

5. SPEZIFIKATIONSSPRACHEN

5.1 Eigenschaften von Spezifikationssprachen

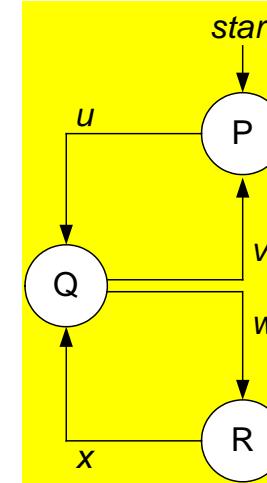
- Beschreiben die gewünschte **Funktionalität** (was?), nicht die Implementierung (wie?).
- Verschiedene **Abstraktionsebenen** (z. B. Hardware: Bauelemente-Ebene, Gatterebene, Register-Transfer-Ebene, Systemebene).
- Sollten für das Anwendungsgebiet geeignete, intuitive **konzeptionelle Modelle** unterstützen.
- Sollten die Basis für **Simulation** des Systemverhaltens bilden (*executable specifications*).
- Sollten eine weitgehend automatische **Synthese** der Implementierung erlauben (heute nur in speziellen Ausnahmefällen möglich).
- Sollten die Basis für die **Dokumentation** des Systemverhaltens bilden.

⇒ **Formale Spezifikationssprachen, Umgangssprache ungeeignet.**

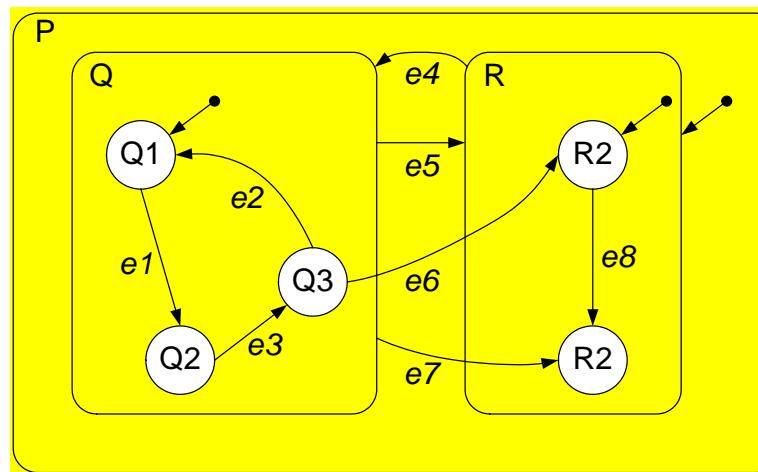
Anforderungen an Spezifikationssprachen für eingebettete Systeme

- Zustandsübergänge (State Transitions):
Eingebettete Systeme weisen typischerweise Zustände und Zustandsübergänge auf.

⇒ *Unterstützung zustandsorientierter konzeptioneller Modelle.*

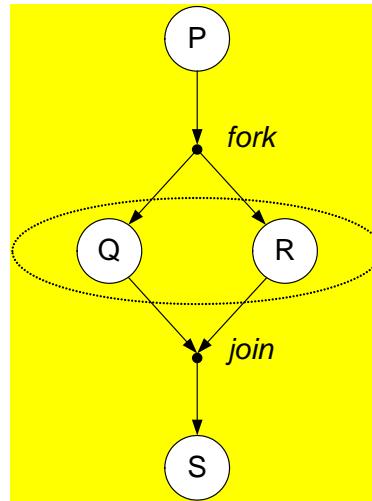


- Verhaltenshierarchien (Behavioral Hierarchies):
Oft komplexes Verhalten, das *hierarchisch* beschreibbar sein muss, um überschaubar zu bleiben.



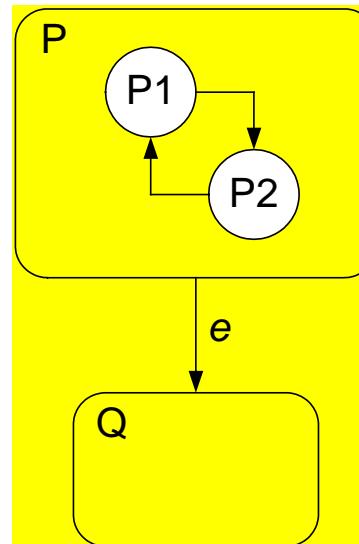
- Nebenläufigkeit (Concurrency):

Meist nebenläufige Verhalten, die beschreibbar sein müssen.



- Ausnahmen (Exceptions):

Manche (Ausnahme-) Ereignisse erfordern sofortige Behandlung, unabhängig vom momentanen Zustand des Systems.



Beispiel: Bei Auftreten von Ereignis e geht das System aus P sofort in den Zustand Q über (unabhängig vom Unterzustand P).

Wünschenswerte Eigenschaften

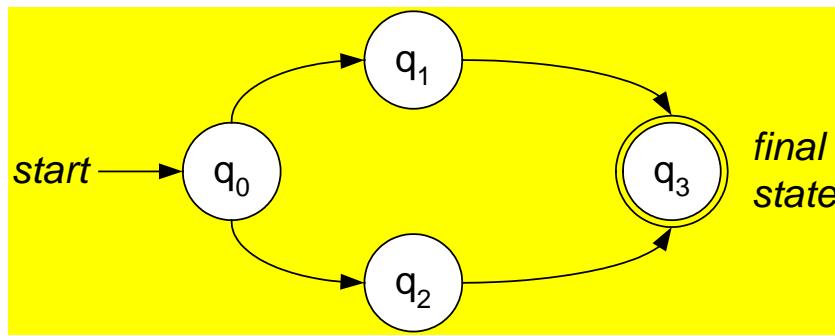
- Programmierkonstrukte (Programming Constructs):

Manche Teilberechnungen lassen sich gut direkt in einer Programmiersprache formulieren.

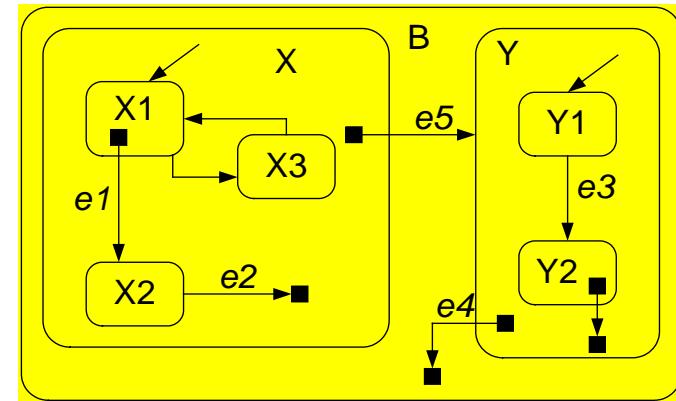
⇒ *Programmierkonstrukte sollten einbettbar sein.*

- Verhaltensterminierung (Behavioral Completion):

Die Terminierung von Teilverhalten sollte klar spezifizierbar sein.



(a) FSM



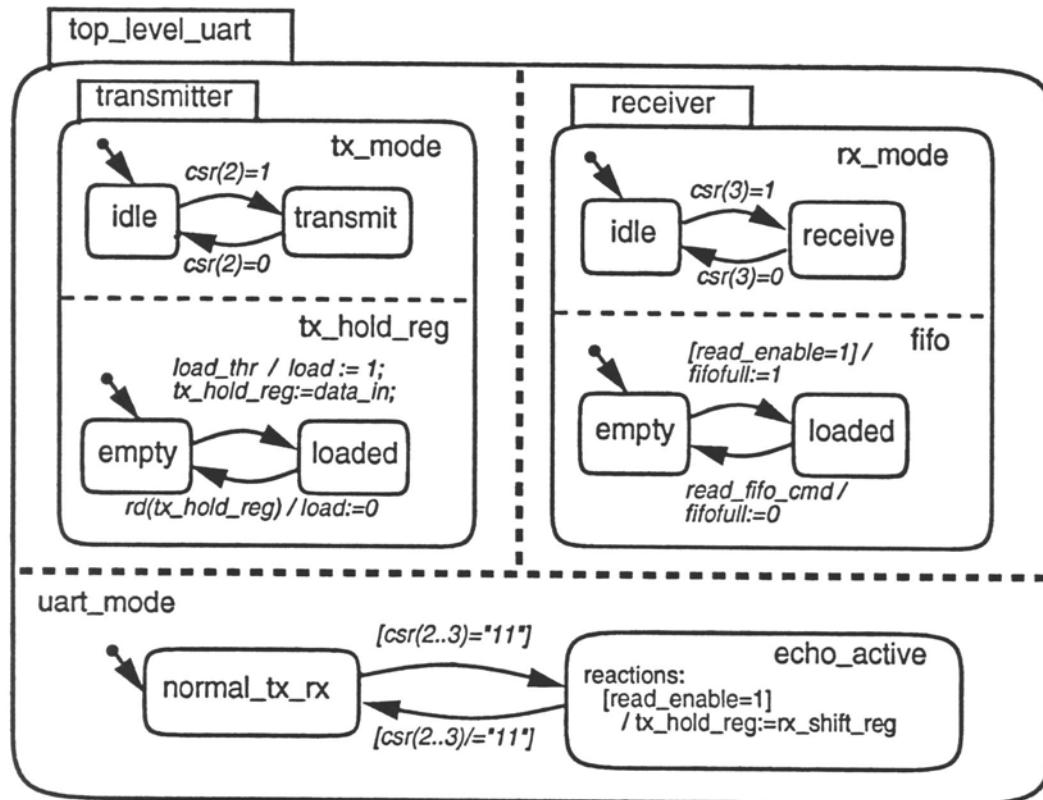
(b) PSM

- Objektorientierung

Eine objektorientierte Modellierung sollte unterstützt werden.

5.2 Zustandsbasierte Spezifikationssprachen: Beispiel Statecharts

Konzipiert für ereignisgetriebene, kontroll-dominierte Systeme (reaktive Systeme).



- Verhaltenshierarchie

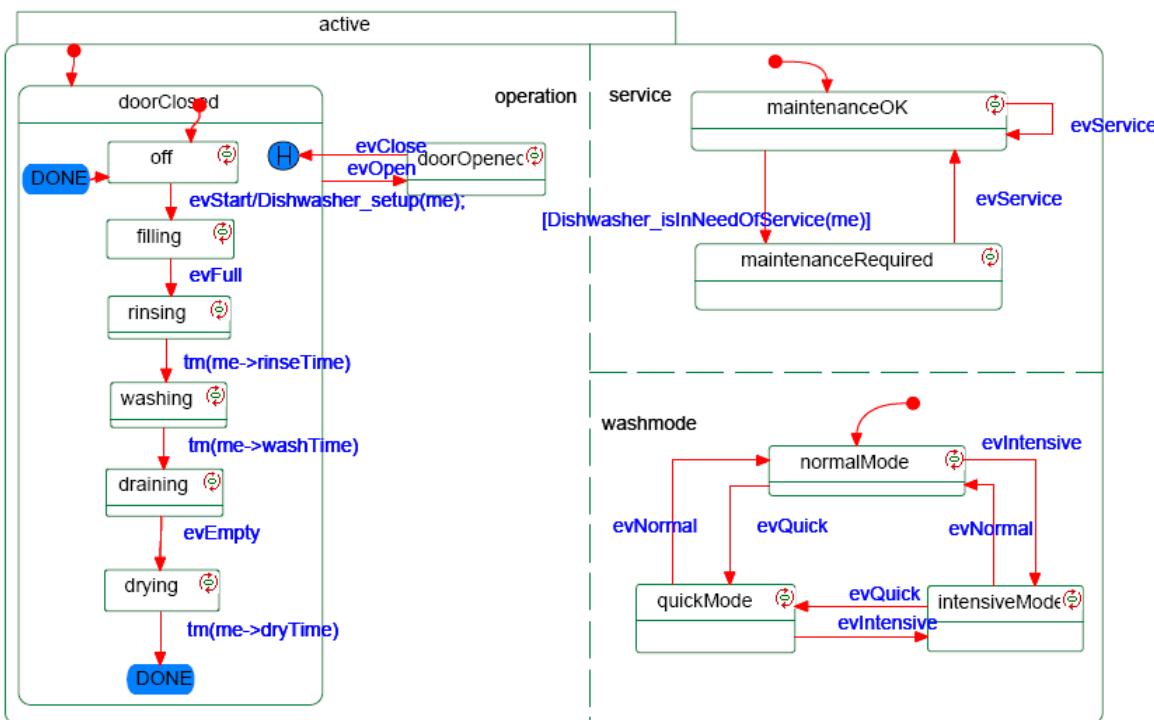
ODER-Dekomposition (sequentiell): Teilzustände mit Übergängen, von denen immer genau einer angenommen wird (hierarchische FSMs, dargestellt durch geschachtelte Kästen).

AND-Dekomposition (nebenläufig): Orthogonale Teilzustände, die gleichzeitig angenommen werden (nebenläufige FSMs, dargestellt durch gestrichelte Linien im Diagramm).

- ⇒ Keine Zustandsexplosion durch nebenläufige Verhalten
- ⇒ Hierarchische Verhaltensstrukturierung

5.2 Zustandsbasierte Spezifikationssprachen: Beispiel Statecharts

Konzipiert für ereignisgetriebene, kontroll-dominierte Systeme (reaktive Systeme).



ODER-Dekomposition
(sequentiell):

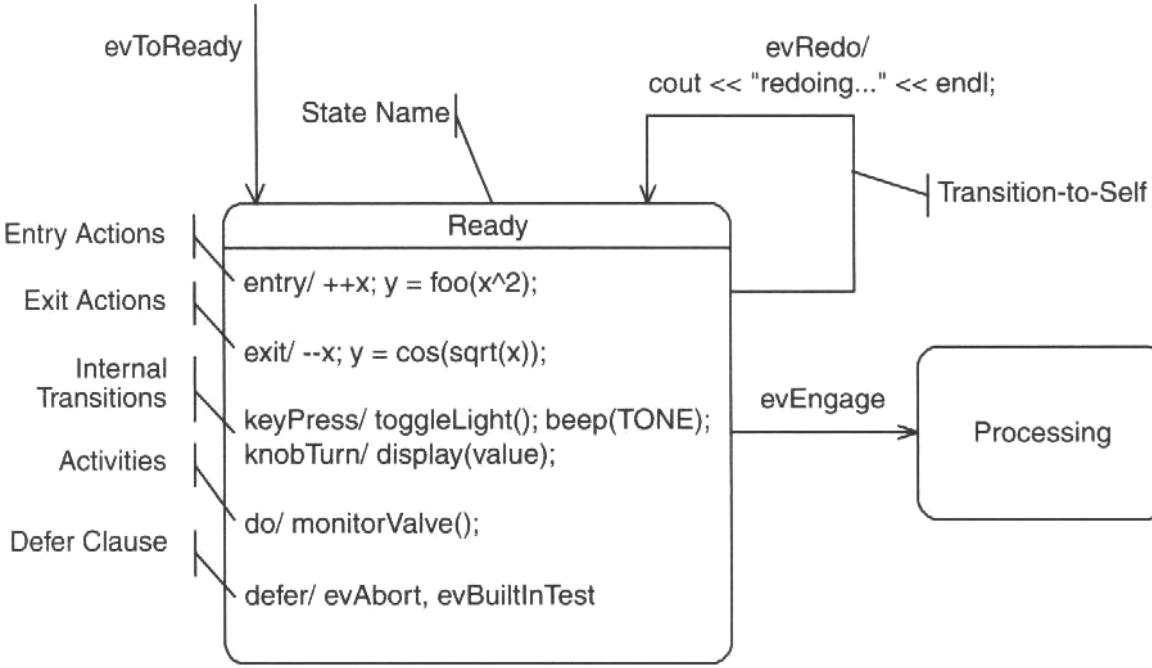
Teilzustände mit Übergängen, von denen immer genau einer angenommen wird (hierarchische FSMs, dargestellt durch geschachtelte Kästen).

AND-Dekomposition
(nebenläufig):

Orthogonale Teilzustände, die gleichzeitig angenommen werden (nebenläufige FSMs, dargestellt durch gestrichelte Linien im Diagramm).

- ⇒ Keine Zustandsexplosion
- ⇒ Hierarchische Verhaltensstrukturierung

Generell mögliche Aktionen in Zuständen:



Reihenfolge der Ausführung bei Event evRedo:

- Exit Action
- Transition Action (Static Reaction)
- Entry Action

Aktionen werden in Zielsprache formuliert (z. B. C, C++), ggf. unter Nutzung von Makros und Funktionen.

Alle Aktionen **atomar**, d. h. bei Eintreten eines Ereignisses nicht unterbrechbar.

Ausnahme: do-Activity (z. B. Kontrollfluss-Diagram)

Defer-Ereignisse werden gemerkt und an Folgezustände weitergereicht, bis es eine passende Transition gibt, die dann ausgeführt wird.

Aufbau einer Transition:

event-name '('parameter-list')' '['guard-expression']' '/' action-list

mit

| Field | Description |
|-------------|---|
| Event name | The name of the event triggering the transition. |
| Parameters | A comma-separated list containing the names of data parameters passed with the event signal. |
| Guard | A Boolean expression that must evaluate to TRUE for the transition to be taken. The expression should not have side effects such as assigning values. |
| Action list | A list of operations executed as a result of the transition being taken. These operations may be of this or another object. |

Arten von Ereignissen (Events):

SignalEvent (asynchrone Kommunikation)

CallEvent (Methodenaufruf, synchrone Kommunikation)

TimeEvent (tm < duration >, after < duration >)

ChangeEvent (wird bei Änderung eines Wertes erzeugt, z. B. bei Hardwareschnittstellen)

Guards:

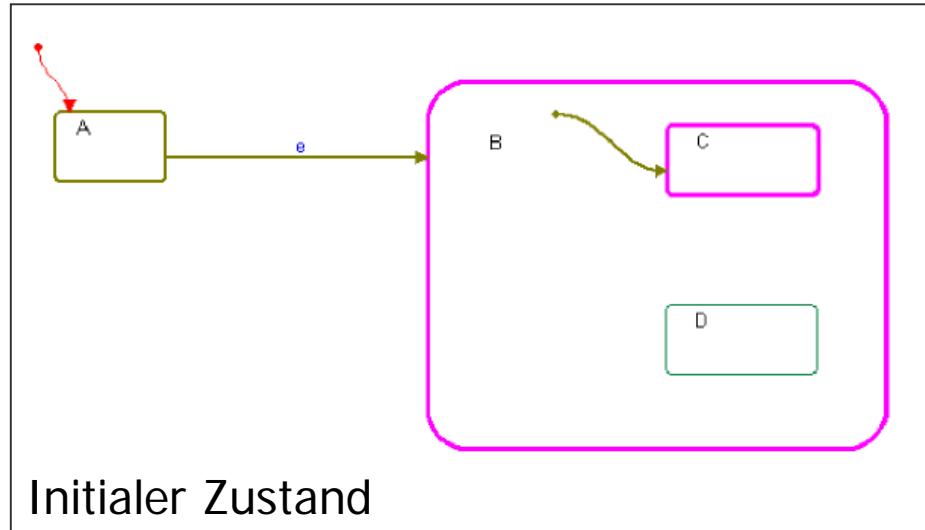
Logische Bedingungen, z. B.:

[X > 0] Dürfen keine Nebeneffekte haben!

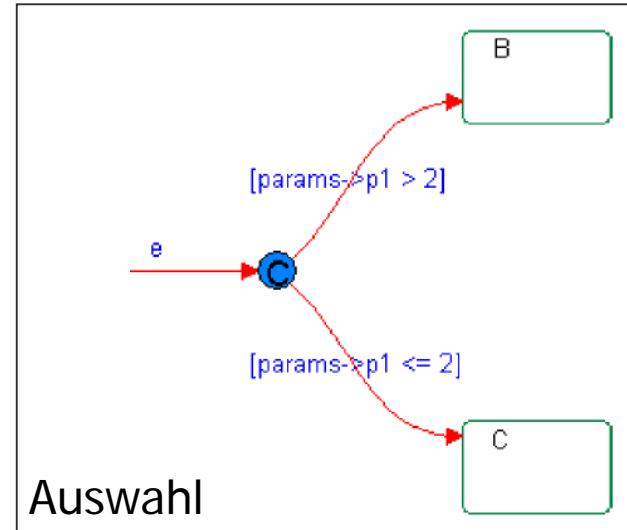
[myLine → IS_IN(Idle)]

TRUE, wenn Objekt unter Pointer myLine im Zustand 'Idle' ist.

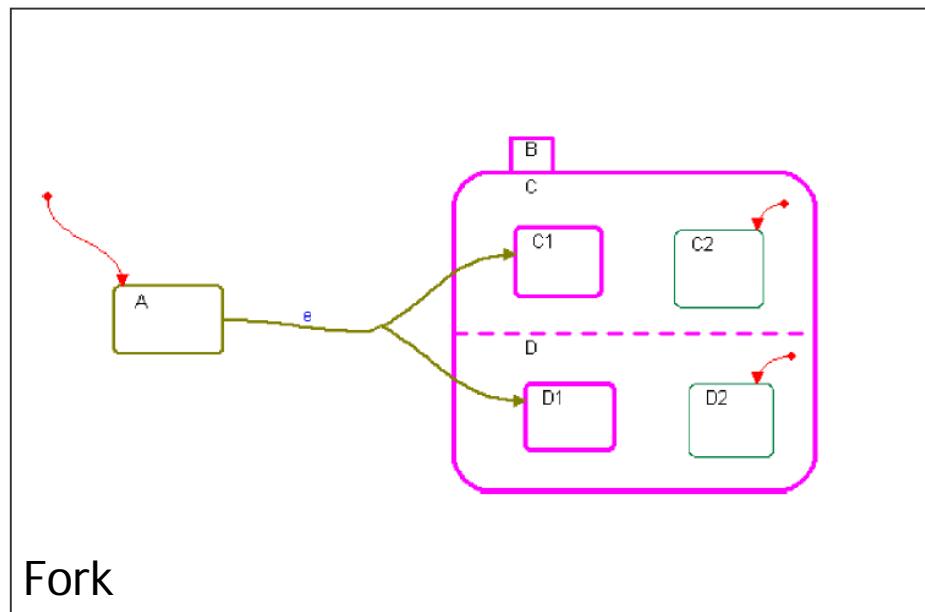
Pseudozustände (Konnektoren etc.)



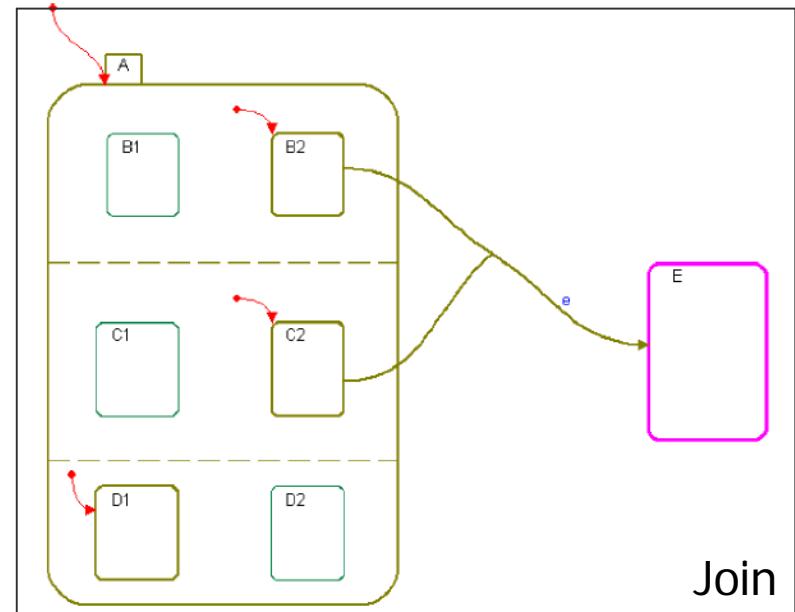
Initialer Zustand



Auswahl

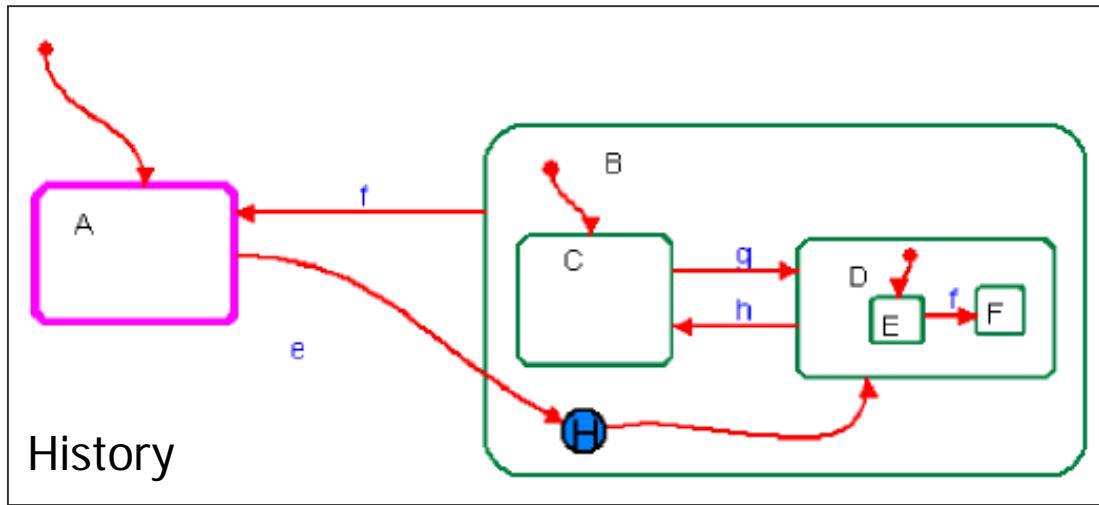


Fork



Join

Pseudozustände (Konnektoren etc.)



5.4 UML (Unified Modeling Language)

Graphische Sprache zur objektorientierten Modellierung.

Vereint verschiedene bisherige Ansätze in einer gemeinsamen Sprache
(Version 1.0: 1997).

Drei Amigos



Grady Booch:

Booch-Methode

James Rumbaugh:

OMT (Object Modeling Technique)

Ivar Jacobson:

OOSE/Objectory (Use Cases)

Standardisierung durch OMG (Object Management Group).

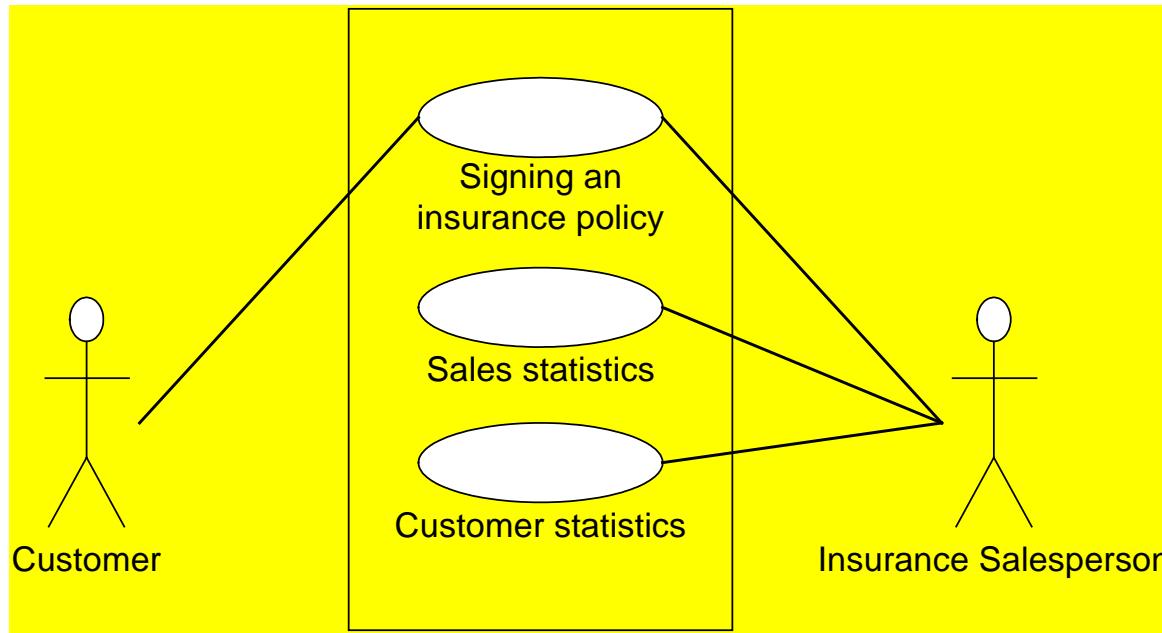
Akzeptanz durch viele bekannte Softwarefirmen.

Besonders für objektorientierte Implementierungssprachen geeignet (C++, Java etc.).

Konzipiert für beliebige (Software-) Systeme, nicht speziell eingebettete Systeme, aber Echtzeiterweiterungen bekannt (Real-Time UML).

5.4.2 UML-Diagramme

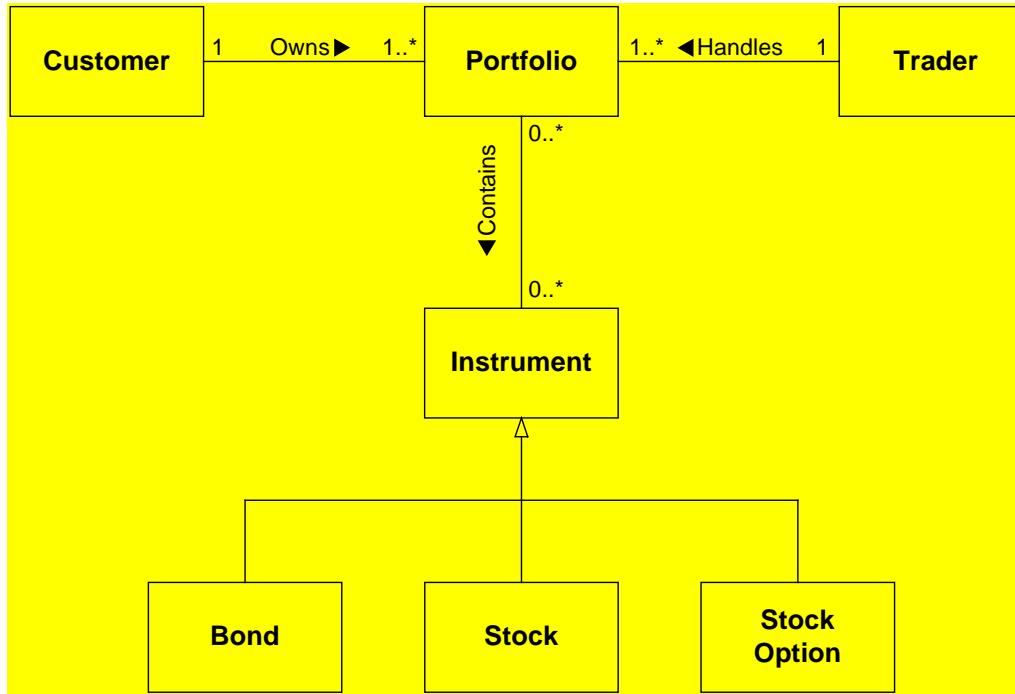
Use Case Diagram



Externe Akteure (Customer, Salesperson) und ihre Verbindung mit dem System über Use Cases (Signing, Sales Statistics, Customer Statistics)

Funktionalität der Use Case selbst aus externer Sicht (äußeres Systemverhalten) wird durch umgangssprachlichen Text oder Activity Diagrams beschrieben.

Class Diagram



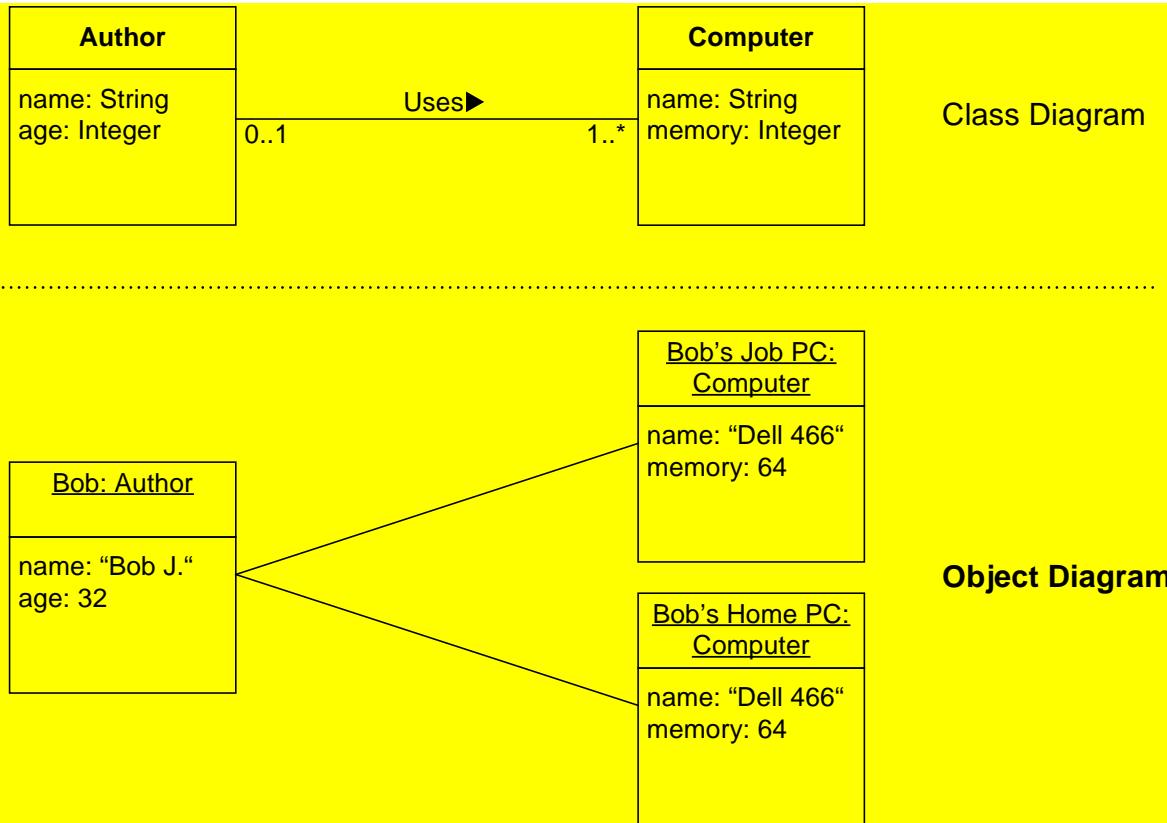
1 Owns ▶ 1..* Bidirektionale Assoziation mit Name **Owns** aus der Perspektive von Klasse **Customer**. Klasse **Customer** kann je exakt 1 Assoziation mit 1 bis beliebig vielen (*) *Portfolios* haben.

Generalisierung: Klassen **Bond**, **Stock** und **Stock Option** sind Subklassen der Klasse **Instrument**.

Statische Struktur der **Klassen** des Systems und ihre Beziehungen untereinander (Assoziation, Abhängigkeit, Spezialisierung, Generalisierung, Aggregation, Komposition).

Gültig während des gesamten Lebenszyklus des Systems.

Object Diagram



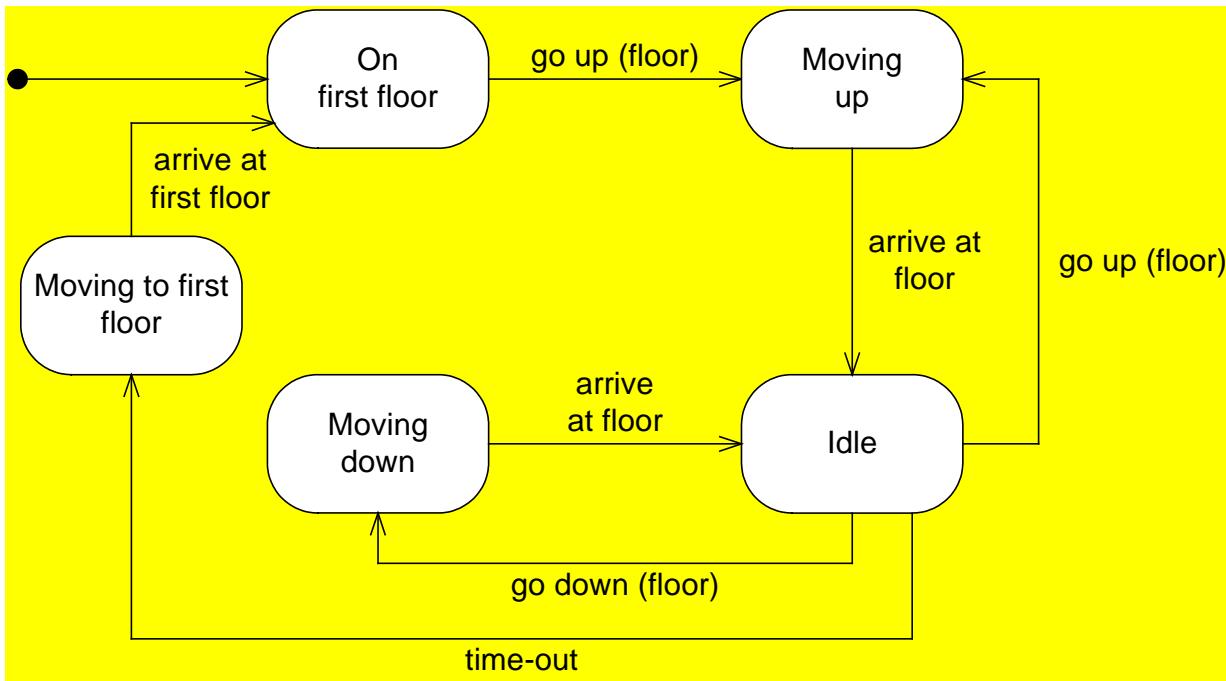
Statische Struktur der Objekte, d. h. *Instanzen* der Klassen und ihre Beziehungen untereinander (s. Class Diagram).

Schnappschuss des Systems zu einem Zeitpunkt.

Können auch in dynamischen Collaboration Diagrams eingesetzt werden.

Beispiel: Objekt Bob als Instanz der Klasse **Author** besitzt („uses“) zwei Objekte Bob's Job PC und Bob's Home PC als Instanzen der Klasse **Computer** (vgl. Class Diagram).

State Diagram



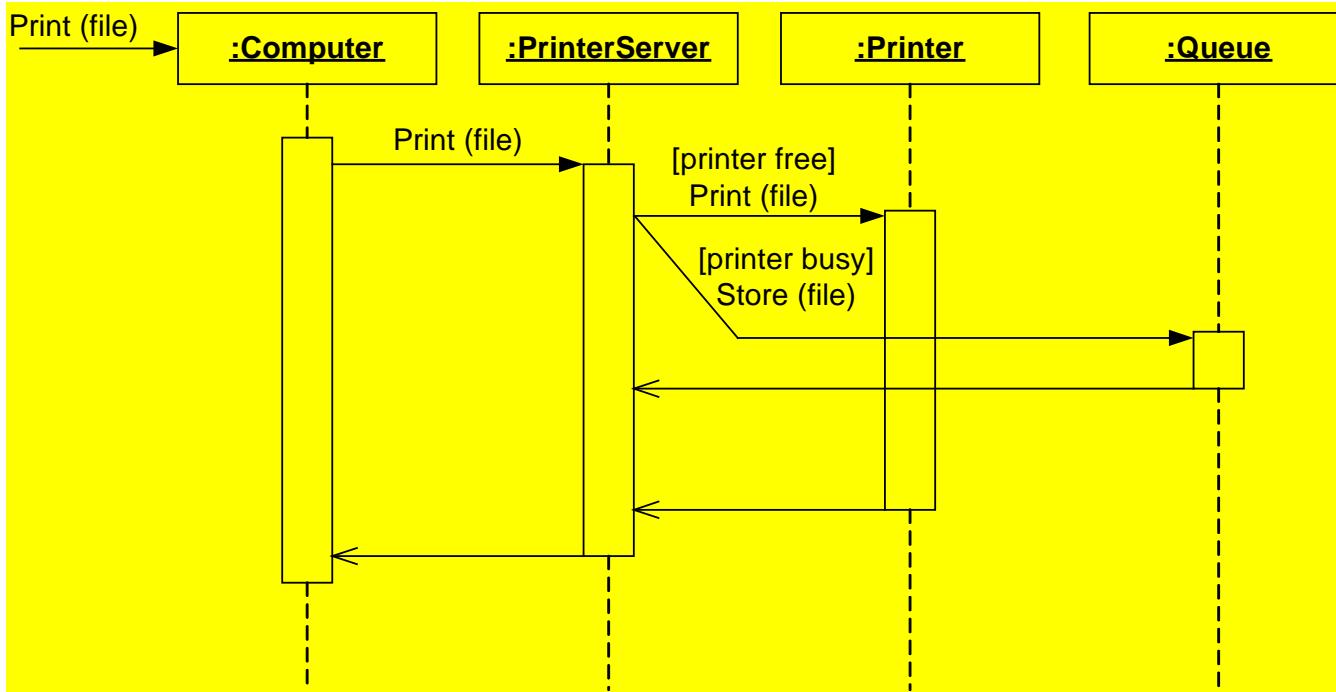
Dynamisches Verhalten von Klassen bzw. Objekten (nicht für alle Klassen sinnvoll).

Zustände (States) und Zustandsübergänge (Transitions). Konzeptionelles Modell des erweiterten endlichen Automaten (siehe StateCharts).

Beispiel: Verhaltensbeschreibung einer einfachen Aufzugssteuerung

Besonders für reaktive, zustandsbehaftete Objekte geeignet.

Sequence Diagram

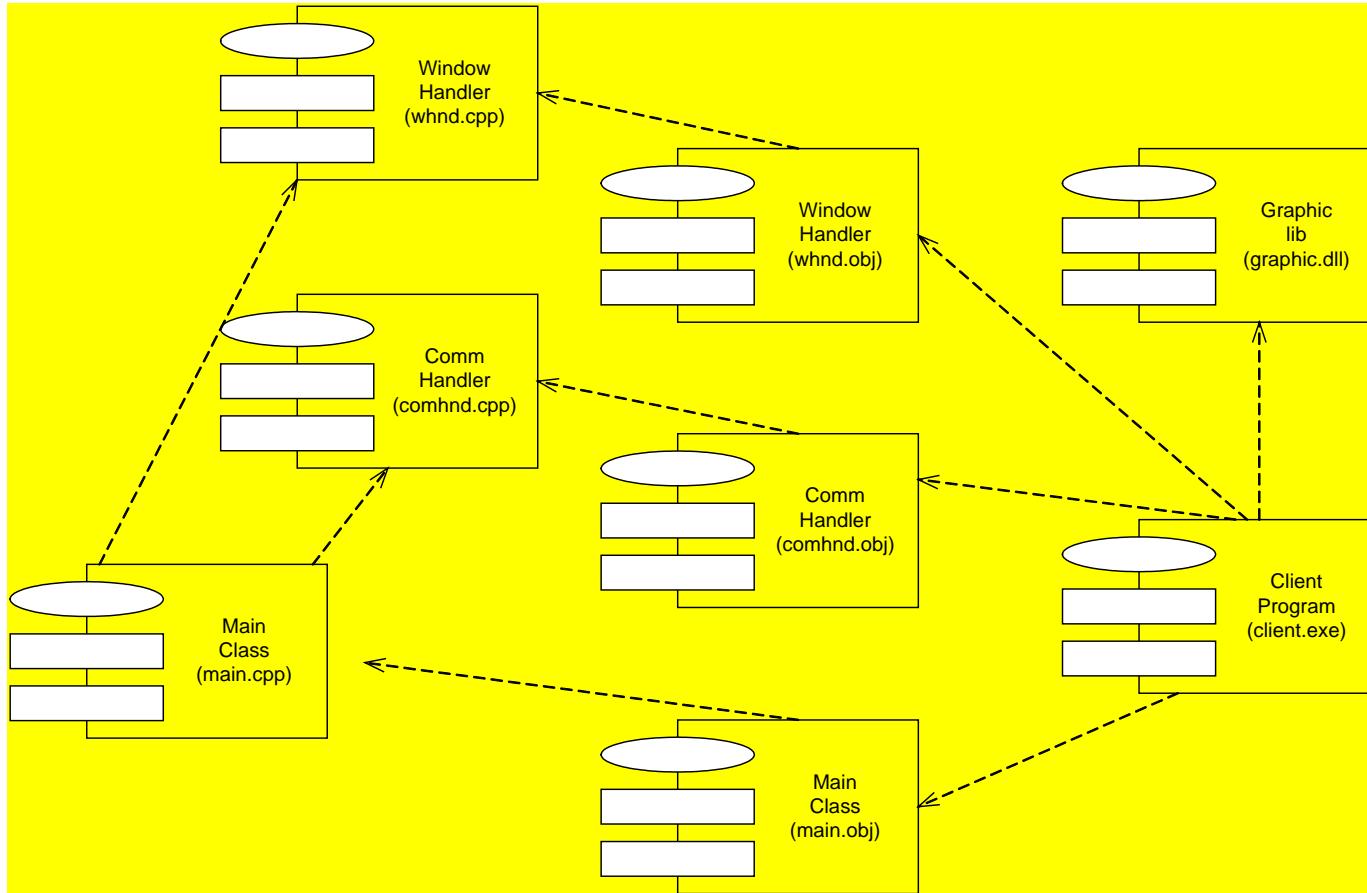


Dynamische Kollaboration zwischen Objekten dargestellt durch Sequenz von Nachrichten zwischen Objekten (vergl. Message Sequence Charts).

Zeitlicher Ablauf von oben nach unten (vgl. Darstellung von Kommunikationsprotokollen).

Besonders für die Spezifikation von Nutzungsszenarien und zur Beschreibung von Testfällen geeignet.

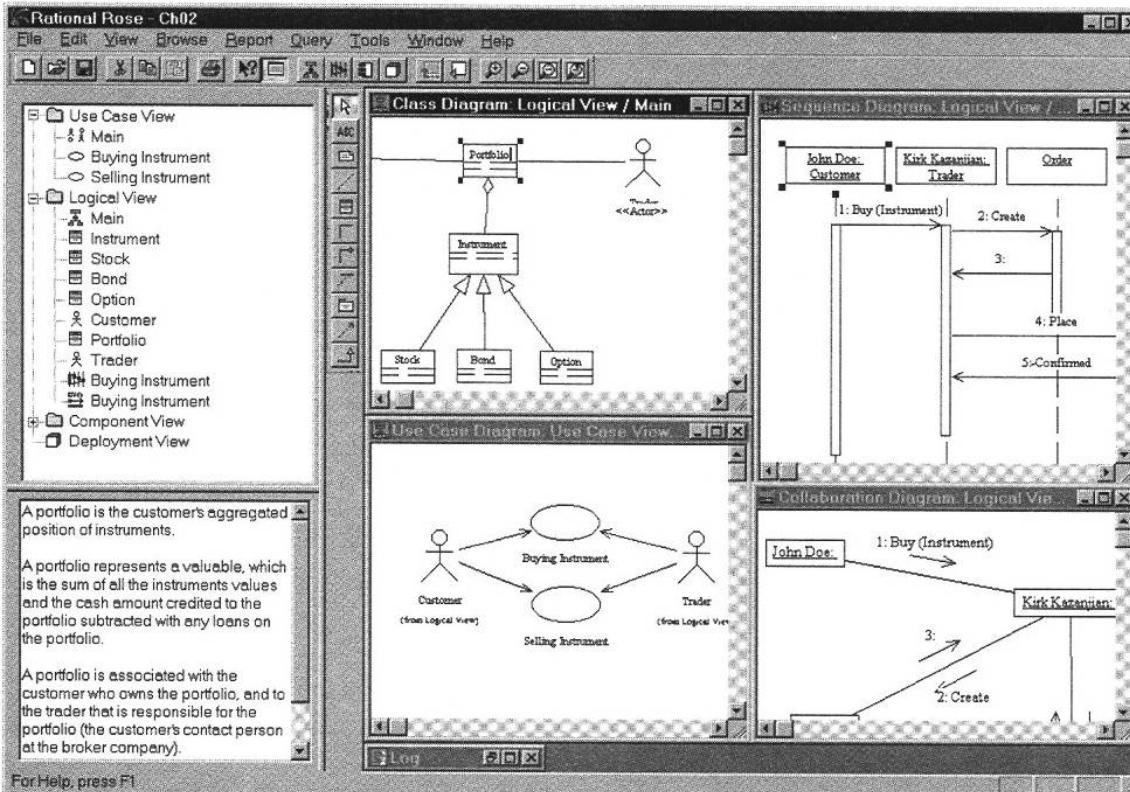
Component Diagram



Physikalische Struktur des **Codes** aus seinen Komponenten (Quellcode, Objektcode, ausführbarer Code).

Legt die Abbildung der logischen Sicht (Klassen) auf die physikalische Sicht (Codekomponenten) fest.

5.4.3 UML-Tools



Graphische Eingabe und systematische Überprüfung der UML-Diagramme.

Konsistenzhaltung zwischen den Diagrammen über gemeinsame Datenbasis (Model Repository).

Multiuser-Unterstützung.

Navigation innerhalb und zwischen Diagrammen.

Generierung von Berichten und Dokumentationen.

Codegenerierung: oft noch recht rudimentär – teilweise nur Schablonen (Skeletons) oder ziemlich ineffizient.

Reverse Engineering (Generierung von Diagrammen aus Code): meist nur begrenzt möglich.

5.4.4 Real-Time UML

Ursprüngliches UML nicht besonders gut für eingebettete Echtzeitsysteme (EES) geeignet, daher 2002 Verabschiedung von

RTP: Real-Time Profile

durch die OMG (Object Management Group, www.omg.org), das Erweiterungen zur Spezifikation von Aspekten wie

- Rechtzeitigkeit (Timeliness)
- Leistung (Performance)
- Scheduling (Schedulability)

enthält.

Weitere UML-Profile, die für EES wichtig sind:

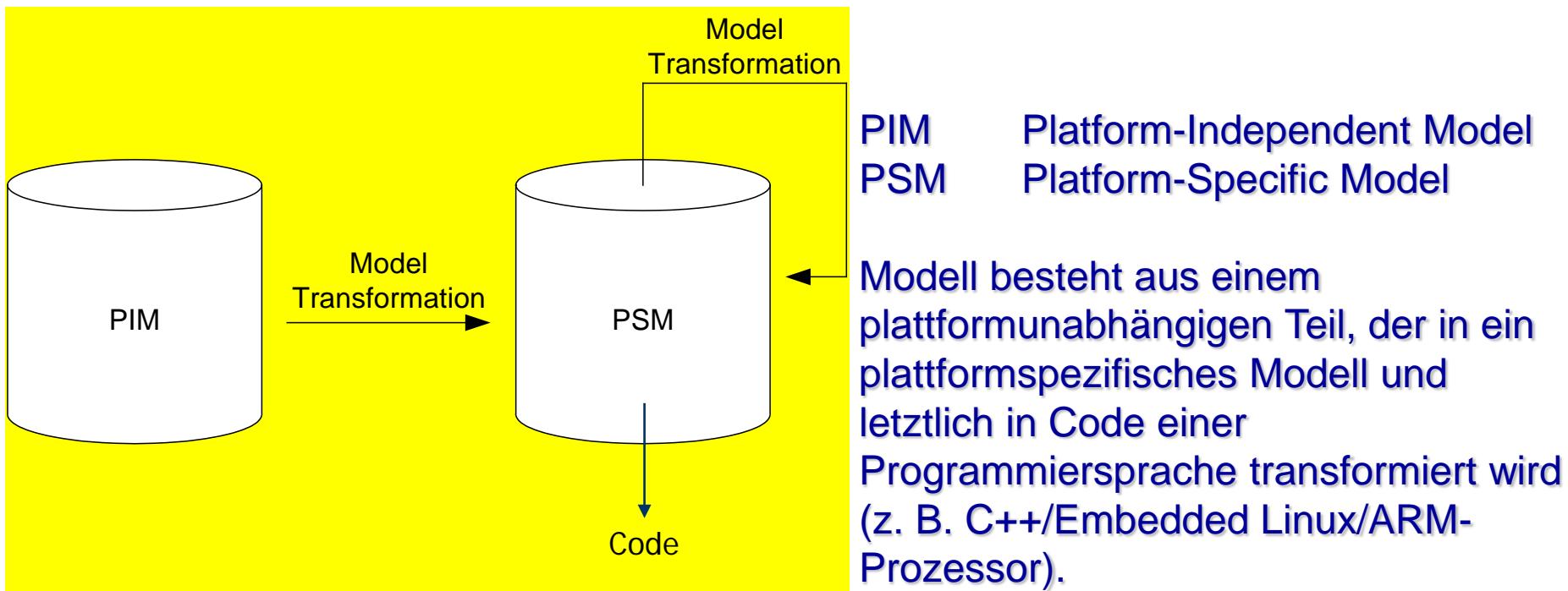
Quality of Service and Fault Tolerance (2002)
System Engineering (2004)

5.5 Real-Time-UML-Werkzeug Rhapsody

5.5.1 Modellierungsansatz

Entwurfswerzeuge wie Rhapsody zur Eingabe der Modelle (UML-Charts) und automatische Codegenerierung und Testunterstützung.

MDA (Model Driven Architecture)



Automatische Codegenerierung aus Diagrammen (C, C++, J, Ada)

Partiell konstruktive Diagramme:

- Use Case Diagrams
- Sequence Diagrams
- (Collaboration Diagrams)

generieren nur Code für einige,
aber nicht alle Elemente des Diagramms.

Voll konstruktive Diagramme:

- Object Model Diagrams
- Statecharts
- Component Diagrams
- (Activity Diagrams)

generieren Code für jedes Element des Diagramms.

Weitere Diagramme und Features kommen mit neuen UML Versionen hinzu (z. B. Timing Diagrams).

Nicht alle Diagramme werden üblicherweise für einen konkreten Entwurf verwendet.

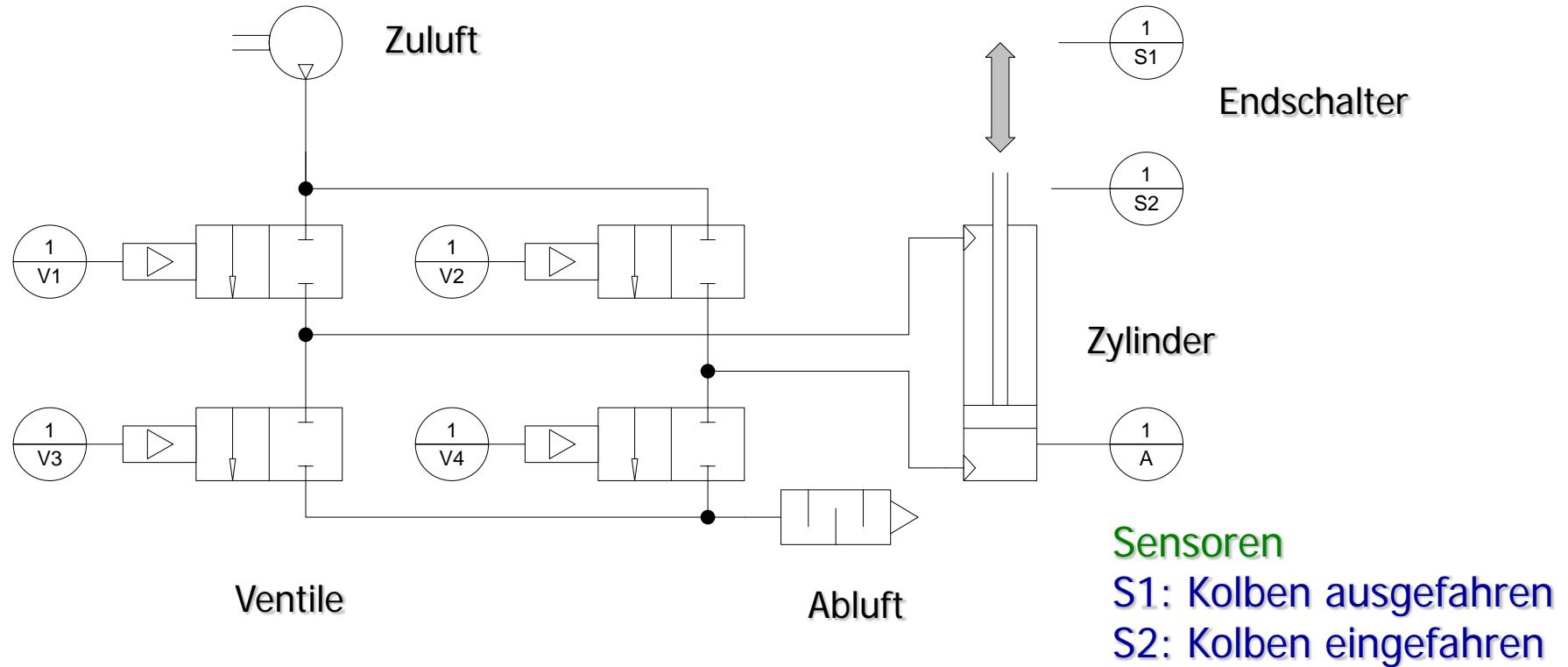
Für *einfache Systeme* reichen z. B. bereits

- Object Diagrams
- Statecharts
- Component Diagrams

Kritik vor allem an UML 2.x:
durch zu viele Diagramme und Features überfrachtet und damit schwer erlernbar und umständlich handhabbar.

5.5.2 Beispiel-Modelle

5.5.2.1 Pneumatik-Zylinder



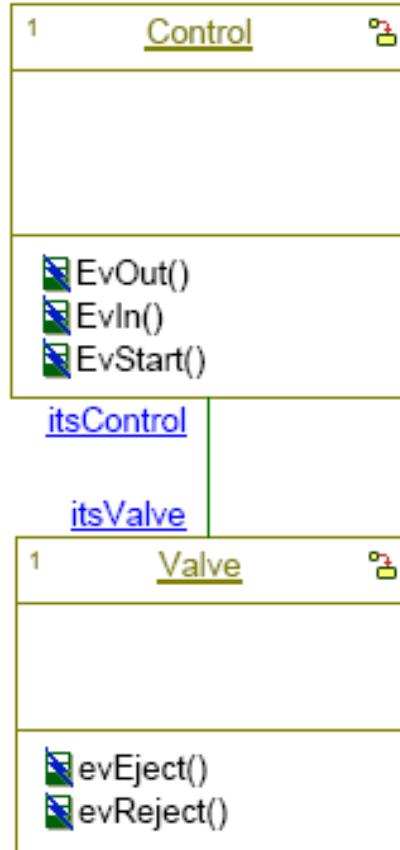
Aktoren

Kolben ausfahren (eject): Ventile V2 (Zuluft) und V3 (Abluft) auf, V1 und V4 zu

Kolben einfahren (reject): Ventile V1 (Zuluft) und V4 (Abluft) auf, V2 und V3 zu

Aufgabenstellung:

Es soll eine Kontrolleinheit entworfen werden, die den Kolben voll ausfährt, und anschließend gleich wieder voll einfährt.



Object Model Diagram

Gibt die Struktur wieder, hier zwei kommunizierende Objekte

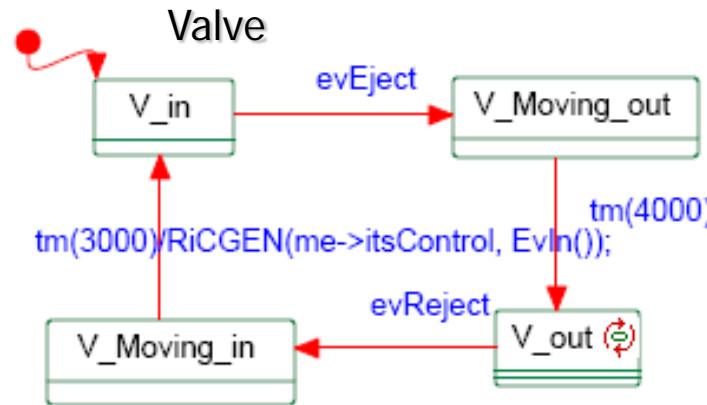
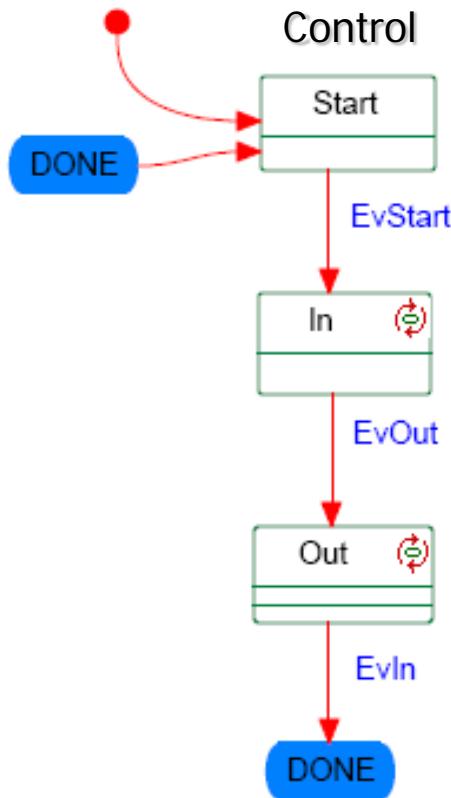
- Control (Kontrolleinheit)
- Valve (Zylinder mit Ventilen)

Verbindung über bidirektionales Link

Die dargestellten Methoden (hier Ereignisse) werden bei der Verhaltensspezifikation der Objekte automatisch eingetragen.

Statecharts

Beschreiben das Verhalten der Objekte



Action in V_out: RiCGEN(me->itsControl, EvOut());

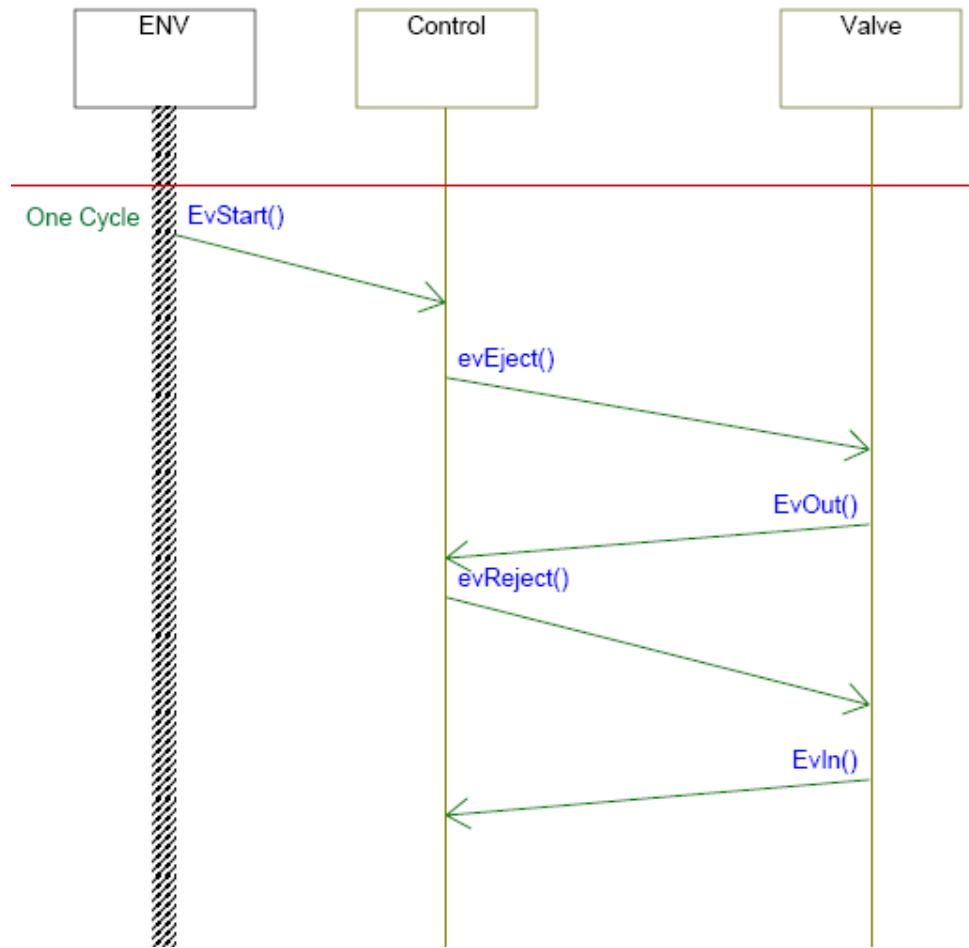
Kommunikation über Ereignisse (Events).

Entry Action in In: RiCGEN(me->itsValve, evEject());
Entry Action in Out: RiCGEN(me->itsValve, evReject());

Modellierung als Moore- oder Mealy-Automat auch gemischt möglich!

Sequence Diagram

Beschreibt die Kommunikation zwischen den Objekten und mit der Umwelt (ENV)



Kann u. a. zum *Testen* verwendet werden (Vergleich von spezifizierter Kommunikation mit tatsächlicher Kommunikation).

Modell kann *simuliert* und mittels Statecharts und Sequence Diagrams *animiert* werden.

Codegenerierung in C, Ausführung auf Zielhardware mittels Adaption Layer möglich!

5.5.3 UML Real-Time Profil

UML nicht speziell für Echtzeitaufgaben konzipiert, daher 2002 Profil für

- *Rechtzeitigkeit (Timeliness)*
- *Leistung (Performance)*
- *Schedulebarkeit (Schedulability)*

von der OMG verabschiedet (Real-Time UML).

UML-Profil

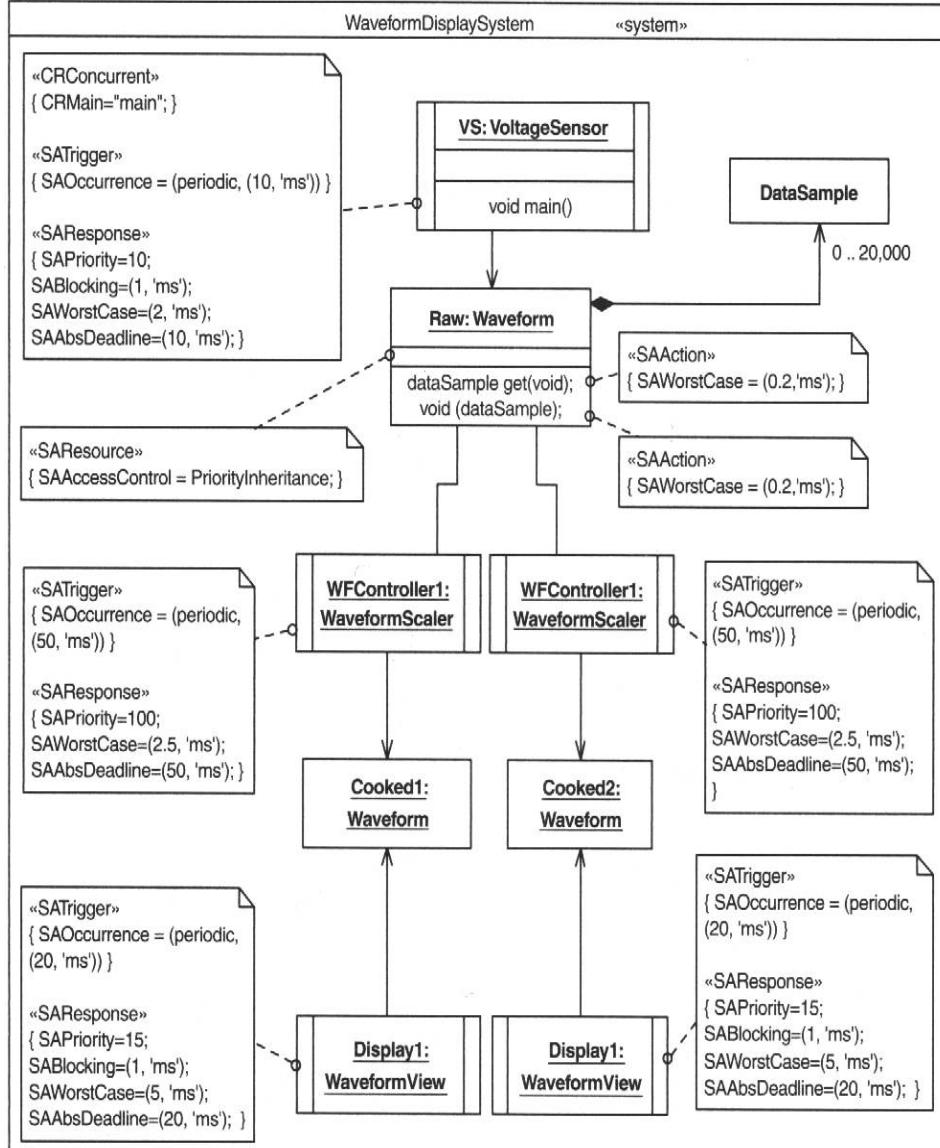
Variante von UML, die auf eine bestimmte Anwendungsdomäne (hier Echtzeitsysteme) spezialisiert ist.

Darf existierende Semantik nicht brechen, sondern nur spezialisieren und erweitern.

Hierzu werden definiert:

- | | |
|--------------------|--|
| <i>Stereotypes</i> | für UML-Metamodellelemente |
| <i>Properties</i> | zur Charakterisierung dieser Elemente |
| <i>Constraints</i> | als zusätzliche Regeln zur Modellbildung |

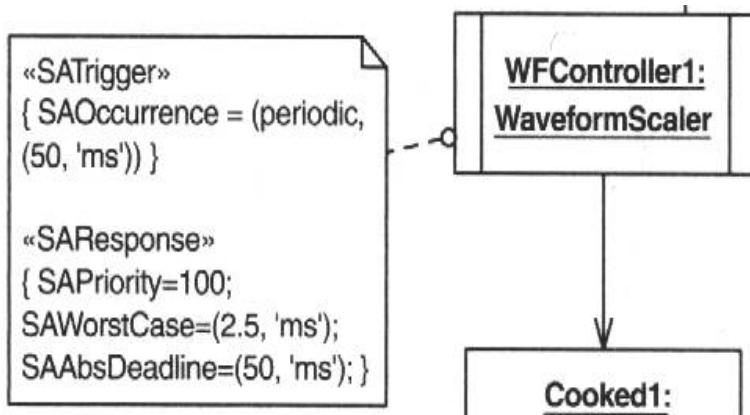
Stereotypes



Stereotypes spezialisieren existierende Elemente (z. B. Klassen) von UML.

Darstellung durch geeignete Icons oder durch
<<Stereotype-ID>>.

Tagged Values und Constraints



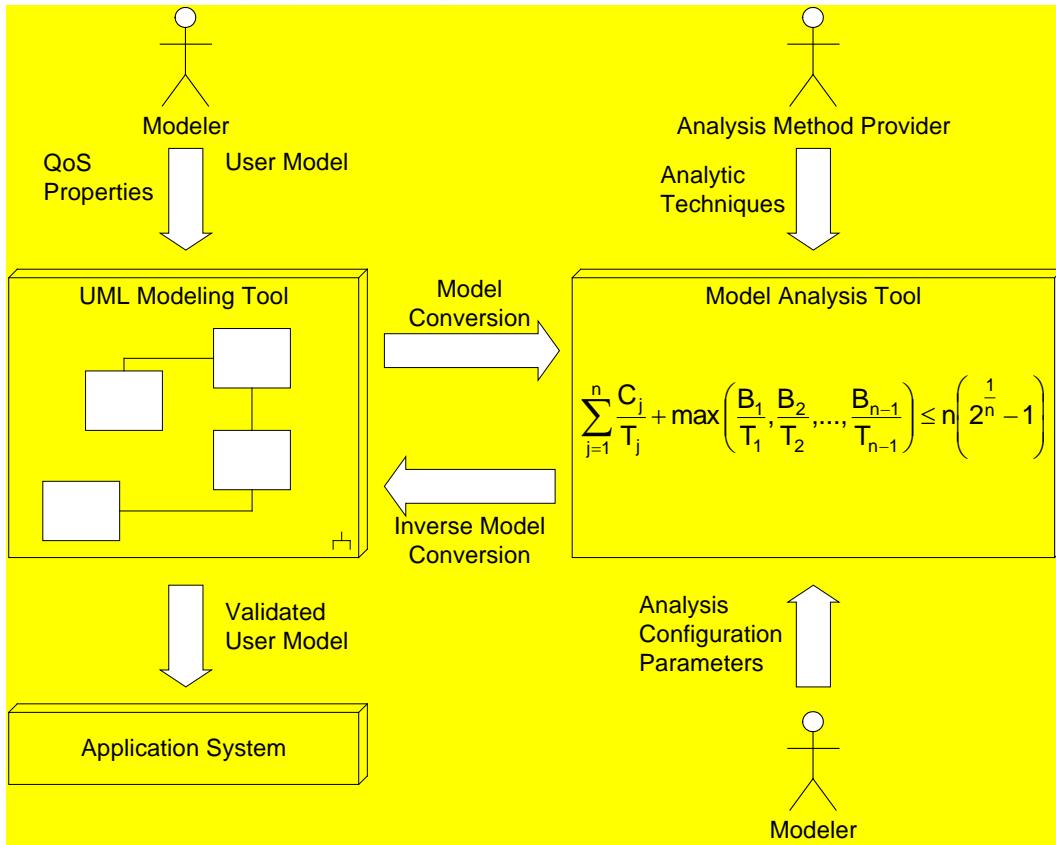
Tagged Values oder *Properties* charakterisieren einen Stereotyp näher.

Constraints (oft als Tagged Values oder Properties formuliert) stellen benutzerdefinierte Regeln für Korrektheit dar.

Angabe in geschweiften Klammern, Zuordnung zu Klassen in Kästen mit umgeknickter oberer rechter Ecke.

Verwendung in anderen Diagrammen (z. B. Sequence Diagrams) entsprechend.

Benutzungsparadigma für RT-Profil



Spezialisierung des Models durch Stereotypes, Tagged Values und Constraints zur Modellierung von

- **Ressourcennutzung**
(RT Resource Modelling)
- **Nebenläufigkeit**
(RT Concurrency Modelling)
- **Zeitverhalten**
(RT Time Modelling)

Einsatz von speziellen Werkzeugen zur

- **Leistungsanalyse**
- **Schedulebarkeitsanalyse**

Unterstützung von RT-Middleware, insbesondere CORBA (Infrastructure Modelling)

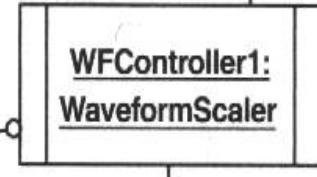
Ausgewählte Stereotypes für Schedulebarkeitsanalyse (SA)

- SAAction: Schedulbare Aktion
 - SAPriority: Priorität im Betriebssystem
 - SABlocking: maximale Blockierungszeit
 - SAWorstCase: Worst-Case Ausführungszeit
 - SAAbsDeadline: Deadline
 - ...
- SATrigger: Auslösung einer Aktion, Subklasse von SAAction
 - SAOccurrence: Auslösung der Aktion, z.B. periodisch alle X ms
 - Alle Attribute von SAAction
 - ...
- SAResponse: Antwort auf einen Stimulus, Subklasse von SAAction
 - Alle Attribute von SAAction
 - ...

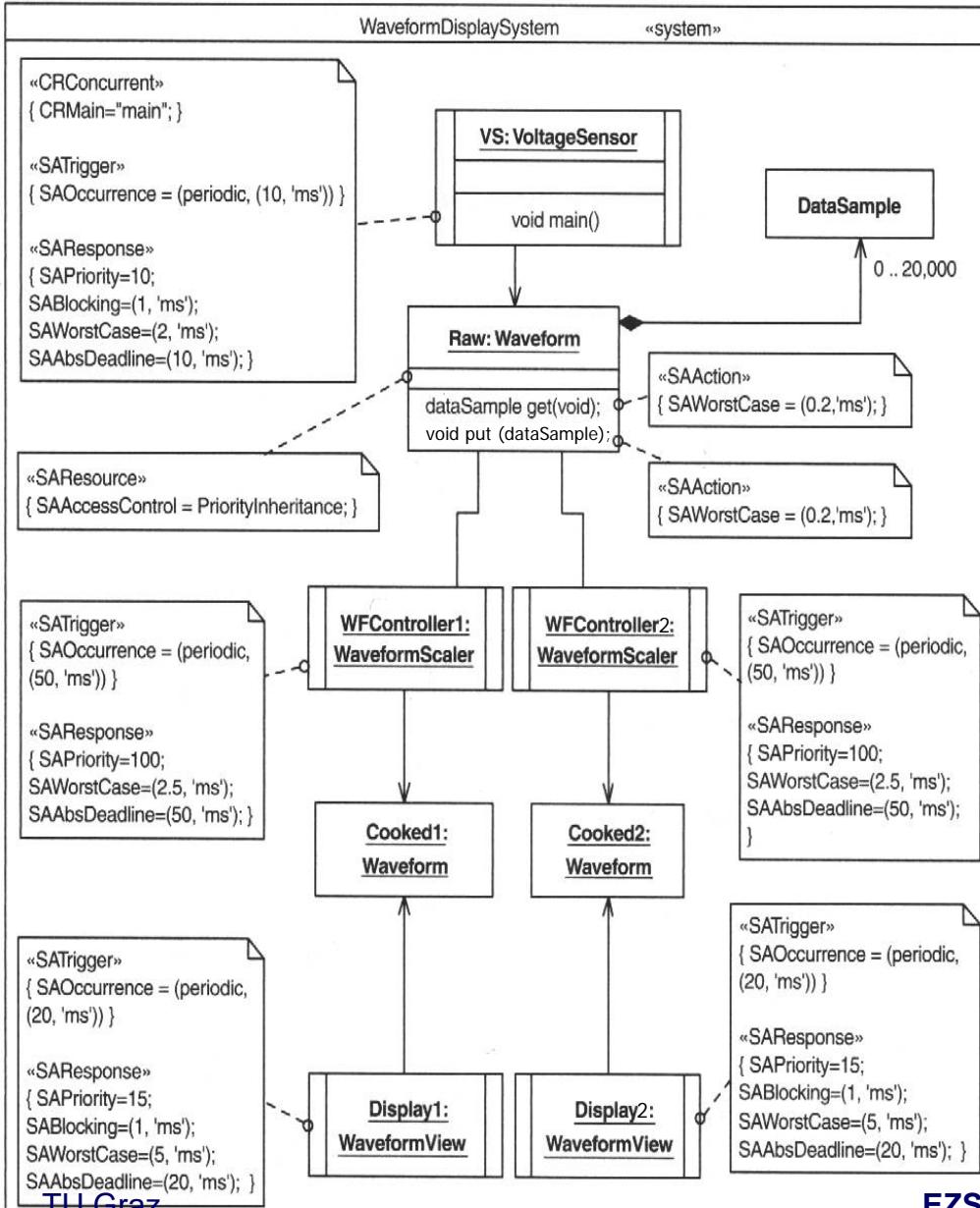
```
«SATrigger»  
{ SAOccurrence = (periodic,  
(50, 'ms')) }
```



```
«SAResponse»  
{ SAPriority=100;  
SAWorstCase=(2.5, 'ms');  
SAAbsDeadline=(50, 'ms'); }
```



Schedulebarkeitsanalyse mit RT-UML



Anhand der Tags wird berechnet, ob Bedingung für RM-Schedule (mit Blockierung) erfüllt ist:

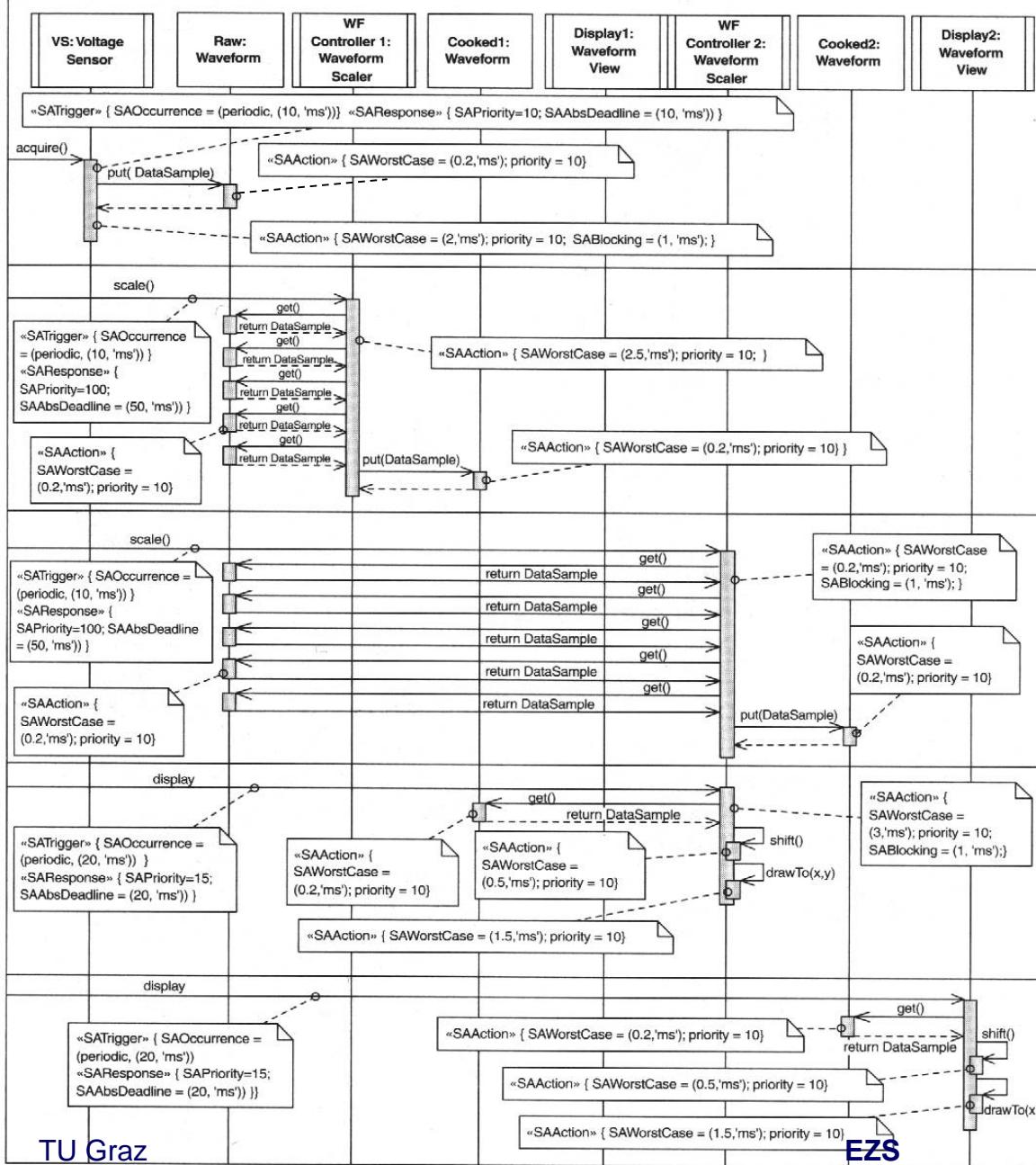
$$U = \frac{2}{10} + \frac{5}{20} + \frac{5}{20} + \frac{2,5}{50} + \frac{2,5}{50} \\ + \max\left(\frac{1}{10}, \frac{1}{20}\right) = 0.9 < 1$$

$U > U_j(5) \approx 0.743 \Rightarrow$ Existenz von RM-Schedule nicht sicher.

Reduktion der Ausführungszeiten von Display-Tasks auf **3 ms**:

$$U = \frac{2}{10} + \frac{3}{20} + \frac{3}{20} + \frac{2.5}{50} + \frac{2.5}{50} \\ + \max\left(\frac{1}{10}, \frac{1}{20}\right) = 0.7 < U_g(5) \approx 0.743$$

⇒ RM-Schedule existiert sicher!



Außer statischer Analyse anhand von OMDs auch **Szenario-Analyse** anhand von SDs möglich.

Bereiche zwischen horizontalen Linien sind parallele Tasks (Schlüsselwort ***par*** oben in SD).

Fazit RT-UML / Rhapsody

- ++ Formale Spezifikation von Anforderungsanalyse bis zur Implementierung.
 - + Unterstützung objektorientierter Techniken.
 - + Automatische Codegenerierung in gängigen Hochsprachen.
 - + Schnelle Reaktionsmöglichkeit auf geänderte Anforderungen.
 - + Test auf Modellebene anhand von Implementierung.
 - + Unterstützung von Echtzeitsystemen durch RT-Profil.
 - + Kürzere Entwurfszeiten, damit geringere Kosten.
-
- Nur auf reine Softwaresysteme beschränkt (Erweiterung existieren).
 - Übersichtlichkeit der Diagramme für sehr komplexe Systeme kritisch.
 - Kontinuierliches Verhalten (bisher) nicht explizit unterstützt.
 - Effizienz des generierten Codes problematisch.
 - Nur für „mächtige“ Zielplattformen (32-Bit-Prozessoren, RT-OS) geeignet.
 - Nachweis harter Echtzeitfähigkeit schwierig.
 - Zeitanalyse, Schedulebarkeitsanalyse etc. nur mit separaten Werkzeugen (nicht integriert).
 - Nachweis von Sicherheit und Zuverlässigkeit fraglich.
-
- ⇒ Für eingebettete Systeme mittlerer Komplexität ohne harte Echtzeitanforderungen oder hohe Sicherheitsanforderungen gut geeigneter Ansatz.