist jedoch ausgeschlossen. Für kommerzielle Zwecke kann beispielsweise das Softwarewerkzeug „Scade" käuflich erworben werden. Scade (früher SAO+/SAGA), das von Esterel Technologies (www.esterel-technologies.com) entwickelt wurde, ermöglicht den graphischen Entwurf eines datenflussorientierten, reaktiven Systems und stellt die hierfür notwendigen Tools zur Verfügung.

4.6.6 Verifikation und automatisches Testen

Formale Verifikation

Bei sicherheitskritischen Systemen ist es besonders wichtig, dass ein Programm verifiziert werden kann. Durch die *formale Verifikation* kann gezeigt werden, dass bestimmte Zustände nie eintreten bzw. das Programm korrekt arbeitet. Zur Verifikation eines Programms werden in Lustre sogenannte synchrone Beobachter eingesetzt. Synchrone Beobachter sind Assertions, die die Eigenschaften des Systems beschreiben. Diese werden wie das Programm auch in der Sprache Lustre implementiert. Nachdem alle synchronen Beobachter in das Programm eingebunden sind, kann mit dem Verifikationstool „Lesar“ (Model Checker, vgl. Abschnitt 6.4.3) geprüft werden, ob die Spezifikation erfüllt wird. Ist dies nicht der Fall, wird ein Gegenbeispiel erzeugt.

Automatisches Testen

Der *automatische Test* ist gewissermaßen das Gegenteil der Verifikation. Es wird hierbei nicht versucht nachzuweisen, dass das Programm den Spezifikationen entspricht wie es bei der Verifikation gemacht wird, sondern es wird versucht ein Szenario zu finden das die Spezifikation verletzt. Dazu wird ein Testdurchlauf generiert, der aus zwei Schritten besteht:

- Erzeugen einer zufälligen Eingabe, die der Systemumgebung entspricht.
- Überprüfung ob die Spezifikation mit dieser Eingabe eingehalten wird.

Lurette

Dieser Testdurchlauf wird solange wiederholt, bis ein Fall gefunden wird, der die Spezifikation verletzt oder ausreichend Durchläufe gemacht sind, so dass die Korrektheit angenommen werden kann. Solche automatischen Tests können mit dem Tool „Lurette“ durchgeführt werden.

Vergleich Esterel, Lustre

Esterel und Lustre sind nicht für alle Problembereiche gleichermaßen einsetzbar. Bei einigen eignet sich der imperative Ansatz von Esterel besser, bei anderen jedoch bietet der Datenfluss-Ansatz von Lustre Vorteile. Da beide Sprachen von mehreren Softwarewerkzeugen zusammen unterstützt werden, ist es möglich, durch gemischsprachliche Programme die Vorteile beider zu vereinen. Dabei sind allerdings ggf. semantische Unterschiede beider Sprachen zu bedenken: Lustre bietet u. a. auch eine Semantik an, welche die perfekte Synchronie, anders als Esterel, nicht zwingend voraussetzt (Bergerand, 1986).

4.6.7 Lustre im Vergleich zu Signal

Auch Signal ist eine Datenfluss-orientierte, synchrone Sprache. Wie bei Lustre ist die Angabe mehrerer Clocks zur Taktung der Datenflüsse möglich. Im Gegensatz zu Lustre besitzen Signal-Programme allerdings nicht nur eine, sondern mehrere Basic Clocks. Anders als bei Lustre ist es somit nicht möglich, Taktintervalle in kleinere Intervalle aufzuteilen. Abgesehen von zu Lustre vergleichbaren Datenoperatoren stellt Signal drei temporale Operatoren zur Verfügung: Einen Verzögerungsoperator, einen „Sample"-Operator sowie einen deterministischen „Merge"-Operator zum Vereinigen zweier Clocks.

Obwohl es sich bei Lustre und Signal um zwei deklarative Sprachen handelt, unterscheiden sie sich doch erheblich. Lustre ist eine *funktionale Sprache,* wobei jeder Operator eine Funktion repräsentiert, welche Eingabesequenzen in Ausgabesequenzen transformiert. Signal dagegen ist eine relationale Sprache. Ein Signal-Programm definiert eine *Relation* zwischen Ein- und Ausgabeflüssen, wodurch – anders als bei Funktionen – Ausgabedatenflüsse auch einen Einfluss auf Eingabedatenflüsse haben können. In anderen Worten könnte man sagen, ein Signal-Programm induziert seine eigenen Bedingungen (engl. constraints). Folglich stellt die Überprüfung von Vollständigkeit und Konsistenz eine größere Herausforderung an den Compiler dar, als dies bei Lustre der Fall ist. Bei streng mathematischer Sichtweise muss hier die Existenz eines eindeutigen Fixpunktes bewiesen werden. Dieser existiert genau dann, wenn die Konjunktion aller Bedingungen eine eindeutig erfüllbare Aussage darstellt. Da das zeitliche Modell aller Clocks eine signifikante Rolle in Signal spielt, liegt dem Compiler ein komplexer temporaler Kalkül zu Grunde.

4.7 Zeitgesteuerter Ansatz am Beispiel von Giotto

Motivation

Bei der Softwareentwicklung eingebetteter, ggf. verteilter Systeme wird derzeit in den meisten Fällen in der Industrie, vor allem in der Automobilindustrie, folgende Vorgehensweise verwendet: Das mathematische Systemmodell und damit die Funktionalität des Systems wird von einem Regelungstechniker mit Hilfe von CASE-Tools wie etwa Matlab oder MatrixX erstellt und simuliert. Soll

dieses Modell dann hinsichtlich seines Echtzeitverhaltens getestet und ggf. optimiert werden, bzw. auf Zuverlässigkeit getestet werden, wird es in der Regel an einen Programmierer weitergegeben, der dann für die Programmierung des tatsächlichen Codes, der auf der realen Plattform ausgeführt wird, verantwortlich ist. Der Programmier muss dabei ggf. periodische Aufgaben wie z. B. das Holen von Sensordaten selbst implementieren und diese Sensorwerte einzelnen Prozessoren zuweisen. Dabei müssen natürlich alle Echtzeitanforderungen eingehalten werden, damit das Verhalten des Systems vorhersagbar bleibt. Hierfür hat der Programmierer zahlreiche unterschiedliche Freiheitsgrade, die ihn in der Regel überfordern. Diese Vorgehensweise ist sehr fehlerbehaftet, und so muss der Programmierer seinen Code immer wieder testen und ggf. verbessern. Wenn der letztendlich entstandene Code dann lauffähig ist, so wurde er für genau eine Zielplattform optimiert und funktioniert daher nur auf dieser einen Plattform vorhersagbar, für die er geschrieben wurde.

Um diese langwierige, mühsame und fehlerträchtige Vorgehensweise zu verbessern, wurde Giotto entwickelt. Giotto ist ein Programmieransatz für eingebettete Systeme, der auf einer möglicherweise verteilten Hardware bzw. verteilten Plattformen zum Einsatz kommen kann. Er besteht aus einer zeitgesteuerten (engl. time-triggered) Programmiersprache, einem Compiler sowie einem Laufzeitsystem (engl. runtime system). Giotto ist besonders gut für die Softwareentwicklung für Systeme mit harten Echtzeitanforderungen und periodischem Verhalten geeignet (Henziger, 2003).

Historie, Anwendungen

Im Gegensatz zu Esterel handelt es sich bei Giotto um einen zeitgesteuerten Ansatz, der an der University of California, Berkeley in den USA im Team des Österreichers Thomas Henzinger entwickelt wurde. Ein Giotto-Programm spezifiziert die genaue Echtzeitinteraktion zwischen Softwarekomponenten und deren Systemumgebung. Unlängst durchgeführte Fallstudien und damit Beispiele für Anwendungen von Giotto sind die Ansteuerung einer Drosselklappe (Stieglbauer, 2003) im Automobil oder Controller für einen autonom fliegenden, unbemannten Helikopter (Henziger, 2003).

Giotto trägt also dazu bei, die komplexe Aufgabe der Entwicklung von Software für Steuerungssysteme zu strukturieren: Domänen-spezifische Aspekte werden von Plattform-spezifischen und zeitbezogene Aspekte werden von Daten-bezogenen Aspekten getrennt (Pree et al., 2003).

Die Giotto-Programmiersprache stellt für den Programmierer ein abstraktes Modell zur Verfügung, indem sie die Plattform-

unabhängigen funktionalen und zeitlichen Belange bzw. Anforderungen strikt von den Plattform-abhängigen Scheduling- und Kommunikationsfragen trennt. Das zeitliche Verhalten eines Giotto-Programms ist dadurch stets vorhersagbar. Damit ist Giotto besonders gut für die Entwicklung sicherheitskritischer Applikationen mit harten Echtzeitanforderungen geeignet.

Compiler

Die Aufgabe des Compilers ist es, das Giotto-Programm in ausführbaren Code zu übersetzen. Dazu muss ein Scheduling für nebenläufige Tasks bestimmt werden. Außerdem entscheidet der Compiler im Falle einer verteilten Plattform, welche Tasks von welchem Prozessor ausgeführt werden. Zu diesem Zweck benötigt der Compiler die sogenannte Worst-Case-Execution-Time eines jeden Tasks, sowie Informationen über die Leistungsfähigkeit der Plattform. Anhand dieser Informationen erstellt der Compiler dann entweder den ausführbaren Code oder – falls das auf der gegebenen Plattform nicht möglich ist – gibt eine entsprechende Fehlermeldung aus.

Das Giotto-Laufzeitsystem setzt sich aus zwei virtuellen Maschinen zusammen (vgl. Abbildung 4.5). Die erste wird als sogenannte Embedded Machine (E-Machine) bezeichnet und ist für die (reaktive) Interaktion mit der Systemumgebung zuständig. Sie interpretiert den eingebetteten Code (kurz: E-Code), der die Ausführung von Software Tasks als Reaktion auf physikalische Ereignisse aus der Umgebung überwacht.

Bei der zweiten virtuellen Maschine handelt es sich um die sogenannte Scheduling Machine (S-Maschine), welche das (proaktive) Zusammenspiel mit der Plattform regelt. Sie interpretiert den Scheduling Code (kurz: S-Code), der die zeitliche Reihenfolge der Ausführung der einzelnen Taks spezifiziert. Damit der Leser einen ersten exemplarischen Eindruck der Sprache gewinnen kann, sei auf folgendes Beispiel verwiesen.

Beispiel: Giotto

Beispiel (Giotto-Programm):
Das folgende Beispiel zeigt den Giotto-Code einer simplen, diskreten Aufzugssteuerung. Ziel dieser Steuerung ist es, den Aufzug zu demjenigen Stockwerk zu fahren, auf dem er angefordert wurde sowie dann die Tür zu öffnen und zu schließen. Der Aufzug kann durch Drücken der entsprechenden Tasten auf dem jeweiligen Stockwerk angefordert werden. Das Giotto-Programm zur Lösung dieser Aufgabe ist im Folgenden abgebildet (übernommen aus www-cad.eecs.berkeley.edu/~fresco/giotto/):

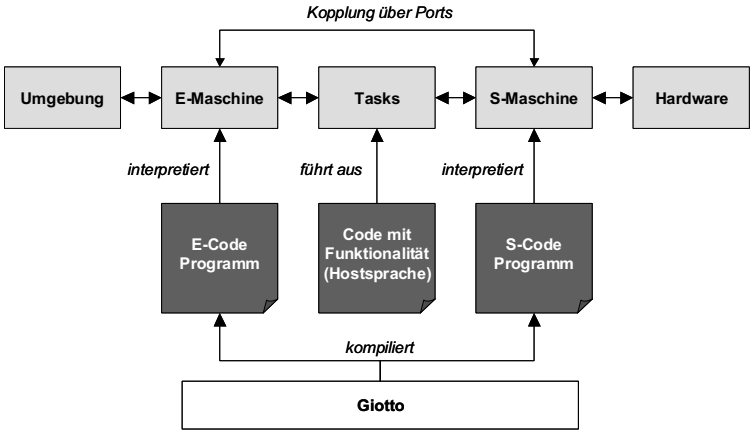


Abb. 4.5: Das Giotto-Laufzeitsystem mit E- und S-Maschine (nach Henzinger, 2003)

```
sensor
   elevator.PortButtons  buttons  uses elevator.GetButtons;
   elevator.PortPosition position uses elevtor.GetPosition;

actuator
   elevator.PortMove     motion  uses elevtor.PutMoveMotor;
   elevator.PortDoor     door    uses elevtor.PutDoorMotor;

output
   elevator.PortMove     tmotion := elevator.InitPortMove;
   elevator.PortDoor     tdoor   := elevator.InitPortDoor;
   bool_port             openwin := elevator.Elevator

task Idle(elevator.PortButtons b) output (tmotion, tdoor)
state () {
   schedule elevator.TaskIdle(b, tmotion, tdoor)
}

task Up(elevator.PortButtons b) output (tmotion, tdoor)
state () {
   schedule elevator.TaskUp(b, tmotion, tdoor)
}

task Down(elevator.PortButtons b) output (tmotion, tdoor)
state () {
   schedule elevator.TaskDown(b, tmotion, tdoor)
}

task Open(elevator.PortButtons b) output (tmotion, tdoor)
state () {
   schedule elevator.TaskOpen(b, tmotion, tdoor)
}

task Close(elevator.PortButtons b) output (tmotion, tdoor)
state () {
   schedule elevator.TaskClose(b, tmotion, tdoor)
}

// Actuator driver

driver Move(tmotion) output (elevator.PortMove m) {
  if constant_true() then copy_elevator.PortMove(tmotion,
m)
}
```

```
driver Door(tdoor) output (elevator.PortDoor d) {
  if constant_true() then copy_elevator.PortDoor(tdoor, d)
}

// Input driver

driver getButtons (buttons) output (elevator.PortButtons b)
{
  if constant_true() then
copy_elevator.PortButtons(buttons, b)
}

// Mode switch driver

driver PGTC(buttons, position) output () {
   if elevator.CondPosGTCall(buttons, position) then
dummy()
}

driver PLTC(buttons, position) output () {
   if elevator.CondPosLTCall(buttons, position) then
dummy()
}

driver PEQC(buttons, position) output () {
   if elevator.CondPosEQCall(buttons, position) then
dummy()
}
driver True() output () {
   if constant_true() then dummy()
}

start idle {

          mode idle() period 500 {
                    actfreq 1 do motion(Move);
                    actfreq 1 do door(Door);
                    exitfreq 1 do up(PLTC);
                    exitfreq 1 do down(PGTC);
                    exitfreq 1 do open(PEQC);
                    taskfreq 1 do Idle(getButtons);
          }
          mode up() period 500 {
                    actfreq 1 do motion(Move);
                    actfreq 1 do door(Door);
                    exitfreq 1 do open(PEQC);
                    taskfreq 1 do Up(getButtons);
          }
          mode down() period 500 {
                    actfreq 1 do motion(Move);
                    actfreq 1 do door(Door);
                    exitfreq 1 do open(PEQC);
                    taskfreq 1 do Down(getButtons);
          }

          mode open() period 500 {
                    actfreq 1 do motion(Move);
                    actfreq 1 do door(Door);
                    exitfreq 1 do close(True);
                    taskfreq 1 do Open(getButtons);
          }

          mode close() period 500 {
                    actfreq 1 do motion(Move);
```

```
                    actfreq 1 do door(Door);
                    exitfreq 1 do idle(True);
                    taskfreq 1 do Close(getButtons);
            }
}
```

Zunächst werden also die für das eingebettete System erforderlichen Sensoren und Aktoren definiert, dann die einzelnen Tasks und Treiber. Die Abarbeitung der Tasks erfolgt in sogenannten Modes (siehe weiter unten). Dieses Beispiel soll lediglich dazu dienen, einen Eindruck zur Syntax zu gewinnen. Sowohl eine detaillierte Erläuterung des Beispiels als auch eine umfangreiche Einführung in Syntax oder gar Semantik von Giotto würde hier den Rahmen deutlich sprengen. Zur Betrachtung des aus diesem Giotto-Code generierten E-Code sei der Leser auf die Seiten www-cad.eecs.berkeley.edu/~fresco/giotto/ verwiesen. Wir geben im Folgenden eine informelle Einführung in die zum Verständnis des obigen Beispiels notwendigen Konzepte.

Ports

Die gesamte Datenkommunikation wird in Giotto über *Ports* abgewickelt. Jeder Port repräsentiert dabei eine Variable des Programms. Jede Variable ist von einem bestimmten Typ und einmalig. Ferner sind Ports persistent, d. h. die Variablen beinhalten stets einen definierten Wert, bis dieser explizit verändert wird. Hierdurch wird indirekt ein Puffer modelliert. Es gibt drei Arten von Ports: *Sensor Ports, Actuator Ports* und *Task Ports*. Erstere enthalten – wie der Name bereits andeutet – Daten, die von Sensoren generiert werden. In die Actuator Ports werden Daten für die ausführenden Elemente des Systems geschrieben. Die Task Ports dienen zur Kommunikation der Tasks untereinander und werden auch zum Datenaustausch zwischen unterschiedlichen Modes verwendet.

Tasks

Bei einer *Task* handelt es sich in Giotto um einen Codeausschnitt, der eine bestimmte Aufgabe ausführt. Jede Task besteht aus Eingabe-, Ausgabe- und lokalen (engl. private) Ports, sowie einer Implementierung mit definierter WCET (Worst Case Execution Time). Die WCET ist die Zeitspanne, welche die Task maximal zur Ausführung benötigt. Jeder Eingabeport darf nur einer Task zugeordnet werden, während die Ausgabeports von mehreren Tasks aktualisiert werden können (wenn die Tasks in verschiedenen Modes sind).

Der funktionale Code der Task kann in einer beliebigen Programmiersprache, also z. B. C, C++, Java, usw. implementiert werden, da der Giotto-Compiler lediglich für die zeitliche Abfolge der Tasks verantwortlich ist. Der funktionale Teil der Task wird vom Compiler der jeweiligen Programmiersprache übersetzt.

Jede Task wird in festen Perioden aufgerufen, deren Dauer ebenfalls im Programm angegeben wird. Dabei können Daten aus dem vorherigen Aufruf in den lokalen Ports für den nächsten Aufruf gespeichert werden. Bei jedem Aufruf der Task werden die Eingabe- und lokalen Ports gelesen und neue Werte für die Task Ports und Actuator Ports berechnet.

Driver

Um die Daten von den Ports zu laden, verwendet eine Task sogenannte *Driver*. Sie wickeln den Datentransport vom bzw. zur Task ab und nehmen, falls nötig, Umrechnungen vor. Das Ausführen eines Drivers geschieht nach der Semantik von Giotto in Null-Zeit. Natürlich wird in der Realität auch für diesen Datentransport tatsächlich Zeit benötigt, welche aber in der zugehörigen WCET der Task subsumiert wird.

Normalerweise ist die für eine Task spezifizierte Ausführungsperiode WCET deutlich länger, als die Ausführung der Tasks tatsächlich dauert (da es sich um eine Wort-Case-Betrachtung handelt). Die Giotto-Semantik legt nicht fest, *wann* die Task in dem verfügbaren Zeitraum ausgeführt wird. Giotto garantiert lediglich, *dass* zur Berechnung diejenigen Werte verwendet werden, die zu Beginn der Periode in den Ports gespeichert waren. Ferner wird garantiert, dass am Ende der Periode die Task vollständig ausgeführt wurde und neue Daten an den Ports anliegen. Die Ausführung einer Task ist also immer zu einem bekannten Zeitpunkt beendet und kann auch nicht vorzeitig abgebrochen werden. Aus diesem Grund ist Giotto für die Entwicklung sicherheitskritischer Echtzeitsysteme mit harten Echtzeitanforderungen geeignet. Das Ergebnis eines Giotto-Programms ist stets deterministisch, weil mit dieser Art der Synchronisation immer vorhergesagt werden kann, welche Werte zu einem bestimmten Zeitpunkt zur Verarbeitung in die Tasks eingelesen werden.

Modes

Die Ausführung eines Tasks kann vom konkreten Betriebszustand des eingebetteten Systems abhängig sein. Beispielsweise darf in einem Flugzeug die Task „Schubumkehr“ auf keinen Fall aktiv sein, wenn sich die Maschine im normalen Flugbetrieb befindet. Hierfür hat man in Giotto sogenannte *Modes* eingeführt. Jeder Mode besteht aus einem festen Satz von Tasks. Ein Giotto-Programm setzt sich dann aus der Menge aller Modes inklusive deren Transitionen zusammen. Zu einem beliebigen Zeitpunkt kann dabei immer nur ein Mode aktiv sein. Jeder Mode ist definiert durch

- eine Zeitperiode (period),
- seine Mode Ports,

- Task Invocations (taskfreq),
- Actuator Updates und
- Mode Switches (exitfreq).

Die *Zeitperiode* gibt an, wie lange die einmalige Ausführung des Modes dauert. Die *Mode Ports* dienen zur Datenkommunikation zwischen verschieden Modes und werden bei einem Wechsel des Modes aktualisiert. Die *Task Invocation* gibt an, welche Tasks in diesem Mode ausgeführt werden. Im Mode wird auch der Transport von Ergebnissen der Tasks zu den ausführenden Teilen des Systems definiert. So wird beispielsweise der Driver, der für den Transport von Daten eines Actuator Ports zum zugehörigen Task verantwortlich ist, vom Mode gestartet. Um zwischen verschiedenen Modes wechseln zu können, gibt es sogenannte *Mode Switches*, die zu den laufenden Tasks weitere hinzufügen bzw. andere entfernen. Diese Mode Switches prüfen in festen Abständen (die sogenannte Switch Frequency), ob der Mode gewechselt werden soll. Dazu überprüft ein Driver den boolschen Wert eines festgelegten Eingabe Mode Ports und entscheidet, ob ein Wechsel des Modes angefordert wurde. Im Mode Switch ist außerdem der Mode festgelegt, in den gewechselt werden soll (der sogenannte Target Mode). Ein Mode Switch ist nur dann zulässig, wenn durch den Wechsel keine laufenden Tasks unterbrochen werden.

Reaktivität

Ein Giotto-Programm spezifiziert die Reaktivität, also innerhalb welcher Zeit welche Funktionen periodisch auszuführen sind und wann die berechneten Ergebnisse weiterzugeben sind. Die Frequenz, mit der eine bestimmte Funktion auszuführen ist, wird spezifiziert. Beispielsweise wird ein Filter mit einer Frequenz von 200Hz ausgeführt, also alle fünf Millisekunden. Funktionen sind in Giotto in sogenannten Modes zusammengefasst. Zu einem Zeitpunkt ist ein Mode aktiv. An den Übergängen zwischen Modes kann auf graphische Weise durch Pfeile und deren Beschriftung angegeben werden, unter welchen Bedingungen zwischen den durch den Pfeil verbundenen Modes gewechselt wird.

Damit wird aber nicht das Scheduling bzw. die Verteilung spezifiziert. Aus Sicht eines Giotto-Entwicklers sind alle Funktionen atomare (nicht unterbrechbare) Ausführungseinheiten, ohne Prioritäten oder interne Synchronisationspunkte. Die Software-Prozesse, die von einem Giotto-Programm angestoßen werden, sind also reine Sub-Routinen (Funktionen) und keine Co-Routinen (Threads). Dem Giotto-Compiler ist es hingegen überlassen, Code für das Zeitverhalten zu generieren, der anderen Code unterbricht. Von dieser Möglichkeit wird in aller Regel auch Gebrauch gemacht,

um das Scheduling und Optimierungen zu bewerkstelligen. Damit sind die einzelnen Code-Stücke, die vom Timing Code angestoßen werden, auf Plattform-Ebene tatsächlich Co-Routinen und nicht Sub-Routinen. Diese Zweiteilung illustriert die Eigenschaften eines guten Software-Modells: Der Programmierer muss sich nicht selbst um die komplexe Aufgabe kümmern, zu welcher Zeit welche Threads nebenläufig ausgeführt werden müssen. Der Giotto-Compiler sorgt für die Effizienz des Plattform-spezifischen, ausführbaren Codes. Damit wird die Fehleranfälligkeit und somit die Zuverlässigkeit erheblich verbessert und aufgrund der Plattform-Unabhängigkeit dennoch die Basis für Komposition und Wiederverwendung geschaffen (Pree et al., 2003).

E-Machine

Die sogenannte Embedded Machine (E-Machine) in Giotto ist eine virtuelle Maschine für die Ausführung von Instruktionen, die das Echtzeitverhalten repräsentieren. Die E-Machine ergänzt das Echtzeit-Betriebssystem und repräsentiert diejenige Komponente innerhalb der Giotto-Middleware, welche zur Ausführung eines Programms benötigt wird. Der Giotto-Compiler übersetzt Plattform-unabhängigen Timing-Code in Plattform-abhängigem Code. Fallstudien (Henzinger, 2003), (Stieglbauer, 2003) haben gezeigt, dass Giotto als Embedded Software Middleware für Steuerungssysteme mit sehr engen Zeitvorgaben geeignet ist. Der durch die zusätzliche Abstraktion entstandene Zusatzaufwand hält sich nach Angaben der Entwickler in einem akzeptablen Rahmen (Pree et al., 2003).

Giotto-basierte Anwendungen profitieren von der Middleware in mehrerlei Hinsicht: Aufgrund der Semantik von Giotto, auf die im Rahmen dieses Studienhefts nicht näher eingegangen wird (vgl. (Henziger et al., 2001)), sind Giotto-Anwendungen deterministisch – das Output-Verhalten der Aktoren bei gegebenem Input an den Sensoren ist also eindeutig vorhersagbar. Das stellt eine Voraussetzung für die formale Verifikation dar. Darüber hinaus wird der Entwicklungsaufwand signifikant reduziert, da der Code für das Zeitverhalten durch einen Compiler erzeugt wird. Dies schließt eine häufige Fehlerquelle aus. Die gute Trennung von Timing Code und Functionality Code erlaubt eine leichte Änderbarkeit des Zeitverhaltens und ein einfaches Hinzufügen oder Modifizieren von Funktionalitäten. Einzelne Giotto-Programme können als Komponenten betrachtet werden; Die formale Semantik von Giotto wurde so definiert, dass bei jeder Komposition die Einhaltung der vom Programmierer einzeln spezifizierten Echtzeitanforderungen gewährleistet ist. Schließlich kann das System auf verschiedene Plattformen portiert werden, wobei lediglich die E-Machine portiert werden muss. Einschlägige Fallstudien hierzu (Henziger, 2003)

haben gezeigt, dass der Portierungsaufwand für die E-Machine etwa eine Personenwoche beträgt. Der Quellcodeumfang hierfür ist ebenfalls sehr gering (Pree et al., 2003).

Der Ansatz von Giotto löst *zwei für die Entwicklung echtzeitkritischer eingebetteter Softwaresysteme typische Probleme*, (1) erstens der Bruch zwischen dem unabhängigen, mathematischen Modell zur Systembeschreibung und des tatsächlich für eine bestimmte Plattform manuell (nach-)optimierten Codes sowie (2) zweitens die enge Verflechtung von funktionalem und zeitlichem Verhalten.

Trennung von Timing und Scheduling

(1) Die Lösung des ersten Problems gelingt durch die *Trennung von Timing (Echtzeitanforderungen) und Scheduling*. Bisher wurde bei der Entwicklung von Software für eingebettete Systeme der Code zwar weitestgehend auf Basis mathematischer Modelle erstellt, dann aber von Hand (nach-)optimiert, um die Zielplattform optimal zu nutzen und alle (harten) Echtzeitanforderungen einhalten zu können. Eingebettete Systeme mit harten Echtzeitanforderungen müssen sich neben der Einhaltung des mathematischen Modells auch um Timing und Scheduling der Aufgaben kümmern. Mit *Timing* ist hierbei die zeitliche Festlegung gemeint, wann welche Task ausgeführt werden soll. Es handelt sich hier also um einen Begriff auf Ebene der Spezifikation, wohingegen das spätere *Scheduling* erforderlich ist, um auf einer gegebenen Plattform diese Tasks so zur Ausführung zu bringen, dass alle Aufgaben gemäss der Echtzeitanforderungen rechtzeitig abgearbeitet werden.

Ein weiterer Punkt ist die starke *Plattformabhängigkeit echtzeitkritischer eingebetteter Software*. Sie kann meist nur auf derjenigen Plattform eingesetzt werden, auf der sie entwickelt wurde und ist auch dort nur schwierig erweiterbar. Giotto führt eine abstrakte Ebene zwischen dem mathematischen Modell und dem von der Plattform ausführbaren Code ein. Diese Ebene wird als eingebettetes Software Modell bezeichnet. In einem eingebetteten Software Modell spezifiziert der Entwickler nur das Timing und die Interaktion (also die Reaktivität) der Prozesse. Es ist dagegen nicht nötig, dass er oder sie sich um das Scheduling und/oder die physische Verteilung (Partitionierung) der Software auf dem eingebetteten System kümmert. In einem Giotto-Programm muss nur festgelegt werden, wann ein Sensor gelesen, seine Werte verarbeitet und die Ergebnisse an den passenden Empfänger weitergeleitet werden. Auf welchem Steuergerät und mit welcher Priorität diese Aufgabe abgewickelt wird, ist im Giotto-Programm nicht spezifiziert (hier stellen lediglich die sogenannten Annotationen in Giotto eine kleine Ausnahme dar). Mit dieser

Trennung muss sich der Programmierer ausschließlich um Plattform-unabhängige Fragen kümmern. Er legt also nur die Reaktivität des Systems auf die Umwelt fest. Die von der Plattform abhängigen Aufgaben des Scheduling und der Partitionierung werden dagegen vom Giotto-Compiler selbst übernommen (für eine detaillierte Beschreibung vgl. Beschreibungen zur E-Machine (Henziger, 2003)).

Verflechtung von funktionalem und zeitlichem Verhalten

(2) Eine zusätzliche Herausforderung stellt bei Echtzeitsystemen die in der Regel *enge Verknüpfung von Timing und Funktionalität* dar. Um das Timing, also alle Echtzeitanforderungen einzuhalten, wird bei der nachgelagerten manuellen Optimierung von Echtzeit-Programmen diese Verknüpfung zwangsläufig vom Programmierer verursacht. Werden Tasks zum Zwecke der Laufzeitperformanz so programmiert, dass sie stets asap (engl. as soon as possible) auf äußere Einflüsse wie z. B. Sensordaten reagieren, wird das Verhalten des Systems in der Regel unvorhersagbar (also nicht-deterministisch) und es entspricht eventuell auch gar nicht mehr dem ursprünglichen mathematischen Systemmodell. Giotto führt eine strikte Trennung von Timing und Funktionalität ein. Ein Giotto-Programm gibt das Timing des Systems exakt vor und überwacht es. Dabei wird in Giotto selbst die eigentliche Berechnung der Funktionalität nicht durchgeführt. Dies geschieht mit Hilfe eines in eine Hostsprache geschriebenen Programms, beispielsweise in C oder Java. Jede dieser Berechnungen wird allerdings als Task gekapselt ausgeführt und kann daher als atomar betrachtet werden: Eine einmal gestartete Taks wird nicht unterbrochen. Ein interner Synchronisationsmechanismus oder Datenaustausch mit anderen Tasks existiert nicht. Die Aufgabe des Giotto-Programmes ist es dabei, fortwährend zu überwachen, dass jede Task innerhalb ihres Zeitfensters ausgeführt wird. Hierin liegt genau die Stärke dieser Trennung von Timing und Funktionalität: Einzelne Tasks können wie Funktionen betrachtet und verifiziert werden, die hintereinander zu bekannten Zeitpunkten ausgeführt werden. Das Verhalten eines auf diese Weise programmierten Systems ist daher nur abhängig von den Sensordaten der Umgebung und somit deterministisch und verifizierbar. Dennoch spielt auch die Optimierung des zeitlichen Verhaltens eine Rolle; der Giotto-Compiler kann dabei das Scheduling eines Betriebssystems noch indirekt durch die Priorisierung der Tasks beeinflussen.

Weiterführende Information

Detaillierte Informationen zu Giotto sowie Softwarewerkzeuge möge der Leser bei Interesse im Internet auf folgenden Seiten nachschlagen: www.eecs.berkeley.edu/~fresco/giotto.

4.8 Zusammenfassung

Programmiersprachen

Wurden im Jahre 2000 noch ca. 80 Prozent aller eingebetteten Softwaresysteme mit der imperativen Programmiersprache C entwickelt, so ist für die Zukunft hier mit einem schrumpfenden Anteil zu rechnen. Mag zwar der direkte Einsatz von C++ bei Embedded Systems nicht sinnvoll sein, so können objektorientierte Sprachen wie Einschränkungen von Java oder die eigens zur Programmierung eingebetteter Systeme entwickelte objektorientierte Sprache Embedded C++ hier schon sehr gute Dienste leisten. Vor dem Hintergrund immer komplexer werdender Systeme mit stetig anwachsenden Qualitätsansprüchen gehört die Zukunft der Softwareentwicklung eingebetteter Systeme jedoch abstrakteren Ansätzen wie etwa graphischen Formalismen (Statecharts, UML, ROOM; siehe Kapitel 5) oder synchronen Sprachen (Esterel, Lustre, Signal).

Esterel

Esterel ist eine textbasierte Entwurfssprache zur ereignisgesteuerten Programmierung eingebetteter Systeme auf Basis komplexer Zustandsübergangssysteme. Esterel wurde unter anderem in Frankreich von Gérard Berry entwickelt und basiert auf dem Grundsatz der sogenannten perfekten Synchronie. Hierbei handelt es sich um die auf den ersten Blick zunächst rein theoretische Annahme, die Reaktion auf ein bestimmtes Eingabeereignis benötige keine Zeit (Nullzeit). Auf den zweiten Blick bzw. in der Praxis bedeutet diese Anforderung, dass ein Ausgabeereignis rechtzeitig, also bevor auf das nächste Eingabeereignis reagiert werden muss, produziert wird. Die Entwurfssprache Giotto dagegen verfolgt nicht die ereignisgesteuerte Philosophie, sondern ist zeitgesteuert.

Giotto

Die mit Hilfe von Giotto erstellten Programme sind portabel, haben ein vorhersagbares Verhalten und entsprechen Echtzeitanforderungen besser als herkömmlich erstellter Code. Die Reaktion auf Ereignisse in Echtzeit geschieht deterministisch, weil das Verhalten der Programme vorhersagbar ist, da dieses nur auf Einflüsse der Umwelt reagiert. Es gibt keinen internen Wettlauf zwischen den Programmteilen, da genau festgelegt ist, wann welche Aufgabe (Task) ausgeführt wird. Der Code ist portabel, weil er weder von der Struktur, dem Scheduling oder der Laufzeitperformanz einer einzelnen Plattform abhängig ist. Nur die Umgebungszeit ist für das Timing wichtig – diese wiederum ist aber von der Plattform unabhängig. Ein mit Giotto erstelltes Programm entspricht exakt dem mathematischen Modell des eingebetteten, möglicherweise verteilten Systems mit harten Echtzeit-

anforderungen. Auf diese Weise sind bei der Systementwicklung keine langwierigen Test- bzw. Optimierungszyklen mehr erforderlich und insgesamt kann die Entwicklungszeit verkürzt werden. Aufgrund der zeitlichen Invarianz bei der Komposition von Giotto-Modulen ist eine Erweiterung der Funktionalität möglich, ohne das zeitliche Verhalten der einzelnen Module zu beeinflussen.