Karlsruhe Institute of Technology

Robotik I: Einführung in die Robotik
Roboterprogrammierung

Tamim Asfour, Rüdiger Dillmann
Institut für Anthropomatik und Robotik

KIT-Fakultät für Informatik, Institut für Anthropomatik und Robotik (IAR)
Hochperformante Humanoide Technologien (H^2T)

Inhalt

- **Motivation**
- Klassische Roboterprogrammierverfahren (Übersicht)
- Graphisches Programmierverfahren: Statecharts
- Symbolische Planung

Motivation - Neue Anforderungen in der Produktion

- Klein- & Kleinstserienfertigung
- Unikatfertigung (z.B. Prototyp)
- Produkte mit:
  - vielen Ausstattungsvarianten
  - hoher Rekonfigurierbarkeit

Flexible Fertigung

Motivation - Neue Anforderungen im Servicebereich

- Handel:
  - Kommissionierung und Palettierung von Waren
  - Bestücken von Regalen
- Qualitätssicherung
- Pflege:
  - Unterstützung von Rehabilitationsmaßnahmen
- Handwerk:
  - Handhabungen in Schreinereien und Schlossereien

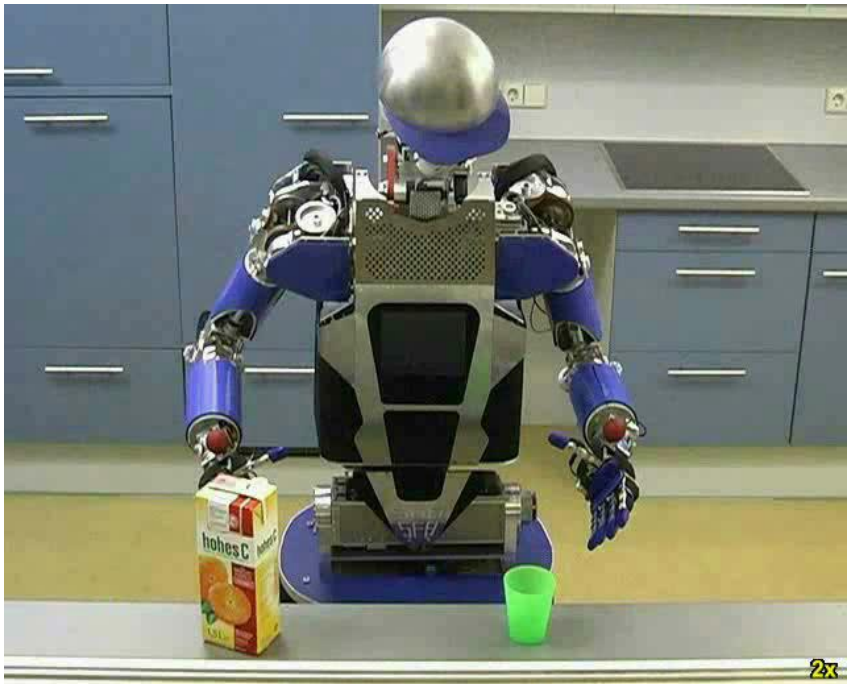
Motivation - Anforderungen der humanoiden Robotik

- Manipulation beliebiger Objekte
- Selbstständiges Lösen komplexer Aufgaben
- Einsatz im menschlichen Umfeld

komplexe Umgebung!

sehr viele Bewegungs-freiheitsgrade!

Programmierung?

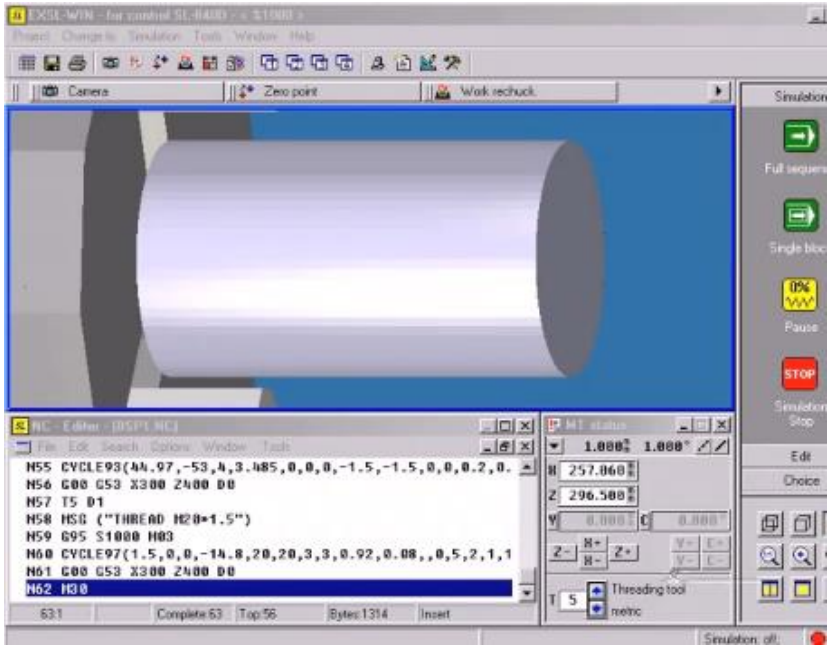


Motivation

Klassische Roboterprogrammierung erfordert Expertenkenntnisse

und ist mit hohem Aufwand verbunden

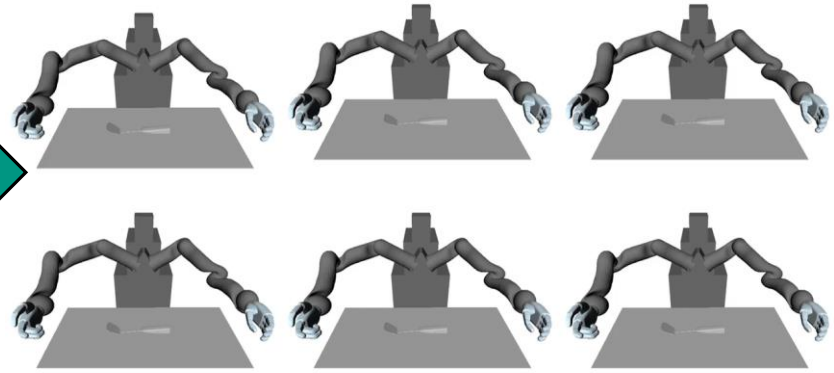
Teach Panel



Textuelle Programmierung

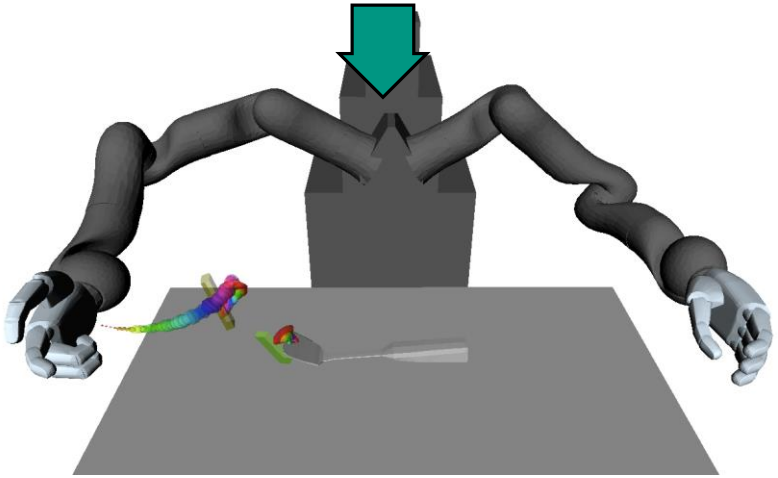
Interaktive Programmierung

Demonstration & Aufzeichnung



Segmentierung & Interpretation

Ausführung



Abstraktion & Simulation

Rainer Jäkel – Learning of generalized manipulation strategies in service robotics

Interaktive Programmierung

Wächter & Asfour (2015)

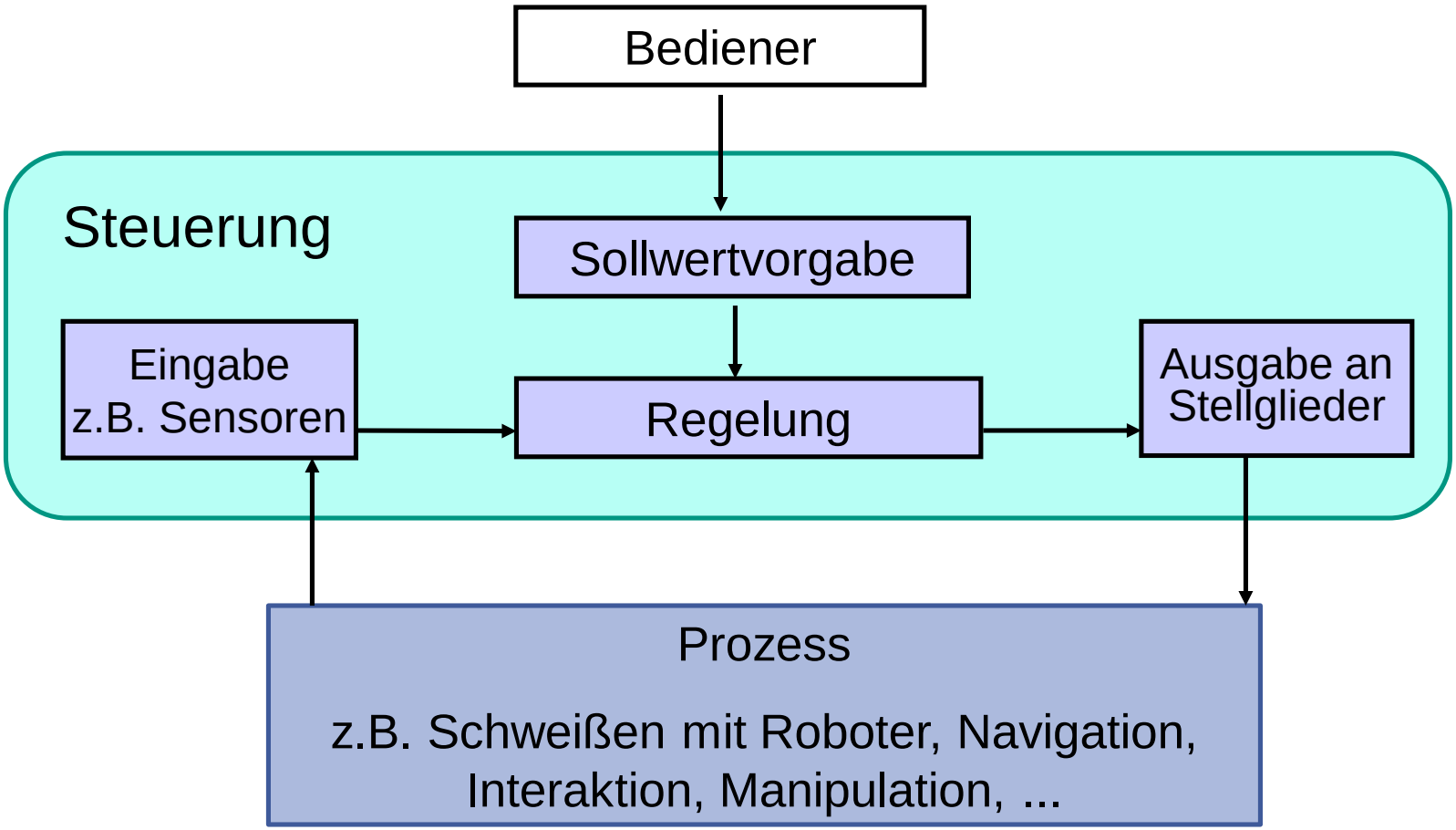
Interaktive Programmierung

Wächter & Asfour (2015)

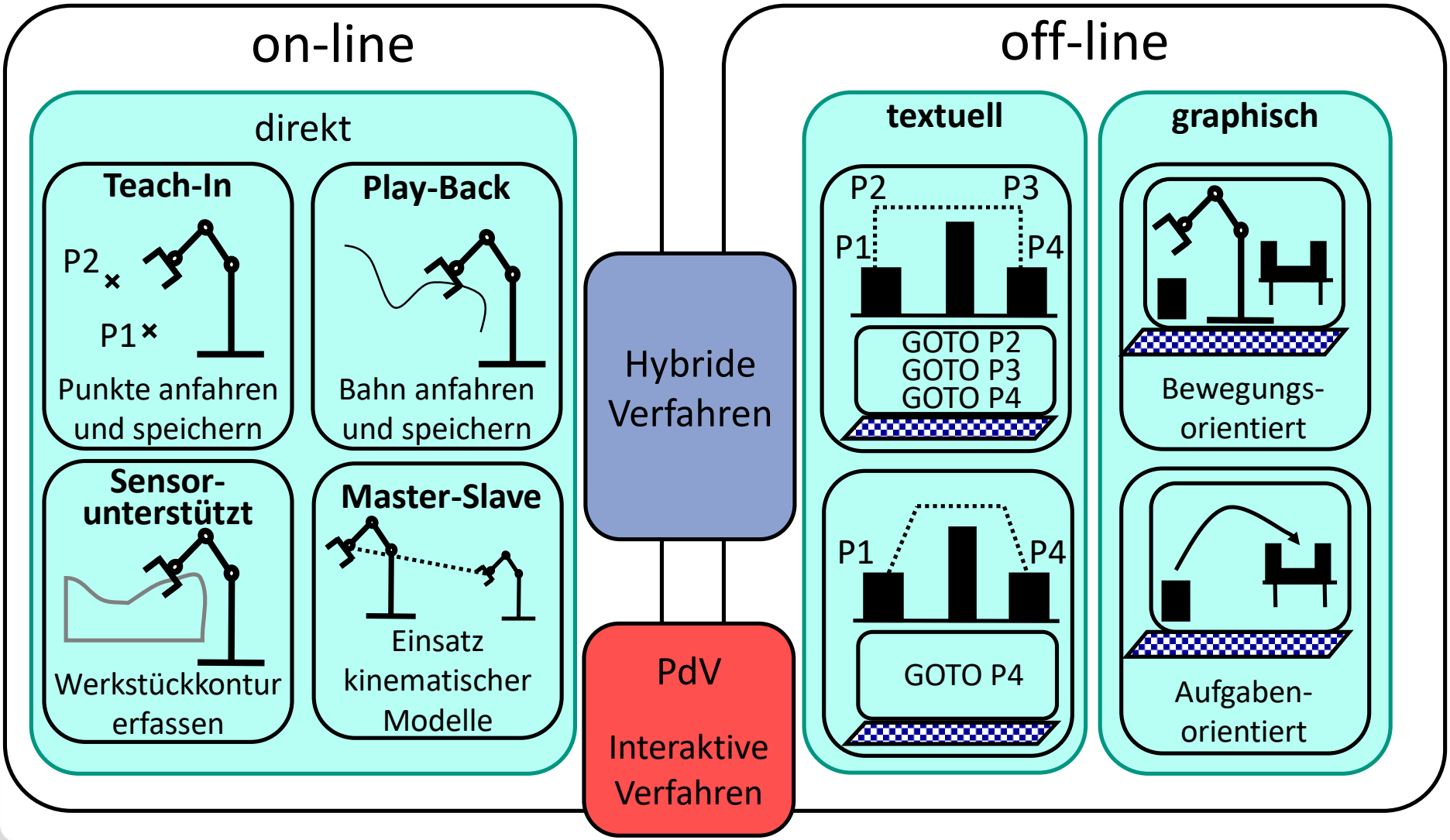
Inhalt

- Motivation
- **Klassische Roboterprogrammierverfahren (Übersicht)**
- Graphisches Programmierverfahren: Statecharts
- Symbolische Planung

Roboterprogrammierung



Roboterprogrammierverfahren



Klassifizierung der Roboterprogrammierverfahren

Kriterien:

I. Programmierort
II. Art der Programmierung
III. Abstraktionsgrad der Programmierung

Klassifizierung der Roboterprogrammierverfahren

Kriterien:

I. **Programmierort**
II. Art der Programmierung
III. Abstraktionsgrad der Programmierung

Roboterprogrammierverfahren

on-line

Die Programmierung erfolgt direkt am Roboter (an der Robotersteuerung).

In der Literatur auch **direkte** oder prozessnahe Programmierung genannt.

Hybride Verfahren

off-line

Die Programmierung erfolgt ohne den Roboter mit Hilfe textueller, graphischer, interaktiver Methoden.

In der Literatur auch **indirekte** oder prozessferne Programmierung genannt.

Klassifizierung der Roboterprogrammierverfahren

Kriterien:

I. Programmierort
- Direkte Programmierung
- Indirekte Programmierung

II. Art der Programmierung
- **Direkte Programmierung**
- **Textuelle Verfahren**
- **Graphische Verfahren**
- **Gemischte Verfahren**

Klassifizierung der Roboterprogrammierverfahren

Kriterien:

I. Programmierort
- Direkte Programmierung
- Indirekte Programmierung

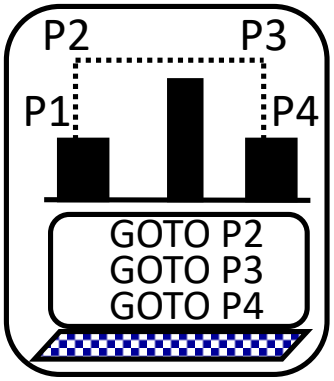
II. Art der Programmierung
- Direkte Programmierung
- Textuelle Verfahren
- Graphische Verfahren
- Gemischte Verfahren

III. Abstraktionsgrad der Programmierung
- **Implizite Programmierung**
- **Explizite Programmierung**

Abstraktionsgrad der Programmierung

Explizite oder roboterorientierte Programmierung

Bewegungen und Greiferbefehle sind direkt in eine Programmiersprache eingebunden.

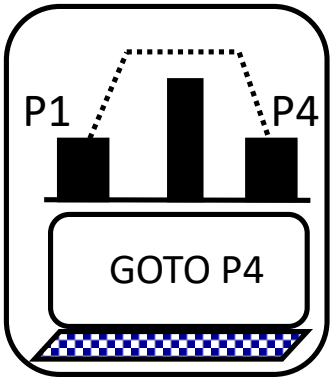


„Wie ist etwas zu tun?“

Abstraktionsgrad der Programmierung

Implizite oder aufgabenorientierte Programmierung

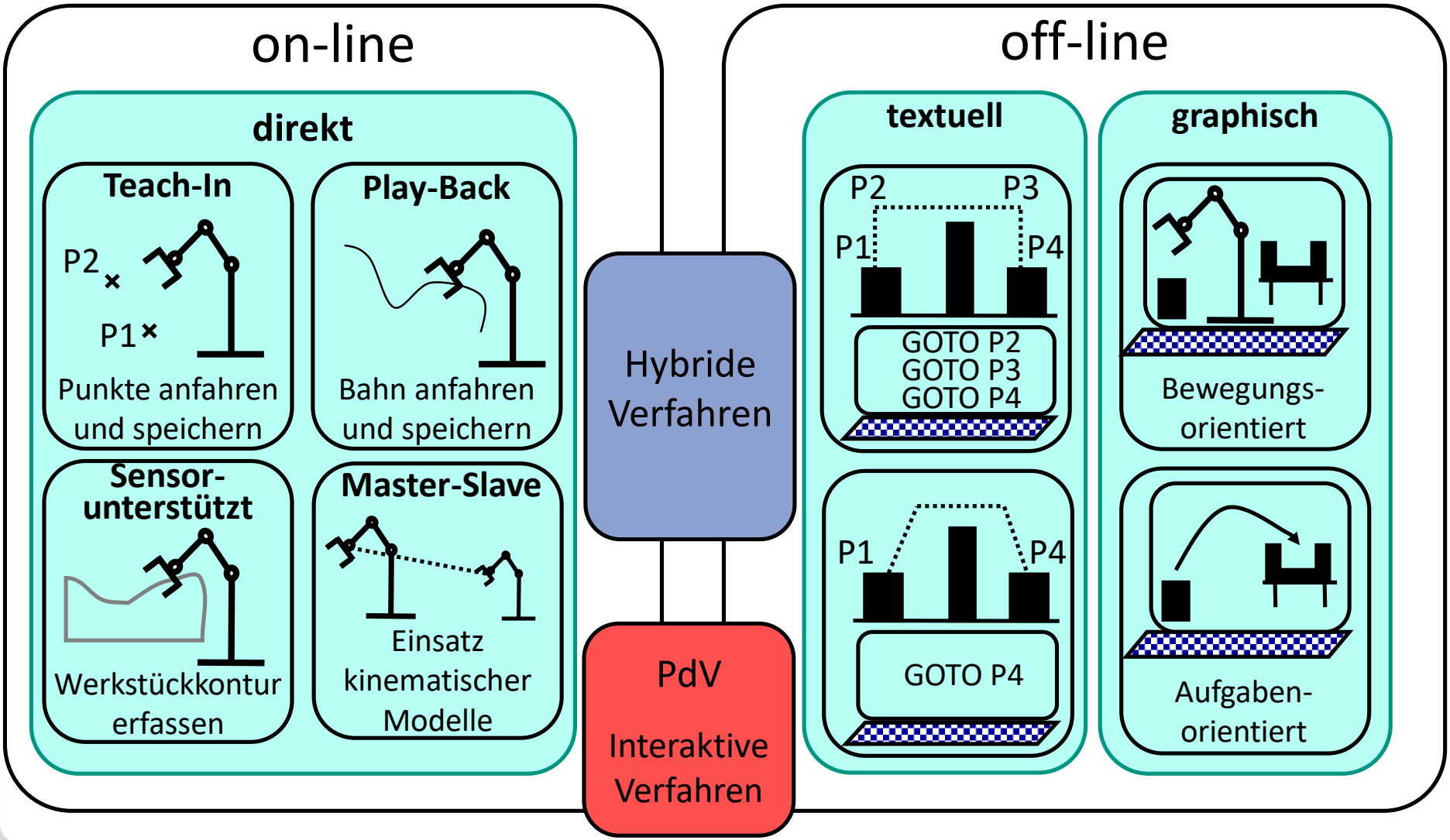
Die Aufgabe, die der Roboter durchführen soll, wird beschrieben, z.B. in Form von Zuständen.



„Was ist zu tun?“

- Abstrakte Form der Programmierung erfolgt in den Phasen
  1. Modellierung der Umwelt
  2. Spezifikation der Aufgaben
  3. Erzeugung der Roboterprogramme
- u. U. erfolgt vor der Ausführung eine Überprüfung des Roboterprogramms (Simulation)

Direkte Programmierverfahren



Direkte Programmierung

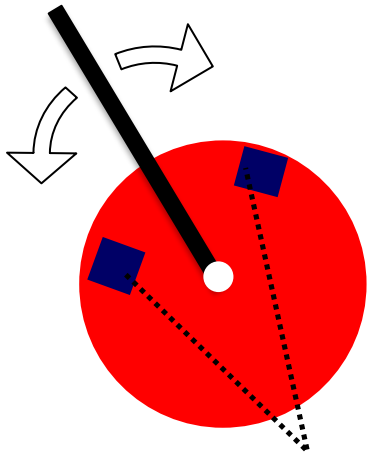
- „Einstellen“ des Roboters
- Teach-In Programmierung
- Play-Back Programmierung
  (manuelle Programmierung)
- Master-Slave Programmierung
  (mit Sonderfall Teleoperation)
- Sensorunterstützte Programmierung

Direkte Programmierung

Direktes Einstellen des Roboters

- Ältestes Programmierverfahren
- Der Bewegungsbereich jedes Gelenks wird durch Stopper eingeschränkt
- Die Bewegungen erfolgen für jedes Gelenk einzeln bis zum Anschlag
- Zuordnung zwischen Anfahrpunkten zu Stopper kann mit Codiermatrizen erfolgen
- Sog.: „Bang-Bang-Robot“

- Nachteil:
  - Sehr kleine Menge von Anfahrpunkten

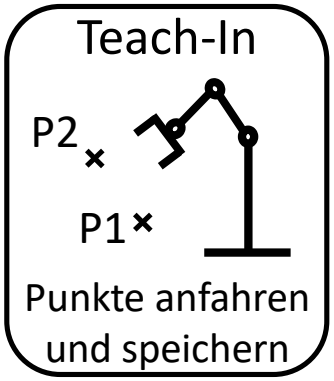


Direkte Programmierung : Teach-In

Teach-In Programmierung

- Anfahren markanter Punkte der Bahn mit manueller Steuerung (Teach Box, Teach Panel)

- Funktionalität einer Teach Box:
  - Einzelbewegung der Gelenke
  - Bewegung des Effektors in 6 Freiheitsgraden
  - Speichern / Löschen von Anfahrpunkten
  - Eingabe von Geschwindigkeiten
  - Eingabe von Befehlen zur Bedienung des Greifers
  - Starten / Stoppen ganzer Programme



Teachbox des Mitsubishi RM-501

5 DoF + Greifer
1,2 kg Nutzlast
Reichweite 45cm

Direkte Programmierung: Teach-In

- Weitere Eingabegeräte:
  - Maus, Joystick
    - schlechte Positioniergenauigkeit
    - nur 2 Freiheitsgrade
    - Unintuitiv bei komplexen Roboterarmen

  - Spacemouse (Teach-Kugel)
    - Kugel wird an Kraft-Momenten-Sensor befestigt
    - Kräfte werden als x,y,z Abweichungen interpretiert
    - Momente als Drehung um die Winkel α, β, γ

    + Erfassung von 6 Freiheitsgraden

    + Hohe Positioniergenauigkeit

Direkte Programmierung: Teach-In

Vorgehensweise beim Teach-In

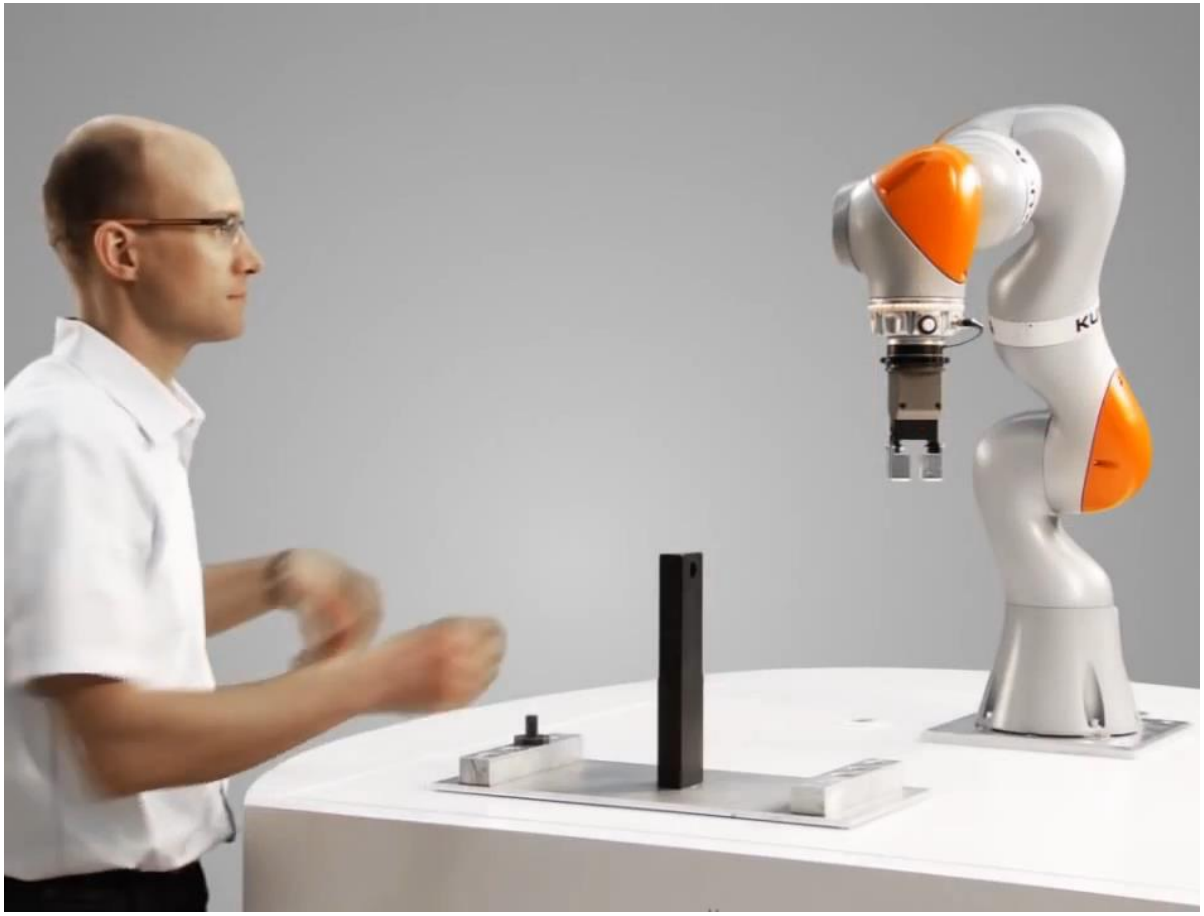
- Anfahren markanter Punkte der Bahn
- Bahn = Folge von Zwischenpunkten
- Speichern der Gelenkwerte
- nachträgliche Ergänzung der gespeicherten Werte um Parameter wie Geschwindigkeit, Beschleunigung usw.
- Anwendung:
  - in der Fertigungsindustrie (Punktschweißen, Nieten)
  - Handhabungsaufgaben (Pakete vom Fließband nehmen)

Direkte Programmierung: Play-Back

Play-Back- (manuelle) Programmierung

- Einstellung des Roboters auf Zero-Force-Control
  (Roboter kann durch den Bediener bewegt werden)
- Auch kinästhetisches Lernen genannt
- Abfahren der gewünschten Bahn
- Speichern der Gelenkwerte:
  - automatisch (definierte Abtastfrequenz) oder
  - manuell (durch Tastendruck)
- Anwendung:
  - mathematisch schwer beschreibbare Bewegungsabläufe
  - Integrierung der handwerklichen Erfahrung des Bedieners
  - Typische Einsatzbereiche sind: Lackieren oder Kleben

Direkte Programmierung: Playback



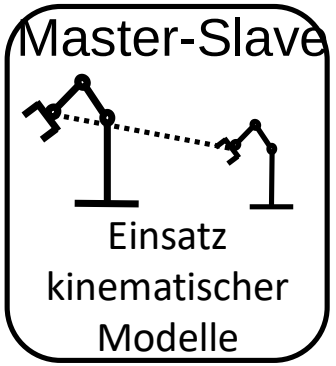
Direkte Programmierung: Play-Back

- Nachteile der Play-Back-Programmierung:
  - schwere Roboter schwierig zu bewegen
  - Direkter Kontakt mit Roboter → dadurch Sicherheitsrisiko
  - hoher Speicherbedarf (bei hoher Abtastrate)
  - schlechte Korrekturmöglichkeiten

Direkte Programmierung: Master-Slave

Master-Slave-Programmierung:

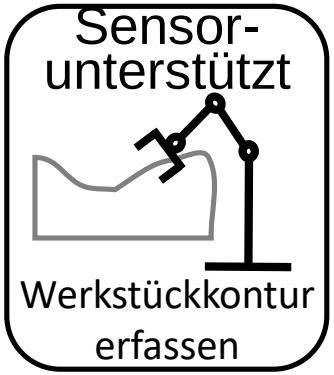
- Bediener führt einen kleinen, leicht bewegbaren Master-Roboter (entspricht einem kinematischen Modell des Slave-Roboters)
- Bewegung wird auf den Slave-Roboter übertragen
- Bewegungen werden synchron ausgeführt
- Slave-Roboter wirkt als Kraftverstärker
- Anwendung:
  - Handhabung großer Lasten bzw. großer Roboter
- Vor- und Nachteile:
  - teuer, da zwei Roboter benötigt werden
  - Möglichkeit, auch schwerste Roboter zu programmieren



Direkte Programmierung: Sensorunterstützt

Sensorunterstützte Programmierung

- Manuell
  - Bediener führt Programmiergriffel (Leuchtstift, Laserstift) entlang der abzufahrenden Bahn
  - Erfassung der Bewegung durch externe Sensoren (z.B. Kameras, Laserscanner)
  - Berechnung der inversen Kinematik
  - Abspeichern der Bahn als Folge der Gelenkwinkel
- Automatisch
  - Vorgabe des Start- und Zielpunktes
  - Sensorische Ertastung der Sollkontur (z.B. über Kraft-Momenten-Sensor)
- Anwendung:
  - Schleifen, Entgraten von Werkstücken

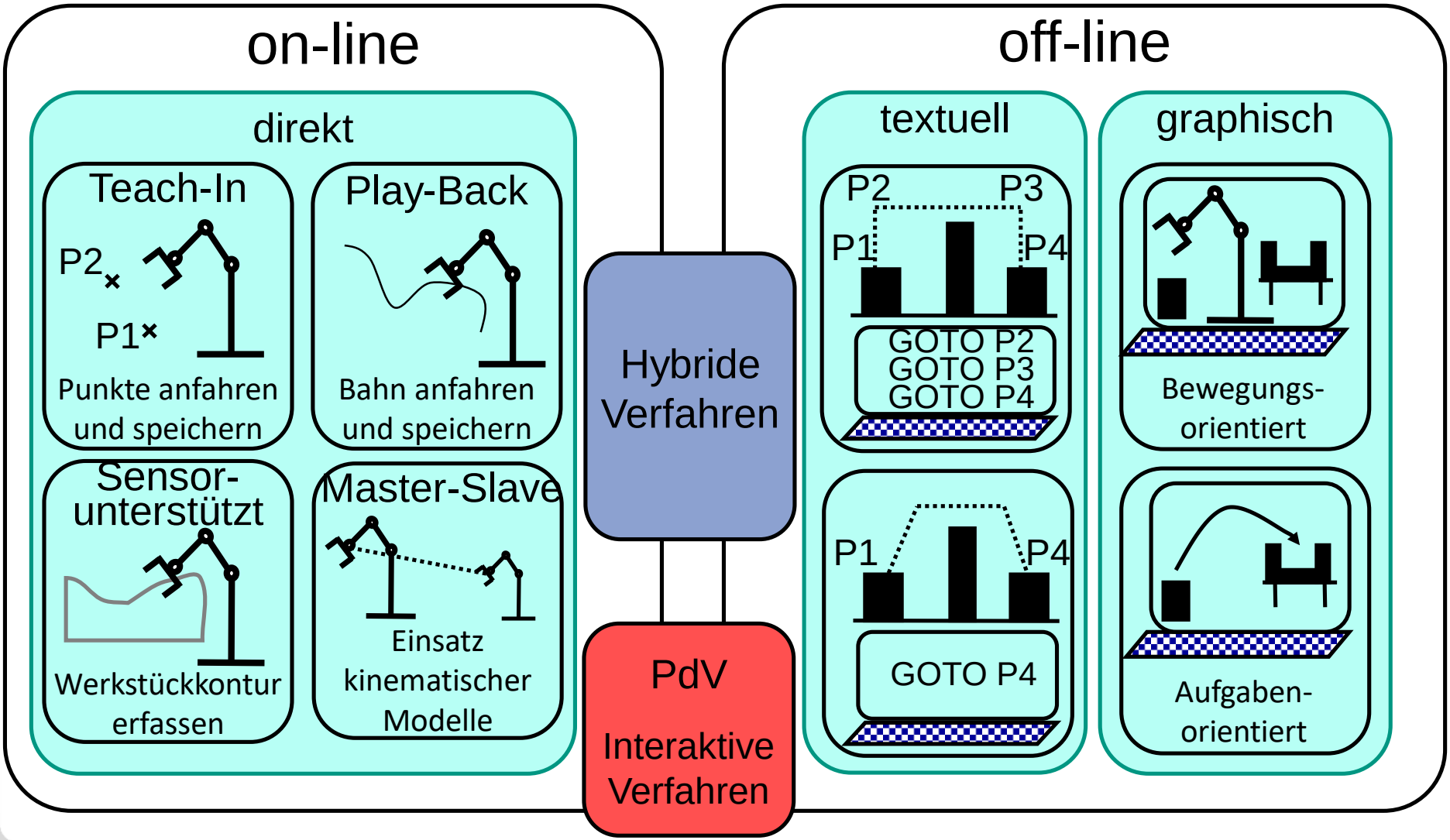


Direkte Programmierung: Zusammenfassung

- **Vorteile:**
  - schnell bei einfachen Trajektorien
  - sofort anwendbar
  - geringe Fehleranfälligkeit
  - Bediener benötigt keine Programmierkenntnisse
  - Kein Modell der Umwelt erforderlich

- **Nachteile:**
  - hoher Aufwand bei komplexen Trajektorien
  - nur mit und am Roboter möglich
  - Spezifisch für einen Robotertyp
  - Verletzungsgefahr durch Roboter
  - Keine Adaption an neue Gegebenheiten

Textuelle Programmierverfahren



Textuelle Programmierverfahren

- Programmierung erfolgt mittels erweiterter, höherer Programmiersprachen wie PASRO, VAL, V+ (Unimation/Stäubli), RAPID (ABB), KRL (KUKA), …

→ Robotersteuerprogramm

- Vorteile:
  - Programmierung kann unabhängig vom Roboter erfolgen
  - strukturierte, übersichtliche Programmierlogik
  - Erstellung komplexer Programme
    (Einbezug von Wissensbasis, Weltmodell, Auswertung von Sensoren)

- Nachteile:
  - Bediener benötigt Programmierkenntnisse
  - keine / schlechte Korrekturmöglichkeiten

Textuelle Programmierverfahren: DIN 66025

- Befehlscodierung nach **DIN 66025**
- Programm = Menge numerierter Sätze

Beispiel:

N70 G00 X20 Z12
Werkzeug im Eilgang (G00) an Position X=20 Z=12 bewegen.
(N = Satznummer)

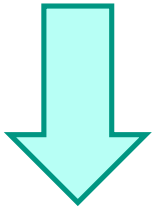
- Sprachen:
  - APT (Automatically Programmed Tools) 1961 MIT
  - EXAPT (Extended Subset of APT) 1966 TH Aachen

- Beispiel

P1=POINT/20,12	Variablendefinition
Rapid	Eilgang
GOTO/P1	Positionierung auf P1

Textuelle Programmierverfahren: SPS

- **VPS: Verbindungsprogrammierte Steuerung (historisch)**
  - Steuerung erfolgt über Hardware
  - „Programmänderung“ = Hardwareänderung



- **SPS: Speicherprogrammierbare Steuerung (englisch: PLC, Programmable Logic Controller)**
  - Steuerungs- und Regelungsablauf werden programmiert
  - große Flexibilität

Textuelle Programmierverfahren: CNC

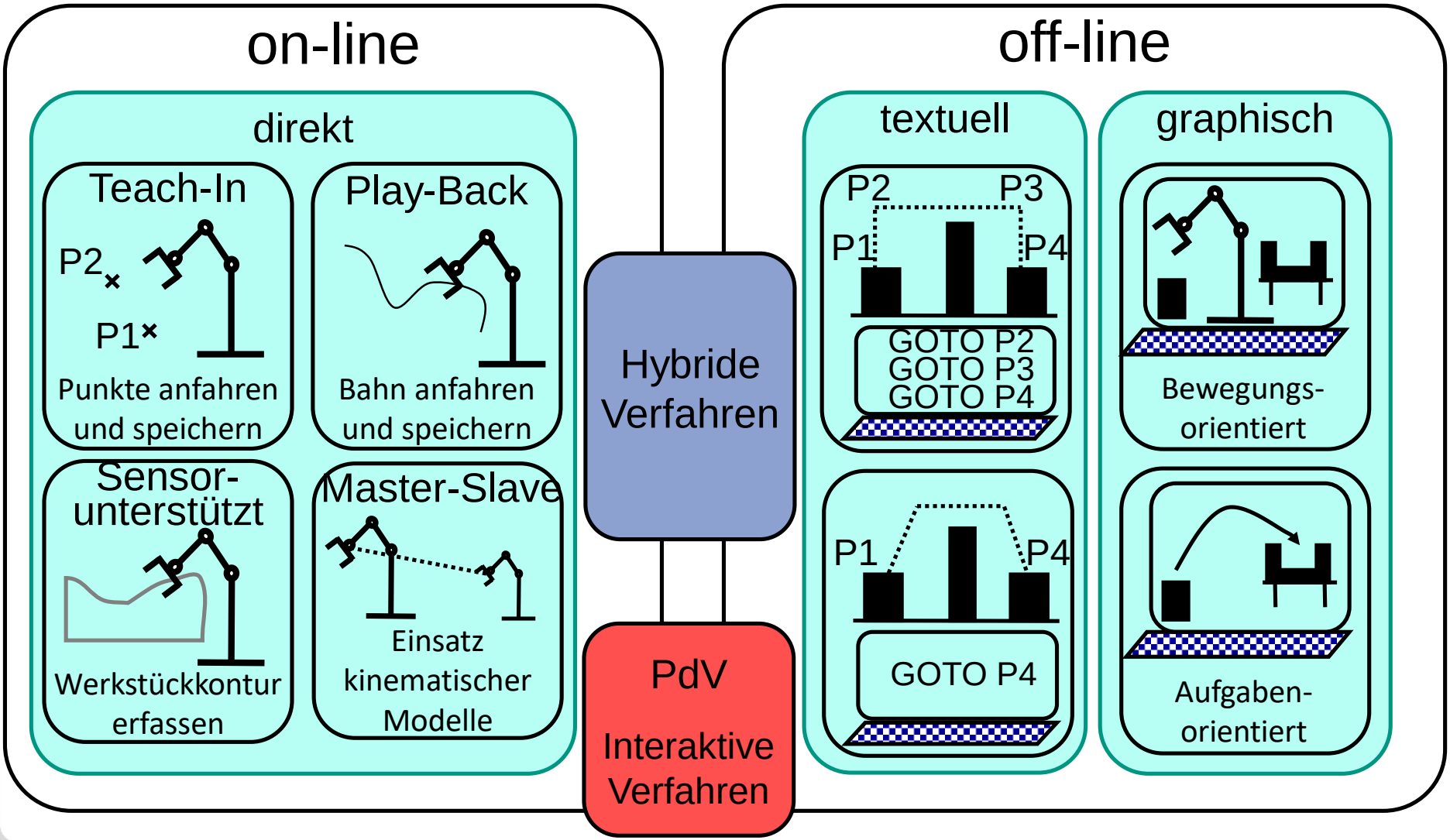
CNC: Computerized Numerical Control

- Steuerung von Werkzeugmaschinen
  - geometrische Beschreibung des Werkstücks bzw. der Bearbeitungsflächen + Kurven (Bahnkurve des Werkzeugs)
  - technologische Beschreibung (z.B. Vorschubgeschwindigkeit oder Spindeldrehzahl)

- Steuerungsarten:
  - Punktsteuerung (Bohrung)
  - Achsenparallele Steuerung (Fräsen)
  - Bahnsteuerung z.B. Strecke oder Kreis (Brennschneidmaschinen)

- Sprachen:
  - APT (Automatically Programmed Tools) 1961 MIT
  - EXAPT (Extended Subset of APT) 1966 TH Aachen

Hybride Programmierverfahren



Hybride Verfahren

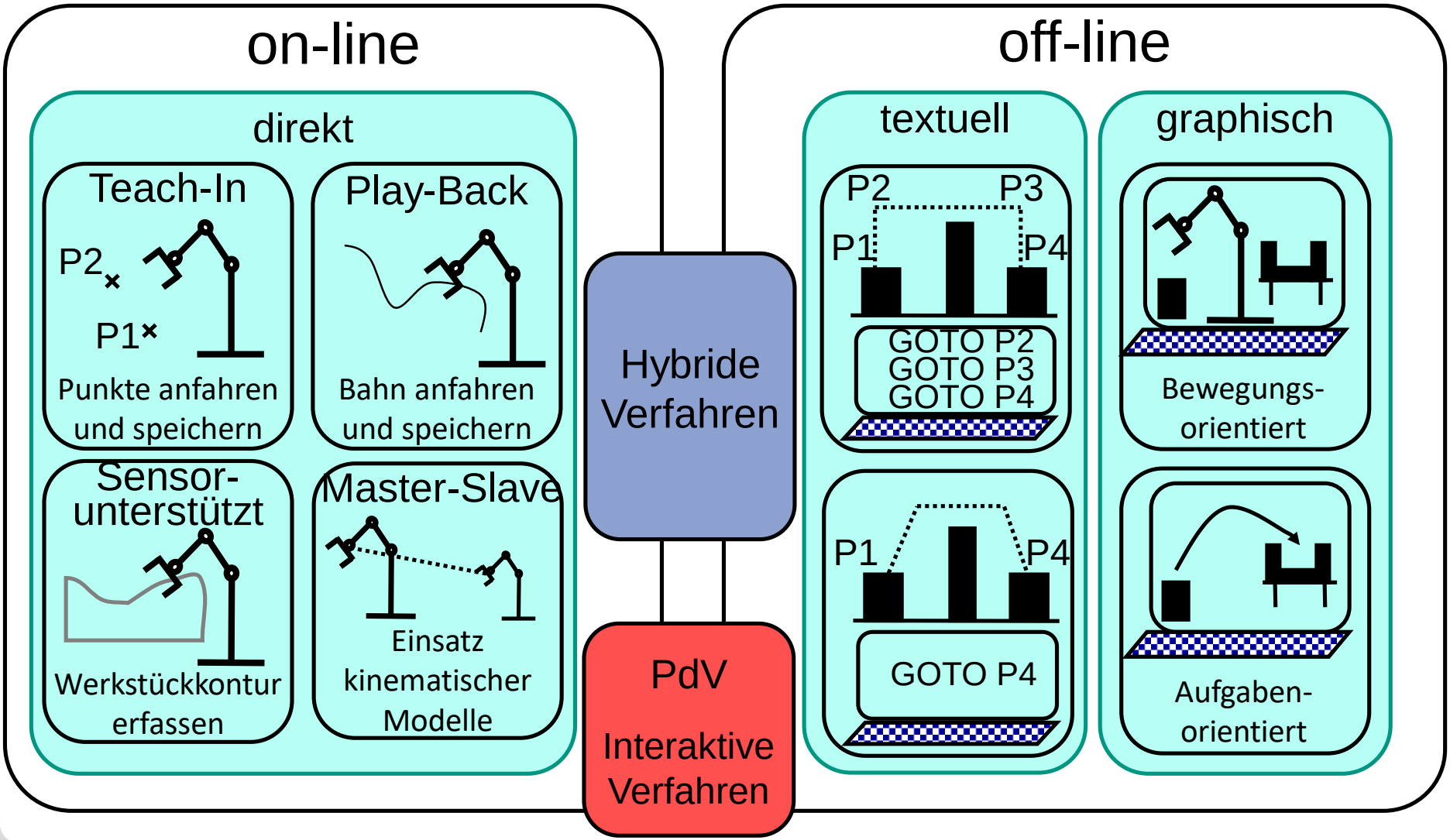
- Graphische Programmierung basierend auf sensorieller Erfassung der Benutzervorführung

  → Simulation der Roboterprogramme

- Vorteile:
  - Programmierer benötigt weniger Programmierkenntnisse
  - einfache Programmierung, leichte Fehlererkennung
  - schnelles Erstellen komplexer Programme (rapid prototyping)

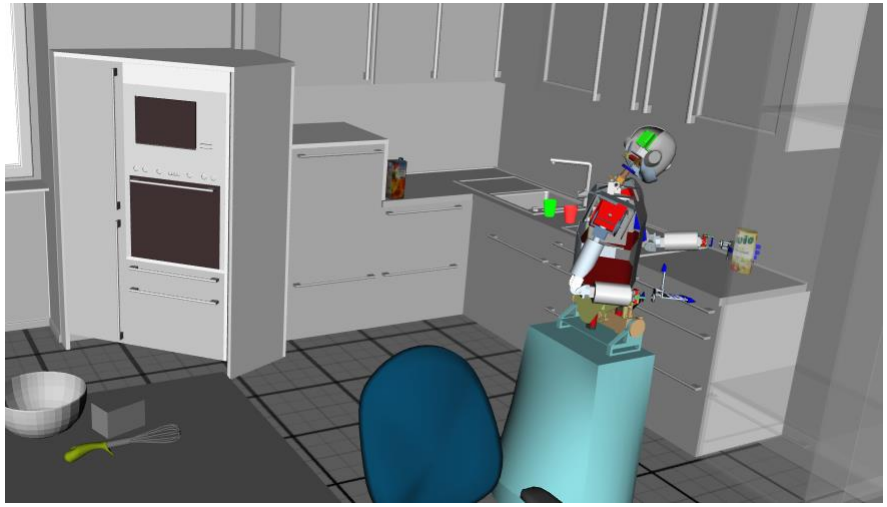
- Nachteile:
  - sensorielle Benutzererfassung noch zu ungenau
  - Leistungsfähige Hardware für Signalanalyse, Modellierung, ...
  - Komplexe Modelle benötigt

Graphische Programmierverfahren

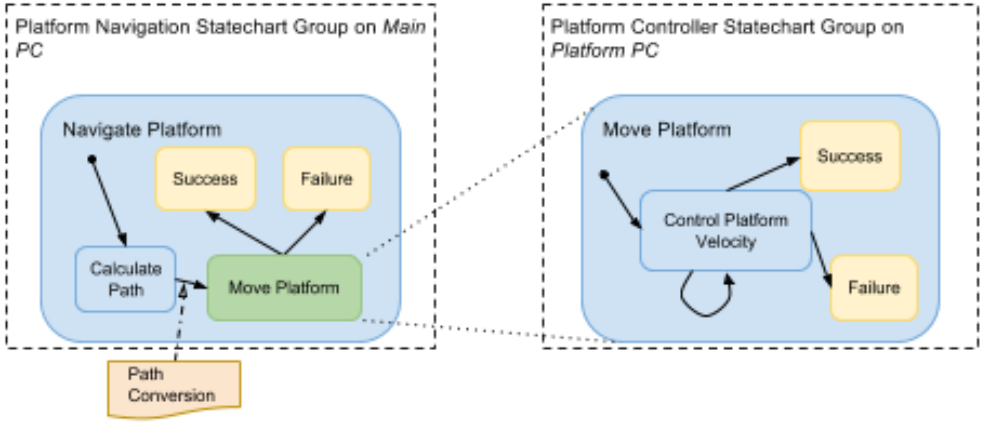


Graphische Roboterprogrammierung

- Zwei grundsätzliche unterschiedliche Varianten
  - Virtuelles Teach-In
    - Manipulation von Roboter und Umgebung in 3D Visualisierung
    - Abspeichern der Bewegungen
    - Benötigt exaktes 3D Modell von Roboter und Umgebung



  - Graphische Modellierungsformalismen
    - Endliche Zustandsautomaten
    - Petri-Netze
    - Statecharts



Virtuelles Teach-in

- Vorteile:
  - Programmierung kann unabhängig vom Roboter erfolgen
  - Programmierer benötigt weniger Programmierkenntnisse
  - einfache Programmierung, leichte Fehlererkennung
  - schnelles Erstellen komplexer Programme (rapid prototyping)

- Nachteile :
  - Leistungsfähige Hardware für Visualisierung und Simulation
  - Komplexe (Rechen-)Modelle für eine realitätsnahe Simulation benötigt
  - Roboter und Umwelt müssen modelliert werden!

Inhalt

- Motivation
- Klassische Roboterprogrammierverfahren (Übersicht)
- **Graphisches Programmierverfahren: Statecharts**
- Symbolische Planung

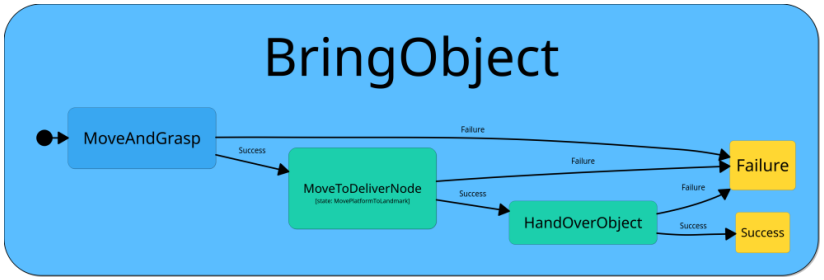
Roboteraktionsrepräsentation mittels Statecharts

- Warum Statecharts zur Programmierung von Robotern?

- **Lernen** von Aktionen aus Beobachtung schwierig (mehr dazu später)
  - Perzeption
  - Embodiment
  - Ausführungsunsicherheiten

- **Textuelle Aktionsprogrammierung** schwierig
  - Systemkomplexität
  - Roboterfähigkeiten stark zustandsbehaftet
  - Fähigkeiten bestehen zumeist aus Subfähigkeiten
  - Textuelle Programmierung unübersichtlich

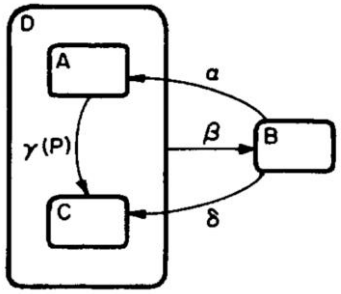
→ Graphische Programmierung von Roboteraktionen: **Statecharts**



Graphische Modellierungsformalismen

- Harel Statechart Formalismus (Harel, 1987)
  - **Graphischer Formalismus** zum Entwurf komplexer Systeme
  - Wichtigste Features:
    - Hierarchisch
    - Interlevel Transitionen
    - Orthogonalität
    - Zustandsaktionsphasen: Entry, Exit, Throughout
  - Limitation: Keine Datenflussspezifikation

→ArmarX Statechart Erweiterung
https://armarx.humanoids.kit.edu/

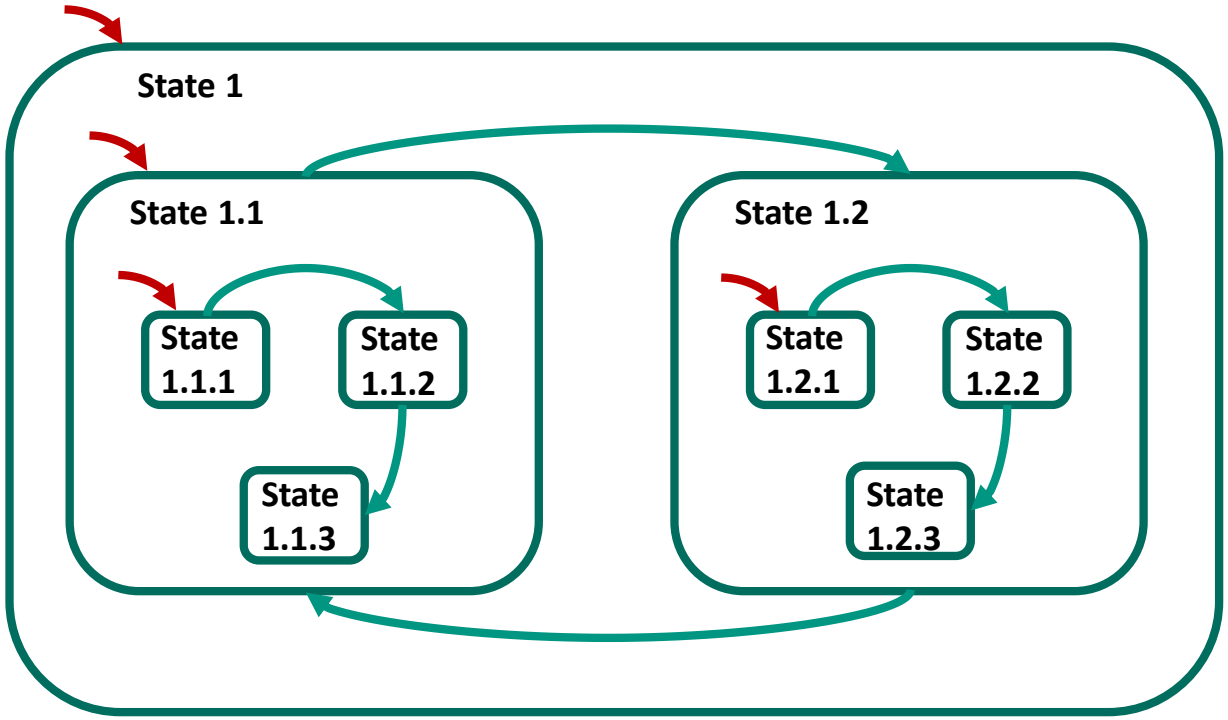


Harel,1987

- A, B, C und D sind Zustände
- Buchstaben auf den Kanten kennzeichnen Ereignisse; in Klammern werden die Bedingungen angegeben

D. Harel, *Statecharts: A visual formalism for complex systems*, Science of computer programming 8.3, pp. 231-274, 1987

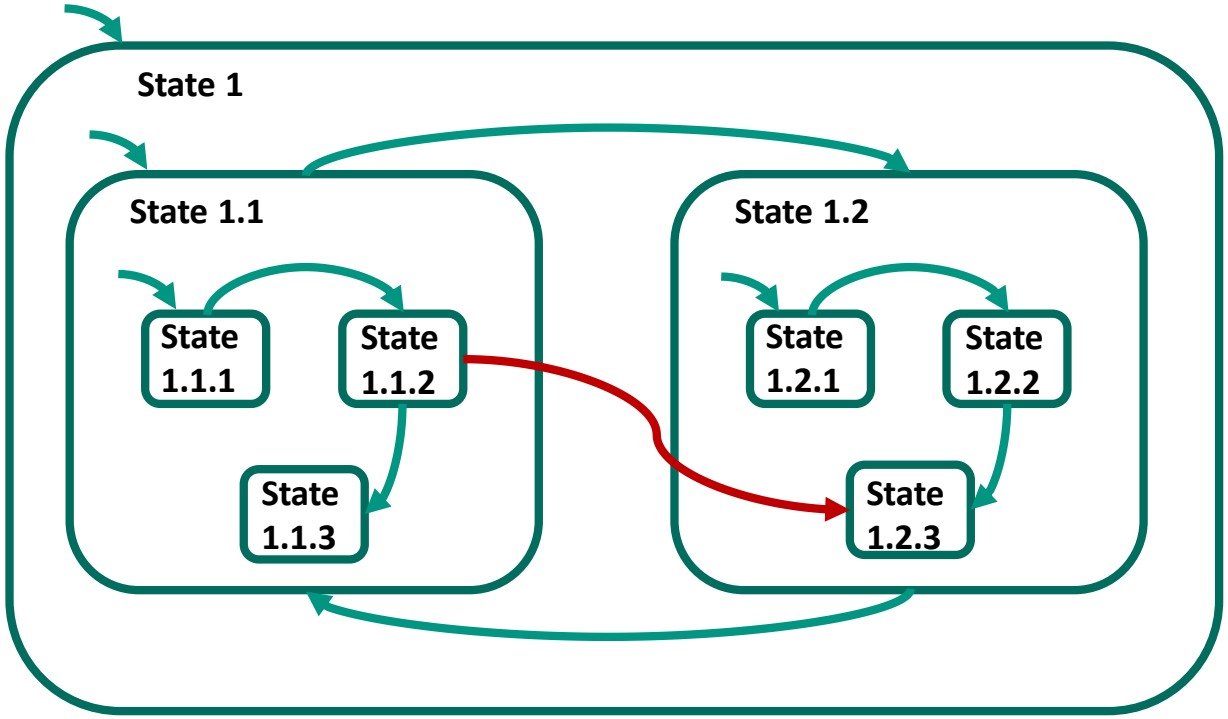
Statecharts (1)



Hierarchie

- State 1 ist Top-Level Zustand
- State 1.1 und State 1.2 sind Unterzustände von State 1
- State 1.1 und State 1.2 enthalten weitere Kind-Zustände
- Beim Betreten eines Zustands wird dessen **initialer Kind-Zustand** betreten

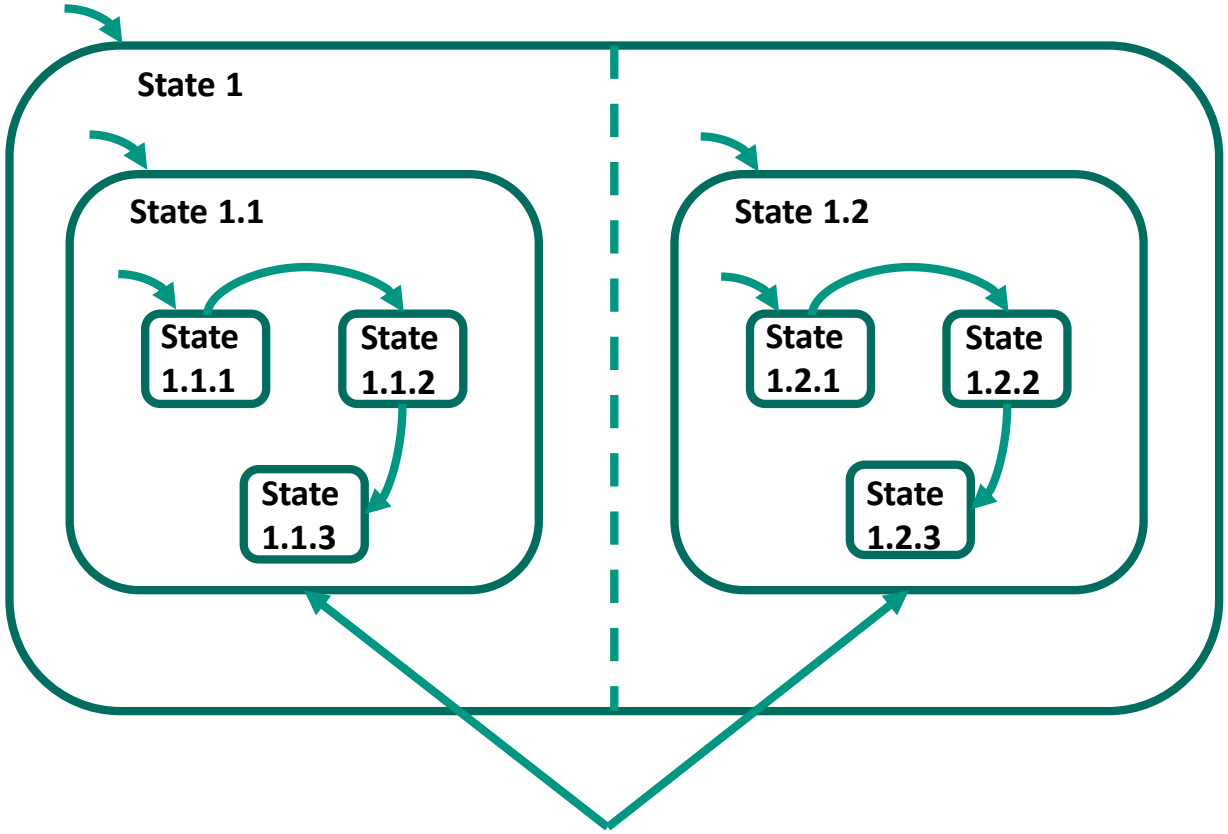
Statecharts (2)



Interlevel Transitionen

- Transitionen können zwischen Hierarchie-ebenen stattfinden

Statecharts (3)

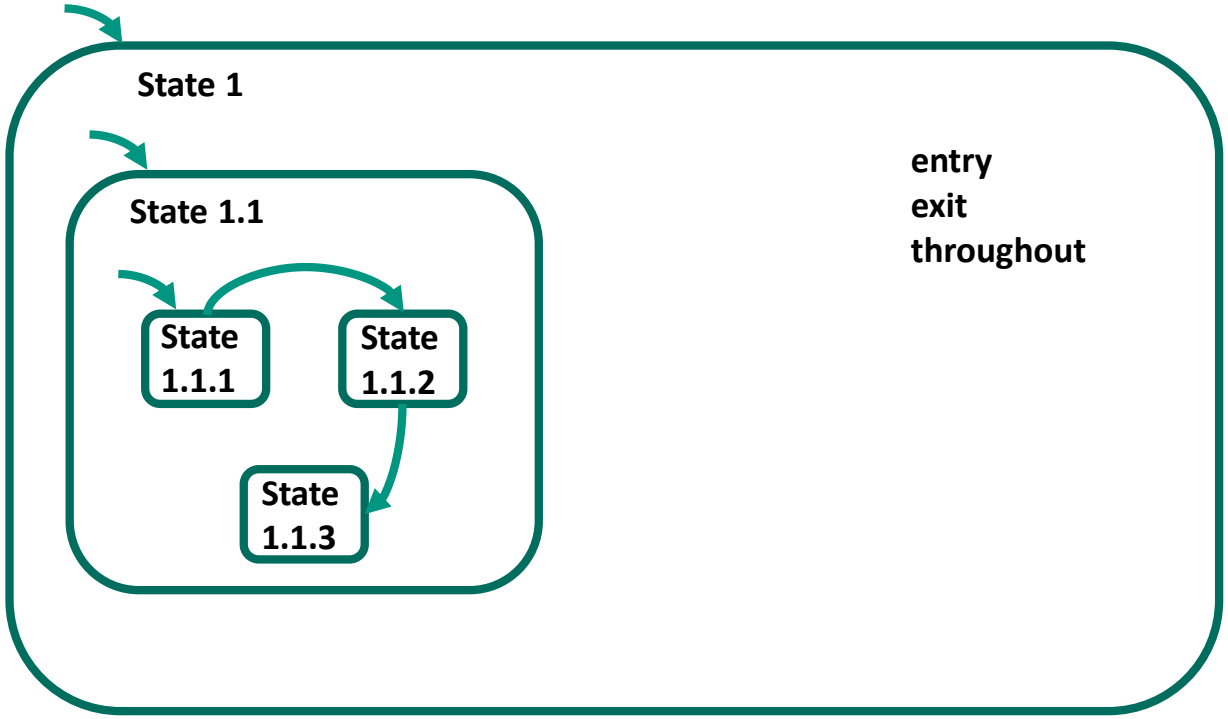


Parallele Ausführung dieser beiden Zustände

Orthogonalität

- Eine **gestrichelte Linie** markiert die **parallele Ausführung** der Zustände State 1.1 und State 1.2

Statecharts (4)

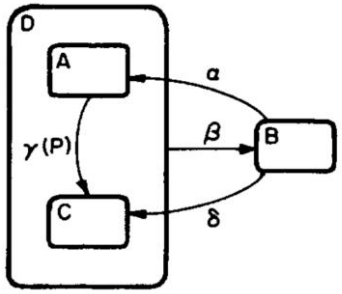


Zustandsphasen

- Beim **Betreten** von State 1 wird **zuerst** **Aktivität *entry*** ausgeführt, bevor State 1.1 betreten wird
- **Aktivität *throughout*** wird ausgeführt, **während** State 1.1 ausgeführt wird
- Nach Beendigung von State 1.1 und **vor dem Verlassen** von State 1 wird **Aktivität *exit*** ausgeführt

Erweiterte Statecharts für Roboterprogrammierung

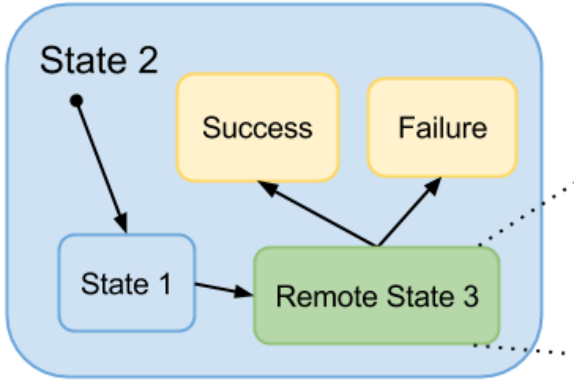
- Limitation Harel Statechart Formalismus: Keine Datenflusss
  - Datenfluss in der Robotik ist notwendig; Ergebnisse von Zuständen werden in Folgezuständen benötigt
    - Beispiel: Objektlokalisierung wird für Visual Servoing oder inverse Kinematik benötig
  - Wiederverwendbarkeit erfordert eine Adaption von Parametern
    - Kontrollparameter
    - Kinematikparameter
    - Objektparameter
    - ….



Harel,1987

Erweiterung des Harel Statechart Formalismus am H²T

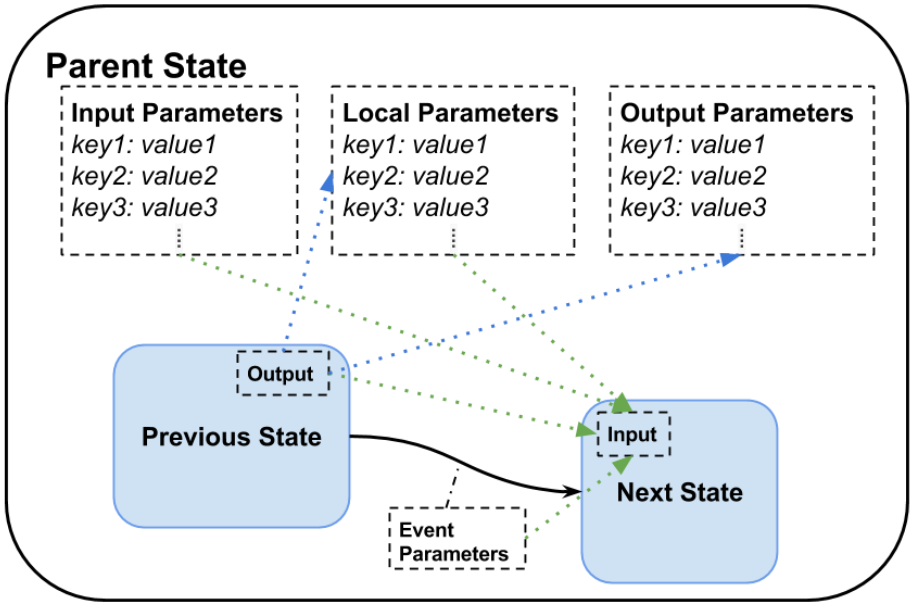
- Datenflussspezifikation: Transitionsbasierter Datenfluss
- Keine Inter-level Transitionen, da sie die Wiederverwendbarkeit verhindern
- Erfolgs- und Misserfolgszustände in jedem Zustand
- Dynamische Struktur
- Verteilung über mehrere Host-PCs
- Verbindung von graphischer Kontroll- und Datenflussspezifikation mit C++ Benutzer-Code



M. Wächter, S. Ottenhaus, M. Kröhnert, N. Vahrenkamp and T. Asfour, *The ArmarX Statechart Concept: Graphical Programming of Robot Behaviour*, Frontiers - Software Architectures for Humanoid Robotics, 2016

Statechart Erweiterungen

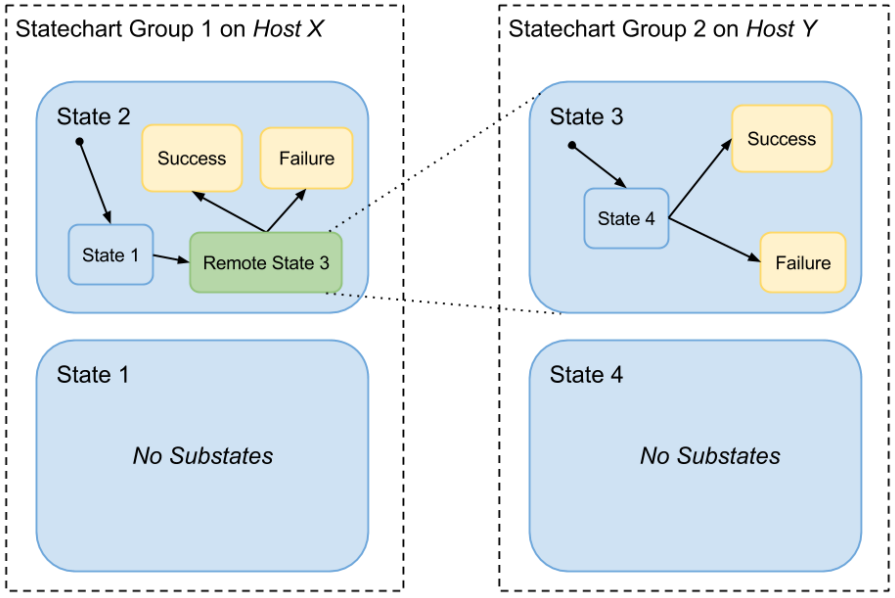
- Transitionsbasierter Datenfluss
  - Beliebige Datentypen
    - Grundlegende Datentypen: int, float, string, …
    - Komplexe Datentypen: Position, 6D-Pose, Matrizen, Listen, …
  - 3 Parameter Container pro Zustand
    - Eingabe (Input), Lokale Parameter (Local), Ausgabe (Output)
  - Parameterabbildung pro Transition
    - Abbildung von **Quell-Parameter** auf Eingabe-Parameter des **Zielzustandes**
  - Spezifizierter Datenfluss
    → Keine Seiteneffekte durch globale Variablen



Parameterabbildungen

Statechart Erweiterungen

- Verteilung über mehrere Host-PCs
  - Transparente Verteilung von Statecharts
    - Netzwerk Middleware *ZeroC Ice (https://zeroc.com/)*
  - Kind-Zustände können Zeiger auf entfernt liegende Zustände sein (grüner Zustand)
  - **Vorteile:**
    - Lastverteilung
    - Erhöhte Robustheit und Fehlertoleranz
    - Näher an eingesetzter Hardware (Sensorik, Aktorik)

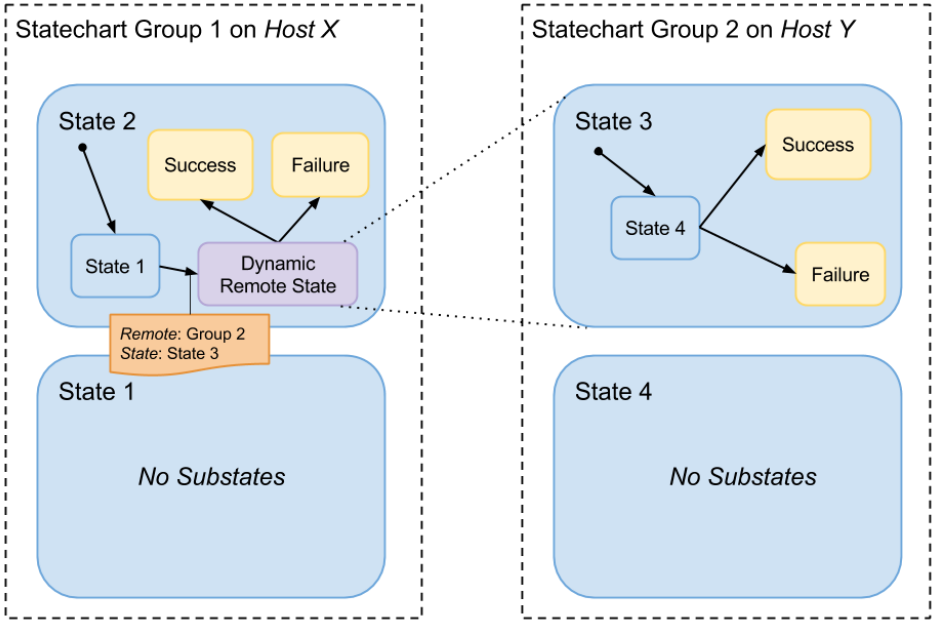


Verteilte Statecharts

Statechart Erweiterungen (2)

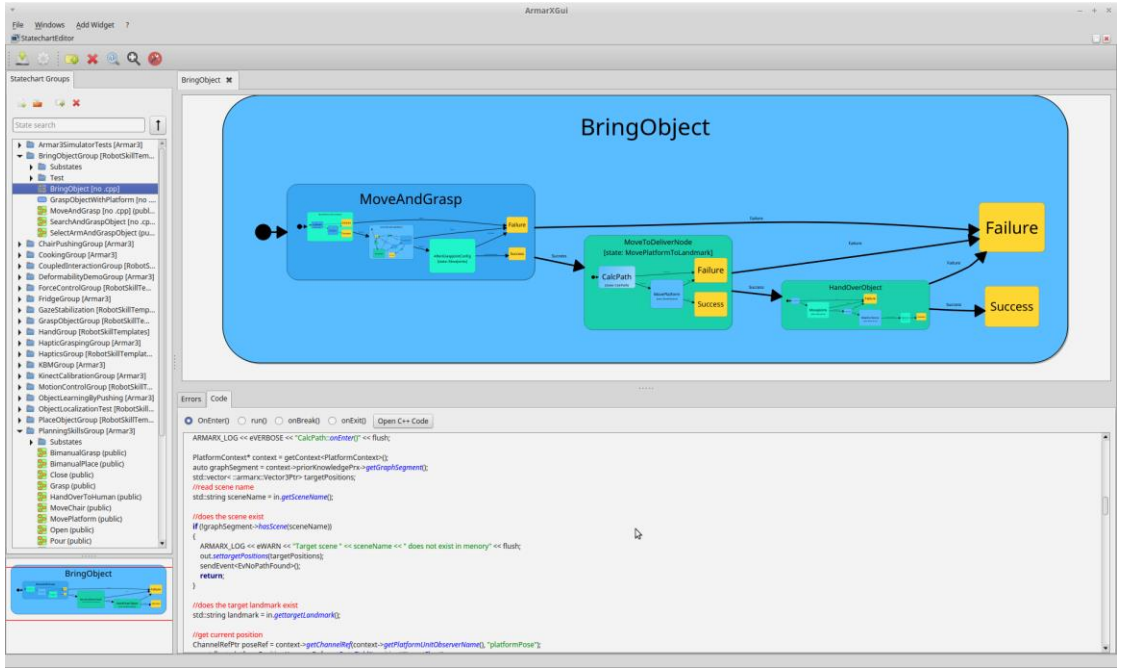
- Dynamische Struktur
  - Austausch von Unterzuständen zur Laufzeit
  - Transparente Verbindung zu beliebigem entfernten Zustand
  - Unterzustand durch Parameterabbildung spezifiziert
  - Einsatz: Ausführung von generierten Plänen

- Vollständig integriert in das Roboter Entwicklungsframework *ArmarX*
  - Graphischer Editor
  - Verbindung zu allen Roboter-komponenten
  - Online Inspektion der aktuellen Ausführung

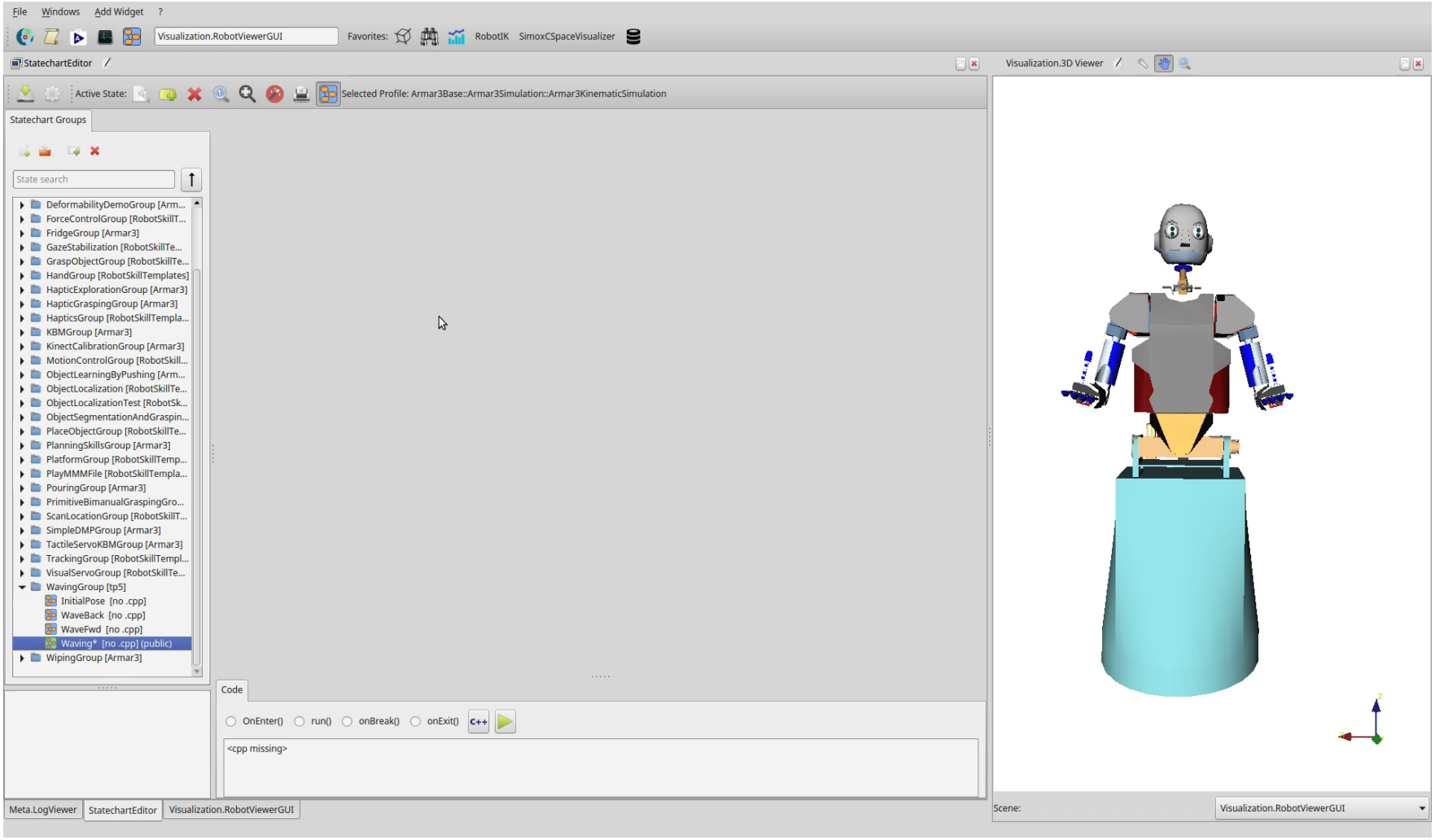


Graphischer Statechart Editor in ArmarX

- Tool zur graphischen Spezifikation von
  - Kontrollfluss
  - Datenfluss
  - Abhängigkeiten zu externen Roboterkomponenten
  - Direkt verlinkt mit C++ Code

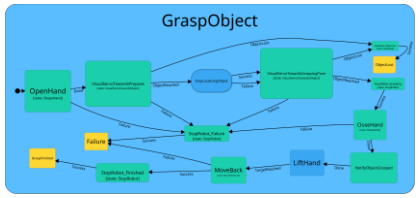


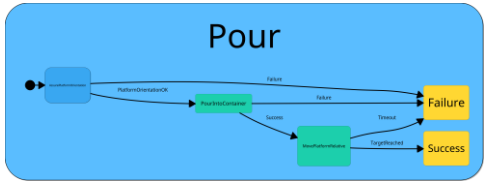
Graphischer Statechart Editor in ArmarX: Demo Video

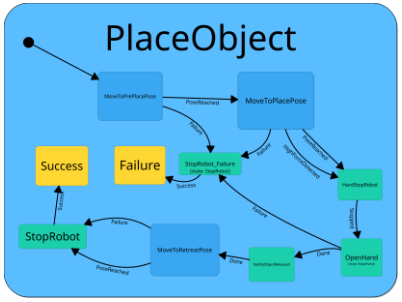


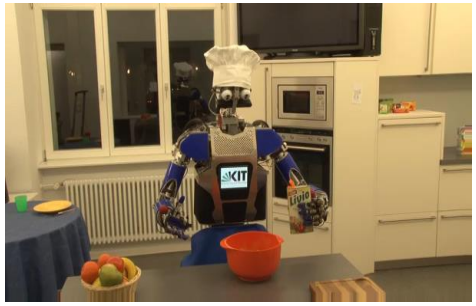
Manipulationsfähigkeiten mittels Statecharts

- ARMAR-III: Zur Zeit ~330 Zustände implementiert









Inhalt

- Motivation
- Klassische Roboterprogrammierverfahren (Übersicht)
- Graphisches Programmierverfahren: Statecharts
- **Symbolische Planung**

Symbolische Planung - Herkunft

- Wortbedeutung von „Plan“
  - Duden: Plan, *Substantiv*
    - *„Vorstellung von der Art und Weise, in der ein bestimmtes Ziel verfolgt, ein bestimmtes Vorhaben verwirklicht werden soll“*
    - Absicht, Vorhaben
    - (DDR) verbindliche Richtlinie für die Entwicklung der Volkswirtschaft
    - Entwurf in Form einer Zeichnung oder grafischen Darstellung, in dem festgelegt ist, wie etwas, was geschaffen oder getan werden soll, aussehen, durchgeführt werden soll
    - Übersichtskarte

Symbolische Planung

- Plan in der Robotik:
  - **Sequenz** von **parametrisierten Aktionen** zum Erreichen eines **definierten Ziels**
- Symbolisch:
  - Repräsentation des Weltzustandes durch **Booleschen Prädikaten** statt kontinuierlichen Werten
- Planung:
  - Finden einer Aktionssequenz zur Lösung eines Zielzustandes
- Klassisches Planungsbeispiel: **Blockwelt**

Zustand:
on(A, table)
on(B, table)

Ziel: on(A,B)
Plan: grasp(A)
stack(A, B)

Symbolische Planung - Problemstellung

- Herausforderung:
  - Wie kann man das **Planungsproblem** so formulieren, dass
    - eine Lösung **einfach/schnell** zu finden ist?
    - die **Existenz** einer Lösung bewiesen/widerlegt werden kann?

- Klassischer Lösungsansatz:
  - Wissensmodellierung:
    - Beschreibungssprache der Umwelt und Aktionen (STRIPS, ADL, PDDL,…)
  - Suche:
    - Suchalgorithmen zum Finden der gültigen Aktionssequenz

- Probleme:
  - Suchraum ist sehr groß (Branching factor)
  - Abbildung der realen Welt auf Symbole „Symbole Grounding Problem“

- Mehr Details in z.B. „M. Ghallab, D. Nau, and P. Traverso, Automated planning: theory & practice. Elsevier, 2004”

STRIPS

- **„ST**anford **R**esearch **I**nstitute **P**roblem **S**olver“ (Fikes, 1971)
- Bekannteste und eine der ältesten Sprache zur Beschreibung von Planungsdomänen. Wurde in 1971 von Fikes und Nilson zur Steuerung des Roboters „Shakey“ entwickelt
- Sehr einfach, daher auch eingeschränkt

- *Action Description Language* (ADL) und *Planning Domain Definition Language* (PDDL) sind von STRIPS abgeleitet

- Bestandteile von STRIPS (auch Planungsdomäne genannt)
  - Zustände
  - Aktionen
  - Ziele

R. Fikes and N. Nilsson (1971). STRIPS: a new approach to the application of theorem proving to problem solving. Artificial Intelligence, 2:189-208.

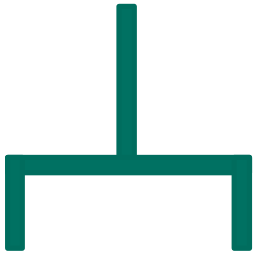
STRIPS: Symbole

- *Funktionsfreie* Sprache erster Ordnung (Prädikatenlogik erster Stufe)
  - Endliche Anzahl an Prädikaten und Konstanten, keine Funktionssymbole
  - Endliche Anzahl an Operatorensymbolen

- Beispiel: Blockwelt
  - Konstantensymbole: A, B, C, … (Namen der Blöcke)
  - Variablensymbole: $u, v, x_1, x_2, \ldots$
  - Prädikate:
    - $handempty$
    - $ontable(bottle)$
    - $on(bottle, table)$
    - $in\ (water, cup)$
    - $at(robot, fridge)$
    - …
  - Operatorensymbole: $pickup, putdown, stack, unstack, grasp, move, \ldots$

STRIPS: Zustände

Repräsentation vom **Zustand der Welt**



- Konjunktion positiver aussagenlogischer Literale aus Prädikaten und Konstantensymbolen
  - $A \wedge on(A, B)$

Zustand:
$ontable(A) \wedge ontable(B) \wedge handempty$

- Einschränkungen:
  - Endliche Anzahl an Literalen
  - *Keine* Variablen: ~~$on(x, B)$~~
  - *Keine* negativen Literale: ~~$\neg on(A, B)$~~
  - *Keine* Funktionen: ~~$on(on(B), B)$~~

- Geschlossene Welt (Closed World Assumption)
  - Jeder Zustand muss vollständig bekannt sein und beschrieben werden können
  - Nicht vorkommende Literale gelten als negative (negiert) Literale

STRIPS: Ziele

Repräsentation von **Zielen**

- Ein Ziel ist ein teilweiser spezifiziert Zustand
  - **Konjunktion** von **positiven** Literalen: $ontable(B) \wedge handempty$
- Erfüllung von Zielen
  - Alle Literale des Zieles müssen im Weltzustand vorkommen

$$ontable(B) \wedge handempty \wedge on(A, B)$$

erfüllt das Ziel

$$ontable(B) \wedge handempty$$

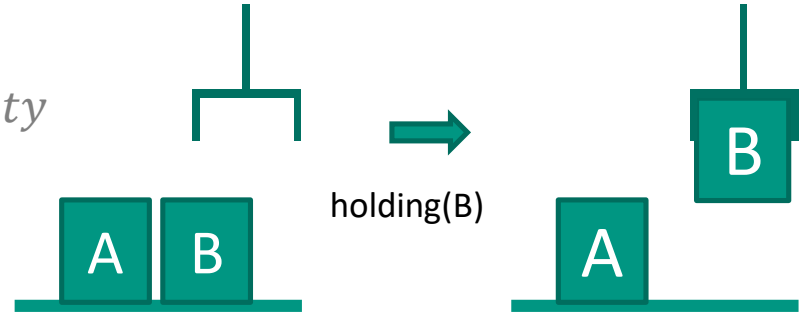
STRIPS: Aktionen

Aktion: Tripel bestehend aus (Deklaration, Vorbedingungen, Effekte)

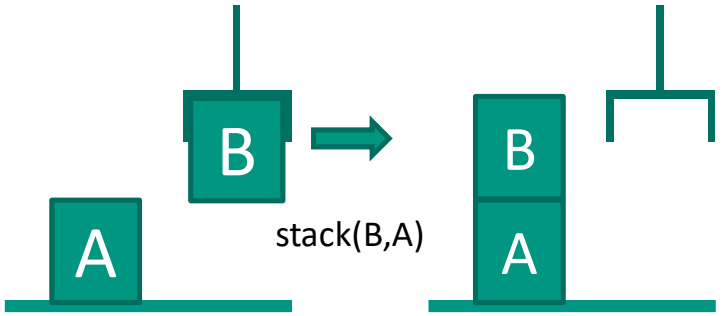
- Deklaration: Name und Parameterliste
  - Beispiel: Greife(h,o,l)
  - Eindeutiger Name
  - Parameterliste muss alle Variablen enthalten, die in den Vorbedingungen und Effekten benutzt werden
- Vorbedingungen V_A (Preconditions)
  - Konjunktionen von Literalen, die wahr sein müssen, um einen Operator anwenden zu können
    - Negative Literale erlaubt
    - Nur Variablen in Literalen
  - Aktion kann nur angewendet werden, wenn alle Vorbedingungen vollständig erfüllt
- Effekte E_A (Effects)
  - Auswirkungen der Aktion auf den Weltzustand
  - Liste von positiven oder negativen Literalen; auch bekannt als Hinzufüge-Listen (add-List) und Lösch-Listen (Delete-List) von positiven Literalen

STRIPS: Aktionen - Beispiele

- grasp(object):
  - Vorbedingung: $clear(object) \wedge handempty$
  - Effekte:
    - Füge hinzu: $holding(object)$
    - Lösche: $clear(object) \wedge handempty$



- stack(object1, object2):
  - Vorbedingung: $clear(object2) \wedge holding(object1)$
  - Effekte:
    - Füge hinzu: $\mathrm{on}(object1, object2) \wedge handempty$
    - Lösche: $clear(object2) \wedge holding(object1)$



STRIPS: Ausführbarkeit

- Eine Aktion A ist ausführbar in allen Zuständen z, die die Vorbedingung V_A erfüllen

- **Ergebnis der Ausführung:**
  - Das Ergebnis z' der Ausführung der Aktion A auf einem Zustand z erhält man durch
    - Entfernen aller **negativen** Literale des Effekts E_A aus z
    - Hinzufügen aller **positiven** Literale des Effekts E_A zu z

STRIPS: Vor- und Nachteile

- Vorteile:
  - Einfache Beschreibungssprache
  - Einfache Planung
  - Lösbarkeitsbeweis einfach
- Nachteile:
  - Viele Restriktionen beim Modellieren
    - Geschlossene Welt
    - Nur positive Literale in Zuständen (Unbekanntes Wissen nicht modellierbar)
    - Keine **Quantoren** ($\forall, \exists$)und **Disjunktionen** ($\vee$) in Zielen, Vorbedingungen, Effekte
    - Keine Typisierung (Nur mit Prädikaten umsetzbar)
    - Nur Boolesche Prädikate
- ADL und PDDL
  - sind mächtiger
  - erlauben bessere Modellierung
  - aber auch komplexer

STRIPS: Suche im Zustandsraum

- Suche: Anwenden von *ausführbaren* Aktionen
- Suche ist einfach umzusetzen
  - Keine Funktionssymbole
    - Endlicher Zustandsraum
    - Standard-Algorithmen zur Baumsuche möglich
  - Transformation der Vorbedingungen
    - Effekte bijektiv → Suche auch rückwärts möglich

- Problem: Suchraum ist sehr groß
    - Branching factor: Anzahl möglicher Verzweigungen pro Zustand
    - Aktion mit n Parametern und m Konstanten in der Domäne
      - m^n Aktionskandidaten
      - Prüfung auf Ausführbarkeit → Alle ausführbaren Aktionen werden weiter verfolgt

Suche im Zustandsraum

- Suchalgorithmen
  - Tiefensuche
    - Findet meistens nicht den kürzesten Plan (z.B. Roboter fährt unnötig herum)
  - Breitensuche
    - Findet immer den kürzesten Plan
    - Hoher Speicherverbrauch -> Oft nicht anwendbar
  - Heuristik basierte Suche
    - Güte der Heuristik ist ausschlaggebend
    - Es gibt keine allgemeingültige Heuristik bis jetzt
  - *FastForward*-Planer (Hoffmann et al., 2001)
    - Gierige Vorwärtssuche mit Heuristik
    - Breitensuche um aus lokalen Minima zu entkommen
    - Findet nicht immer den kürzesten Plan, aber in vielen Fällen einen kurzen
  - Planungsgraphen
    - Analyse der Prädikate und Aktionen um eine Heuristik aufzustellen