Überblick und allgemeine Prinzipien

Eigenschaften von Software

- Korrektheit
  - Übereinstimmung eines Programms mit seiner Spezifikation
    - ohne Spezifikation ist die Korrektheitsfrage nicht entscheidbar
    - bei informaler Spezifikation ist die Korrektheitsfrage nicht eindeutig entscheidbar
- Zuverlässigkeit
  - dauerhafte Einsetzbarkeit, der Anwender kann sich erlauben von der SW abzuhängen
    - Zuverlässigkeit ist ein relatives Kriterium, das vom Einsatzzweck der SW und vom Schadenspotential der Nichtverfügbarkeit der SW ab
    - Zuverlässigkeit von SW wird bei neuer SW zuweilen nicht erwartet (Bananen-SW), wesentlicher Unterscheid zu fast allen anderen industriellen Produkten
  - Zusammenhang zwischen Korrektheit und Zuverlässigkeit

	zuverlässig	nicht zuverlässig
korrekt	alle Anforderungen spezifiziert und erfüllt	unspezifizierte Anforderungen
nicht korrekt	nicht korrekte Fälle treten nicht auf (Überspezifikation)	Fehlern verursachen Fehlverhalten

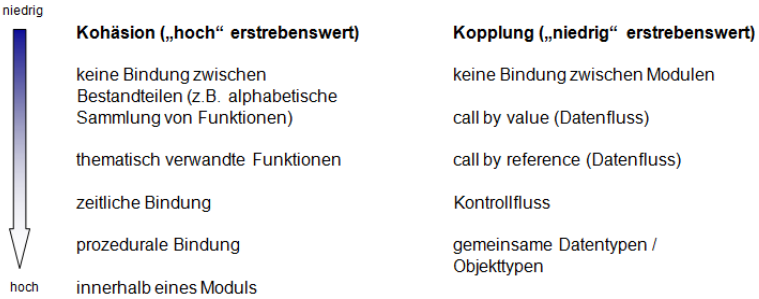
- Robustheit
  - Toleranz gegenüber nicht spezifizierter Bedienung / nicht spezifizierten Rahmenbedingungen
    - auch nicht robuste SW kann korrekt sein
    - nicht robuste SW kann leicht nutzlos werden
    - wesentliche Robustheitsanforderungen sollten deshalb spezifiziert werden
- Performanz
  - Erfüllung der Anforderungen an Antwortzeitverhalten
  - prinzipielle Überprüfungsmethoden: Messen, Berechnen, Simulieren
  - häufiger Umgang mit der Eigenschaft Performanz
    - Erstellen einer initialen Version
    - Verbesserung zum Zweck des Erreichens der notwendigen Performanz
    - Problem: deutliche Verbesserungen erfordern zuweilen grundlegendes Redesign
- Benutzungsfreundlichkeit
  - SW ist benutzungsfreundlich, wenn die Anwender sie für einfach zu benutzen halten
    - Gestaltung der Dialoge, einfache Konfigurierbarkeit, einleuchtender Aufbau
  - alles andere als das Empfinden der Anwender zählt nicht
  - häufiger Ansatz: Standardisierung der GUI
- Usablity
  - Usability eines Produktes ist das Ausmaß, in dem es von einem bestimmten Benutzer verwendet werden kann, um bestimmte Ziele in einem bestimmten Kontext effektiv, effizient und zufriedenstellen zu erreichen
  - testbare Attribute finden, um Anforderungen an SW abzuleiten: Erlernbarkeit, Effizienz, Reliabilität, Zufriedenheit
  - Anforderungen an Usability: Zugänglichkeit, Anpassbarkeit, Konsistenz, Fehlermanagement, explizite Nutzerkontrolle, Anleitung, minimiere kognitive Last, natürliche Bilder, Feedback
- Wartbarkeit
  - leichte Handhabbarkeit einer SW nach ihrer Auslieferung

  - Arten der Wartung
    - korrektive Wartung: Beseitigung von Fehlern
    - adaptive Wartung: nachträgliche Anpassung an neue Anforderungen
    - perfektive Wartung: Verbesserung im Hinblick auf nicht-funktionale Anforderungen
  - Wartbarkeit hängt stark von Strukturierung ab
  - Wartbarkeit nimmt zumeist mit der Lebensdauer der SW ab
- Wiederverwendbarkeit
  - Wahrscheinlichkeit, mit der SW in einem anderen Kontext wiederverwendet werden kann (oft bezogen auf Komponenten)
  - Wiederverwendbarkeit entsteht nicht zufällig, sondern muss geplant sein (infrastrukturell unterstützt werden)
  - Beispiele: parametrisierte Datenstrukturen, Klassenbibliotheken
- Portierbarkeit
  - die Portierbarkeit einer SW ergibt sich aus dem Aufwand, der nötig ist, um eine SW auf einer anderen Plattform lauffähig zu machen (im Verhältnis zu ihrem Entwicklungsaufwand)
  - hoher Grad an Portierbarkeit durch Verkapselung von Plattformabhängigkeiten, vgl. Ansätze der modellgetriebenen SW-Entwicklung
- Interoperabilität
  - gegeben, wenn sich eine SW mit geringem Aufwand mit anderer SW integrieren lässt
  - PC-Werkzeuge sind mittlerweile großenteils interoperabel
  - hochintegrierte SW-Systeme sind meist weniger interoperabel, weil sie „alles“ können
  - Interoperabilität wird immer mehr zu einem Muss, weil kaum noch SW in völlig neuen Gebieten eingesetzt wird

SW Engineering Prinzipien

- Prinzip = Grundsatz, den man seinem Handeln zugrundelegt
- Technik = Vorschrift zur Durchführung einer Tätigkeit (was ist wie zu tun)
- Methode = planmäßig anwendbare, begründete Technik zur Erreichung vorgegebener Ziele
- Sprache = syntaktische Regeln zur Unterstützung einer Methode oder Technik, eine Sprache besteht aus ihrer Syntax und ihrer Semantik
  - formale Sprachen haben eine formale Syntax- und Semantikdefinition (z.B. Aussagenlogik)
  - semiformale Sprachen haben eine formale Syntax, aber keine klar definierte Semantik (UML)
  - informale Sprachen haben weder eine formale Syntax noch Semantik (Deutsch, Englisch)
- Werkzeuge = Rechnerunterstützung für Techniken und Methoden
  - Prinzipien → Methoden/Techniken → Sprachen → Werkzeuge
  - Dekomposition → OO/funktional → UML/SADT → Together J/Prados
- Strukturierung
  - separation of concerns / Unterteilung in Aspekte
  - Ziel: Komplexität handhabbar machen
  - Beispiele von Aspekten: Funktionalität, Robustheit, Performanz und Ressourcenverbrauch, Organisation der SW-Entwicklung, Konfigurations-Management
  - Abhängigkeit zwischen Aspekten: Performanz und Datenintegrität, Funktionalität und Zielplattform
  - Arten der Unterteilung
    - zeitliche Unterteilung: Anforderungsanalyse/Entwurf/Programmierung/Test
    - qualitative Unterteilung: Effizienz/Robustheit/Korrektheit

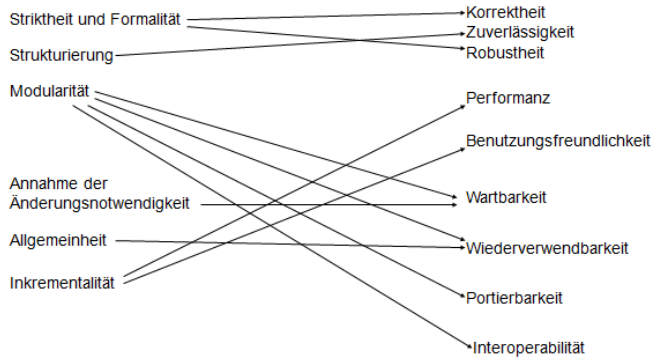
        - perspektivische Unterteilung: Verwendung von Datenstrukturen/Datenfluss/Kontrollfluss
        - Dekomposition (Unterteilung in Bestandteile): Komponente 1, Komponente 2 (→ diese Unterteilung ist so wesentlich, dass sie unter Modularität gesondert betrachtet wird)
    - Bewertung
        - +: Überschaubarkeit und Handhabbarkeit
        - -: globale Optimierungen geraten außer Sicht
        - Pragmatik: übergreifende Entscheidungen müssen vor der Unterteilung und der lokalen Bearbeitung der Aspekte getroffen werden
- Modularität
    - Unterteilung eines komplexen Systems in Komponenten / Module
    - Modularität
        - als Ergebnis der Dekomposition
        - als Grundlage der Komponierbarkeit
        - als Voraussetzung für Wiederverwendung und für lokale Änderungen
    - Bestandteile eines Moduls: Operationen, Objekttypen, Beziehungen zwischen Objekttypen
    - Ziel: hohe Kohäsion, geringe Kopplung



- Abstraktion
    - Trennung von wichtigen und unwichtigen Merkmalen, Konzentration auf das Wesentliche
    - Wichtigkeit und Unwichtigkeit ist relativ zum Zweck der Abstraktion
    - gängige Abstraktionen
        - die Signatur einer Operation abstrahiert von der Realisierung der Operation
        - die Konstrukte einer Programmiersprache abstrahieren von Prozessorendetails
        - ein Datenflussdiagramm abstrahiert von den Aufrufstrukturen zwischen Komponenten
- Änderbarkeit / Annahme der Änderungsnotwendigkeit
    - SW ist Gegenstand von Änderungen
    - Ursachen von Änderungen
        - korrektive/perfektive/adaptive Wartung
        - Erweiterung der Funktionalität wegen Erkenntnisgewinn während der Entwicklung
    - Chance der Änderbarkeit: flexible Erfüllung zusätzlicher Kundenwünsche, Produktfamilien
    - Risiken derÄnderbarkeit: alles wird geändert!
    - Entwurf änderbarer SW (wesentlicher Unterschied zu Fertigungsprozessen)
    - häufige Änderungen als Ursachen des Konfigurations-Managements
    - Änderungen als Ursache für Wartungsintensität und Wartung als eigenständiges Berufsbild
- Allgemeinheit
    - SW wird durch Änderungen für verwandte Aufgaben und Zwecke eingesetzt
    - „allgemeine“ SW lässt sich für verschiedene, verwandte Zwecke einsetzen, Rahmenbedingungen können als Parameter ergänzt werden
    - Analogie zur Entwicklung von Küchen und Autos
    - Allgemeinheit verursacht Aufwand

- Inkrementalität
  - Inkrementalität einer Tätigkeit bedeutet, dass diese Tätigkeit in Schritten vorgenommen wird, nach jedem Schritt wird rückgekoppelt
  - basiert auf der Annahme, dass Änderungen nötig sind (und Aufwand verursachen)

Prinzipien und Ziele



Spezifikation

Modellierungskonzepte

- Modellierung eines Systems durch Sichten

Sicht	Konzept	Notation
Funktional	Funktionshierarchie	Funktionsbaum
	Arbeitsablauf	Aktivitätsdiagramme, EPKs
	Informationsfluss	Datenflussdiagramm
Datenorientiert	Datenstrukturen	Data Dictionary
	Entitätstypen und Beziehungen	Entity Relationship Diagramm
Objektorientiert	Klassenstrukturen	Klassendiagramm
Algorithmisch	Kontrollstrukturen	Pseudocode, Programmablaufplan, Struktogramm
Regelbasiert	Wenn-dann-Strukturen	Regeln, Entscheidungstabellen
Zustandsbasiert	Endlicher Automat	Zustandsautomat, Aktivitätsdiagramm
	Nebenläufige Strukturen	Petri-Netz
Szenariobasiert	Interaktionsstrukturen	Sequenz-/Kollaborationsdiagramm

- weitere Modellierungskonzepte
  - algebraische Spezifikation: formale Spezifikation des Verhaltens von Modulschnittstellen
  - Object-Z: formale OO-Spezifikation des Systemverhaltens
  - OOA: abgeleitet aus Modellierungskonzepten ER, Klassenstrukturen, Kontrollstrukturen, Endlicher Automat, Interaktionsstrukturen, Arbeitsablauf
  - Model Driven Architecture: Spezifikation modellgetriebener Architekturen
- Anforderungen an gute Spezifikationsmethoden
  - verständliche und überschaubare Spezifikationstexte
  - präzise und eindeutige Semantik
  - Abstraktion von irrelevanten Details
  - Erlernbarkeit, Problemangemessenheit

Was bedeutet Spezifizieren?

- Spezifizieren bedeutet beschreiben. Letztlich geht es in der gesamten SW-Entwicklung um Beschreibung! Wir spezifizieren einen Entwurf / Testfall / Testergebnis / Benutzerhandbuch.
- Im Folgenden verstehen wir unter dem Spezifizierer denjenigen, der – aufbauend auf dem Anforderungsdokument – das *Verhalten* des zu entwickelnden SW-Systems beschreibt.
- Unter der Spezifikation (auch Verhaltensspezifikation) verstehen wir eine Beschreibung des Verhaltens des SW-Systems. Die Spezifikation enthält keine Angaben darüber, wie dieses Verhalten realisiert werden soll.
- das Ergebnis der Spezifikation ist das Spezifikationsdokument (kurz: Spezifikation)
- Hinweis: Unterscheidung zwischen Spezifikation und Anforderungsanalyse! Auch wenn in der Praxis beide Aktivitäten (und auch beide Rollen) eng miteinander verzahnt sind, unterschieden wir im Folgenden dazwischen, weil beide Aktivitäten auf unterschiedliche Zielgruppen abzielen.
- ein mögliches Ergebnis der Spezifikation: prinzipielle Nicht-Machbarkeit
- Prototyping als Hilfsmittel der Spezifikationsüberprüfung
- Unter dem Spezifizierer verstehen wir diejenige Rolle, deren Verantwortlichkeit es ist, das Problem aus Anwendersicht zu beschreiben. WAS soll mit der neuen Software getan werden, wen soll sie wobei unterstützen? Es geht nicht um das WIE einer möglichen Realisierung.
- Hinweis: Die genannte Trennung zwischen WAS und WIE ist üblich und eine nützliche Richtlinie, aber nicht 100%ig trennscharf. *Manchmal* müssen ein paar Details des WIE beschrieben werden, um das WAS zu verstehen.
- Trennung zwischen Problembeschreibung und Beschreibung des Modells des Problems
- Beispiel: Spezifikation der SW-Steuerung für einen Telefondienst im Hotel
  - Alternative 1: Um ein Ferngespräch zu führen, muss der Anwender den Hörer abnehmen. Nach maximal 3 Sekunden ertönt ein Ton. Der Anwender wählt eine 9. Nach maximal drei weiteren Sekunden ertönt ein Freizeichen und der Anwender kann eine Telefonnummer wählen.
  - Alternative 2: Das System umfasst vier Zustände: wartend, Ton, Freizeichen, Verbunden. Um vom Zustand wartend in den Zustand Ton zu kommen, muss der Anwender den Hörer heben. Um vom Zustand Ton in den Zustand Freizeichen zu kommen muss der Anwender eine 9 wählen.
  - daraus folgt:
    - eine natürlichsprachliche Beschreibung muss das Problem beschreiben, nicht das formale Modells des Problems, ansonsten wird der Entwurf vorweggenommen
    - Alternative 1 ist eine sinnvolle Spezifikation
    - Alternative 2 ist eine Entwurfsbeschreibung (sie beschreibt das WIE, nicht das WAS!)
- formale Spezifikation
  - Entwicklung einer formalen Spezifikation vermittelt zusätzliche Erkenntnis
  - Eindeutigkeit/Chance der Verifizierbarkeit (Vollständigkeit, Widerspruchsfreiheit, Redundanz)
  - Handhabbarkeit und Anwendbarkeit auf Probleme relevanter Komplexität
  - Notwendigkeit von Werkzeugunterstützung
- Ablauf der SW-Entwicklung: Vorstellung (Nachdenken), Darstellung (Spezifizieren), Herstellung (Implementieren)
- Aufgaben der Spezifikation im Entwicklungsprozess
  - Korrektheit der Spezifikation muss nachprüfbar sein
  - d.h. Korrektheit der Implementierung muss nachprüfbar sein
  - automatische Entwurfshilfen (z.B. Prototypen) aus Spezifikation ableitbar

Algebraische Spezifikation

das Spezifikationsproblem

- betrachtet abstrakte Datentypen als algebraische Strukturen, d.h. als Mengen mit darauf definierten Operationen, für die bestimmte Axiome gelten
- dient zur formalen Spezifikation des Verhaltens von Modulen/-Schnittstellen (Unterschied zur Spezifikation des Verhaltens eines gesamtes SW-Systems)
- dient der Verifikation der Implementierung gegen die Spezifikation
- durch das Zusammensetzen solcher Module kommt man zu SW-Systemen, deren Verhalten durch die Spezifikation der Module und durch die Konstruktion der Module zu einem SW-System definiert ist
- die Konstruktion von SW-Systemen aus Modulen ist Gegenstand des Entwurfs
- algebraische Spezifikation eines abstrakten Datentyps besteht aus einem Syntax- und einem Semantikteil
  - Syntax: Menge von Vereinbarungen von Zugriffsoperationen
    - Operationsname: Definitionsbereich → Wertebereich
  - Semantik: Menge von Gleichungen, die Beziehungen zwischen Zugriffsoperationen in Form von Axiomen beschreiben

Beispiel Kellerverwaltung

algebra stack-of-nat0 **introduces sorts** nat0, bool, stack-of-nat0;
operations
create: → stack-of-nat0
push: stack-of-nat0, nat0 → stack-of-nat0
pop: stack-of-nat0 → stack-of-nat0
top: stack-of-nat0 → nat0
isempty: stack-of-nat0 → bool
constraints create, push, pop, top, isempty
so that for all st: stack-of-nat0, n: nat0
isEmpfty(create()) = true
isEmpfty(push(st, n)) = false
pop(create()) = create()
pop(push(st, n)) = st
top(create()) = 0
top(push(st, n)) = n
end stack-of-nat0;

Beispiel Spezifikation des Datentyps String

- Ziel: Beschreiben was mit Zeichenketten passieren kann
  - Erzeugen (new), Sorte String wird gebraucht
  - Verketten (append)
  - Zeichen anhängen (add), Sorte Char wird gebraucht
  - Länge ermitteln (length), Sorte Nat wird gebraucht
  - Ermitteln, ob leer (isEmpty), Sorte Bool wird gebraucht
  - Gleichheit zweier Zeichenketten (equal)

algebra StringSpec **introduces sorts** String, Char, Nat, Bool;
operations
new: → String
append: String, String → String
add: String, Char → String
length: String → Nat
isEmpty: String → Bool
equal: String, String→ Bool

Syntax

- algebra StringSpec legt nur die Syntax fest, aber was bedeuten die Operationen?
- die Semantik der Operationen wird durch Gleichungen beschrieben
- Diese Gleichungen formulieren Eigenschaften, die von der Realisierung der Operationen nicht verletzt werden dürfen. Die Gleichungen heißen deshalb auch Axiome.
- Alle späteren Realisierungen der Operationen, die die Axiome nicht verletzen, sind spezifikationskonform! Lücken in den Axiomen sind deshalb fast zwangsläufig die Ursache für spätere Missverständnisse.

constraints new, append, add, length, isEmpty, equal
so that for all s, s1, s2: String, c: Char
isEmtpy(new()) = true
isEmpty(add(s, c)) = false
length(new()) = 0
length(add(s, c)) = length(s) + 1
append(s, new()) = s
append(s1, add(s2, c)) = add(append(s1, s2), c)
equal(new(), new()) = true
equal(new(), add(s, c)) = false
equal(add(s, c), new()) = false
equal(add(s1, c), add(s2, c)) = equal(s1, s2)
end StringSpec;

Semantik

- zur Ermittlung der erforderlichen Axiome und zur Sicherstellung der Beweiskraft von Induktionsbeweisen wird zwischen erzeugenden und inspizierenden Operationen unterschieden
  constraints new, append, add, length, isEmpty, equal
  so that StringSpec generated by [new, add] **for all** s, s1, s2: String, c: Char

Vollständigkeit der Algebra

- Erweiterung der Algebra String um Konstanten ‘a’, ‘b’ (formal: a: → ‘a‘)
- Frage: equal(add(s, ‘a‘), add(s, ‘b‘)) = false?
- Sollte intuitiv gelten, lässt sich aber nicht beweisen. Die angegebene Algebra ist unvollständig, weil sich nicht alle als wahr angenommenen Aussagen beweisen lassen.
- Vervollständigung durch zusätzliche Operation
  equalC: Char, Char → Bool
  mit den Axiomen
  equalC(‘a‘,‘a‘) = true
  equalC(‘a‘,‘b‘) = false
  und der Ersetzung des Axioms
  equal(add(s1, c), add(s2, c)) = equal(s1, s2) durch:
  equal(add(s1, c1), add(s2, c2)) = equal(s1, s2) ∧ equalC(c1, c2)

Beispiel Texteditor

- Operationen
  - Erzeugen einer neuen Datei: newF
  - Testen, ob eine Datei leer ist: isemptyF
  - Anfügen einer Zeichenkette an eine Datei: addF
  - Einfügen einer Zeichenkette an einer bestimmten Position einer Datei: insertF
  - Hintereinanderhängen zweier Dateien: appendF

algebra TextEditor **introduces sorts** Text, String, Char, Bool, Nat;

operations

newF: () → Text
isEmptyF: Text → Bool
addF: Text, String → Text
insertF: Text, Nat, String → Text
appendF: Text, Text → Text
lengthF: Text → Nat
equalF: Text, Text → Bool
addFC: Text, Char → Text (Hilfsoperation für addF)

constraints newF, isEmptyF addF, insert, append, lengthF, equalF, addFC

so that for all f, f_1, f_2: Text; s: String; c: Char; cursor: Nat

isEmpftyF(newF()) = true;
isEmptyF(addFC(f, c)) = false;
addF(f, newS()) = f;
addF(f, addS(s, c)) = addFC(add(f, s), c);
lengthF(new F()) = 0;
lengthF(addFC(f, c)) = lengthF(f) + 1;
appendF(f, newF()) = f;
appendF(f_1, addFC(f_2, c)) = addFC(appendF(f_1, f_2), c);
equalF(newF(), newF()) = true;
equalF(addF(), addFC(f, c)) = false;
equalF(addFC(f, c), newF()) = false;
equalF(addFC(f1, c1), addFC(f2, c2)) = equalF(f1, f2) ∧ equalF(c1, c2);
insertF(f, cursor, newS()) = f;
(((equalF(f, appendF(f_1, f_2) ∧ (lengthF(f_1) = cursor-1))
⟹ equalF(insertF(f, cursor, s), appendF(addF(f_1, s), f_2))) = true;

end TextEditor;

Signatur und Algebra

- Definition „Algebraische Struktur“
  - $\Sigma = (S, F)$ heißt algebraische Struktur bzw. Signatur
    - S eine Menge von Sorten, z.B. $S = \{Nat, Bool\}$
    - F eine Menge von Operationssymbolen, z.B. $F = \{zero, add, equal\}$
    - auf F ist eine Abbildung definiert $type: F \rightarrow S^* x\ S$
    - für $type(f) = s_1, \dots, s_n, s)$ schreiben wir $f: s_1, \dots, s_n \rightarrow s$
    - Konstanten sind die Abbildungen aus F, deren Vorbereich die leere Menge ist

- Definition Untersignatur
  - seien zwei Signaturen $\Sigma = (S, F)$ und $\Sigma' = (S', F')$ mit den Eigenschaften $S \subseteq S'$ und $F \subseteq F'$ gegeben
  - dann heißt Σ Untersignatur von Σ'
- Algebren
  - eine Algebra interpretiert eine Signatur Σ, indem sie die Symbole in Σ mit konkreten Mengen und Abbildungen füllt
  - Beispiel

Signatur Σ-Test	Σ-Algebra A
Nat	$A_{Nat} =_{def} N$
Bool	$A_{Bool} =_{def}$ {true, false}
zero: → Nat	$zero_A =_{def} 0$
one: → Nat	$one_A =_{def} 1$
succ: Nat → Nat	$succ_A (n) =_{def} n + 1$
add: Nat x Nat → Nat	$add_A (n, m) =_{def} n + m$
equal: Nat x Nat → Bool	$equal_A (n, m) =_{def}$ true if n = m else false
equalBool: Bool x Bool → Bool	$equalBool_A (a, b) =_{def}$ true if a = b else false
T: → Bool	$T_A =_{def}$ true
F: → Bool	$F_A =_{def}$ false

  - für alle $s \in S$ sei A_s eine Menge, dann bezeichnet man A_s auch als Trägermenge von A zu s
- „Undefiniert“
  - Wie lassen sich Operationen explizit als „undefiniert“ kennzeichnen?
  - Beispiel Nulldivision oder erstes Element einer leeren Liste
  - Variante 1: ein undefiniertes „bottom“-Element wird per Definition jeder verwendeten Menge zugeordnet
  - Variante 2: ein undefiniertes Element wird bei Bedarf explizit eingeführt

    algebra stack-with-bottom **introduces sorts** nat0, bool, stack-of-nat0;

    operations

    create: → stack-of-nat0

    bottom: → stack-of-nat0

    isbottom: stack-of-nat0 → bool

    …

    constraints create, bottom, isbottom, …

    so that for all st: stack-of-nat0, n: nat0

    isbottom(top(create()) = true

    isbottom(top(st)) = isempty(st)

    …

- homogene/heterogene Algebra
  - eine homogene Algebra besteht aus genau einer Menge (Trägermenge) und einer Menge von Operationen
    - Σ-Algebra B zu Signatur Σ-Test
    - Gruppen, Ringe, Körper, Verbände
  - eine heterogene Algebra besteht aus mehr als einer Menge (Trägermenge) und einer Menge von Operationen
    - Σ-Algebra A zu Signatur Σ-Test
    - Vektorräume (Vektor und Skalare als Träger)

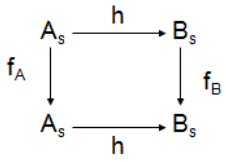
Homomorphismen

- Isomorphie = strukturelle und symmetrische Gleichheit von Mengen, gegenseitige Ersetzbarkeit
- Homomorphie = asymmetrische Gleichheit
- Beispiel

Σ-Algebra $A = (A_s, f_A)$ Σ-Algebra $B = (B_s, f_B)$

$A_s = \{a,b,c\}$ $B_s = \{X,Y,Z\}$

$f_A = \{a \to b, b \to c, c \to a\}$ $f_B = \{X \to Z, Y \to Z, Z \to Z\}$



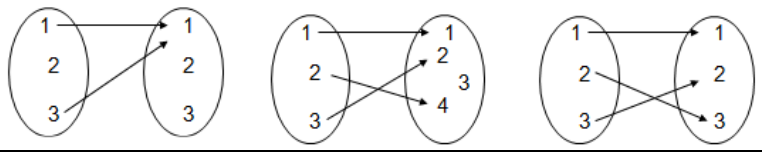
h Homomorphismus-Abbildung, wenn

- Wird a auf X abgebildet, so muss auch das Bild von a (nämlich b) auf das Bild von X (nämlich Z) abgebildet werden.

Damit gibt es nur einen Homomorphismus, nämlich den, der a,b,c auf Z abbildet.

- ein Homomorphismus drückt aus, wie sich die eine Algebraische Struktur A in einer zweiten B „gleichwertig“ (nicht ersetzbar) wiederfindet
- ein Homomorphismus umgekehrt von B nach A braucht nicht zwangsläufig zu existieren, B kann weitaus komplexer sein
- ein Homomorphismus ist eine strukturerhaltende, gerichtete Abbildung zwischen zwei Algebren
- ein Homomorphismus formuliert eine „Verträglichkeit“ zwischen zwei Algebren
- da Algebren nicht nur Mengen enthalten, sondern auch Abbildungen, geht der Begriff des Homomorphismus über die Verträglichkeit zweier Mengen hinaus → die Abbildungen müssen also auch miteinander verträglich sein
- spezielle Homomorphismen
  - Sei $\Sigma = (S, F)$ eine Signatur. Ein Σ-Homomorphismus $h: A \to B$ heißt
    - injektiv: die Werte des Abbildes sind höchstens einem Wert des Ur-Bildes zugeordnet
    - surjektiv: alle Werte des Abbildes haben mindestens eine Entsprechung im Urbild
    - bijektiv: umkehrbar eindeutig, gleichmächtige Mengen, injektiv und surjektiv gleichzeitig

nicht injektiv, nicht surjektiv | injektiv, nicht surjektiv | bijektiv



- ein Σ-Homomorphismus heißt isomorph, wenn alle Funktionen in h bijektiv sind, also: wenn Homorphisus bijektiv
- Idee des Isomorphie-Nachweises
  - Homomorphie zeigen
  - Bijektivität aller Funktionen aus der Familie der Funktionen des Homomorphismus zeigen (eine pro Sorte), also Injektivität und Surjektivität dieser Funktionen

Terme

- **Σ-Grundterm**
  - Signatur: Syntax
  - Algebra: Semantik
  - Terme: Wörter der Sprache
    - beschreibt syntaktische Struktur auf Ebene der Signatur
    - Interpretation durch die jeweilige Algebra, z.B. succ(add(n1, n2))
    - Grundterme: Terme ohne Variablen
    - Terme: Terme mit (oder ohne) Variablen
- **Auswertung** von Σ-Grundterm
  - Sei $\Sigma = (S, F)$ eine Signatur und A eine Σ-Algebra. Die Auswertung (Evaluation/Interpretation) $eval(A)$ der Terme zur Signatur Σ in A ist eine Familie von Abbildungen
    $$eval(A) = (eval(A)_s : T(\Sigma, \mathrm{s}) \to \mathrm{A_s})_{\mathrm{s}\in\mathrm{S}}$$
  - Beispiel
    $$eval(A)_{Nat}\left(succ\left(add(one, one)\right)\right)$$
    $$= succ_A\left(eval(A)_{Nat}\left(add(one, one)\right)\right)$$
    $$= succ_A\left(add_A\left(eval(A)_{Nat}(one), eval(A)_{Nat}(one)\right)\right)$$
    $$= succ_A\left(add_A(one_A, one_A)\right)$$
    $$= succ_A\left(add_A(1,1)\right)$$
    $$= succ_A(2)$$
    $$= 3$$
- Signatur mit **Variablen**
  - Sei $\Sigma = (S, F)$ eine Signatur. Eine Familie $X = (X_s)_{s\in\mathrm{S}}$ von Mengen, die durch S indiziert ist, heißt S-sortiert (S-inidiziert). Deren Elemente heißen Variablen.
  - Dann heißt $\Sigma = (S, F, X)$ Signatur mit Variablen.
- **allgemeine Terme**
  - ein Grundterm mit Variablen wird allgemeiner Term genannt
  - Variablen spielen dieselbe syntaktische Rolle wie Konstantensymbole, unterschieden sich jedoch in der Semantik
    - Konstanten haben in Algebra feste Bedeutung
    - Variablen müssen für eine Termauswertung erst belegt werden
- Auswertung von allgemeinen Σ-Termen
  - Sei $\Sigma = (S, F, X)$ eine Signatur mit Variablen, A eine Σ-Algebra und $ass: X \to A$ eine Variablenbelegung („assignment“).
  - Die erweiterte Auswertung $xeval(ass): T_\Sigma(\mathrm{X}) \to \mathrm{A}$ der Terme der zur Signatur Σ in A (bezgl. ass) ist eine Familie von Abbildungen.
    $$(xeval(ass)_s : T(\Sigma, \mathrm{s})(\mathrm{X}) \to \mathrm{A_s})_{\mathrm{s}\in\mathrm{S}}$$

- Beispiel
  - Signatur Σ-Nat, Algebra A
  - $X_{Nat} = \{n_1, n_2\}$
  - Variablenbelegeung $v: X \to A$, $v_{Nat}(n_1) = 1909, v_{Nat}(n_2) = 5$

$$
\begin{aligned}
&xeval(v)_{Nat}(add(succ(0), succ(n_2)) \\
&= add_A\left(xeval(v)_{Nat}(succ(0)), xeval(v)_{Nat}(succ(n_2))\right) \\
&= add_A\left(succ_A(xeval(v)_{Nat}(0)), succ_A(xeval(v)_{Nat}(n_2))\right) \\
&= add_A\left(succ_A(v_{Nat}(0)), succ_A(v_{Nat}(n_2))\right) \\
&= add_A(succ_A(0), succ_A(5)) \\
&= add_A(1{,}6) \\
&= 7
\end{aligned}
$$

Algebraische Spezifikation

- Spezifikation: Auflistung (syntaktisch formulierbarer Eigenschaften einer Algebra)
- Signaturen repräsentieren einfachste Form der Spezifikation
- Idee: Wie kann die Interpretation einer Signatur durch eine Algebra als Funktion dargestellt werden?
  - Bildung von Formeln über den Termen einer Signatur
- Spezifikationen erweitern also Signaturen um Formeln, die mit Hilfe von Termen syntaktisch formuliert werden
  - Algebraische Gleichungslogik
- **Σ-Gleichung / Grundgleichung**
  - Sei A eine Algebra mit der Signatur $\Sigma = (S, F, X)$. Für $t, t' \in T(\Sigma, X)_{s \in S}$ heißt $e \equiv t =_s t'$ eine Σ-Gleichung.
  - e heißt Grundgleichung, falls weder t noch t' Variablen enthalten
  - Beispiel: Signatur Σ-Nat, Σ-Algebra A
    - $eval(A)_{Nat}(zero) = eval(A)_{Nat}(add(zero, zero))$ gültig
    - $eval(A)_{Nat}(zero) = eval(A)_{Nat}(one)$ ungültig!
    - $xeval(ass)_{Nat}(add(n_1, n_2)) = xeval(ass)_{Nat}(add(n_2, n_1))$ gültig
- **Menge der wohldefinierten Formeln**
  - Zu einer Algebra mit der Signatur $\Sigma = (S, F, X)$ und einer S-sortierten Variablenmenge X wird die Menge der wohldefinierten Formeln über der Signatur Σ mit $WFF(\Sigma)$ bezeichnet
  - $WFF(\Sigma)$ ist die kleine Menge mit folgenden Eigenschaften
    1. Jede Gleichung ist in $WFF(\Sigma)$
    2. Mit $G, H \in WFF(\Sigma)$ sind auch $\neg G, (G \wedge H), (G \vee H) \in WFF(\Sigma)$
    3. Falls $x \in X_s, G \in WFF(\Sigma)$, dann ist auch $(\forall x: G), (\exists x: G) \in WFF(\Sigma)$
- **Algebraische Spezifikation**
  - Sei $\Sigma = (S, F, X)$ eine Signatur mit Variablen und E eine Menge von Σ-Gleichungen.
  - Dann heißt $SP = < \Sigma, E >$ algebraische Spezifikation, falls $E \subseteq WFF(\Sigma)$.
  - Beispiel: Algebraische Spezifikation Spec-Nat erweitert Signatur Σ-Nat um die Σ-Gleichungen
    - $equal_{Nat}(zero, zero) = true$
    - $equal_{Nat}(succ_{Nat}(n_1), succ_{Nat}(n_2)) = equal_{Nat}(n_1, n_2)$
    - $succ_{Nat}(n_1) = add_{Nat}(n_1, 1)$
    - $succ_{Nat}(zero) = one$
- **Modell** ist algebraische Spezifikation + Algebra A, wenn alle Gleichungen in A gültig sind

- Semantik einer Algebra
  - alles, was in der Semantik vorkommen soll, kann in der Signatur bezeichnet werden (**no-junk-Prinzip**)
    - Das heißt, eine Algebra, die Elemente in ihren Trägermengen hat, die wir mit Grundtermen nicht bezeichnen können, können wir mit der Signatur nicht "gemeint" haben, den sonst hätten wir eine "größere" Signatur ausgewählt.
    - Alles was von der Signatur (mittels der Grundterme) nicht erfasst wird, wird als „junk" bezeichnet.
    - Einschränkung von no-junk: lässt keine Trägermenge zu, die mächtiger als Nat ist
  - alles, was in der Semantik vorkommen soll, kann eindeutig in der Signatur bezeichnet werden (**no-confusion-Prinzip**)
    - Das heißt, wir interpretieren die durch eine Signatur gegebene Spezifikation minimal. Sie beschreibt nichts „Überflüssiges". Wäre eine Algebra mit der Signatur „gemeint" gewesen, die ein Element in einer Trägermenge hat, das Bald zweier verschiedener Grundterme ist, hätte die Signatur „kleiner" ausfallen sollen.
    - Einschränkung von no-confusion: verbietet verarbeitende Operationen
- Widersprüchlichkeit und Redundanz
  - eine Algebra ist widersprüchlich, wenn sich $true = false$ beweisen lässt
  - eine Algebra ist redundant, wenn die Menge der beweisbaren Aussagen gleich bleibt auch dann, wenn ein Axiom gestrichen wird

von einer Datenstruktur zum Modell

- Datenstruktur ist aus formeller Sicht eine Algebra
- Datenstruktur besteht aus verschiedenen Datenbereichen und einer Menge von Operationen auf diesen Bereichen, z.B. INT, REAL, NAT, STRING, STACK, QUEUE

1) Aufstellung einer Signatur Σ
   - Sorten (Datenbereiche) und Operationen (Methoden, Funktionen)
2) Formulieren von Σ (Grund-)Termen
   - ermöglichen die Bezeichnung der Elemente der Datenbereiche
   - no-junk-Prinzip: eine Algebra soll nur Elemente enthalten, die durch Σ-Terme bezeichnet werden können
     - Stellt unsere Signatur genügend syntaktische Ausdrücke bereit, um alle Elemente des Datenbereichs zu bezeichnen? → evtl. Signatur erweitern
     - Lassen sich alle Element der Implementierung (Algebra) durch die Signatur bezeichnen? → evtl. Algebra einschränken
3) Aufstellungen von Gleichungen
   - no-confusion-Prinzip: Elemente der Datenbereiche lassen sich durch verschiedene Grundterme beschreiben
   - gesucht: eine Menge E von Gleichungen, um die Gleichheit von Elementen zu beschreiben, die in der Algebra gültig ist (Korrektheit von E) und die Menge aller im Modell gültigen Grundgleichungen generiert (Vollständigkeit von E)
4) Ableiten von Algebren aus Spezifikation
   - aus Signatur kann eine „Grundtermalgebra" abgeleitet werden (initiale Semantik)
   - aus Spezifikation (Signatur + Gleichungen) kann eine „Quotientenalgebra" abgeleitet werden (initiale Semantik)
   - Quotientenalgebra ist idealerweise isomorph zu unserem Modell (Ausgangs-Algebra)

5) Homomorphismus und Algebra
   - wenn es eine strukturerhaltende Abbildung (Homomorphismus) von einer Quotiententermalgebra auf eine Algebra X gibt, dann entspricht X der Spezifikation
   - strukturelle Induktion als Werkzeug für den Bewies der Existenz einer solchen Abbildung

Vor- und Nachteile der algebraischen Spezifikation

Vorteile	Nachteile
▪ Spezifikation weitestgehend implementierungsunabhängig ▪ vollständig formale Spezifikation, daher für Verifikation und automatische Werkzeuge einsetzbar ▪ theoretisch fundiert ▪ kann auf abstrakte Datenobjekte, Datentypen und Klassen angewandt werden ▪ Wirkung von Zugriffsoperationen ergibt sich durch wechselseitige statische Abhängigkeiten, nicht durch isolierte Betrachtung ▪ mit einiger Erfahrung kann man algebraische Spezifikationen lesen ▪ leicht erweiterbar, da zusätzliche Operationen nur jeweils eine neue Syntaxregel sowie zusätzliche Axiome aufbauend auf Grundoperationen erfordern	▪ Konstruktion algebraischer Spezifikationen nicht trivial ▪ Fälle wie z.B. Grenzbedingungen können leicht übersehen werden ▪ schwierig auf Konsistenz und Vollständigkeit zu prüfen ▪ Operationen mit mehreren Wertebereichen sind schwierig zu beschreiben → In der Praxis hat sich die algebraische Spezifikation nicht durchgesetzt!

Object-Z

Was ist Object-Z?

- klassenbasierter Ansatz zur formalen Spezifikation objektorientierter Systeme
- Object-Z Klasse spezifiziert
  - die möglichen Zustände eines Klassenobjekts
  - die möglichen Zustandsübergänge
  - die Kommunikation des Objekts mit seiner Umgebung
  - Systeme durch Identifikation der zugehörigen Objekte und deren Interaktion
- Wiederverwendung möglich durch Vererbung, Instanziierung
- Erweiterung der formalen Spezifikationssprache Z auf konservative Weise, d.h. Syntax und Semantik von Z bleiben in Object-Z erhalten

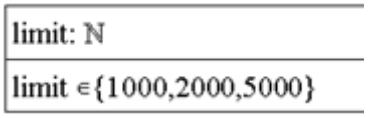
einführendes Beispiel

- Spezifikation eines Systems zur Verwaltung von Kreditkartenkonten
- d.h. Spezifikation der Basisfunktionalität der Kreditkartenkontenobjekte und deren Interaktion
  - Spezifikation der Kontenobjekte, deren Zustände und Operationen
    - Kontenobjekt gekennzeichnet durch zwei Zahlen: aktueller Kontostand und Kreditlimit
    - Operationen: Geld abheben, mit Karte einkaufen, Geld einzahlen
  - Spezifikation des Systems und Operationen zur Definition von Aktionen zwisch. den Objekten

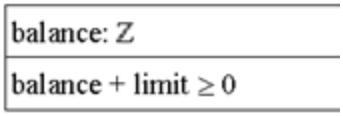
das Object-Z Klassenkonstrukt

- eine Klasse besteht aus 6 Teilen
  - Klassenname und -box, Sichtbarkeitsliste („Visibility List“), Definition von Konstanten, Definition von Objektzuständen und Veränderungen („State Schema“), Definition initialer Bedingungen („Initial Schema“), Definition von Operationen auf Objekten („Operation Schemas“)

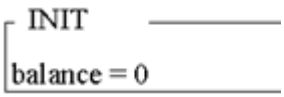
- Visibility List ⊦ (limit,balance,INIT,withdraw,deposit,withdrawAvail)
  - führt Eigenschaften auf, die sichtbar in der Umgebung eines Objekts der Klasse CreditCard sind
  - spezifiziert Schnittstelle des Objekts
  - auf Element der Schnittstelle kann von außen zugegriffen werden
  - falls keine Visibility List existiert, sind alle Eigenschaften implizit sichtbar
- Constant Definition (Open Schema Box)



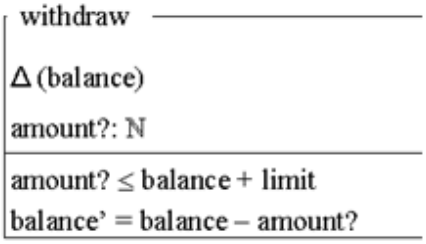
  - ist axiomatische Definition
    - Deklarationsteil (oben)
      - deklariert Konstantennamen und -typ
    - Prädikatenteil (unten)
      - Formeln der Prädikatenlogik 1. Ordnung
      - definiert Constraints auf den Konstante
- State Schema

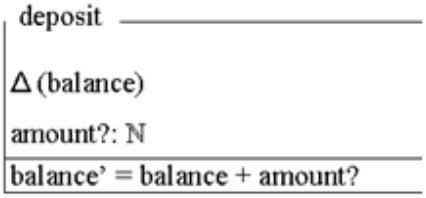


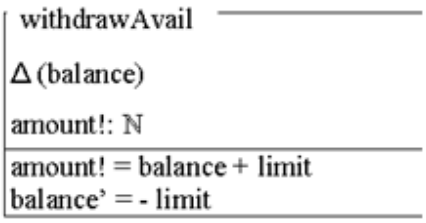
  - Deklaration von Zustandsvariablen (oben)
  - Prädikate auf Zustandsvariablen und Konstanten (unten)
  - Zustandsvariablen und Konstanten bilden zusammen die *Attribute* einer Klasse
  - Prädikate sind *Klasseninvarianten*, d.h. müssen stets erfüllt sein
  - Schema bestimmt eine Menge von gültigen Instanzen der Variablen (hier: balance)
- Initial Schema



  - trägt stets den Namen INIT
  - spezifiziert Prädikate auf Attributen
  - Prädikat zusammen mit den Klasseninvarianten definiert *initiale Bedingungen*
  - Objekte sind in sog. inititaler Konfiguration, falls die Werte der Attribute die initialen Bedingungen erfüllen
  - Initial Schema ist eine Bedingung, keine Operation
- Operation Schema
  - spezifizieren die Veränderungen eines Objekts der Klasse CreditCard
  - Δ (variable): Liste von (sog. primären) Variablen, die durch die Operation verändert werden
    - hier keine Konstanten erlaubt
    - nicht aufgeführte Variablen können nicht geändert werden
  - Einführung einer Menge von *Kommunikationsvariablen*, die Informationen zwischen Objekt und seiner Umgebung weitergeben
    - ?: Inputvariable, !: Outputvariable
  - Spezifikation der Attributwerte vor und nach einer Operation (z.B. withdraw spezifiziert den Wert von balance)
  - variable‘: variable nach Anwendung einer Operation
  - Prädikat (z.B. $amount? \leq balance + limit$) nicht erfüllbar, so ist Operation nicht anwendbar



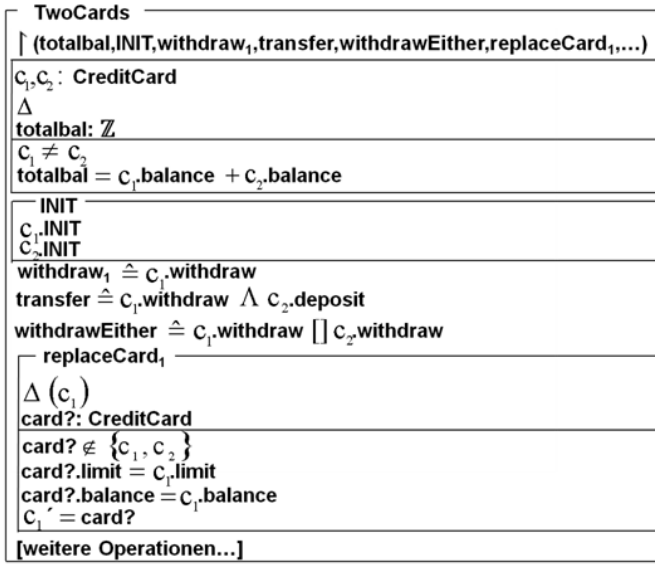




  - Auswirkungen einer Operation kann festgestellt werden durch Verknüpfung des Schema Prädikats mit den Klassenvariablen
  - in Klassenvariablen zusätzlich Variablen‘ einsetzen

Instanziierung und Interaktion von Objekten

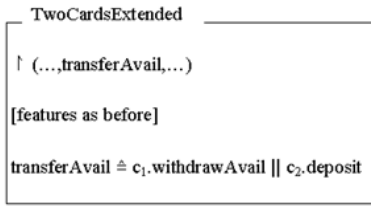
- bis jetzt: ein Objekt modelliert
- jetzt: Menge von Objekten und deren Interaktion modellieren



- Objekte als Attribute einer Klasse
  - c_1, c_2 sind Attribute und zugleich Objekte der Klasse CreditCard, die Werte dieser Attribute sind Objektidentitäten
  - c_1, c_2 sind zwei unterschiedliche Objekte
  - Identität (nie änderbar) unterscheidet sich vom Zustand (ersetzbar) eines Objekts
- sekundäre Variablen
  - platziert unter dem Δ Symbol
  - Wert ändert sich entsprechend der Änderungen von Konstanten und primären Variablen
- Initial Schema
  - Objekte der Klasse TwoCards sind in initialer Konfiguration, wenn Kreditkartenobjekte in initialer Konfiguration sind
- Promote von Operationen
  - $withdraw_1$ ist Operation von TwoCards
  - wird als Anwendung der $withdraw$-Operation auf $withdraw_1 \triangleq c_1.withdraw$ definiert (sog. Promote)
- Verknüpfungsoperator Λ (conjunction operator)
  - Wert $amount?$ ist Input für $withdraw$ UND $deposit$, d.h. wird von c_1 abgehoben und auf c_2 eingezahlt
    $transfer \triangleq c_1.withdraw \Lambda c_2.deposit$
  - Verknüpfungsoperator ist kommutativ und assoziativ
  - Kommunikationsvariablen werden bei Verknüpfung gleichgesetzt
    - Bsp: $withdrawAvailBoth \triangleq c_1.withdrawAvail \Lambda c_2.withdrawAvail$
    - Operation nur anwendbar, wenn gilt: $c_1.balance + c_1.limit = c_2.balance + c_2.limit$
    - Umbenennung von amount durch other umgehen
      $withdrawAvailBoth \triangleq c_1.withdrawAvail \Lambda c_2.withdrawAvail[other!/amount!]$
- Auswahloperator [] (choice operator)
  - wählt zwischen zwei Operationen aus $withdrawEither \triangleq c_1.withdraw[]c_2.withdraw$
  - Bedingungen
    - gleiche Kommunkationsvariable für beide Komponenten

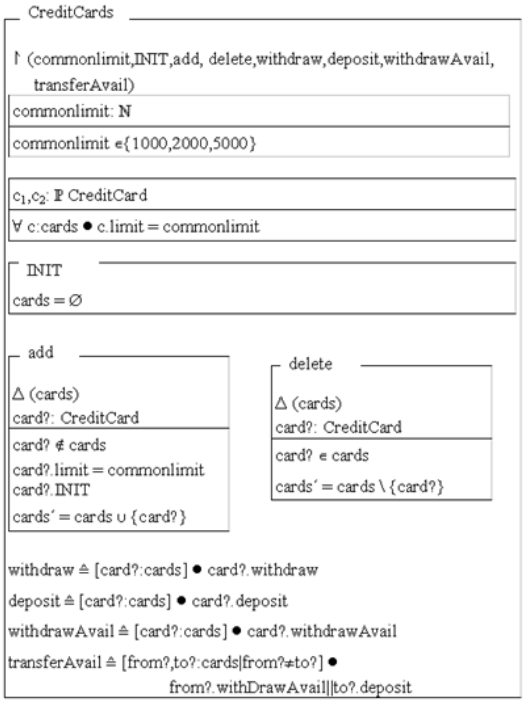
  - sind beide Operationen nicht anwendbar (weil bei beiden Konten der Betrag das Kontolimit übersteigt), ist die gesamte Operation nicht anwendbar
  - ist genau eine der Operationen anwendbar, wird diese angewendet
  - sind beide anwendbar, wird eine gewählt → nichtdeterministische „angelic choice“
- Auswahloperator ist kommutativ und assoziativ
- Objektersetzung
  - $replaceCard_1$ modelliert Ersetzung von Objekt c_1 durch Objekt $card?$

Inter-Objekt-Kommunikation



- Parralleloperator || (parallel operator)
  - Operator zur Modellierung der Interobjekt-Kommunikation
  - $transferAvail$ modelliert Abbuchung des max. Betrags von c_1 und Buchung auf c_2
  - Komposition der beiden Operationen $withdrawAvail$ und $deposit$ durch Parralleloperator ||
  - Parallel-Operator verhält sich wie Verknüpfungsoperator, aber erlaubt Inter-Objekt-Kommunikation zwischen den Inputs und Outputs der Operatoren, in beide Richtungen. Namensgleichheit ist bedeutsam.
  - Kommunikation über identische Kommunikationsvariablen
  - Inter-Objekt-Kommunikation funktioniert in beide Richtungen (sogar in einer Operation)
  - Parralleloperator ist kommutativ, aber nicht assoziativ
  - im Fall der Operation $transferAvail$ wird zunächst $withdrawAvail$ der Klasse $CreditCard$ auf das Objekt c_1 angewendet
  - Ergebnis: $amount!$
  - parallel wird $deposit$ auf c_2 angewendet, das erfordert eine Eingabe $amount?$
  - da $amount!$ und $amount?$ gleichgesetzt werden, kommt es hier zu der versteckten Kommunikation im Rahmen des Parallel-Operators
  - keine Assoziativität, da versteckte Kommunikation nur zwischen 2 Operationen möglich
  - ohne Inter-Objekt-Kommunikation verhält sich der Parallel- wie der Verknüpfungs-Operator

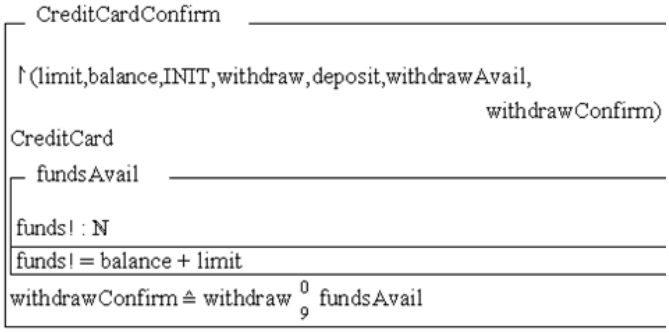
Objekt-Aggregation



- Aggregate als Attribute
  - Banksystem besteht aus einer Menge (Aggregation) von Objekten
  - Attribute als Aggregate
    - $cards$ ist Menge von Objekten der Klasse $CreditCard$
    - das Limit jedes Objekts der Menge $cards$ entspricht der Konstante $commonlimit$
    - initial ist die Menge $cards$ die leere Menge
- Hinzufügen und Löschen von Objekten
  - add spezifiziert Addition eines neuen Kreditkartenobjekts zur Menge $cards$
  - $delete$ spezifiziert das Entfernen eines Objekts aus der Menge $cards$
  - Annahme: add wird auf ein Objekt $cards$ der Klasse $CreditCards$ angewendet. Die Identität des hinzuzufügenden Objektswird über Input-Variable ermittelt.
  - Hinweis: im Unterschied zu objektorientierten Programmiersprachen kümmert sich niemand darum, ob das Objekt existiert, entsprechend werden Objekte auch nicht zerstört
- Selektion von Objekten
  - $SelectObjects \bullet OperationExpression$
    - $[card? : cards]$: Wähle ein Objekt $card?$ aus der Menge $cards$
    - führe auf diesem Objekt eine Operation durch

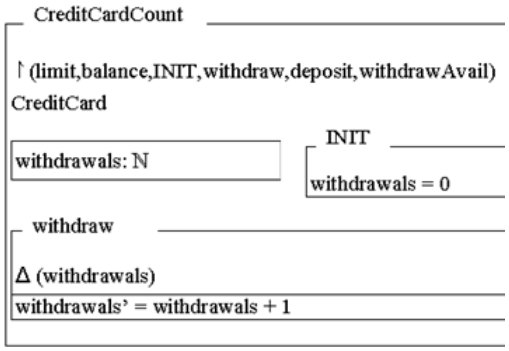
Vererbung

- Erweiterung von $CreditCard$ um eine Operation, die ein $withdraw$ durch die Ausgabe der verfügbaren Summe bestätigt

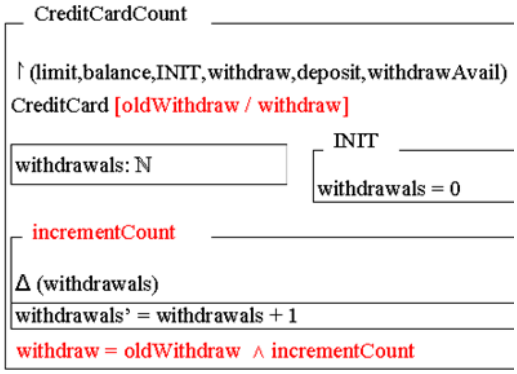


- Wiederverwendung der spezifizierten Eigenschaften von Object-Z Klassen (superclass) zur Spezifikation neuer Object-Z Klassen (subclass)
- neue Klasse $CreditCardConfirm$ erbt von der Klasse $CreditCard$, alle Features von $CreditCard$ werden Features von $CreditCardConfirm$
- Achtung: Visibility List wird nicht vererbt, neue Liste für $CreditCardConfirm$
- explizite Definition der Operationen $fundsAvail$ und $withdrawConfirm$, die nicht geerbt werden
- $fundsAvail$
  - Rückgabe des Überziehungslimits
  - nach außen hin (über Visibility List) nicht sichtbar, nur interne Hilfsoperation
- $withdrawConfirm$
  - Abbuchung von $amount?$ und Rückgabe des Überziehungslimits
  - $\overset{0}{9}$ sequentieller Kompositionsoperator
  - erlaubt unidirektionale Interobjekt-Kommunikation, deswegen nicht-kommutativ und nicht-assoziativ
  - der sequentielle Kompositionsoperator verhält sich wie ein Parallel-Operator, bei dem nur von links nach rechts weitergereicht werden kann

- Vereinigung von Eigenschaften



  - Anzahl der Abbuchungen soll gezählt werden
  - Attribute, Prädikate und Operationen der Superklasse werden mit denen der Subklasse vereinigt
  - Achtung: nur möglich, wenn keine Typkonflikte
- Umbenennung und Redefinition von Operationen



  - Vereinigung zweier Operationsschemata auch explizit möglich
  - $withdraw$-Operation der Klasse $CreditCard$ wird in $oldWithdraw$ umbenannt
  - $withdraw$ steht für Redefinition in Klasse $CreditCardCount$ zur Verfügung
- Mehrfachvererbung
  - Object-Z unterstützt Mehrfachvererbung
  - Konflikte müssen manuell gelöst werden
  - der Mechanismus der Umbenennung reicht aus, um das umfassend zu tun

Polymorphismus

- Typ eines Objekts ist nicht auf eine Klasse festgelegt, sondern kann jeden Typ, der in der Vererbungshierarchie definiert wurde annehmen: $card? : \downarrow CreditCard$
- Kompatibilitätsbedingungen für Signaturen
  - jedes Attribut in der Visibility List von $CreditCard$ muss ein Attribut der Visibility List jeder Subklasse ($CreditCardCount$, $CreditCardConfirm$) sein
  - ist $INIT$ in der Visibility List von $CreditCard$, muss $INIT$ auch in der Visibility List jeder Subklasse sein
  - jede Operation in der Visibility List von $CreditCard$ muss eine Operation in der Visibility List jeder Subklasse sein und die Input- und Outputvariablen dieser Operationen müssen den gleichen Typ und Namen besitzen
- gelten diese Bedingungen, wird eine Vererbungshierarchie auch Polymorphische Hierachie genannt

- Beispiel

GenCreditCards

↑(INIT,add,delete,withdraw,deposit,transferAvail,withdrawConfirm)

cards: ℙ ↓ CreditCard

INIT
cards = ∅

add
Δ(cards)
card?: ↓ CreditCard
card? ∉ cards
card?.INIT
cards´ = cards ∪ {card?}

delete
Δ(cards)
card?: ↓ CreditCard
card? ∈ cards
cards´ = cards \ {card?}

withdraw ≙ [card?:cards] • card?.withdraw

deposit ≙ [card?:cards] • card?.deposit

withdrawConfirm ≙ [card?:cards|card? ∈ CreditCardConfirm] • card?.withdrawConfirm

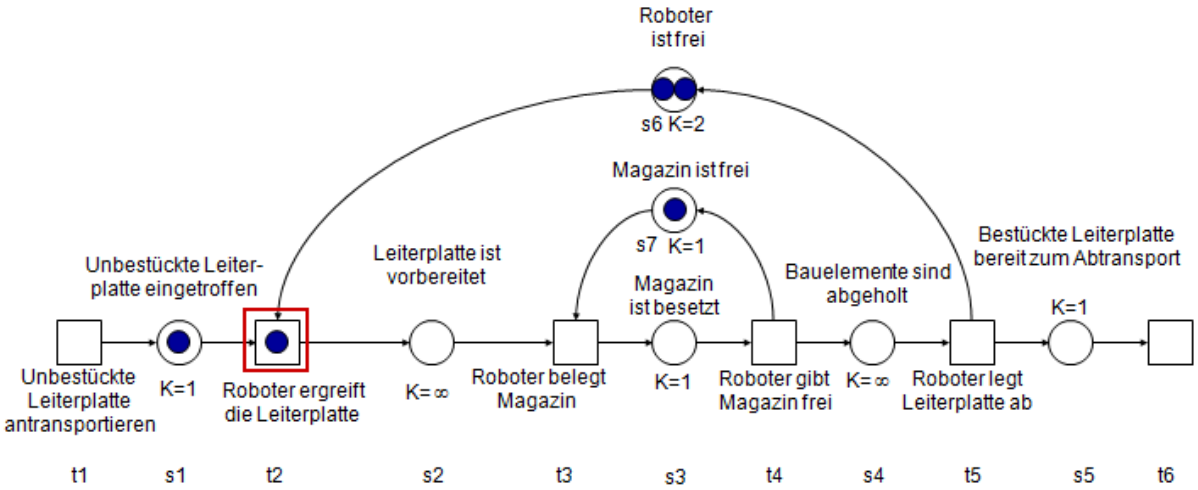
transferAvail ≙ [from?,to?:cards|from?≠to?] • from?.withDrawAvail||to?.deposit

  - $cards$ ist eine Menge von Objekten, die den Typ $CreditCard$, $CreditCardCount$ oder $CreditCardConfirm$ annehmen können
  - Operationen (z.B. $withdraw$) beziehen sich jeweils auf den Typ des Objekts
  - Ausnahme: $withdrawConfirm$ nur anwendbar auf Objekte des Typs $CreditCardConfirm$

Petri-Netze

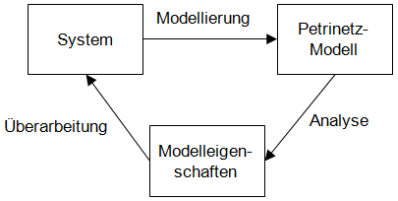
Beispiel

- 2 Roboter bestücken Leiterplatten mit elektronischen Bauelementen
- Bauelemente werden auf Fließband antransportiert
- ist einer der Roboter frei, nimmt er die Leiterplatte vom Fließband
- sind beide Roboter frei, wird nichtdeterministisch entschieden, wer die Leiterplatte nimmt
- nur jeweils ein Roboter darf auf Bauelemente-Magazin zugreifen
- nur jeweils eine Leiterplatte wird zu einem Zeitpunkt abtransportiert

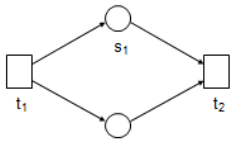


Grundlagen

- Modellierung, Analyse, Simulation von dynamischen Systemen mit nebenläufigen und nichtdeterministischen Merkmalen
- Erlaubten die Beschreibung von Kontroll- und Datenfluss

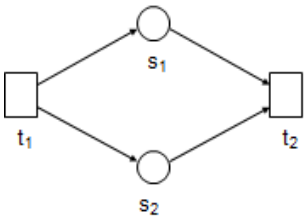


- gerichteter Graph, besteh aus zwei verschiedenen Sorten (Stellen, Transitionen) von Knoten (bipartiter Graph)
  - Stelle: Zwischenablage von Informationen
  - Transitionen: Verarbeitung von Informationen
- Kanten verbinden Stellen mit Transitionen, niemals Stellen mit Stellen oder Transitionen mit Transitionen
- Stellen werden mit Objekten (sog. Token oder Marken) belegt
- Durchlauf der Token durch Petri-Netz beschreibt dynamisches Verhalten des Systems
- Petrinetze erlauben einen natürlichen Begriff von Parallelität
- Lokalitätseigenschaften von Petrinetzen: das Schalten einer Transition wird nur von ihrer direkten Umgebung beeinflusst und beeinflusst auch nur diese
- Jede Stelle repräsentiert eine Aussage. s ist genau dann markiert, wenn die durch s repräsentierte Aussage wahr ist. Das Eintreten eines Ereignisses t macht die Aussagen im Vorbereich von t falsch und im Nachbereich von t wahr.
- Definition
  - ein (Petri-)Netz ist $N = (S, T, F)$ ist folgendermaßen definiert
    (i) $S \neq \emptyset, T \neq \emptyset$
    (ii) $S \cap T \neq \emptyset$
    (iii) $F \subseteq SxT \cup TxS$
  - „Petri“ in Klammern, weil zum Petrinetz auch eine Festlegung des dynamischen Verhaltens gehört
  - $s \in S$ heißen S-Elemente oder Stellen,
    $t \in T$ heißen T-Elemente oder Transitionen,
    $f \in F$ heißen Kanten
  - je nach Definition des dynamischen Verhaltens (in Form einer sog. Schaltregel) entstehen spezielle Netztypen, in denen die S- und T-Elemente besondere Namen haben

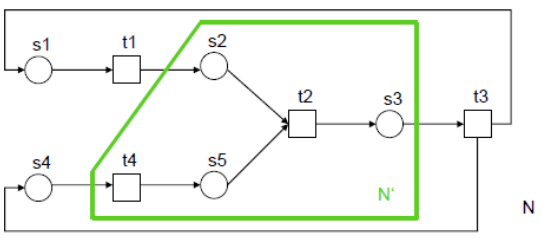


$S=\{s_1, s_2, s_3, s_4\}$
$T=\{t_1, t_2, t_3\}$
$F=\{(t_1, s_1), (t_1, s_2), (s_1, t_2), (s_2, t_2), (s_3, t_3), (t_3, s_4)\}$

- Vorbereich und Nachbereich eines S-Elements
  - $\bullet s = \{t \in T | (t, s) \in F\}$ ist der Vorbereich von s
  - $s \bullet = \{t \in T | (s, t) \in F\}$ ist der Nachbereich von s
- Schlichtheit
  - ein Netz heißt schlicht, wenn keine zwei Stellen denselben Vor- und Nachbereich haben
    $\forall x, y: \bullet x = \bullet y \wedge x \bullet = y \bullet \Longrightarrow x = y$
  - ein nicht schlichtes Netz deutet ggf. daraufhin hin, dass nicht konsequent modelliert wurde



- Teilnetze und Netzteile
  - ein Netz $N' = (S', T', F')$ ist Teilnetz des Netzes $N = (S, T, F)$, wenn $S' \subseteq S, T' \subseteq T$ und $F' = F \cap ((S' \text{ x } T') \cup (T' \text{ x } S'))$
  - N' heißt Netzteil, falls gilt $F' \subseteq F \cap ((S' \text{ x } T') \cup (T' \text{ x } S'))$
  - Beispiel
    - N' ist Teilnetz von N
    - würde in N' bspw. Kante $(s2, t2)$ fehlen, wäre N' ein Netzteil von N



- relativer Rand
  - relative Rand eines Netzteils = diejenigen seiner Knoten, die über Kanten mit dem Restnetz verbunden sind
  - $Rand(N', N) = \{x \in S' \cup T' | \, x \bullet \cup \bullet x \backslash (S' \cup T') \neq \emptyset\}$
    - N' heißt stellenberandet, wenn $Rand(N', N) \subseteq S'$
    - N' heißt transitionsberandet, wenn $Rand(N', N) \subseteq T'$
    - im Beispiel ist N' weder stellen- noch transitionsberandet, da $Rand(N', N) = \{s2, s3, t4\}$
- Schlinge
  - Netzteil der Form $(\{s\}, \{t\}, \{(s, t), (t, s)\})$
  - wird eingesetzt, um Elemente zu modellieren, die benutzt aber nicht verändert werden (z.B. ein Kochbuch beim Kochen)
  - braucht man bei der Beschreibung von Geschäftsprozessen häufig, verhindert oft aussagekräftige Analyseergebnisse

Stellen/Transitions-Netze

- Anzahl der Marken pro Stelle wird betrachtet
  - Zustandsbetrachtung
  - Wann kann im nächsten Zustand passieren?
- verschiedene Arten von Petrinetzen mit anonymen Marken
  - Stellen/Tranistions-Netz (S/T-Netz)
  - Bedingungs/Ereignis-Netz (B/E-Netz)
- Markierung allgemein
  - „Verteilung der Marken auf die Stellen“
  - die initiale Markierung ist Bestandteil des Anfangszustands eines Netzes
    - basierend auf der inititalen Markierung warden aktivierte Transitionen ermittelt, deren Schalten dann zu Folgemarkierungen führen
    - unter den Folgemarkierungen sind dann (möglicherweise) wieder andere Transitionen aktiviert, dies setzt sich fort bis keine Transition mehr aktiviert ist
  - eine Markierung, unter der keine Transitionen aktiviert ist, heißt tote Markierung
- S/T-Netze
  - Stellen können mehr als eine Marke enthalten
  - Kanten haben Gewichte
  - beim Schalten wird den Stellen des Vorbereichs/des Nachbereichs der aktivierten Transitionen so viele Token entnommen/hinzugefügt, wie das Gewicht der Kante anzeigt
  - Stellen mit Kapazitäten (max. Tokenmenge)
  - es darf nur geschaltet werden, wenn Kapazität der jeweiligen Stelle nicht überschritten wird
- ein 6-Tupel (S, T, F, K, W, M_0) heißt Stellen-/Transitions-Netz (S/T-Netz), falls gilt:
  - i) (S, T, F) ist ein Netz aus Stellen S und Transitionen T
  - ii) $K: S \to N \cup \{\infty\}$ erklärt eine (möglicherweise unbeschränkte) Kapazität für jede Stelle
  - iii) $W: F \to N$ bestimmt zu jedem Pfeil des Netzes ein Gewicht
  - iv) $M_0: S \to N_0$ ist eine Anfangsmarkierung, die die Kapazitäten respektiert, d.h. für jede Stelle $s \in S$ gilt $M_0(s) \leq K(s)$
- ein Netz $N = (S, T, F, K, W, M_0)$ wird auch mit (N, M_0) bezeichnet
- Sei N ein S/T-Netz.
  - Eine Abbildung $M: S \to N_0$ heißt Markierung von N, mit $\forall s \in S: M(s) \leq K(s)$
  - $M(N)$ ist die Menge aller Markierungen von N

- eine Transition $t \in T$ heißt M-aktiviert (schreibe $M[t>$), falls gilt
  $\forall s \in \bullet t: M(s) \geq W(s,t)$
  $\forall s \in t \bullet: M(s) \leq K(s) - W(t,s)$
- eine M-aktivierte Transition $t \in T$ bestimmt eine Folgemarkierung M' von M durch

$$M'(s) := \begin{cases} M(s) - W(s,t) & \text{falls } s \in \bullet t \setminus t \bullet \\ M(s) + W(t,s) & \text{falls } s \in t \bullet \setminus \bullet t \\ M(s) - W(s,t), +W(t,s) & \text{falls } s \in \bullet t \cap t \bullet \\ M(s) & \text{sonst} \end{cases}$$

- wir sagen dann: t schaltet von M nach M' und schreiben: $M[t > M'$.

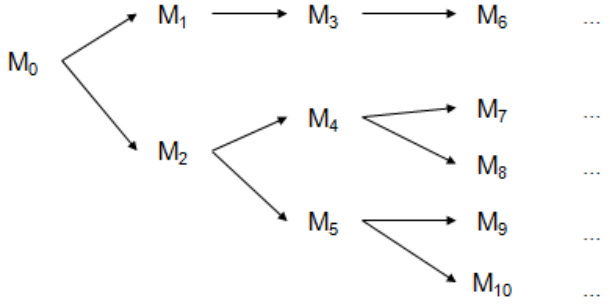
- Es seien (N, M_0) ein S/T-System und $w = t_1 \dots t_n \in T^*$. Die Markierung $M'' \in M(N)$ heißt Folgemarkierung unter M_0 unter w (schreibe $M_0[w > M''$), wenn
  $w =$ leer $\wedge\ M'' = M_0$ oder
  $\exists M' \in M(N): M_0[t_1 \dots . t_{n-1} > M' \wedge M'[t_n > M''$
- w heißt Schaltfolge, w heißt aktiviert unter M_0 (schreibe $M_0[w>$)
- $M_0 M_1 \dots M_n$ heißt die w zugeordnete Markierungsfolge, falls
  $\forall i = 1, \dots, n: M_i := M_0 t_1 \dots t_i$ (M_i erreichte Markierungen)
- $M_0 t_1 M_1 \dots t_n M_n$ heißt die w zugeordnete verallgemeinerte Markierungsfolge
- $[M_0 >_N := \{M | \exists w \in T^*$ mit $M_0[w > M\}$ heißt Erreichbarkeitsmenge des Systems

Erreichbarkeit

- Erreichbarkeitsmenge im Beispiel der Roboter

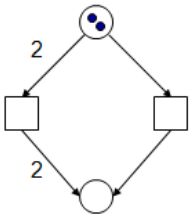
Nr.	s1	s2	s3	s4	s5	s6	s7	Schaltungen
M0	1	0	1	0	0	2	0	t2->M1 t4->M2
M1	0	1	1	0	0	1	0	t4->M3
M2	1	0	0	1	0	2	1	t2->M4 t5->M5

- Erreichbarkeitsgraph

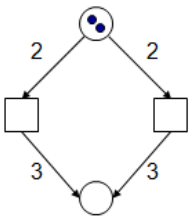


schlichte S/T-Netze

- wenn zwei Transitionen den gleichen Vor- und Nachbereich haben, ist das Netz nicht schlicht



Schlichtes Netz



Nicht schlichtes Netz

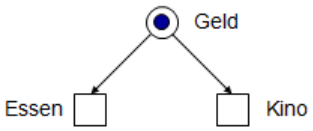
Inzidenzmatrix

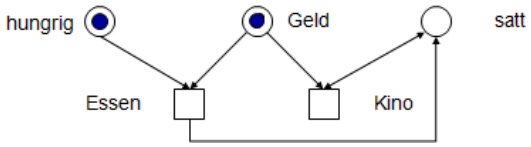
	t1	t2	t3	t4	t5	t6
s1	1	-1				
s2		1	-1			
s3			1	-1		
s4				1	-1	
s5					1	-1
s6		-1			1	
s7			1	-1		

- $s1$ bekommt eine Marke von $t1$ und gibt eine Marke an $t2$ ab
- mit Hilfe der Matrixdarstellung lassen sich S-Invarianten berechnen
- eine S-Invariante ist eine Menge von Stellen, für die die Summer der Marken durch das Schalten von Transitionen unverändert bleibt
- S-Invarianten weisen darauf hin, dass keine bearbeiteten Elemente verloren gehen
- das Fehlen von S-Invarianten deutet manchmal auf Modellierungsfehler hin
- eine variablenfreie Lösung i der Gleichung $N^T * i = 0$ identifiziert eine S-Invariante

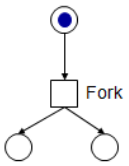
Grundsituationen in S/T-Netzen

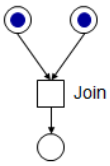
- **Konflikt**
  - nicht-nebenläufige gleichzeitige Aktiviertheit von Transitionen
  - konfliktierende Transitionen nehmen sich gegenseitig Input-Marken weg
  - zwei in Konflikt stehende Transitionen t_1, t_2 haben mindestens eine gemeinsame markentragende Inputstelle $s \in \bullet t1 \cap \bullet t2$



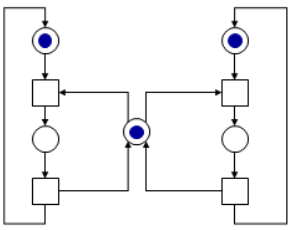


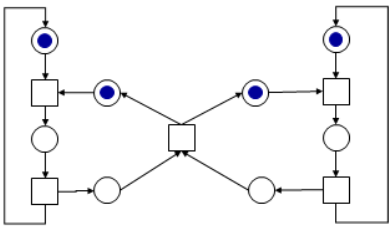
  - konfliktlösende Einbettung
    - ist durch Einführung zusätzlicher Eingabestellen möglich
    - hierdurch wird Information über zu wählende Alternative ergänzt
- **Synchronisation**
  - Einführen von Abhängigkeiten zwischen Transitionsfolgen, d.h. Wegnahme von Nebenläufigkeit
  - Beispiel: Fork-Join-Synchronisation zur Aufspaltung und Zusammenführung eines Ereignisstroms



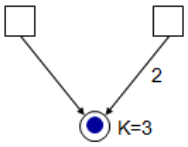


  - Synchronisation von 2 Prozessen

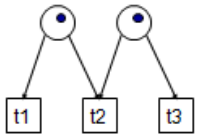




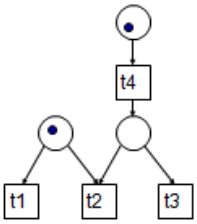
- **Kontakt**
  - eine Transition bringt so viele Marken auf eine Outputstelle, dass eine andere Transition nicht mehr schalten kann (weil Kapazität ausgelastet)



- **Konfusion**
  - ein Konflikt kann von „dritter Seite“ aufgelöst oder herbeigeführt werden



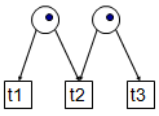
kft(t1,M) = {t2}
kft(t2,M) = {t1,t3}
kft(t3,M) = {t2}



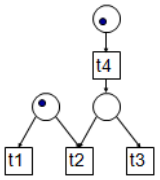
kft(t1,M)= ∅
kft(t2,M)= ∅

kft = Konfliktmenge

  - ein Tripel $(M, t1, t3)$ mit $t1 \neq t3$ und $kft(t1, M) \neq kft(t1, M[t3>)$ nennt man Konfusion
  - Konfliktvergrößerung: $kft(t1, M) \subset kft(t1, M[t3>)$
  - Konfliktverminderung: $kft(t1, M[t3>) \subset kft(t1, M)$
  - symmetrische Konfusion: $kft(t1, M) \backslash kft(t1, M[t3>) \neq \emptyset$
  - asymmetrische Konfusion: $kft(t1, M[t3>) \backslash kft(t1, M) \neq \emptyset$



Konfliktbeseitigung



Konflikteinführung

kft(t1,M) = {t2}
kft(t2,M) = {t1,t3}
kft(t3,M) = {t2}

kft(t1,M)= ∅
kft(t2,M)= ∅

kft(t1,M) = {t2} ≠ kft(t1,M[t3>)=∅
→ Konfusion

kft(t1,M) = ∅ ≠ kft(t1,M[t4>) = {t2}
→ Konfusion

kft(t1,M) ⊃ kft(t1,M[t3>)
→ Konfliktverminderung

kft(t1,M) ⊂ kft(t1,M[t4>)
→ Konfliktvergrößerung

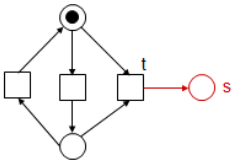
kft(t1,M) = {t2} \ kft(t1,M[t3>) = ∅ ≠ ∅
→ symmetrische Konfusion

kft(t1, M[t4>) = {t2} \ kft(t1,M) = ∅ ≠ ∅
→asymmetrische Konfusion

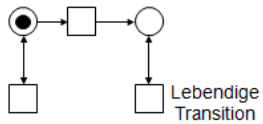
Analyse von Systemen

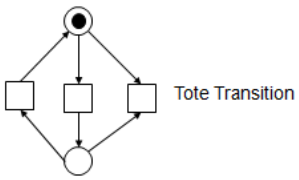
- Simulation kann nicht zeigen, dass bestimmte Situationen nicht auftreten, da Simulationen immer nur einen Ausschnitt aus der Menge aller möglichen Verhalten darstellen
- Simulation kann zeigen, dass bestimmte Situationen auftreten können, sagt aber nichts über deren Eintrittswahrscheinlichkeit
- Beweis von Eigenschaften
  - statische Eigenschaften: solche, die unabhängig von Markierungen sind und nur von der Netztopologie abhängen → Deadlocks, Traps
  - dynamische Eigenschaften: solche, die von der Menge der erreichbaren Markierungen abhängen → Standardhilfsmittel: Erreichbarkeitsgraphen

- Analyse von (S/T-)Systemeigenschaften
  - Annahmen zur Vereinfachung: betrachtete S/T-Systeme sind endlich, schlicht und schwach zusammenhängend
  - untersuchte Eigenschaften: Sicherheit, Lebendigkeit
- **Sicherheit**
  - Seien $Y = (S, T, F, K, W, M_0)$ ein S/T-System und $B: S \to N_0 \cup \{\infty\}$ eine Abbildung, die jeder Stelle eine „kritische Markenzahl“ zuordnet. Y heißt B-sicher bzw. B-beschränkt, wenn $\forall M \in [M_0 >, s \in S: M(s) \leq B(s)$
  - Man spricht von 1-sicher, 2-sicher usw., wenn B=1, 2 usw. Y heißt beschränkt, wenn es eine natürliche Zahl b gibt, für die Y b-sicher ist.
  - eine Stelle s heißt b-sicher, wenn Y mit $B(s) = b$, ansonsten $B(s') = \infty$ B-sicher ist
  - Unterschied zwischen Kapazität und Sicherheit
    - Kapazität begrenzt Stellenmarkierung (a priori-Begrenzung)
    - Sicherheit beobachtet Stellenmarkierung (a posteriori-Begrenzung)
  - Beispiel: Verkehrsplaner modelliert Ampelsystem. An einer bestimmten Stelle s darf sich nur ein Auto aufhalten. Modelliert er $K(s) = 1$, kann die Analyse über die Korrektheit der Modellierung nichts aussagen. Durch Modellierung $K(s) = \infty$ kann in der Analyse geprüft werden, ob $B(s) = 1$.
  - Beispiel: Transition t soll nie schalten dürfen. Durch Hinzufügen einer Beobachtungsstelle s und der Bedingung $B(s) = 0$ kann diese Sicherheitseigenschaft ausgedrückt werden.

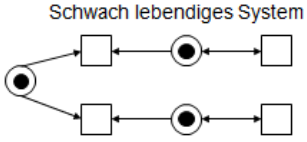


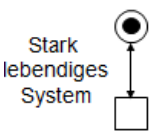
- **Lebendigkeit**
  - eine Transition t eines S/T-Systems $Y = (N, M_0)$ heißt **aktivierbar**, wenn sie mindestens unter einer Folgemarkierung aktiviert ist
  - sie heißt **lebendig**, wenn sie unter allen Folgemarkierungen aktivierbar ist
  - sie heißt **tot**, wenn sie unter keiner erreichbaren Markierung aktiviert ist

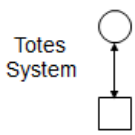




  - ein S/T-System $Y = (S, T, F, K, W, M_0)$ heißt **deadlockfrei** oder **schwach lebendig**, wenn unter jeder erreichbaren Markierung mindestens eine Transition aktivierbar ist





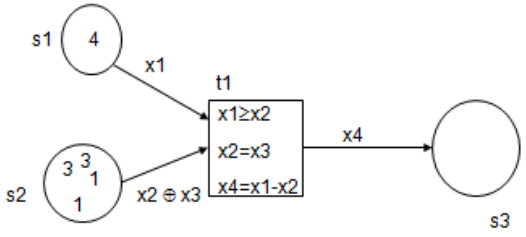


  - ein S/T-System heißt **lebendig** oder **stark lebendig**, wenn aus jeder erreichbaren Markierung jede Transition aktivierbar ist
    - Eigenschaft permanent aktiver Systeme, die nie auch nur teilweise ausfallen
    - Berücksichtigung partieller Ausfülle („graceful degradation“) führt zur schwachen Lebendigkeit, BSP
  - es **heißt** tot, wenn keine Transition aktiviert ist
    - bedeutet häufig einen Deadlock

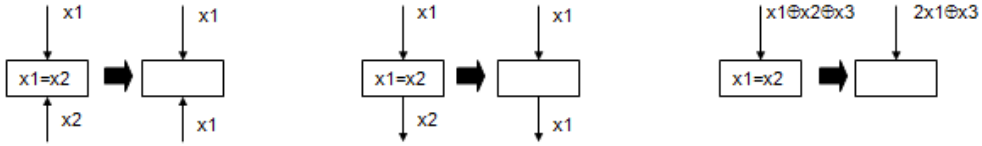
- **Deadlock**
  - ein Deadlock ist eine Menge von S-Elementen, die – wenn einmal unmarkiert – immer unmarkiert bleibt, da der Vorbereich der Menge Teil des Nachbereichs ist
- Aussagen zur Lebendigkeit
  - Ein S/T-System ist genau dann tot, wenn keine Transition aktivierbar ist, d.h. alle Transitionen tot sind.
  - Ein S/T-System ist genau dann schwach bzw. stark lebendig, wenn keine erreichbare Markierung tot ist bzw. alle seine Transitionen lebendig sind
  - Ein S/T-System ist genau dann stark lebendig, wenn sein Erreichbarkeitsgraph von jedem Knoten ausgehend für jedes t aus T einen Pfad besitzt, in dem t als Etikett (Label) vorkommt.
  - Eine erreichbare Markierung ist genau dann tot, wenn von ihr im Erreichbarkeitsgraph keine Kante ausgeht.
- **Synchronie**
  - Synchronieaussagen beantworten Fragen der Art, wie oft Ereignisse einer Art A und Ereignisse einer Art B im Verhältnis zueinander stattfinden können
  - das maximale sofortige Vorauseilen einer Transitionsmenge vor der anderen σ_a nennt man verankerten Synchronieabstand
  - das maximale gelegentliche Vorauseilen σ_f nennt man freien Synchronieabstand
  - beiden können im Extremfall die Werte 0 oder ∞ annehmen
- **Fairnessabstand**
  - Seien $E_1 \subseteq T$ und $E_2 \subseteq T$ zwei Transitionsmengen eines S/T-Systems. Dann bezeichnet man die maximale sofortige Anzahl der möglichen Vorkommen einer Transitionsmenge ohne die andere als **verankerten Fairnessabstand** von E_1 und E_2.
  - Das maximale gelegentliche voneinander nicht unterbrochene Vorkommen nennt man den **freien Fairnessabstand** von E_1 und E_2.

Systeme mit individuellen Marken

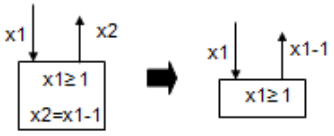
- individuellen Marken werden Werte zugeordnet
  - zusätzliches Ausdrucksmittel, um Marken zu unterscheiden und mit ihnen zu rechnen
- komplexere Kantengewichte
- zusätzliche Schaltbedingungen
- verschieden Arten von Petrinetzen („höhere Netze“)
  - Prädikat/Transition-Netze (P/T-Netz), Coloured Net, Realationennetze etc.
- die Grundform der Netze bezeichnet man als IM-Netze
- Idee: Grundmodell ähnlich S/T-Netzen
- zusätzliche Ausdrucksmittel
  - individuelle Marken
  - Benennung verbrauchter und erzeugter Marken auf Kanten
  - Schaltbedingungen auf Transitionen
  - zur Angabe des Schaltens einer Transition gehört nun nicht nur die Transition, sondern auch die Wahl der Input-/Outputmarken
  - im Beispiel sind zwei Schaltungen möglich (x1=4, x2=x3=1,x4=1 und x1=4, x2=x3=3, x4=1)
  - die erste wird folgendermaßen notiert: $M1[t1(x1 = 4, x2 = x3 = 1, x4 = 3) > M2$, (kurz $M1[t1(4,1,1,3) > M2$)



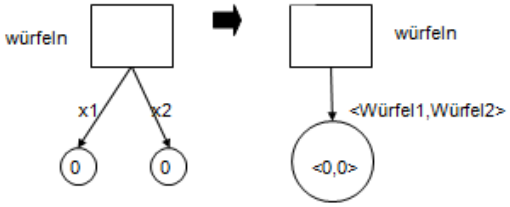
- Achtung: Definition von Datentypen pro Stelle notwendig, um Typkonflikte zu vermeiden
  - z.B. $\mathbb{N}$ für s1 und s2, $\mathbb{Z}$ für s3
- Vereinfachung durch gleiche Variablennamen und Konsistenzregel
  - Identität durch Gleichbenennung von Markenvariablen ersetzen
  - bei Schaltung von Transitionen gilt die gleiche Variablenbelegung



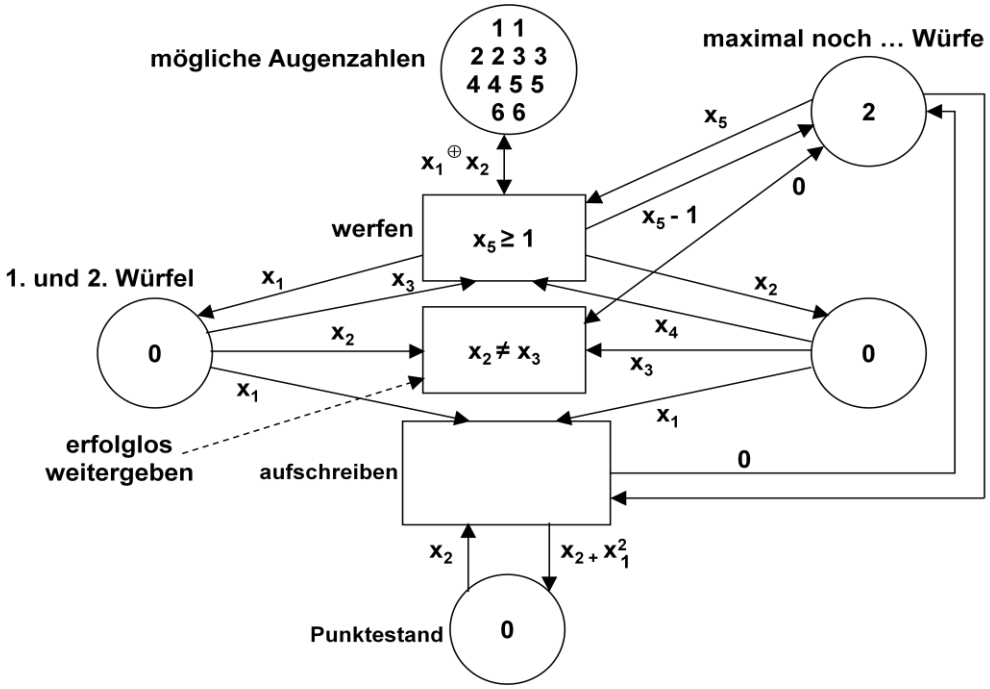
- Vereinfachung durch Verwendung von Ausdrücken an Kanten
  - Ableitung der Ausdrücke von Transitionsprädikaten



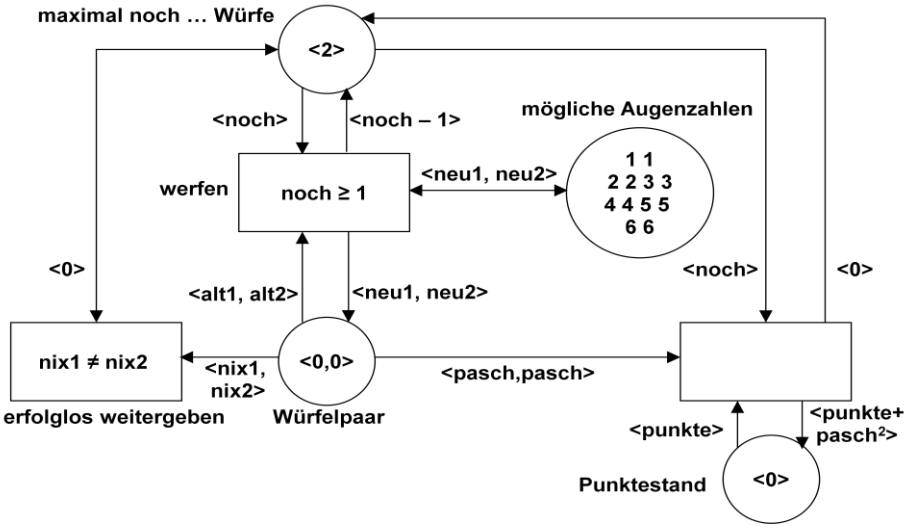
- Vereinfachung durch Tupel
  - Ersetzung der Variablen durch suggestivere Namen
  - Zusammenfassung von Variablen in Tupeln
  - Stellen haben dann als Datentyp kartesisches Produkt



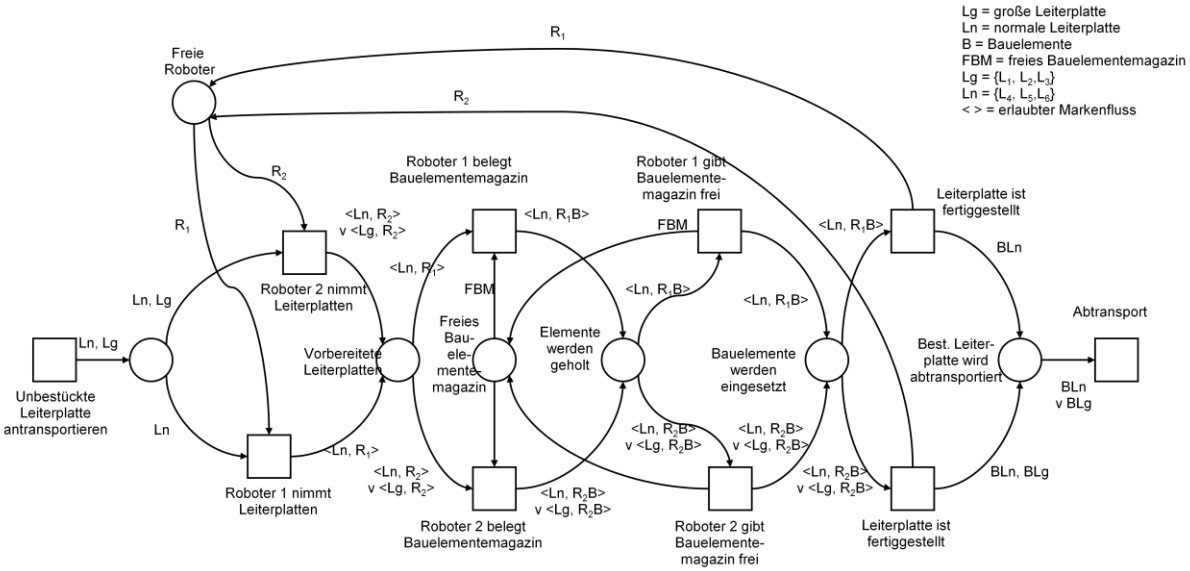
- Beispiel: Würfeln, Regeln
  - jeder Spieler würfelt bis zu zweimal mit 2 Würfeln
  - sobald er einen Pasch wirft, addiert er das Produkt der Augenzahlen zu seinem Punktekonto
  - im folgenden Beispiel: Würfelspiel mit Ausdrücken an Kanten



- nächstes Beispiel: Würfelspiel mit Tupel an Kanten



  - Problem in vorherigen Beispielen: Aufschreiben kann unmittelbar erfolgen. Dies liegt an der ungeeigneten Initialmarkierung. Könnte durch ein Prädikat, das festlegt, dass „erfolglos aufschreiben“ nur mit Werten ungleich Null schalten kann, behoben werden.
- Erweiterung des Bestückungsroboter-Beispiels
  - Roboter 2 kann (anders als Roboter 1) auch übergroße Leiterplatten bestücken
  - die Leiterplatten werden aufgeteilt in 2 Mengen: L_n (normale Platten) und L_g (große Platten)

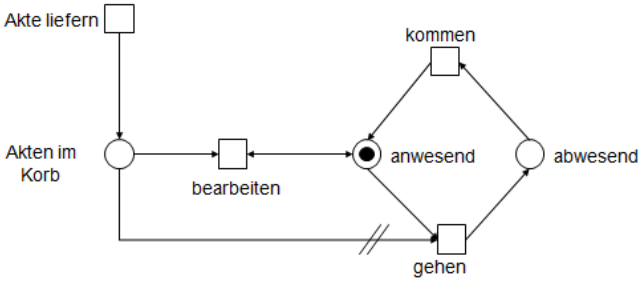


- ein IM-System ist ein 7-Tupel (S, T, F, D, P, W, M_0)
  - S Stellen, T Transitionen, F Flussrelation, D Grundmengen, P Schaltbedingungen, W Kantengewichte, M_0 Anfangsmarkierung
  - (S, T, F) ist ein Netz
  - D Grundmengen
    - $\forall s \in S: D(s)$ ist ein Stellentyp, d.h. Grundmenge, aus der alle Marken auf s stammen
    - $D = (D_s)_{s \in S}$
    - S/T-Netz: $D = \{\bullet\}$
  - M_0 Markierung
    - eine Markierung M_0 des IM-Netzes ist auf jeder Stelle eine Multimenge über deren Typ, $M_0: S \to P_m(\cup D(s))$
  - W Kantengewichte / Kantenanschriften

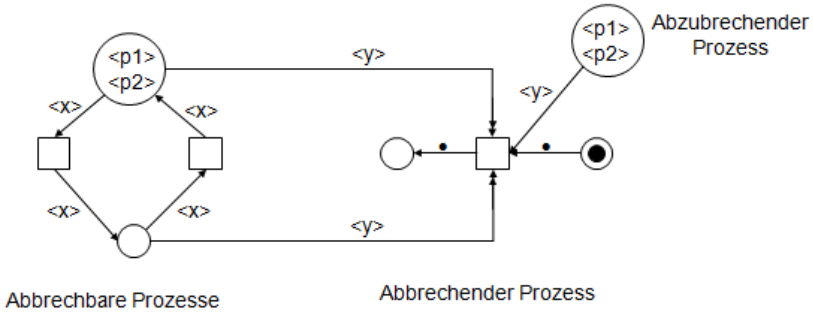
  - ersetze Kantenanschriften durch Zahlenmengen (z.B. x2⊕x3 durch {2,3})
- P Schaltbedingungen
  - Menge $V(T)$ = alle (typgerechten) Variablenbelegungen einer Transition t des IM-Systems
  - Transitionsprädikat P definiert zusätzlich zum Vorliegen hinreichend vieler Inputmarken weitere qualitative Bedingungen, die die Menge der schaltfähigen Belegungen einschränkt
- IM-Systeme können durch verhaltensgleiche S/T-Systeme modelliert werden

Nicht-Standard-Netze

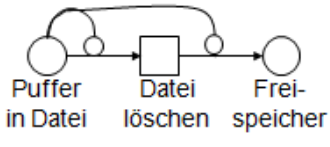
- Netze mit Verbotskanten
  - Aktiviertheit einer Transition hängt u.a. (d.h. den anderen Stellen aus dem Vorbereich der Transition) davon ab, dass die Stelle, die mit Verbotskante verknüpft ist, keine Marke trägt



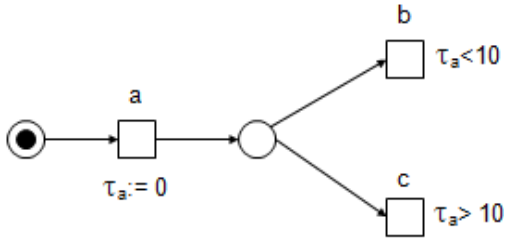
- Netze mit gewichteten Verbotskanten
  - Aktiviertheit einer Transition hängt u.a. (d.h. den anderen Stellen aus dem Vorbereich der Transition) davon ab, dass die Stelle, die mit Verbotskante verknüpft ist, weniger Marken als das Gewicht der Verbotskante trägt
- Verbotskante mit Beschriftung
  - Aktiviertheit einer Transition hängt u.a. davon ab, dass die Stelle, die mit Verbotskante verknüpft ist, keine Marken es Datentyps der Verbotskanteninschrift trägt (für Prioritäten)
- Netze mit Abräumkanten
  - Ziel: Modellierung des Abbruchs von Prozessen
  - Abräumkanten entnehmen ihren Ausgangsstellen im Falle der Schaltung sämtliche zu dem Zeitpunkt dort vorhandenen Marken
- Abräumkanten im IM-Systemen
  - Ziel: Modellierung des Abbruchs eines Prozesses aus einer Menge von Prozessen
  - Abräumkanten entnehmen ihren Ausgangsstellen im Falle der Schaltung die zu dem Zeitpunkt dort vorhandenen Marken des spezifizierten Datentyps



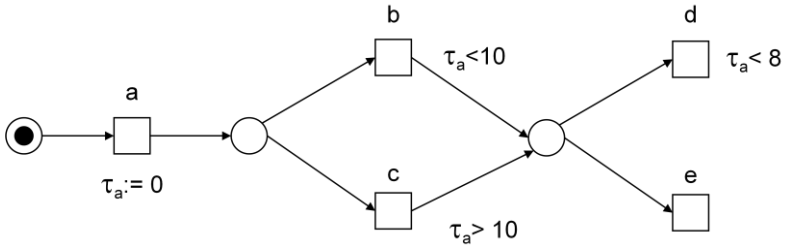
- Netze mit markierungsgesteuerten Kantengewichten
  - Stelle („Gewichtungsstelle“) modelliert das Gewicht einer Kante



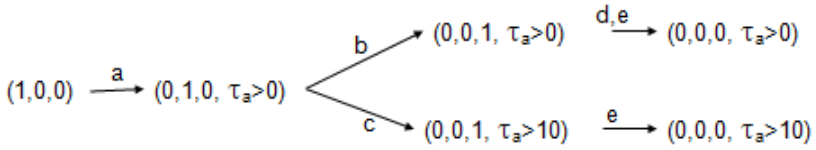
- Netze mit Zeitbegriffen
  - bislang: nur qualitative (vorher/nachher) oder implizite (Kausalität) Zeitbetrachtungen
  - hier: quantitative (wie lange, wann) Zeitbetrachtung
  - Modellierung von Zeit an
    - Stellen
      - Mindestverweildauer von Marken
      - Marken können durch von ihnen aktivierte Transitionen reserviert werden
    - Transitionen
      - Schaltdauer
- Netze mit Zeitbegriffen: Timer-Netze
  - Ziel: Modellierung von Systemen mit zeitabhängigen Entscheidungen (z.B. Kommunikations-protokolle, Durchsatzanalysen, Realzeitanwendungen)
  - zunächst geschieht a, τ_a die seit a verstrichene Zeit
  - b geschieht nach weniger als 10 Sekunden *oder* c geschieht nach mehr als 10 Sekunden



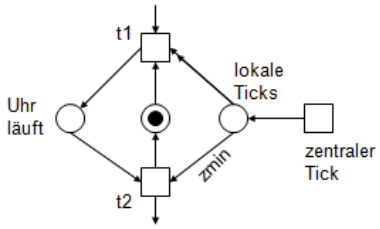
  - ist Transition nicht atomar (d.h. geschieht nicht zu einem Zeitpunkt), so sollte sie verfeinert werden
  - Erreichbarkeitsanalyse muss Zeitaspekt berücksichtigen, sonst möglicherweise inkonsistente Schaltfolgen
  - zunächst geschieht a, τ_a die seit a verstrichene Zeit
  - b geschieht nach weniger als 10 Sekunden *oder* c geschieht nach mehr als 10 Sekunden
  - anschließend d *oder* e, aber d nur, wenn nach a weniger als 8 Sekunden vergangen sind
  - acd ist ungültige Schaltfolge



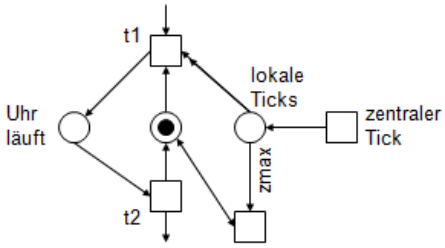
  - Erreichbarkeitsanalyse muss aus diesen Gründen zur Zeitanalyse erweitert werden
    - neben der Markierung und möglicher Schaltungen wird der *Zeitenraum* angegeben, d.h. alle nach der letzten Schaltung möglichen Stände aller gestarteten Timer τ
    - Transition ist *zeitkonsistent* aktiviert, wenn Schaltbedingung von mindestens einem Punkt im Zeitenraum der gegenwärtigen Markierung erfüllt wird
    - bei Schaltung wird der neuen Markierung ein neuer Zeitenraum zugeordnet



- Netze mit Zeitbegriffen: Modellierung von Uhren
  - Idee: Modellierung von Uhren als Teilnetze
  - Einhaltung einer Mindestzeitbedingung
    - zu Beginn schaltet $t1$, dadurch Abräumen der lokalen Ticks
    - Uhr läuft, Marken sammeln sich in „lokale“ Ticks
    - $t2$ kann frühestens schalten (d.h. Uhr stoppt), wenn $zmin$ Marken erreicht



  - Einhaltung einer Höchstzeitbedingung
    - zu Beginn schaltet $t1$, dadurch Abräumen der lokalen Ticks
    - Uhr läuft, Marken sammeln sich in „lokale“ Ticks
    - $t2$ kann beliebig schalten, spätestens dann, $zmax$ Marken erreich



Methodik

- kein Standard zur Erstellung von Petri-Netzen vorhanden
- vorgeschlagenes Vorgehen
  1) Stellen und Transitionen auf hohem Abstraktionsniveau identifizieren
  2) Beziehungen ermitteln
  3) Verfeinerung und Ergänzung
  4) Festlegung der Objekte
  5) Schaltregeln identifizieren
  6) Netztyp festlegen
  7) Anfangsmarkierung festlegen
  8) Analyse, Simulation

Fazit

positiv	negativ
▪ einfache und wenige Elemente ▪ graphisch gut darstellbar ▪ durch Marken übersichtliche Visualisierung des Systemzustands ▪ Syntax und Semantik formal definiert ▪ Werkzeuge zur Erstellung, Analyse, Simulation, Code-Generierung vorhanden ▪ Petri-Netze gut geeignet zur Modellierung von Systemen mit kooperierenden Prozessen ▪ besitzen größere Mächtigkeit als Zustandsautomaten, da zu einem Zeitpunkt mehrere Zustände darstellbar	▪ höhere Petri-Netze schwer zu erstellen und zu analysieren ▪ von anderen Modellierungs-konzepten isoliert ▪ statische Struktur ▪ keine Methode zur Erstellung von Petri-Netzen vorhanden

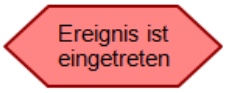
Ereignisgesteuerte Prozesskosten (EPK)

Ereignisgesteuerte Prozessketten (EPK)

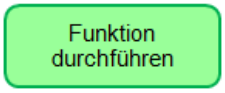
- Bestandteile
  - Reihenfolge der Tätigkeiten (Kontrollfluss), Erzeugung und Austausch von Daten (Datenfluss), betriebliche Organisation, verwendete Betriebsmittel
- Grundlagen
  - dargestellt werden primär Kontrollflüsse
  - drei Basissymbole: Ereignisse, Funktionen, Konnektoren
  - Basissymbole werden durch Kanten verbunden: Kontrollfluss-Kanten, Daten-/Materialfluss-Kanten, Zuordnungs-Kanten
  - zusätzliche Informationen: organisatorische Einheiten, Informations-/Materialobjekte, Anwendungssysteme, Prozesswegweiser

Basissymbole der EPK

- Ereignisse



  - ein Ereignis stellt das Eintreten eines Zustandes dar
  - es steuert oder beeinflusst den weiteren Ablauf
  - es ist jedoch selber passiv und trifft keine Entscheidungen
  - Ereignisse verbrauchen weder Ressourcen noch Zeit
  - jeder GP hat mindestens ein Start- und ein Endereignis
  - syntaktische Empfehlung: Substantiv gefolgt vom Partizip eines Verb
- Funktion



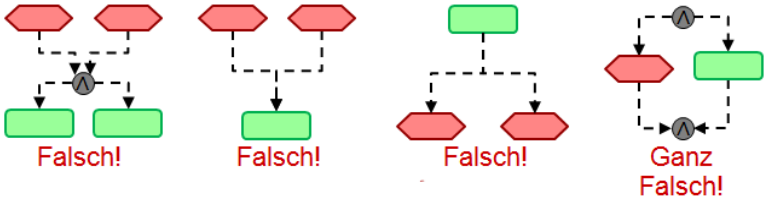
  - eine Funktion ist eine fachliche Tätigkeit oder Aufgabe
  - sie nimmt meist Zeit in Anspruch und kann Ressourcen verbrauchen
  - sie kann über den weiteren Verlauf des Prozesses entscheiden, ist also aktiv
  - Funktionen in GPs sollen einem oder mehreren Unternehmenszielen dienen
  - syntaktische Empfehlung: Substantiv und Infinitiv
- Kontrollflusskante
  - Kontrollflusskanten bringen Funktionen und Ereignisse in zeitliche und logische Abfolgen
  - Ereignisse und Funktionen folgen einander streng abwechselnd!
  - jedes Objekt ist durch eine Kante verbunden, es gibt keine losgelösten Objekte
  - eine Kante verbindet immer genau zwei Objekte miteinander
- Konnektoren
  - Konnektoren leiten Verzweigungen ein (split) oder beenden sie (join)
    - "Und"-Konnektoren umschließen Teilprozesse, die gleichzeitig bzw. unabhängig voneinander ausgeführt werden

    - „Oder"-Konnektoren umschließen alternative Pfade, wobei mehrere Alternativen gleichzeitig möglich sind

    - „Exklusiv-Oder"-Konnektoren umschließen Alternativen, von denen nur jeweils eine ausgeführt werden darf

  - „Oder"- und „X-Oder"-Verzweigungen (splits) können nur auf Funktionen folgen (da sie einer Entscheidung bedürfen und Ereignisse keine Entscheidungskompetenz haben)
  - „Und"-Verzweigungen sowie alle Zusammenführungen (joins) können sowohl Funktionen als auch Ereignissen folgen
  - Konnektoren dürfen auf Konnektoren folgen

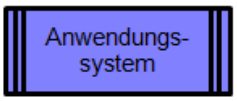
- verbreitete Modellierungsfehler
  - ein Konnektor ist entweder ein Split oder ein Join, nie beides gleichzeitig
  - Konnektoren sind die einzigen Symbole, deren Kontrollflüsse verzweigen oder zusammenführen; Ereignisse und Funktionen haben nie mehr als einen Ein- und einen Ausgang
  - alle Eingänge und alle Ausgänge eines Konnektors müssen selben Typ sein

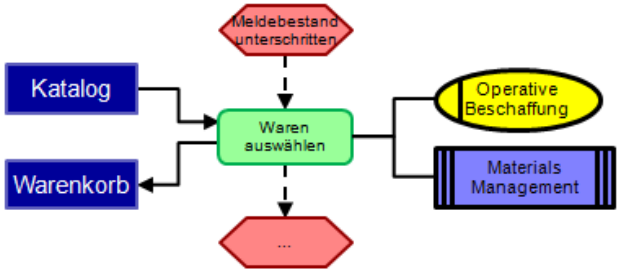


  - dasselbe Symbol, das eine Verknüpfung beginnt, muss sie auch beenden („Wohlgeformtheit“)
  - Quersprünge sind zu vermeiden. Sie mögen theoretisch möglich sein, sind aber in der Praxis fast immer falsch, irreführend oder zumindest überflüssig
- zusätzliche Informationen
  - Daten- und Materialflusskanten sowie Informationsobjekt (bzw. Daten- oder Materialobjekt)

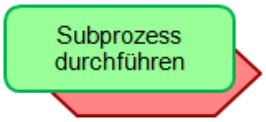
    - das Informationsobjekt repräsentiert Dinge (auch Personen, die nicht dem Unternehmen angehören, z.B. Kunden) der realen Welt
    - Informationsobjekte dienen als Input (Nutzung, Verbrauch, lesender Zugriff) oder Output (Erstellung, schreibender Zugriff) für Funktionen
    - Datenflusskanten verbinden die Informationsobjekt mit den Funktionen
  - organisatorische Einheiten sowie Zuordnungs-Kanten
    - organisatorische Einheiten repräsentieren Rollen oder Personen, die verantwortlich für die (mit Zuordnungskanten) verknüpften Funktionen sind

  - Anwendungssystem
    - Anwendungssysteme repräsentieren EDV-Unterstützung für einen Prozessschritt und werden über Zuordnungskanten mit Funktionen verbunden





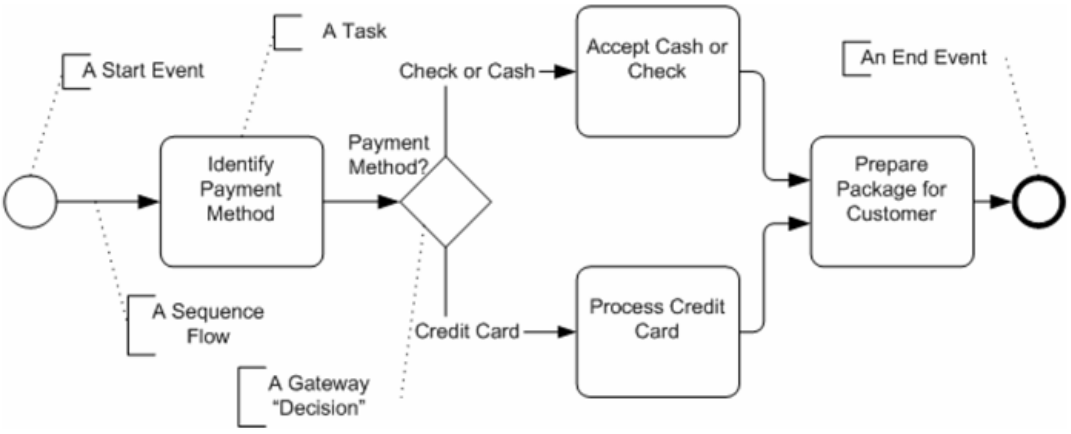
  - Prozesswegweiser
    - erlauben komplette Prozesse in andere Prozesse zu integrieren
    - innerhalb des übergeordneten Prozesses werden die Prozesswegweiser weitestgehend wie Funktionen behandelt



    - anders als normale, atomare Funktionen dürfen Wegweiser auch am Start oder Ende des Prozesses stehen
    - Prozesswegweiser dienen der Erstellung von Hierarchien in Prozessen
    - sie können auch zur Platzersparnis eingesetzt werden

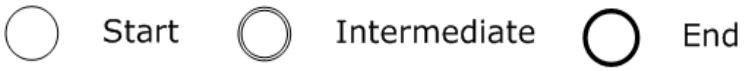
Business Process Modeling Notation (BPMN)

Beispiel: einfacher GP

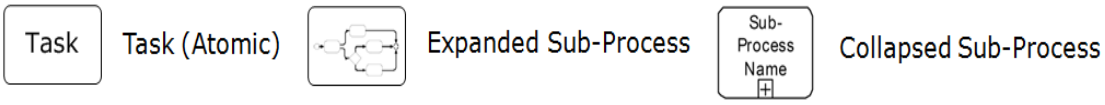


Basissymbole

- fünf grundlegende Symbolkategorien
  - Flussobjekte: Ereignisse, Aktivitäten, Gateways
  - verbindende Objekte: Sequenzfluss, Nachrichtenfluss, Assoziationen
  - Schwimmbahnen: Pools, Bahnen
  - Daten: Datenobjekte, Nachrichten
  - Artefakte: Gruppen, Annotationen
- Ereignisse
  - ein Ereignis ist etwas, das im Verlauf eines Prozesses oder einer Choreographie passiert
  - Ereignisse beeinflussen den Ablauf
  - Ereignisse haben i.d.R. eine Ursache (trigger) und Auswirkungen (results)
  - es gibt drei Ereignistypen, die durch interne Marker weiter unterschieden werden können

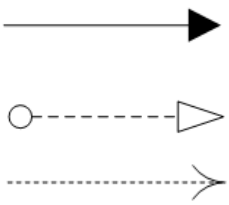


- Aktivitäten
  - Aktivitäten repräsentieren Arbeiten, die im Prozess ausgeführt werden
  - Aktivitäten können atomar (Task) oder zusammengesetzt (Sub-Prozess) sein



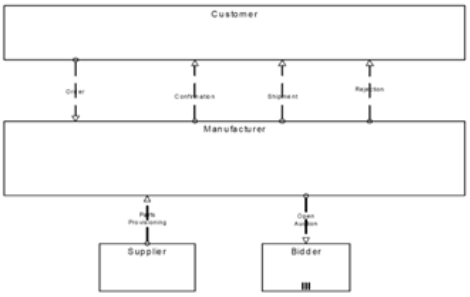
- Gateways
  - Gateways leiten Verzweigungen im Sequenzfluss ein und beenden sie
  - interne Markierungen legen die Art der Verzweigung fest

- verbindende Objekte
  - der Sequenzfluss gibt die Reihenfolge der Aktivitäten vor (entspricht in etwa dem „Kontrollfluss“ in anderen Notationen
  - mit dem Nachrichtenfluss wird der Nachrichtenaustausch zwischen den Prozessteilnehmern modelliert
  - Assoziationen verknüpfen Daten, Text und andere Artefakte
  - Assoziationen können auch Ein- und Ausgaben für Aktivitäten repräsentieren
  - Assoziationen können gerichtet oder ungerichtet sein

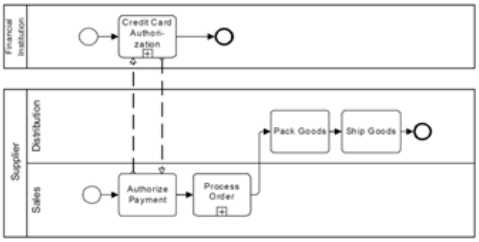


- Pools und Lanes
  - Gemeinsamkeiten
    - Pools und Lanes repräsentieren die Teilnehmer/Verantwortlichen eines Prozess
    - sie können für Organisationen,Personen/Rollen oder Systeme stehen
    - mit Pools werden Prozesse partitioniert, mit Lanes unter-partitioniert
    - Pools und Lanes können auch als Black Boxes (unter Ausblendung innerer Abläufe) dargestellt werden
  - Unterschiede
    - Pools stellen physische Trennungen da, Prozesse innerhalb eines Pools sind in sich abgeschlossen
    - Sequenzflüsse können die Poolgrenzen nicht überschreiten
    - Pools werden vielmehr durch Nachrichtenflüsse miteinander verknüpft
    - Nachrichten können an die Poolgrenze gerichtet werden, oder in den Pool hinein an ein spezifisches Objekt
    - Nachrichtenflüsse innerhalb eines Pools (auch zwischen mehreren Bahnen desselben Pools) sind hingegen unzulässig
    - stattdessen werden Bahnen (innerhalb desselben Pools) durch Sequenzflüsse verknüpft

Blackbox-Darstellung



Sequenz und Nachrichtenfluss

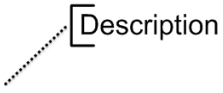


- Datenobjekte
  - Datenobjekte werden von den Aktivitäten gebraucht oder erzeugt
  - sie können individuelle Objekte oder ganze Sammlungen davon repräsentieren
  - Datenobjekte werden mit dem sie umschließenden Prozess instanziiert und zerstört – sie haben keine Persistenz über den Prozess hinaus

- Nachrichten
  - Nachrichten symbolisieren den Inhalt einer Kommunikation
  - Nachrichten werden an Nachrichtenflüsse assoziiert
  - schwarz ausgefüllt = sendend, weiß = empfangend

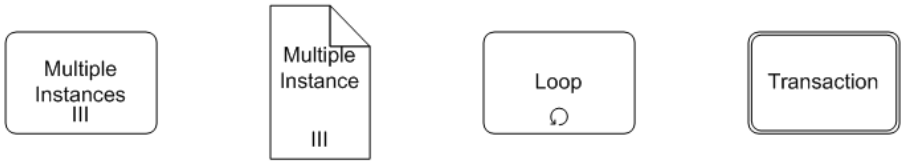
- Gruppen
  - Gruppen fassen Aktivitäten „der selben Kategorie“ zusammen
  - sie können Pools und Lanes überspannen und haben keinen Einfluss auf den Ablauf
  - Kategorien dienen als logisches oder fachliches Ordnungsmerkmal der Analyse, Dokumentation oder Hervorhebung

- Annotationen
  - Text-Annotationen ermöglichen weiterführende Angaben, Kommentare, Notizen etc.



erweiterte Konstruktionen

- Gateway Steuerung
  - Exklusive = XOR (genau einer)
  - Inklusive = OR (einer oder mehrere)
  - Parallel = Nebenläufigkeit
  - zudem ereignisbasierte und komplexe Gateways (freie Definition der Regeln)
- Sub-Prozesssteuerung
  - Transaktionen kennzeichnen zusammengehörige Prozessbereiche, sie werden entweder vollständig ausgeführt oder vollständig abgebrochen
  - Schleifen kennzeichnen Prozessbereiche, die mehrfach in Folge ausgeführt werden
  - das „Multiple Instances“-Symbol kennzeichnet Prozessschritte, die parallel mehrfach ausgeführt werden
  - es kann auch bei Objekten verwendet werden, die mehrfach instanziiert werden



Orchestrierung vs. Choreographie

- Orchestrierung und Choreographie sind Begriffe aus der Dienstkomposition
- Orchestrierung beschreibt dabei das Innenleben der Dienste: Wie läuft die Ausführung des Dienstes ab?
- Choreographie abstrahiert vom Innenleben und beschreibt das Zusammenspiel zwischen den Diensten – Welche Nachrichten werden wann ausgetauscht?
- EPK, WS-BPEL und auch die bisher vorgestellten Konzepte der BPMN sind eher im Bereich der Orchestrierung einzuordnen.
- BPMN 2.0 unterstützt auch Choreographie.
- Eine Alternative dazu ist z.B. die Web Services Choreography Description Language (WS-CDL)

Modellbasierte SW-Entwicklung

Semantik von Modellen

- Vorteile formaler Semantik
  - Struktur bzw. Verhalten des modellierten Systems ist eindeutig beschrieben
  - Struktur und Verhalten können automatisch analysiert werden
  - Fehler und Inkonsistenzen innerhalb eines Modells können automatisch gefunden werden
- Nachteile formaler Semantik
  - formale Modelle eines „echten“ Systems sind sehr komplex und unübersichtlich
  - Erstellung und Wartung formaler Modelle ist sehr aufwändig
  - Modellierer muss den Formalismus gut kennen