Modellierung verteilter Systeme

Grundlagen der Programm und Systementwicklung

Sommersemester 2012

Prof. Dr. Dr. h.c. Manfred Broy

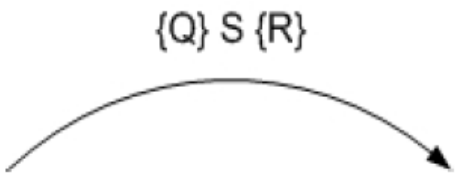
Unter Mitarbeit von Dr. M. Spichkova, J. Mund, P. Neubeck

Lehrstuhl Software & Systems Engineering

Modellierung der Kontrollzustände als Prädikate

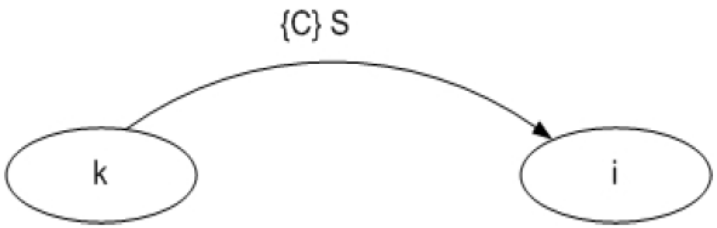
- Kontrollzustände beschreiben Mengen von Zuständen
- Zustandsmengen durch Zusicherungen p: $\Sigma \rightarrow$ B darstellbar
- Um Kontrollzustände $k_1, k_2, \ldots, k_n$ eindeutig zu unterscheiden können wir
  - eine Variable pc („program counter“) mit Sorte $\{k_1, k_2, \ldots, k_n\}$ oder
  - boolesche Variablen $k_1, k_2, \ldots, k_n$ einführen, wobei immer genau ein k_i wahr gesetzt ist

- Beschreibung der markierten Übergänge durch Vorbedingung Q, Aktion S und Nachbedingung R



Zustandsübergangssysteme als Programme

- Programm besteht aus
  - Initialisierung für den Anfangszustand
  - und einer großen Endlosschleife
- Nehme Variable pc zur Identifikation der Kontrollzustände an
- Jeder Übergang wird übersetzt in eine bedingte Anweisung innerhalb der Schleife



do

. . .

[] pc = k ∧ C **then** S

. . .

od

Programme als Zustandsübergangssysteme

- Transformation des Programms in Form mit Marken (engl. Labels), sodass vor und nach jeder Anweisung (zusammengesetzt und elementar) eine eindeutige Marke steht
- neue Variable pc über der Menge der Marken einführen
- Zu Beginn: Ersetze das durch Marken angereicherte Programm

  m_0: S; m_1

  durch

  pc := m_0

  do

  [] true $\wedge$ pc = m_0 **then** S; pc := m_1

  od
- Ersetze iteriert bewachte Anweisungen der Form

  [] G $\wedge$ pc = m **then** S; pc := m‘

  wie folgt

Programme als Zustandübergangssysteme ff.

- *Sequentielle Komposition*

$$[]\ G \wedge pc = m_0 \textbf{ then } m_0 : S_0; m_1 : S_1; pc := m_2$$

wird zu

$$[]\ G \wedge pc = m_0 \textbf{ then } m_0 : S_0; pc := m_1$$

$$[]\ pc = m_1 \textbf{ then } m_1 : S_1; pc := m_2$$

- ***if*-Anweisung**

$$[]\ G \wedge pc = m \textbf{ then } m : \textbf{if } \ldots\ []\ G_i \textbf{ then } m_i : S_i\ []\ \ldots\ \textbf{fi};\ pc := m'$$

wird zu

$$[]\ G_1 \wedge G \wedge pc = m \textbf{ then } m_1 : S_1; pc := m'$$

$$\ldots$$

$$[]\ G_n \wedge G \wedge pc = m \textbf{ then } m_n : S_n; pc := m'$$

$$[]\ \neg(G_1 \wedge \cdots \wedge G_n) \wedge G \wedge pc = m \textbf{ then abort}; pc := error$$

Programme als Zustandübergangssysteme ff.

- *Schleife*

$$[]\ G \wedge pc = m\ \textbf{then}\ m'' : \textbf{do}\ \ldots []\ G_i\ \textbf{then}\ m_i :\ S_i\ []\ \ldots \textbf{do}; pc := m'$$

wird zu

$$[]\ G \wedge pc = m \qquad \textbf{then}\ m : nop; pc := m''$$

$$[]\ G_1 \wedge pc = m'' \qquad \textbf{then}\ m_1 : S_1; pc := m''$$

$$\ldots$$

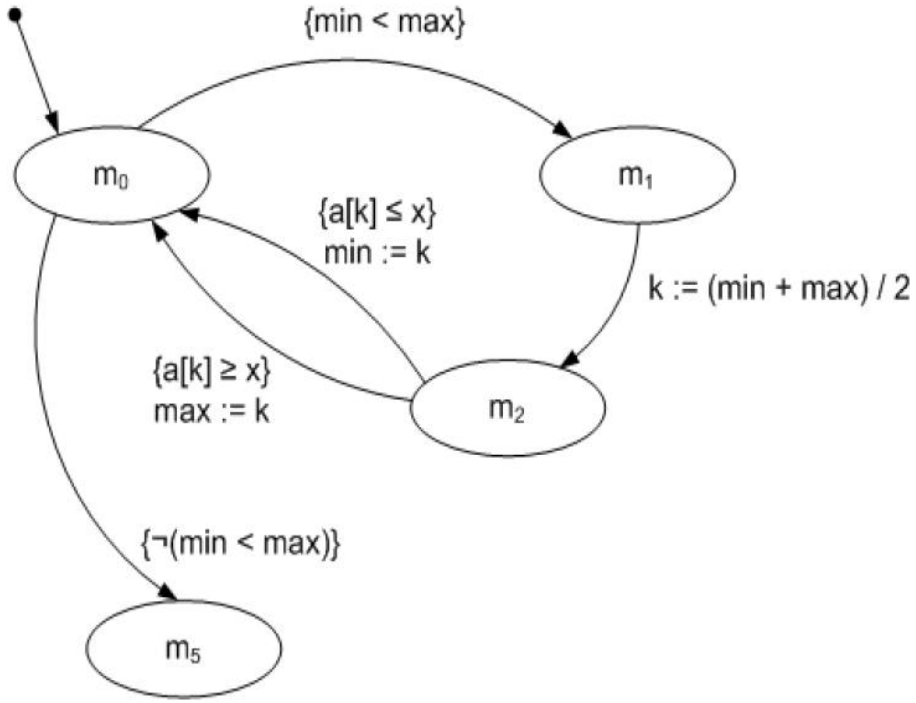
$$[]\ G_n \wedge pc = m'' \qquad \textbf{then}\ m_n : S_n; pc := m''$$

$$[]\ \neg G_1 \wedge \ldots \wedge \neg G_n \wedge pc = m'' \qquad \textbf{then nop}; pc := m'$$

- Programm in einfacher Form kann dann in ZÜS übersetzt werden
  - Kontrollzustände sind durch die Markierungen gegeben
  - Bewachte Anweisungen ergeben Übergänge

Beispiel: Übersetzung Programm nach ZÜS

do $min < max$ **then** $k := (min + max)/2;$
if $a[k] \leq x$ **then** $min := k$
[] $a[k] \geq x$ **then** $max := k$ **fi**
od



- Die Granularität des Programms beeinflusst die der erzeugten Zustandsmaschine
- Vergleiche
  - k := (min + max)/2
  - k := min + max; k := k / 2
- Nur im ersten Fall gilt die Invariante $min \leq k \leq max$

Klassen und Objekte als Zustandsmaschinen

- Objektattribute bestimmen den Zustandsraum

  sort ObjectState = state(att1 : AttSort1, ...)

- Methodenaufrufe als Eingabe

  sort Call = methcall(name : MethodId, par : Parameters)

- Rückgabewerte als Ausgabe

  sort Return = callreturn(name : MethodId, par : Parameters, ret : ReturnValues)

- Betrachten Methodenaufruf als atomar, d.h. als einen Übergang

$$\Delta : \text{ObjectState} \times \text{Call} \rightarrow \text{ObjectState} \times \text{Return}$$

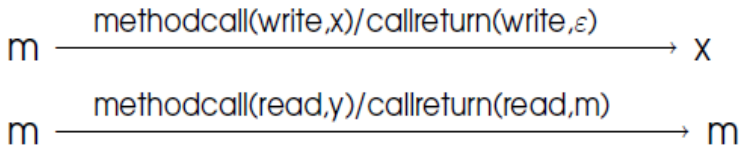
- Unteraufrufe können so nicht modelliert werden!

Beispiel: Lesen- und schreiben eines Attributes

- Für die Klassendefinition

```
class Container =
    { attribute
        a: var M;
      methods
        write = (x: M): a := x;
        read = (y: var M): y := a
    }
```

- erhalten wir folgende Übergänge



Modellieren von Unteraufrufen

- Erlauben statt eines Rückgabewertes einen weiterleitenden Unteraufruf
- Außerdem ergänzen wir Objektidentifikatoren

sort Call = methcall (oid: ObjectId,
name: MethodId,
par: Parameters)

sort Return = callreturn (oid: ObjectId,
name: MethodId,
par: Parameters
ret : ReturnValues)

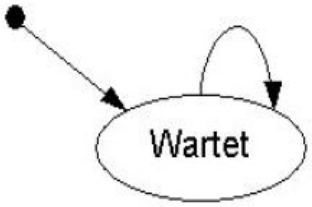
sort Message = Call | Return

Δ : ObjectState $\times$ Message $\rightarrow$ ObjectState $\times$ Message

Beispiel: Bankkonto objektorientiert

```
class Konto
{ attribute stand: var Betrag
// Der Betrag b wird vom Konto abgenommen
// wenn dieser die Höhe des Kontostands nicht überschreitet
// wird done auf true gesetzt, sonst auf false
  method change = (b:Betrag, done: var Bool)
      Pre: true
      Post: (stand´< b ∧ stand`= stand´ ∧ done`= false)
            ∨(stand´≥ b ∧ stand`= stand´- b ∧ done`= true)
}
```

methcall(change,k, b,d) / methreturn(change,k,b,d)
{(stand´< b ∧ stand` = stand´ ∧ done` = false)
∨ (stand´ ≥ b ∧ stand` = stand´-b ∧ done` = true) }



Beispiel: Bankkonto objektorientiert ff.

```
class Konto_Manager
{

  // Überweisung von Konto a nach Konto c, den Betrag b = 0,
  // wenn die Transation erfolgreich war ist, wird done true, sonst false

method trans = (b: Betrag, a,c: Konto, done: var Bool)
...
}
```