2 Domänenorientierte Software-Entwicklung - DESC Methode

Dieses und die folgenden Kaptitel betreffen das Software Engineering von Embedded Systemen. Es werden Konzepte, Methoden und Implementierungstechniken vermittelt, die den Bau robuster und zuverlässiger Systeme ermöglichen. In diesem ersten Kapitel wird in einer allgemeinen Problem Analyse ein gemeinsamer methodischer Rahmen für die folgenden vier Kapitel geschaffen.

2.1 Embedded Software Engineering

Bei der Entwicklung von Embedded Systemen werden in der industriellen Praxis für einzelne Teilprobleme unterschiedliche Methoden und Tools verwendet, wenn überhaupt mit Methoden gearbeitet wird. Das Problem ist aber, das kaum umfassende Methoden existieren, in welchen die Teilmethoden kohärent zusammenführt sind. Plattformen mit Tool Chains integrieren zwar Entwicklungswerkzeuge, auf der methodischen Ebene entstehen aber Konglomerate mit fehlendem semantischen Zusammenhang zwischen den Teilmethoden.

DESC Methode (Domain-oriented Embedded System Construction)

Die DESC-Methode ist als umfassende Software-Entwicklungsmethode für Embedded Real-time Systeme konzipiert. In diesem Kaptitel werden in einer generischen Problemanalyse die typischen Problemkreise identifiziert, die bei der Entwicklung von Embedded Systemen eine zentrale Rolle spielen. Auf die Methoden, die DESC für die Entwicklung in diesen Problemkreisen anbietet, wird in den folgenden Kapiteln eingegangen.

Das domänenorientierte Korrespondenzprinzip für Embedded Systeme

Das Prinzip verlangt für die embedded Software einen starken Zusammenhang mit den Eigenschaften der Anwendungsdomäne:

> Die Software eines Embedded Systems muss bezüglich Struktur
> und Verhalten der technischen Umgebung entsprechen.

Mit dem Prinzip soll erreicht werden, dass die Software verständlich ist, und dass nicht durch unnötige Strukturkonflikte zusätzliche Softwarekomplexität entsteht.

Will man Übereinstimmungen mit der Umgebung formal erfassen können, werden folgende zwei Entwicklungskonzepte wichtig:

Modell-basierte Software-Entwicklung.

Das Korrespondenzprinzip verlangt für die Software klare Übereinstimmungen mit der Struktur und dem Verhalten der technischen Umgebung. Solche Übereinstimmungen sind nur realisierbar, wenn die Software mit formalen Modellen spezifiziert wird, die automatisiert in ausführbaren Code tranformiert werden können. Es ist kaum machbar, klare Beziehungen zwischen den Konstrukten einer Programmiersprache und der realen Umgebung herzustellen, weil die Sprachsemantik vollständig prozessor-orientiert ist. Modell-basierte Software-Entwicklung wird schon seit langem praktiziert um die Softwarekomplexität in den Griff zu bekommen und ist mittlerweilen auch auf dem Weg, für die allgemeine Softwareentwicklung *State of the Art* zu werden.

Kompositionelle Modell-Architektur.

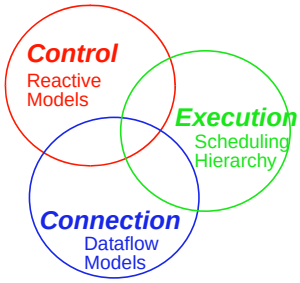
Kompositionelles Arbeiten mit Modell-Komponenten bedeutet, dass diese als sog. *Units of Construction* über explizite Konnektoren zu Systemen zusammengesetzt werden, was durch sog. *Architecture Description Languages (ADL's)* unterstützt wird. Die in DESC verwendeten Modellierungssprachen weisen ähnliche Eigenschaften auf wie *Architecture Description Languages*, indem kompositionelle Konstruktionstechniken das domänenorientierte Vorgehen optimal unterstüt-

zen. Es kann z. B. mit Komponenten begonnen werden, die als Agenten mit der technischen Umgebung interagieren. Die kooperative Funktionalität der embedded Software entsteht dann in weiteren Schritten, indem Komponenten erstellt werden, die mit den Agenten entsprechend interagieren.

Das Konzept hilft durch Anwendung des Geheimnisprinzips die wachsende Komplexität einzuschränken, ähnlich wie beim Arbeiten mit Modulen oder Objekten. Der Unterschied liegt bei der Verwendung expliziter Konnektoren bei der Konstruktion von Interaktion zwischen Komponenten. Komponenten sind damit inerte Bausteine, ähnlich wie die realen Komponenten einer technischen Maschine.

Domain-oriented Separation of Concerns: Steuerung, Verbindung, Ausführung

Eine domänenorientierte Analyse der Entwicklungsproblematik von Embedded Systemen ergibt, dass die Steuerfunktionen und deren Verbindung zu den Rechnerschnittstellen streng getrennt zu entwickeln sind. Mit den Steuerfunktionen werden die funktionalen Anforderungen erbracht, und diese beziehen sich nur auf die technischen Prozesse, nicht auf die Sensoren und Aktoren. Die Steuersoftware kann deshalb unabhängig von den Sensoren und Aktoren in einer eigenen Schicht (Layer) entwickelt werden, in der sie virtuell mit den technischen Prozessen interagiert. Die virtuelle Interaktion ist mit der Verbindungssoftware in einer tieferen Schicht zu implementieren. Ein weitere Entwicklungsaufgabe ist schliesslich die embedded Software so auszuführen, dass die Echtzeitanforderungen an das System erfüllt sind.



DESC unterstützt die Entwicklung in den drei generischen Problembereichen mit entsprechenden Methoden:

- Steuersoftware: DPC Methode
- Verbindungssoftware: DEC Methode
- Ausführungssoftware: DRS Methode

Abb. 3: Generische Problembereiche bei der Embedded System Entwicklung

Für die Software-Entwicklung in den drei Problembereichen von *Abbildung 3* stellt DESC entsprechende Methoden und Werkzeuge zur Verfügung:

Domain-oriented Process Control - DPC.

Die Steuersoftware wird mittels reaktiver Verhaltensmodelle spezifiziert. Nebenläufige Reaktive Maschinen, die aus synchron kooperierenden Zustandsmaschinen bestehen, modellieren die Steuerfunktionen. Für vollständig zeitgetriggerte Steuerkomponenten werden hier Methoden der Regeltechnik verwendet.

Werkzeug: Geeignete State Machine Tools (CIP, SDL 2000, Statecharts)

Domain-oriented Embedded Connection - DEC.

Die Verbindungssoftware, welche die Reaktiven Maschinen mit den Computerschnittstellen verbindet, wird als parallele Datenflussmaschinen spezifiziert, welche Funktionen für I/O-Zugriffe und Embedded Middleware Dienste ausführen.

Werkzeug: DEC Tool

Domain-oriented Real-time Scheduling - DRS.

Die Steuer- und die Verbindungssoftware ist auf zwei Ebenen auszuführen. Die Ausführungssteuerung der Datenflussmaschinen und der Reaktiven Maschinen erfolgt non-preemptiv innerhalb nebenläufiger Embedded Units, während die Embedded Units RTOS-basiert als Fixed Priority Server ausgeführt werden.

Werkzeug: DRS Tool als DEC Tool Plugin (zur Zeit in Entwicklung)

2.2 Embedded System Problemanalyse mit Problem Frames

Domänenorientierung ist ein Entwicklungsprinzip für Software-Applikationen. Die domänenorientierte Sicht ist eine problem-orientierte Sicht, das heisst es wird von der primären Aufgabenstellung ausgegangen, die sich immer auf die Applikationsdomäne bezieht. Bei Embedded Systemen ist das die Entwicklung einer Applikation, die als Software-Maschine ständig bestimmte Interaktionen mit den technischen Prozessen der Systemumgebung unterhält [1, 2].

Um den domänenorientierten Entwicklungsansatz für Embedded Systeme zu verstehen ist es notwendig, sich grundsätzlich mit der Entwicklungsproblematik solcher Systeme auseinanderzusetzen. Für eine entsprechende Problem-Analyse wenden wir die neue Requirements Engineering Methode der *Problem Frames* von Michael Jackson [3] an.

> “Understanding the problem and structuring the solution is the primary need.
> Only when that has been done the subject matter for computation is available”.
>
> (Michael Jackson)

Mit der Methode werden die Anforderungen eines Software-Entwicklungsproblems analysiert, indem die in Anforderungen implizit enthaltenen Beziehungen zwischen Anwendungsdomänen und der zu entwickelnden Software grafisch in Problemstrukturen (Problem Frames) dargestellt werden. Separation of Concerns wird unterstützt, indem schwierige Probleme in einfachere zerlegt werden können (problem decomposition).

Das Resultat der im folgenden diskutierten Analyse des allgemeinen Embedded Systemen Problems ist eine Zerlegung des allgemeinen Entwicklungsproblem in die drei Teilprobleme *Control, Connection* und *Scheduling Problem*. Die Analyse führt damit direkt zu einer generischen Software-Architektur für Embedded Systeme und zu einem entsprechend strukturierten Entwicklungsprozess der zeigt, wie die drei Problemkreise in separaten Entwicklungsthreads bearbeitet werden können.

2.2.1 Problemanalyse mit Problem Frames

Der Problem Frames Approach ist eine grafische Notation, um Problemstrukturen von Software-Entwicklungsprobleme darzustellen. Eine Problemstruktur setzt *gegebene Grössen* und *gesuchte Grössen* in einer Aufgabenstellung zueinander in Beziehung. Bei Software-Entwicklungsproblemen sind das Domänen der Systemumgebung und die zu programmierende Maschine. Die Problemstellung wird in Anforderungen festgehalten.

Problem Domains - Die gegebenen *Problembereiche*

> *Problem-Domänen* sind diejenigen Bereiche der realen Welt, für die das Programm einen Nutzen hat. Sie repräsentieren die gegeben Grössen des zu lösenden Problems.

Machine Domain - Die *Software-Maschine(n)*, die konstruiert werden muss.

> Die *Maschine* entsteht, indem ein Programm entwickelt wird, das auf einem Computer ausgeführt werden kann.

Requirements - Die *Anforderungen* an das System

> Die *Anforderungen* beschreiben, welchen Effekt das ablaufende Programm in der realen Welt haben muss. Anforderungen beziehen sich immer auf die *Problemdomänen.* Sie repräsentieren die Problemstellung.

Die ***Aufgabe*** ist immer Konstruktion der *Software-Maschine* aufgrund der Verbindungen zu den *Problem-Domänen,* so dass die *Anforderungen* erfüllt sind.

Die Struktur eines Problems wird grafisch durch ein *Problem Frame* beschrieben.

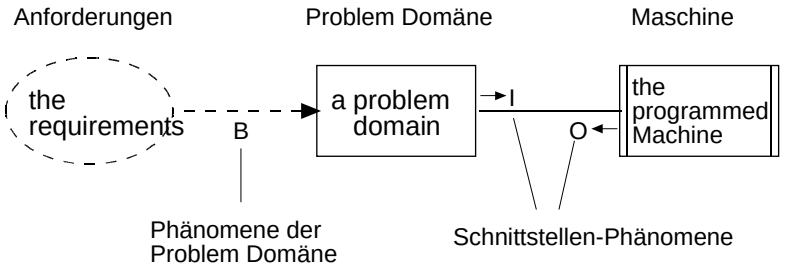


Abb. 4: Problem Frame

In *Abbildung 4* sind die Problemteile des allgemeinen Software-Entwicklungsproblems dargestellt. Die *gesuchte* Grösse, die programmierte Maschine, hat im Gegensatz zu den Problemdomänen *zwei* vertikale Begrenzungslinien. Die Problemdomäne und die Maschine sind mit einer Linie verbunden, die eine gemeinsame Schnittstelle (*Interface*) darstellt. Die Anforderungsbeschreibung ist durch ein gestricheltes Oval repräsentiert. Die Verbindung zur (hier einzigen) Problemdomäne zeigt, auf was sich die Beschreibung bezieht. Der Pfeil sagt, das die Anforderungen einschränkenden Charakter haben.

Die Verbindungen sind mit Symbolen beschriftet, die Mengen von Phänomenen (Zustände, Ereignisse, Aktionen, ...) spezifizieren:

> Die Menge B in der Abbildung sind diejenigen Phänomene der Problemdomäne, die in der Beschreibung der Anforderungen verwendet werden.
>
> I und O sind Schnittstellen-Phänomene, die den verbundenen Domänen gemeinsam sind (*shared phenomena*). Die Richtung des Pfeiles zeigt an, wer die Phänomene bewirkt, und wo sie einen Effekt haben. Solche gemeinsamen Phänomene sind notwendig, damit zwischen verschiedenen Domänen Interaktion stattfinden kann. In unserem einfachen generischen Beispiel stellt I die Input- und O die Outputdaten des Computers dar.

Ein Problem Frame eines konkreten SW-Problems ist in der Regel ein Netz bestehend aus mehreren Problemdomänen, das u. U. auch mehrere Maschinen und Anforderungsbeschreibungen enthalten kann.

Problem Frames und Methoden

Die Idee der Problem Frames basiert auf der Erkenntnis, dass es sich lohnt, bei der Lösung eines Problems Lösungen verwandter Probleme zu betrachten. Ein Problem Frame beschreibt abstrakt die Problemstruktur einer bestimmten Problemklasse, das heisst einer Gesamtheit von Problemen desselben Typs. Problem Frames können zur Beschreibung einfacher elementarer Problemtypen verwendet werden, es kann aber auch die Problemstruktur einer ganzen Anwendungskategorie, wie z. B. die Kategorie der Embedded System Probleme, analysiert werden. Gelingt es, aufgrund der abstrakten Problemstruktur konstruktive Lösungsansätze zu entwickeln, entsteht eine Software-Entwicklungsmethode.

> “A good software development method prescribes a specific problem frame and fully exploits its properties.” (M. Jackson)

Der Anwendungsbereich der Methode sind diejenigen Software-Probleme, deren Struktur als Instanzen des entsprechenden Problem Frame betrachtet werden können.

Komplexe Software-Entwicklungsprobleme können nicht einem einfachen Problem Frame zugeordnet werden. Solche Problem umfassen in der Regel immer ein Vielzahl von Teilproblemen. Das Ziel der Problemanalyse ist in diesem Fall, ein Zerlegung des Problems in Teilprobleme zu finden, die bekannten Problem Frames zugeordnet werden können (decompose your problem, fit the subproblems to specific problem frames and use their methods).

2.2.2 Das allgemeine Embedded System Problem

Zuerst wird kurz die allgemeine Problemstellung bei der Entwicklung einer Embedded Anwendung formuliert. Wir gehen davon aus, dass die System-Engineering-Probleme gelöst (Hardware) sind und eine geeignet SW-Entwicklungsumgebung zur Verfügung steht.

Gegeben

Das physikalische Verhalten der technischen Prozesse muss bekannt sein, technische Details sind in entsprechenden Dokumenten festgehalten.

Die zu verwendenen Prozessoren (Zielsysteme), IO-Module und verfügbaren Kommunikationssysteme (Feldbusse) sind festgelegt und entsprechend dokumentiert.

Die Sensor- und Aktor-Verbindungen zwischen technischen Prozessen und Prozessoren sind installiert und die Eigenschaften dieser Interaktionswandler sind dokumentiert.

Anforderungen

Funktionale Anforderungen (Functional Requirements) beschreiben, wie sich die gesteuerten Prozesse zu verhalten haben und welchen Einfluss die Bediener auf das System haben.

Zeit-Anforderungen (Timing Requirements) beschreiben, wie schnell das Embedded System auf bestimmte Prozess-Ereignisse reagieren muss. Bei Produktionssystemen kommen oft noch Anforderungen an den Durchsatz (Performance) dazu.

Aufgabe

Die Aufgabe ist embedded Programme zu entwickeln, so dass bei deren Ausführung auf den Zielsystemen bei den technischen Prozessen ein Verhalten bewirkt wird, das die Anforderungen erfüllt.

Um die typischen Schwierigkeiten besser verstehen können, die bei der Software-Entwicklung eines eines Embedded Systems auftreten, wird die allgemeine Problemstruktur anhand von Problem Frames diskutiert.

Die allgemeine Struktur des *Embedded System* Problems wird in *Abbildung 5* durch das entsprechende Problem Frame beschrieben.

Problem Frame - Embedded System Problem

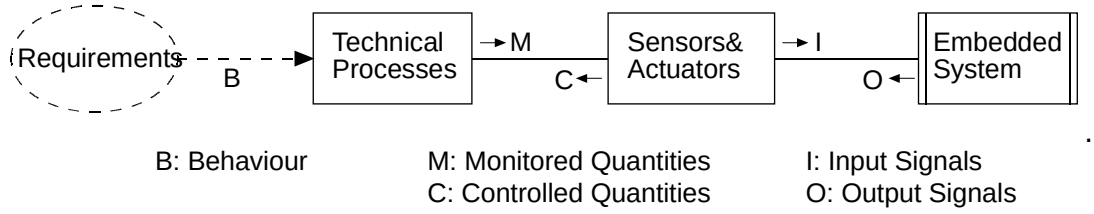


Abb. 5: Problem Frame - Embedded System Problem

Das *Embedded System* ist der Computer, der programmiert werden soll. Die *Technical Processes*, die überwacht und gesteuert werden müssen, bilden die primäre Problemdomäne. Zu dieser Domäne werden hier der Einfachheit halber auch die Bediener der Anlage und die Anzeigen gezählt. Weiter können auch andere, bereits produktive embedded Systeme zur Umgebung des zu entwickelnden Embedded System gehören. Charakteristisch für Embedded Systeme ist immer auch eine zweite Problemdomäne, die installierten *Sensors&Actuators*, über welche das Verhalten der *Technical Processes* durch den Rechner erfasst, bzw. beeinflusst werden kann. Diese Interaktionswandler ermöglichen die notwendige Interaktion zwischen den *Technical Processes* und dem *Embedded System*.

Die Interaktion zwischen den *Technical Processes* und den *Sensors&Actuators* beruht auf den überwachten und gesteuerten Zustandsgrössen M und C, über welche die Sensoren und Aktoren physikalisch wechselwirken (mechanisch, hydraulisch, elektrisch, ...). M steht für *Monitored Quantities* und C für *Controlled Quantities*. Die Verbindung zwischen den *Sensors&Actuators* und dem *Embedded System* besteht aus den elektronischen Input- und Outputsignalen I und O.

Die *Requirements* beziehen sich auf bestimmte Phänomene der *Technical Processes*. Die Menge dieser Phänomene wird mit B (*Behaviour*) bezeichnet. Ein Kessel muss zum Beispiel gefüllt werden, wenn ein Knopf gedrückt wird, ausser das System befinde sich im Alarmzustand. Die Menge B umfasst somit die Ereignisse *PushButton und VesselFilled*, die Aktionen *ValveOpen und ValveClose* sowie den Zustand *Alarm*.

Die ***Aufgabe*** ist ein Programm für ein Embedded System zu entwickeln, das bei den technischen Prozessen ein Verhalten bewirkt, so dass die Requirements erfüllt werden. Das muss aufgrund der elektronischen Input und Output-Signale I und O geschehen. Die Input-Signale I liefern Information von Sensoren, die mit den überwachten Zustandsgrössen M interagieren, und über die Output-Signale O kann mittels Aktoren auf die gesteuerten Zustandsgrössen C eingewirkt werden.

Eine fundamentale Entwicklungsschwierigkeit bei Embedded Systemen

Mit der Struktur des Embedded System Problems in *Abbildung 5* wird eine Entwicklungsschwierigkeit ersichtlich, die typisch ist für fast alle Embedded Systeme:

Die Interaktion mit den *Technical Processes* ist *nicht transparent.*

Die *Requirements* beziehen sich auf die realen Prozessphänomene B. Der technische Kontext der embedded Software ist jedoch durch die Input und Outputschnittstellen I und O der elektronischen Signale gegeben.

Um die Prozesse korrekt zu steuern ist damit immer noch eine zusätzliche Aufgabe zu lösen. Es muss der nicht-triviale Zusammenhang zwischen den Signal Schnittstellen I und O und den realen Prozessphänomenen B hergestellt werden. Wie gross die Schwierigkeiten sein können wird klar, wenn man versucht, eine funktionale Spezifikation zu schreiben, die sich nur auf den Signal-Input und Signal-Output bezieht.

Damit ist schnell klar, dass die Embedded Software entsprechend modularisiert werden muss. Dabei genügt es nicht, durch herkömmliche Hardware-Abstraktion die Signaldaten lediglich in verarbeitungsfreundlichere Formate aufzubereiten (Umwandlung der Hardwareformate, Skalierung, Filterung, Verifikation, etc.). Die Software muss applikationsspezifisch in Verbindungssoftware und funktionale Software aufgeteilt werden. Dabei sind der Verbindungssoftware der funktionalen Software Schnittstellen zur Verfügung zu stellen, die einen klaren Bezug zu den Prozessphänomenen haben, auf denen die funktionalen Anforderungen beruhen.

Diese Modularisierung mit ihrem generischen Charakter kann am besten in der allgemeinen Problemanalyse für Embedded Systeme gefunden werden, indem das Embedded System Problem in Teilprobleme zerlegt und jedes durch eine entsprechende Problemstruktur beschrieben wird. Dabei wird sich zeigen, dass immer noch ein weiterer Problemkreis ins Spiel kommt, der die real-time-mässige Ausführung der embedded Software betrifft.

2.3 Separation of Concerns: Funktion, Verbindung, Scheduling

Bei der Entwicklung eines Embedded Systems sind wie gesagt drei Problemkreise in Betracht zu ziehen. Der erste Problemkreis betrifft die Systemfunktionalität und ist rein logischer Natur. Es müssen Steuerfunktionen entwickelt werden, die auf das aktuelle Verhalten der Prozesse reagieren und dabei bestimmte Aktionen bei den Prozessen bewirken. Die Steuerfunktionen sind dafür verantwortlich, dass sich die technischen Prozesse so verhalten, wie das die funktionalen Anforderungen verlangen. Der zweite Problemkreis betrifft die Verbindung der technischen Prozesse mit den Steuerfunktionen. Die Lösung dieser Probleme ist stark von der verwendeten Einbettungstechnologie abhängig. Der dritte Problemkreis ist die Ausführung der embedded Software. Hier muss dafür gesorgt werden, das die zeitlichen Anforderungen an das System erfüllt werden, d. h. die Ausführung der Steuerfunktionen muss so geplant werden, das auch im schlimmsten Fall die verlangten Reaktionszeiten eingehalten werden. Das lösen solcher Real-time Schedulingprobleme kann schwierig sein, vor allem weil auch die Ausführungszeiten der funktionalen und der Verbindungsoftware zu

berücksichtigen sind. Damit wird auch klar, dass das Scheduling erst spezifiziert werden kann, wenn die funktionalen und die Verbindungskomponenten lauffähig entwickelt worden sind.

Ziel des folgenden Unterkapitels ist, diese drei unterschiedlichen Problemkreise voneinenander zu trennen.

2.3.1 Idee der Virtuellen Interaktion

Die Problematik der nicht-transparenten Interaktion mit den technischen Prozessen besteht, weil Prozessphänomene vom Computer aus rein technischen Gründen nur über Sensoren erkannt, bzw. über Aktoren erzeugt werden können. Die funktionalen und zeitlichen Anforderungen werden aber unabhängig von dieser Interaktionsproblematik formuliert, und nur diese Anforderungen sind aus Kundensicht wichtig. Für die Realisierung des Systems ist die korrekt funktionierende Verwendung von Sensoren und Aktoren jedoch eine notwendige Grundvoraussetzung. Trotz diesen Abhängigkeiten sollte es aber möglich sein, die Steuerfunktionen unabhängig von der Verbindungssoftware zu entwickeln.

Aus domänenenorientierter Sicht wird schnell klar, dass von der zu realisierenden Interaktion zwischen Prozessen und Steuerfunktionen abstrahiert werden muss. Die Idee ist eigentlich einfach. Wir benötigen ein einfaches generisches Interaktionsmodell, das uns erlaubt, die Interaktion zwischen den technischen Prozessen und den Steuerfunktionen unabhängig von der realen Interaktion über Sensoren, Aktoren und Verbindungssoftware zu spezifizieren. Diese modellierte Interaktion wird als *virtuell* bezeichnet, weil auf der Prozessseite reale *Ereignisse* scheinbar eine Interaktionspropagation zu den Steuerfunktionen bewirken, und weil die vom Embedded System bewirkte Interaktionspropagation scheinbar auf der Prozessseite reale *Aktionen* produziert.

Mit der spezifizierten virtuellen Interaktionsverbindung können die Steuerfunktionen unabhängig von der Verbindungssoftware und dem RT-Scheduling entwickelt werden. Als zweite zentrale Aufgabe muss die virtuelle Verbindung implementiert werden. Auf der Prozessseite ist die Implementation durch die installierten Sensoren und Aktoren gegeben. Diese Interaktionswandler müssen nun auf der Rechnerseite durch entsprechende Verbindungssoftware ergänzt werden, die als *Embedded Connection* bezeichnet wird. *Embedded Connection* und *Embedded Scheduling* implementieren dann zusammen die *Virtual Interaction.*

2.3.2 Generische Problemaufteilung (Problem Decomposition)

Das Konzept der virtuellen Interaktion ermöglicht eine domänenorientierte Aufteilung des allgemeinen ‘*Embedded System’*-Problems in drei generische Teilprobleme, nämlich in ein *Steuerproblem*, in ein *Verbindungsproblem* und ein *Scheduling Problem* (generic problem decomposition).

Das Control Problem

Die Aufgabe beim Control Problem (Steuerproblem) ist die Konstruktion von Steuerfunktionen, die abhängig von auftretenden Prozessphänomene Einwirkungen auf die Prozesse initiieren, so dass die funktionalen Anforderungen erfüllt werden. Das Control Problem wird, basierend auf der Spezifikation einer virtuellen Interaktionsverbindung, unabhängig vom Verbindungs- und Ausführungsproblem gelöst.

Das Connection Problem

Die Aufgabe beim *Connection Problem* (Verbindungsproblem) ist die Implementierung der virtuellen Interaktionsverbindung. Die Entwicklung der *Embedded Connection* umfasst u. A. das Einlesen des Signal-Inputs, die Ereignisdetektion, die Aktionsinitiierungen und das Absetzen des Signal-Outputs.

Das Scheduling Problem

Aufgabe beim Scheduling Problem ist das Real-time Scheduling der Verbindungs- und Steuerfunktionen. Damit wird auch klar, dass die Implementierung der virtuellen Interaktion auch für die Einhaltung der zeitlichen Anforderungen verantwortlich ist.

Die generische Struktur der drei Teilprobleme wird durch entsprechende Problem Frames dargestellt.

Problem Frame - Control Problem

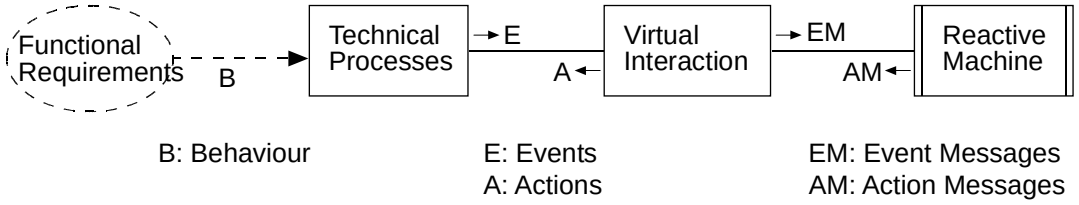


Abb. 6: Struktur des Control Problem

Abbildung 6 zeigt die Struktur des Control Problem. Neu ist die Domäne *Virtual Interaction*. Diese abstrakte Problem-Domäne stellt ein virtuelles Interaktionsmedium dar, welches auftretende Prozessereignisse und Aktionsinitiierungen scheinbar als Anregungen (Stimuli) zwischen den *Technical Processes* und der neu eingeführten *Reactive Machine* übertragen. Die *Reactive Machine* stellt die funktionalen Software dar.

Die virtuelle Interaktion eines Systems wird durch Festlegung der Schnittstellenstimuli E und A der *Technical Processes* spezifiziert. E (*Events*) bezeichnet diejenigen Prozessereignisse, die einen Einfluss auf die Steuerung haben müssen. A (*Actions*) spezifiziert die Menge von Einwirkungen auf die Prozesse, die von der Steuerung verursacht werden können. Die *Event Messages* EM auf der Embedded-Seite triggern die *Reactive Machine*, welche als Reaktion eine oder mehrere *Action Messages* der Menge AM erzeugen kann. Dabei entsprechen die *Event Messages* EM und die *Action Messages* AM eins-zu-eins den Events E und den Actions A auf der Prozessseite.

Die Leistung der Interaktionsabstraktion wird anhand der grafischen Problemstruktur besonders deutlich: Die Verbindung zwischen den Prozessen und der funktionalen Software ist aufgrund der einfachen Semantik der virtuellen Interaktion vollständig transparent geworden. Die Reaktive Maschine kann als passive Software-Komponente mit klar definierten Schnittestellen realisiert werden. Die Input- und die Output-Schnittstellen bestehen aus Triggerfunktionen, die den *Event* und *Action Messages* entsprechen.

Es bleibt die Frage, wo die Zeitanforderungen geblieben sind. Die Antwort ist einfach: Es gibt im Control Problem bezüglich der Ausführung auf dem Rechner keine Zeitanforderungen (Wartezeiten gehören zu den funktionalen Anforderungen). Es wird lediglich verlangt, dass bestimmte Ereignissequenzen bestimmte Aktionssequenzen erzeugen. Einzig die Ausführungszeiten der Reaktiven Maschine haben einen Einfluss auf das quantitative Zeitverhalten. Diese Ausführungszeiten werden als Nebenergebnis der funktionalen Problemlösung betrachtet (Messung auf dem Target) und sind als bekannte Grössen bei der Lösung des Scheduling Problems zu berücksichtigen.

Problem Frame - Connection Problem

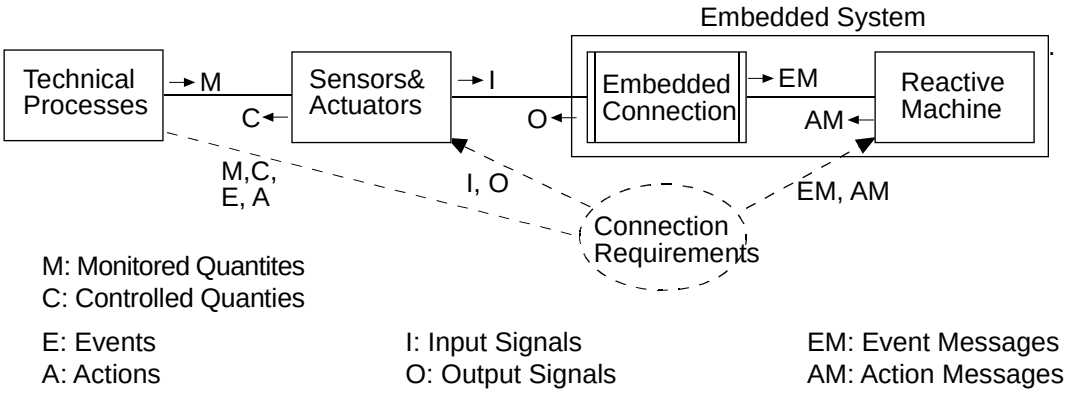


Abb. 7: Struktur des Connection Problem

In der Struktur des Connection Problems in *Abbildung 7* erscheint die *Reactive Machine* neben den *Technical Processes* und den *Sensors&Actuators* als gegebene Problemdomäne. ***Aufgabe*** ist, die virtuelle Ineraktionsverbindung mit den installierten Sensoren und Aktoren und der zu konstruierenden *Embedded Connection* zu implementieren. Die Anforderungen an diesen Software-Interaktionsadapter werden im Diagramm als *Connection Requirements* bezeichnet. Diese Anforderungen beschreiben, wie aus dem Verhalten der Sensor Inputsignale I aufgetretene Ereignisse zu detektieren sind, und wie für bestimmte Aktionsinitiierungen die Outputsignale O for die Aktoren gesetzt werden müssen. Diese Anforderungen nehmen damit auch Bezug auf die *Events* E und *Actions* A der *Technical Processes*.

Machbarkeits-Anforderungen an die spezifizierten Event und Actions

Damit aufgrund der spezifizierten Event und Action Messages das geplante System konstruiert werden kann, müssen für die entsprechenden *Ereignisse* und *Aktionen* folgende zwei Bedingungen erfüllt sein:

Erfüllbarkeit der funktionalen Anforderungen
Die *Ereignisse* und *Aktionen* bestimmen die mögliche Funktionalität des auf virtueller Interaktion basierenden Systems. Diese mögliche Funktionalität muss die angeforderte beinhalten.

Implementierbarkeit der virtuellen Interaktion
Die spezifizierten Ereignisse müssen über die installierten Sensoren vom Rechner erfasst, und die spezifizierten Aktionen vom Rechner über die installierten Aktoren bewirkt werden können.

Diese zwei Bedingungen spielen eine grundlegende Rolle während der ganzen Entwicklung eines Embedded Systems, weil sie zu Abhängigkeiten zwischen den verschiedenen Phasen des Entwicklungsprozesses führen.

Problem Frame - Scheduling Problem

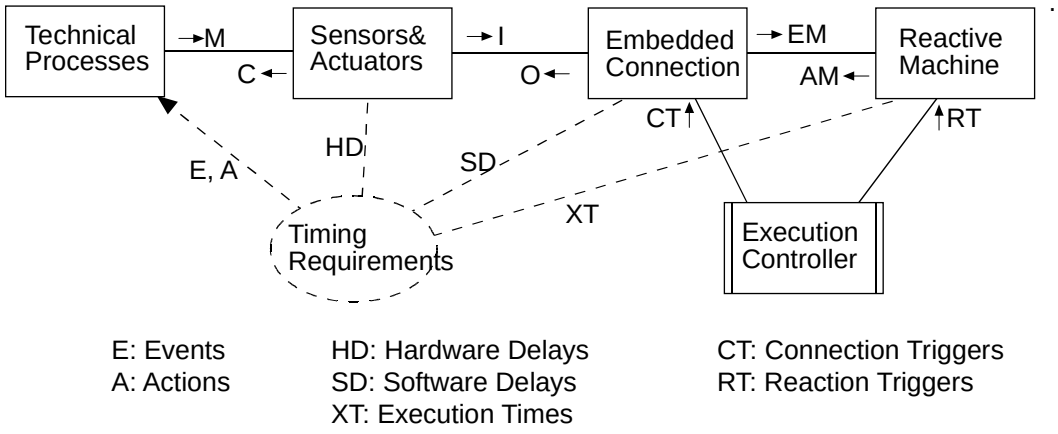


Abb. 8: Struktur des Ausführungsproblems

Die Lösung des Scheduling Problems muss dafür sorgen, dass die *Timing Requirements* erfüllt werden, d. h. bei der Implementierung der virtuellen Interaktion muss auch bestimmt werden, wann welche *Event Messages* zu verarbeiten sind. Unter Umständen können viele Ereignisse gleichzeitig auftreten. Für die Sensoren und Aktoren ist das kein Problem, weil diese in der Regel parallel arbeiten. Das Problem ist die sequentielle Ausführung der embedded Software. Die Detektion parallel auftretender Ereignisse und deren Verarbeitung durch die Reaktive Maschine muss geeignet sequentialisiert werden. Die dazu notwendige Spezifikation der Ausführungssteuerung wird als *Real-time Scheduling* bezeichnet. Hier liegen die wesentlichen Schwierigkeiten, aber auch das Potenzial, wenn bei knappen Ausführungsressourcen harte Antwortzeiten einzuhalten sind.

Das Lösen von Schedulingproblemen gehört zum zentralen Aufgabenkreis der Interaktionsimplementierung. Die ***Aufgabe*** ist ein *Execution Controller* (Ausführungssteuerung) zu konstruieren, bzw. zu konfigurieren, welche die Ausführung der Interaktionsfunktionen der *Embedded Connection* und der *Reactive Machine* so steuert (CT und RT), dass die *Timing Requirements* eingehalten werden. Diese Anforderungen sind typischerweise als Reaktionszeiten des Systems formuliert, die sich jeweils auf bestimmte Events (E) und Actions (A) beziehen. Dabei sind sowohl Verzögerungen der *Sensors&Actuators* (HD) und der *Embedded Connection* (CD) als auch die Ausführungszeiten (XT) der *Reactive Machine* zu berücksichtigen. Die gesamte Scheduling-Aufgabe wird zum Teil mit, und zum Teil ohne RT-Betriebssystem gelöst.

2.4 Semantik der virtuellen Interaktion

Das zeitliche Verhalten der virtuellen Interaktion muss durch ein einfaches, aber formales Interaktionsmodell definiert sein. Jede Implementierung der virtuellen Interaktion eines Embedded Systems muss die abstrakten Verhaltenseigenschaften des Interaktionsmodells erfüllen.

Operationelle Semantik der virtuellen Interaktionsverbindung

Immer wenn eines der spezifizierten Prozessereignisse auftritt, wird im virtuellen Interaktionsmedium eine entsprechende Ereignismeldung erzeugt, die eine Reaktion der Reaktiven Maschine auslöst. Die aktivierte Reaktive Maschine kann im virtuellen Interaktionsmedium Aktionsmeldungen erzeugen, die bei den technischen Prozessen entsprechende *Aktionen* bewirken.

Asynchronität. Die Übertragungsdauer für eine Meldung ist durch die Spezifikation der virtuellen Interaktionsverbindung nicht bestimmt (asynchrones Interaktionsmedium).

Sequentielle Event und Action Channels als nebenläufige Übertragungseinheiten. Die Ereignisse, bzw. Aktionen verschiedener Prozesse können typischerweise gleichzeitig auftreten, solche desselben Prozesses treten hingegen meistens in bestimmten Sequenzen auf. Dementsprechend wird das virtuelle Interaktionsmedium in parallele Kanäle aufgeteilt, d. h. die Ereignismeldungen eines Prozesses werden über einen *Event Message Channel* und die Aktionsmeldungen über einen *Action Message Channel* übertragen.

Ereignismeldungen, bzw. Aktionsmeldungen, die zu verschiedenen Kanälen gehören, können zeitlich parallel (concurrent) übertragen werden. Bei Meldungen, die zum selben Kanal gehören, verläuft die Übertragung sequentiell geordnet. Die damit erzeugte sequentielle Übereinstimmung im Verhalten von Prozessen und reaktiven Maschinen wird wichtig werden bei der Entwicklung der reaktiven Modelle.

Run-to-Completion Semantik der Reaktiven Maschine. Eine Reaktive Maschine ist eine sog. sequentielle Maschine, d. h. es wird nur eine Ereignismeldung aufs Mal verarbeitet und kann nicht durch eine weitere Ereignismeldung unterbrochen werden. Diese ununterbrechbare Ausführung von Steuerfunktionen wird mit *Run-to-Completion Semantik* bezeichnet. Bei der Implementierung einer Reaktive Maschine erfordert diese Bedingung, dass auch parallel übertragene Ereignismeldungen sequentiell zu verarbeiten sind.

Die Implementierung der virtuellen Interaktionsverbindung hat die Spezifikation der *Virtuellen Interaktion* zu erfüllen. Insbesondere muss bei detektierten Ereignissen, die sequentiell abhängig sind, die gleiche Sequenz bei den Ereignismeldungen ausgelöst werden. Analog muss bei sequentiell abhängigen Aktionen die Reihenfolge der Aktionsinitiierungen eingehalten werden.

Mehr zu Ereignissen und Aktionen

Ereignisse und *Aktionen* sind reale Prozessphänomene.

Attribute

Ereignisse und *Aktionen* können Attribute haben, deren Werte vom virtuellen Interaktionsmedium übertragen werden (Parameter von Event und Action Messages).

Beispiele

Das Ereignis *Identification* tritt auf, wenn ein Barcode-Leser eine neues Palett identifiziert. Das Attribut *barcode (int)* hält den gelesenen Barcode fest.

Die Aktion *ValveControl* bewirkt bei einem Ventil ein Änderung der Durchflussrate. Das Attribut *valveflow (int)* bestimmt die neue Stellung der Ventilklappe.

Sampling ist ein periodisches Abtastereignis eines Reglers. Als Attribute werden ein Temperatur- und ein Druckewert erfasst: *temperature (float), pressure (float).*

Die Aktion *Control* setzt bei der Reglerstrecke neue Stellwerte. Als Attribute werden die Leistung der Heizung und der Pumpe neu gesetzt: *heatpower (float), pumppower (int).*

Sporadische und periodische Ereignisse

Ein *Ereignis* ist entweder ein *sporadisches* oder ein *periodisches* Prozessereignis. *Sporadische* Prozessereignisse werden durch die physikalische Dynamik der Prozesse zu nicht vorhersagberen Zeitpunkten verursacht. *Periodische* Prozessereignisse sind Zeitereignisse (Abtastereignisse), die in regelmässigen Zeitabständen auftreten. Sie werden benützt, um wiederholt die Zustandswerte eines Prozesses als Ereignisattribute einer Reaktiven Maschine zu übermitteln (Regler).

Eigenschaften der ereignis-basierten Semantik der virtuellen Interaktion

Die beschriebene oprationelle Semantik der virtuellen Interaktion hat ganz bestimmte Gründe.

Ereignisse als Schnittstellenphänomene

Die virtuell an das virtuelle Interaktionsmedium gekoppelten Prozessphänomene sind reale Phänomene. Es sind Phänomene, die der Rechner über Sensoren erfassen oder über Aktoren bewirken kann. Das Erfassen oder Bewirken solcher Phänomene durch den Rechner erfolgt jedoch immer in diskreten Arbeitsschritten. Aus diesem Grund kommen auch auf der Prozessseite nur diskrete Stimuli als Schnittstellenphänomene in Frage; das können auch Zeitereignisse sein, die periodisch die Werte von Zustandsgrössen liefern, wie z. B. die Abtastereignisse einer Regelstrecke. Kontinuierliche Interaktion mit dem Rechner (shared continuous states) wird damit ausgeschlossen, was ja auch gut von den Implementierungsmöglichkeiten her verstanden wird: Kontinuierliche physikalische Interaktion zwischen Prozessen und Sensoren wird immer beim Einlesen der Sensorzustände zeitlich diskretisiert.

Zustandsgrössen als virtuelle Prozess-Schnittstellen als Alternative?

Es gibt Methoden, welche die überwachten und gesteuerten Zustandsgrössen als virtuelle Schnittstellen der technischen Prozesse verwenden und die aktuellen Werte periodisch übertragen (time driven control). Das ist z. B. bei SPS-basierten Systemen üblich. Die Zustandsgrössen werden als Prozessvariablen bezeichnet und im SPS-Takt übertragen. Die Implementierung dieser zustandsbasierten virtuellen Interaktion ist einfacher, dafür wird aber die Detektion sporadischer Prozessereignisse in die funktionale Software verlagert, z. B. in Bedingungen von sog. Ablaufdiagrammen. Dabei leidet jedoch meistens die Kompositionalität der funktionalen Software. Problematisch wird der zustandsbasierte Ansatz beim RT-Scheduling. Das Ausführungsproblem kann nicht mehr ausserhalb der funktionalen Software gelöst werden, weil das Auftreten zeitkritischer Ereignisse innerhalb der funktionalen Software festgestellt wird. Die Folge sind Schwierigkeiten, wenn bestimmte Antwortzeiten garantiert werden müssen.

Propagationszeit

Der asynchrone Charakter der virtuellen Interaktion berücksichtigt die endliche Propagationszeit in den Interaktionsketten zwischen den realen Prozessen und den Steuerfunktionen. Schon die Natur der Implementierung der virtuellen Interaktion zeigt, dass die virtuelle Interaktion i. A. nicht instantan sein kann. Reale Prozesse und die Ausführung der Rechnerprogramme laufen immer asynchron ab. Mit andern Worten, die Anregung des virtuellen Interaktionsmediums durch einen Prozess und die verursachte Aktivierung einer Steuerfunktion kann nicht zum selben Zeitpunkt stattfinden. Dazwischen vergeht Zeit. Wieviel Zeit vergehen kann, ist durch die Implementierung der virtuellen Interaktion bestimmt. Das ist vor allem der Fall, wenn mehrere Ereignisse gleichzeitig auftreten, oder wenn die Sensoren und Aktoren an Peripherieknoten angeschlossen sind, die über einen Feldbus mit dem steuernden Prozessor kommunizieren.

Funktionale Spezifikationen und Modelle

Die Aufgabe im Steuerproblem ist die Entwicklung der Steuerfunktionen, so dass die funktionalen Anforderungen erfüllt werden. Dazu muss aus den funktionalen Anforderungen eine funktionale Spezifikation FS hergeleitet werden die beschreibt, wie und wann bei der Verarbeitung von Ereignismeldungen und Aktionsmeldungen zu erzeugen sind. Formal stellt eine solche Spezifikation eine *Sequentielle Funktion* dar, die jeder Sequenz von Ereignismeldungen die zu erzeugende Sequenz von Aktionsmeldungen zuordnet:

EM = Menge der Event Messages, AM = Menge der Action Messages

FS: EM* -> AM* (X* = Menge aller X-Sequenzen)

Die Definition *Sequentieller Funktionen* ist in der Praxis jedoch kaum machbar, weil es zu viele Sequenzen gibt. Man verwendet deshalb Zustandsmaschinen ZM, die für ein Ereignis abhängig von einem inneren Zustand S die Aktionen erzeugen: ZM: EM x S -> AM x S (S = Menge States)

2.5 Domänenorientierte Software Architektur

Die Anwendung des Konzeptes des virtuellen Interaktion führt direkt zu einer Systemarchitektur mit zwei Schichten (Layers), einer funktionalen *Steuerschicht*, in der virtuell mit den technischen Prozesse interagiert wird, und eine *Verbindungsschicht*, in der die virtuelle Interaktion mittels Sensoren und Aktoren und entsprecheneder Interaktionssoftware implementiert ist.

Wird ein Embedded Systems in mehrere nebenläufige Teilsysteme aufgeteilt (*Embedded Units)* treten die erwähnten zwei Schichten pro Teilsystem auf.

2.5.1 Embedded Units

Es kann aus verschiedenen Gründen sinnvoll oder sogar notwendig sein, die Steuerungsaufgaben eines Embedded Systems auf mehrere nebenläufig ausführbare Reaktiven Maschinen zu verteilen. Jede dieser Steuerkomponenten muss dann softwaremässig in die Rechnerumgebung eingebettet werden. Die dabei entstehenden Ausführungseinheiten werden als *Embedded Units* bezeichnet.

Im folgenden sind drei typische Fälle beschrieben, in denen ein Embedded System aus mehreren Embedded Units besteht.

Verteiltes Embedded System. Wenn das Embedded System auf mehreren Prozessoren implementiert wird ist klar, dass mindestens pro Prozessor eine *Embedded Unit* entwickelt werden muss. Der Grund für den Einsatz von mehrerer Prozessoren kann die physikalische Verteiltheit der Anlage sein, grosse Anforderungen an die Rechenleistung, oder unter Umständen auch strenge funktionale Separationskriterien, die z. B. die Betriebssicherheit betreffen.

Funktionale oder entwicklungstechnische Gründe. Quasi-nebenläufige Komponenten auf demselben Prozessor zeichnen sich in jeder Hinsicht durch ihre starke Modularität aus. Solche Steuerkomponenten können nur über asynchrones Message Passing oder geschützte globale Variablen

interagieren. Mit solchen Embedded Units kann zum Beispiel eine starke Isolierung von Steuerfunktionsgruppen erzwungen werden.

Funktionen unterschiedlicher Geschwindigkeitsklassen auf einem Prozessor. Bei Systemen mit harten Echtzeitbedingungen kann es vorkommen, dass die Antwortzeiten für bestimmte Steuerfunktionen kleiner sein müssen als die Ausführungszeiten anderer Steuerfunktionen. In solchen Fällen müssen zwei oder mehrere Geschwindigkeitsklassen gebildet werden und pro Klasse eine separate *Embedded Unit* entwickelt werden. Die langsamen Embedded Units können dann von den schnelleren unterbrochen werden (preemptives Scheduling). Eine Aufteilung einer Embedded Unit kann zu einer grossen Herausforderung bei der Strukturierung der funktionalen Software werden. Wenn die funktionale Struktur und die Geschwindigkeitsklassen nicht übereinstimmen, müssen schwierige Strukturkonflikte gelöst werden. Dazu kommt, dass zur Entwicklungszeit der funktionalen Steuer-Software die zeitlichen Ausführungsprobleme meistens nur zum Teil bekannt sind.

2.5.2 Generische Domänen-orientierte Software-Architektur

Die Aufteilung des allgemeinen Embedded System Problems in ein *Control* und ein *Connection* Problem basiert auf dem Konzept der virtuellen Interaktion. Das damit unterstützte Abstraktionsprinzip führt zu einer generischen Systemarchitektur mit zwei entsprechenden Schichten.

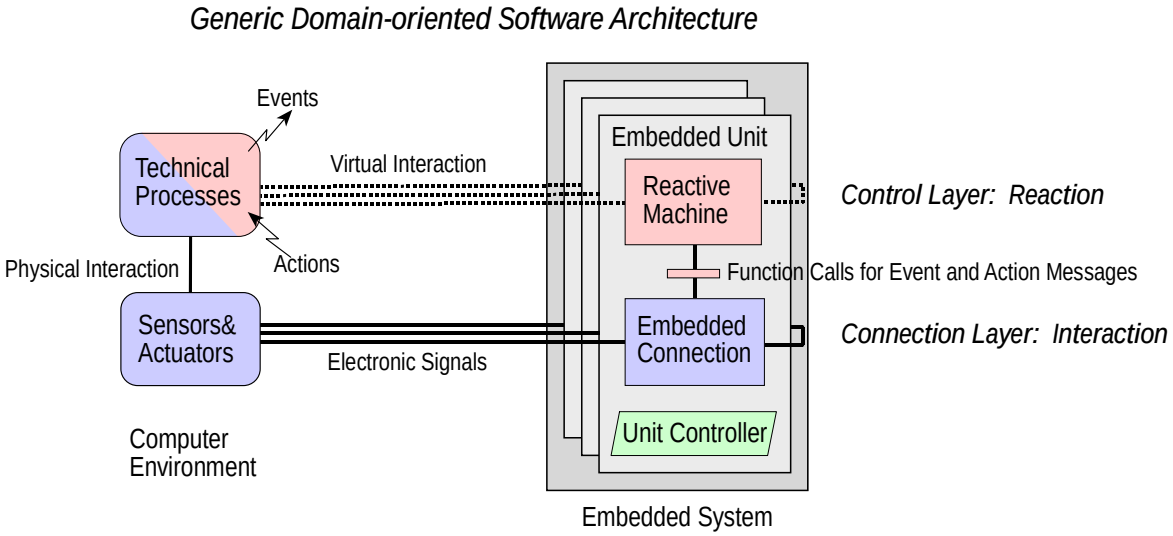


Abb. 9: Domänenorientierte Software-Architektur eines 1-Prozessor-Systems

Real auftretende Events werden virtuell als *Event Messages* zur *Reactive Machine* übertragen, und die durch *Reactive Machine* erzeugten *Action Messages* erzeugen virtuell bei den technischen Prozessen die entsprechenden realen Aktionen. Die Steuerfunktionen können unabhängig von der Einbettungstechnologie in der Steuerschicht (*Control Layer*) als Reaktive Maschine entwickelt werden, weil nur die Spezifikation der virtuellen Interaktionsverbindung bekannt sein muss. Die virtuelle Interaktion selbst ist durch die Hardware- und Software-Komponenten der Verbindungsschicht (*Connection Layer*) zu realisieren.

Embedded Units sind nebenläufige Softwarekomponenten, die eine *Reactive Machine*, die entsprechende Verbindungssoftware (*Embedded Connection*) und einen *Unit Controller* als Ausführungssteuerung enthalten. Der Informationsaustausch zwischen Embedded Units erfolgt über sequentielle Message Passing Kanäle, die auch als virtuelle Verbindung spezifiziert werden.

Die Steuerfunktionen werden unabhängig von der Einbettungstechnologie in der Steuerschicht (*Control Layer*) als Reaktive Maschine entwickelt, indem nur die Spezifikation der virtuellen Interaktionsverbindung verwendet wird. Die virtuelle Interaktion selbst ist durch die Hardware- und Software-Komponenten der Verbindungsschicht (*Connection Layer*) realisiert.

Die generische Systemarchitektur bestimmt auch den Ablauf des Entwicklungsprozesses. Sobald die Virtuelle Interaktion spezifiziert ist, können die *Reaktiven Maschinen* und die *Embedded Connection* in parallelen Entwicklungsthreads unabhängig voneinander entwickelt werden.

Auswirkung auf das Scheduling

Die Systemstruktur mit den nebenläufigen Embedded Units, die selber als domänenorientierte Embedded Systeme wirken, wird entsprechend zu einer zweistufig hierarchischen Scheduling Struktur für das Gesamtsystem führen. Sobald nämlich mehrere *Embedded Units* im Spiel sind, wird das Scheduling der anstehenden Ereignismeldungen auf zwei zeitlichen Ebenen stattfinden, der zeitlichen Ebene des Prozessors und der zeitlichen Ebene der einzelnen Embedded Units. Auf der Prozessorebene muss dafür gesorgt werden, dass bei der Zuteilung der Prozessorzeit an die verschiedenen *Embedded Units* bestimmte Vorgaben eingehalten werden. Diese Aufgabe kann z. B. durch das Multitasking eines *Real-Time Operating System* (*RTOS*) übernommen werden.

Auf der zeitlichen Ebene einer *Embedded Unit* wird das Scheduling für die Ausführung der Ereignismeldungen aufgrund der garantierten Prozessorbenutzungszeit realisiert. Diese Aufgabe ist als system-spezifisches Software-Engineeringproblem zu lösen, weil hier eine deterministische non-preemptive Ausführung der Reaktiven Maschine und die Einhaltung vorgegebener Zeitschranken für die Verarbeitung bestimmter Ereignisse garantiert werden muss.

2.6 Der domänenorientierte Entwicklungsprozess

Zum Software-Entwicklungsprozess

Die Produkte eines Software-Entwicklungsprozesses sind *Systembeschreibungen*. Die ersten Beschreibungen im Entwicklungsprozess sind meistens Systemanforderungen sowie Beschreibungen der Domänen der Embedded System Umgebung, auf die sich die Anforderungen beziehen. Die letzte produzierte Systembeschreibung ist immer der ausführbare binäre Code, er ist das Endprodukt. Dazwischen sind verschiedene intermediäre Systembeschreibungen notwendig wie Spezifikationen, Modelle, Entwürfe und selbstverständlich auch der Quellcode. Unterschiedliche Systembeschreibungen sind notwendig, weil damit verschiedene Aspekte aus unterschiedlichen Sichten dargestellt werden können. Aus informalen Beschreibungen entstehen im Entwicklungsprozess typischerweise formalere, und aus abstrakten Darstellungen konkretere Beschreibungen. Formale Beschreibungen können auch automatisiert in andere Beschreibungen umgesetzt werden, wie zum Beispiel Verhaltensmodelle in generierten Quellcode, oder, heute selbstverständlich, Quellcode in erzeugten binären Objektcode.

Beschreibung des Software-Entwicklungsprozesses durch Abhängigkeitsgrafen

Es stellt sich die Frage, wie der Entwicklungsprozess selbst zu beschreiben ist. Es sind schon viele Darstellungen vorgeschlagen worden, die aber die meisten zu allgemein gehalten sind. Für Embedded Systeme sollte der Entwicklungsprozess anwendungsspezifisch formuliert werden können.

Wir wählen hier eine Darstellung mittels *Abhängigkeitsgrafen*. Solche Abhängigkeitsgrafen legen die im Entwicklungsprozess zu erstellenden Systembeschreibungen fest und definieren gleichzeitig die Abhängigkeiten zwischen diesen Beschreibungen. Der grosse Vorteil dieser Darstellung ist, dass die Struktur des Entwicklungsprozesses anwendungsspezifisch festgelegt werden kann.

Abhängigkeitsgraf eines Entwicklungsprozesses

Azyklischer gerichteter Graf von Systembeschreibungen

Ein Abhängigkeitsgraf umfasst folgende Elemente:

Die Knoten eines Abhängigkeitsgrafen stellen *Systembeschreibungen* dar.

Eine gerichtete Verbindung von A nach B bedeutet, das die Beschreibung A von der Beschreibung B abhängig ist.

Der Graf darf keine Zyklen haben (Verhindern von Huhn-Ei-Problemen).

Ein Knoten kann auch einen Teilprozess des Entwicklungsprozess darstellen. Der Teilprozess wird durch einen weiteren Abhängigkeitsgrafen dargestellt.

Die iterative Dynamik des Software-Entwicklungsprozesses

Wenn eine Systembeschreibung verändert wird, muss in allen abhängigen System-Beschreibungen mit Änderungen gerechnet werden (Versionen).

Ein fundamentaler Zusammenhang

Die Struktur des Software-Entwicklungsprozesses ist durch die Struktur der Software-Architektur bestimmt:

Die Software-Architektur definiert die Interaktionsverbindungen zwischen den Softwarekomponenten und damit die Abhängigkeiten der entsprechenden Beschreibungen.

Schichtenarchitekturen ermöglichen aufgrund der unterstützten Abstraktionen parallele Abläufe im Entwicklungsprozess.

Der domänenorientierte Entwicklungsprozess der DESC-Methode

Abbildung 10 zeigt den Abhängigkeitsgrafen des generischen DESC-Entwicklungsprozesses für ein Embedded System, das aus mehreren Prozessoren mit mehreren *Embedded Units* besteht.

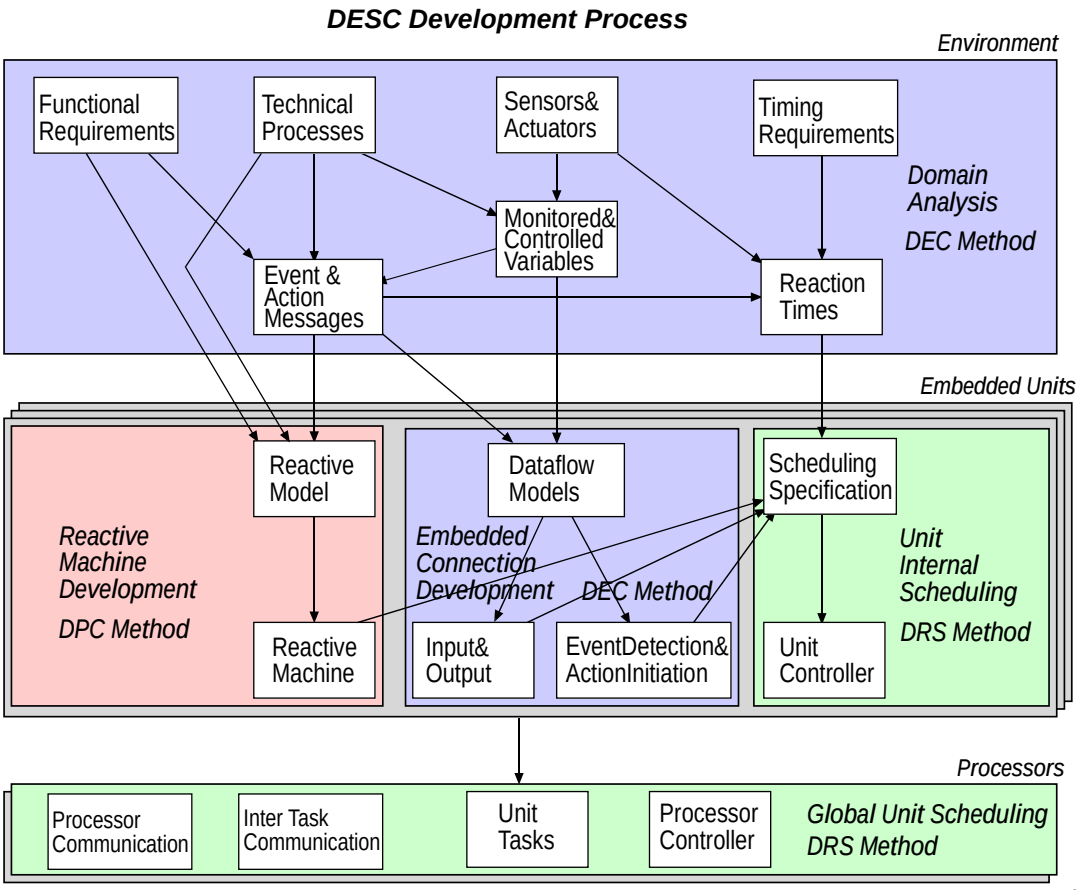


Abb. 10: Domänenorientierter Entwicklungsprozess

Die Entwicklung beginnt mit der *Domain Analys*is, in welcher die Voraussetzung für die generische Aufteilung in die Teilprozesse *Reactive Machine Development*, *Connection Development* und *Scheduling Development* geschaffen wird (Separation of Concerns). Die Spezifikation der *Monitored&Controlled Variables* und davon abhängig die Festlegung der *Event&Action Messages* definieren die virtuelle Verbindung zwischen den technischen Prozessen und den Reaktiven Maschinen, was die Aufteilung in *Reactive Machine* und *Connection Development* bestimmt. Weiter kommt in der *Domain Analys*is die Spezifikation der *Reaction Times* für das Real-time Scheduling dazu.

Die Entwicklung der *Reactive Machine* einer Embedded Unit erfolgt durch Modellierung der Steuerfunktionen (*Reactive Model*) als kooperierende Zustandsmaschinen. Der Source Code der *Reactive Component* wird automatisch erzeugt.

Die Implementierung der virtuellen Verbindung einer *Reactive Machine* mit den technischen Prozessen ist in einem parallelen Entwicklungsprozess zu realisieren. Dabei werden Interaktionsfunktionen implementiert für den Zugriff auf die I/O-Schnittstellen (*Input&Output*) und für die Detektion von Ereignissen und die Iniziierung von Aktionen (*EventDetection&ActionInitiation*). Vorgängig sind in einem ersten Schritt mit Datenflussmodellen (*Dataflow Models*) die Schnittstellen und die Interaktionspfade für die Interaktionsfunktionen zu definieren.

Die zeitliche Ausführung der Reaktiven Maschine und der Interaktionsfunktionen durch den *Unit Controller* einer Embedded Unit wird in einem weiteren Teilprozess festgelegt, der von den spezifizierten *Reaction Times* und den Ausführungszeiten der implementierten Software abhängt.

Die Embedded Units auf den verschiedenen Prozessoren werden schliesslich als Real-time Tasks installiert *(Unit Tasks*) und in der Regel RTOS-basiert durch einen *Processor Controller* ausgeführt.

Anhang 1 - Beispiel für ein Problem Frame

Das Software Werkstück Problem

Die Aufgabe ist einen einfachen Editor zu entwickeln, mit welchem formatierte Texte oder einfache Graphiken erzeugt werden können. Das *Software Werkstück Problem Frame* stellt in *Abbildung 11* die entsprechende Problemstruktur dar.

Problem Frame - Software Werkstück Problem

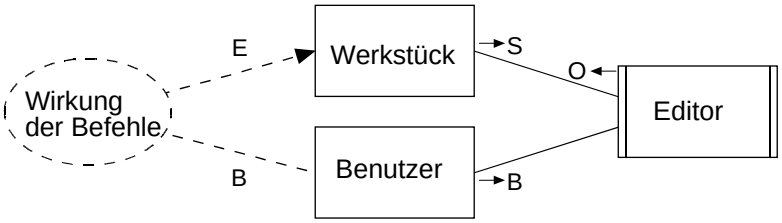


Abb. 11: Problem Frame - Werkstück-Problem

Die zu entwickelnde ‘Maschine’ ist der *Editor.* Das Problem Frame hat zwei Problemdomänen, der *Benutzer* des Editors und das *Werkstück* (Software-Produkt). Das Software Werkstück Problem Frame ist etwas speziell, weil ein *Werkstück*, das ja Teil der realen Welt des Benutzers ist, aus Software besteht. Die Anforderungen *Wirkung der Befehle* beschreiben, welche Effekte (E) die Benutzer-Befehle (B) im editierten Werkstück haben sollen. Der Editor ist durch diese Befehle getrieben und erzeugt entsprechende Änderungen im Werkstück, indem er Operationen aus O ausführt. Um die Änderungen korrekt auszuführen, muss er den aktuellen Zustand (S) des Werkstücks inspizieren können. Speziell an dieser Problemstruktur ist, das in den Anforderungen direkt die Schnittstellen-Phänomene B zur Maschine verwendet werden (das ist oft so bei SW-Produkten, weil der Entwickler die Schnittstellen zum Rechner definieren kann).

Die ***Aufgabe*** ist unter Benutzung der Verbindungen zum Benutzer und zum Werkstück einen Editor zu konstruieren, der für eingegebene Befehle am Werkstück die angeforderte Wirkung erzeugt.

Bei einem Text-Editor gibt es in B zum Beispiel einen Befehl *newLine*. Die Anforderungen verlangen, dass auf diesen Befehl im editierten *Werkstück* (Text) eine neue Zeile erzeugt wird. Im Fall, dass die aktuelle Zeile die 80-ste der Seite ist, muss aber zuerst ein Seitenumbruch eingefügt werden. Die erzeugten Operationen sind dann entweder *line feed*, *carriage return,* oder *line feed, form feed*, *carriage return.* Die Entscheidung wird aufgrund des Wertes der aktuellen Zeilennummer des *Werkstücks* (S) getroffen.

Anhang 2 - Anforderungsspezifikationen für Softwaressysteme

Für die Entwicklung einer Lösung eines Softwareproblems sind verschiedene Systembeschreibungen sinnvoll, die unterschiedliche Aspekte beleuchten. Das gilt auch für die Entwicklung von Lösungen von Teilproblemen. Zu oft werden solche Beschreibungen nicht oder nur skizzenhaft erstellt. Ohne solche Systembeschreibungen ist es jedoch kaum möglich, den iterativen Entwicklungsprozess in den Griff zu bekommen. Folgende drei Beschreibungstypen haben direkt mit der Problemstellung zu tun:

D - Beschreibung der Problemdomänen

Damit das gestellte Problem überhaupt verstanden werden kann, ist eine Beschreibung der betroffenen Welt, d. h. der Problemdomänen notwendig. Solche Beschreibung sind in der Regel informal.

A - Anforderungen

Die Anforderungen beschreiben das Verhalten des zu entwickelnden Programms aus der Sicht des Anwenders. Bei der Formulierung solcher Anforderungen werden immer Objekte und Phänomene der realen Problemdomänen zueinander in Beziehung gesetzt. Die Anforderungen beschreiben nicht das zu erstellende Programm, sondern den Nutzen, den es in seiner Umgebung haben muss. Auch Anforderungen sind in der Regel informale Beschreibungen.

P - Das gesuchte Programm

Das Programm oder Unterprogramm (Source Code) ist eine formale Systembeschreibung, die das Verhalten des programmierten Computers spezifiziert. Sie ist das Endprodukt. (Eigentlich ist der automatisch erzeugte ausführbare Binärcode das Endprodukt).

Mit dem Informationsgehalt der Anforderungen und der Beschreibung der Problemdomänen ist es möglich, das gesuchte Programm zu entwickeln und zu verifizieren. Das ist aber meistens sehr schwierig, und es muss mit vielen Entwicklungsfehlern gerechnet werden. Der Hauptgrund ist der folgende: Die Anforderungen beziehen sich auf die Umgebung, für das Programm hört die Welt aber bei den Eingaben und Ausgaben auf. Mit andern Worten, es fehlt eine Beschreibung, die das Verhalten des Programms in Bezug auf die Ein- und Ausgaben beschreibt, gewissermassen eine Anforderungsbeschreibung für den Programmierer. Das ist die Anforderungsspezifikation:

S - Spezifikation

Die Spezifikation (genauer Anforderungsspezifikation) beschreibt das erwünschte Verhalten eines Programms in Bezug auf dessen technische Schnittstelle. Bei sequentiell auftretenden Eingaben zum Beispiel muss für jede gültige Sequenz von Eingaben spezifiziert sein, welche Ausgaben durch das Programm erzeugt werden müssen, oder bei zyklisch gelesenen Input-Variablen sind Bedingungen (Prädikate) für das zu berechnende Resultat zu formulieren. Spezifikationen können sowohl informale als auch formale Beschreibungen sein, das hängt stark von der Problemstellung ab. Eine formale Spezifikation für ein sequentielles Programm sollte für jede mögliche Eingabesequenz definieren, ob sie gültig ist, und für jede gültig Eingabesequenz die erzeugte Ausgabesequenz spezifizieren. Solche sog. *extensionale* Spezifikationen sind aber selten machbar ist, weil es in der Regel sehr viele Eingabesequenzen gibt.

Mit *Zustandsmaschinen* ist für die Spezifikation reaktiver Systeme aber ein machbarer Weg gefunden worden. Eine deterministische Zustandsmaschine definiert die gültigen Eingabesequenzen und die erzeugten Ausgabesequenzen als formales Verhaltensmodell. Solche Verhaltensmodelle werden als *intensionale* Spezifikationen bezeichnet. Intensionale Spezifikationen sind nicht mehr als reine Beziehungen zwischen Input- und Outputsequenzen formuliert, sondern es werden zusätzlich interne Zustände (States) festgelegt. Das Verhalten wird mittels Zustandsübergängen definiert, die zustandabhängig spezifizieren, welche Eingabe (Event) welchen Ausgabe (Action) erzeugt.

Die minimalen Phasen für die Konstruktion eines Programmes

1. Ausgehend von D und A kann S erstellt werden.
2. Aufgrund von S kann P entwickelt oder generiert werden.