Verified Fiasco

Ein Projekt zur formalen Analyse und zum Beweisen der totalen Korrektheit des Mikrokern-Betriebssystems Fiasco

Christoph Haase (Vortragender)
`www.inf.tu-dresden.de/~s0158714`

Hendrik Tews
`wwwtcs.inf.tu-dresden.de/~tews`

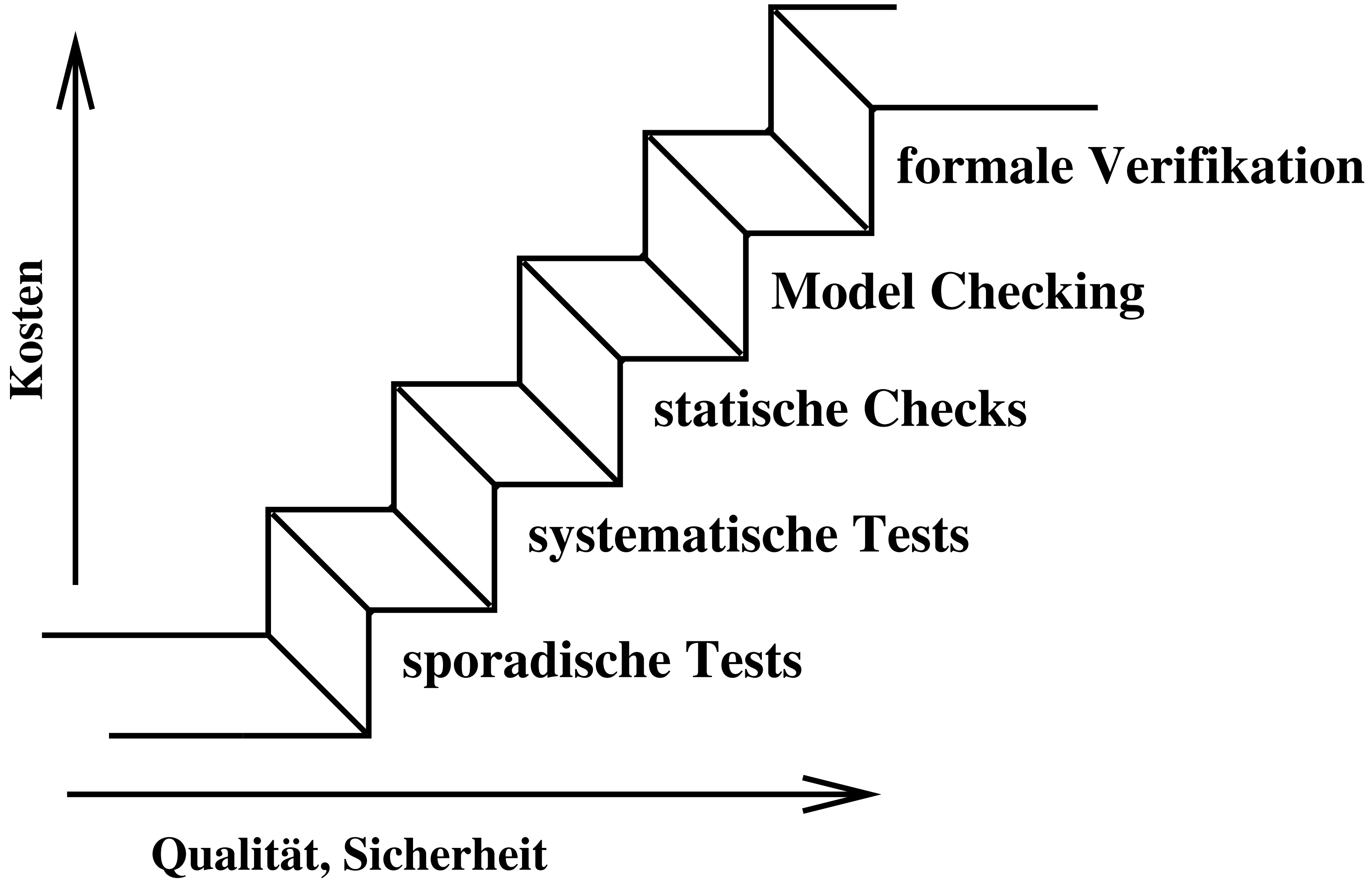
Institut für Theoretische Informatik, TU Dresden

Softwarefehler größeren Ausmaßes

- 1978: F16 Jagdflugzeug stellt sich beim Überqueren des Äquators auf den Kopf
- 1992: Einzug der Grünen in Schleswig-Holsteins Landtag
- 1994: Pentium-Bug
- 1996: Absturz der Ariane 5 Rakete

Möglichkeiten der Qualitätsverbesserung

- Software ausliefern und auf Feedback der Kunden warten
- Sporadische Tests
- Einfache, systematische Checks
- Statische Checks
- Model Checking
- Verifikation


formale Verifikation
Model Checking
statische Checks
systematische Tests
sporadische Tests
Kosten
Qualität, Sicherheit

Softwareverifikation

- Sicherstellen der korrekten Funktionsweise eines Programms
- Überprüfen von (Teilen von) Software hinsichtlich einer Spezifikation
- Resultiert in einem allgemein gültigen mathematischen Beweis

dazu notwendig

- geeignetes mathematisches Modell des Programms (*Semantik*) und der Spezifikation (z. B. *HOL*)
- Werkzeug zum interaktiven, automatisierten Führen von Beweisen

Grenzen von Softwareverifikation

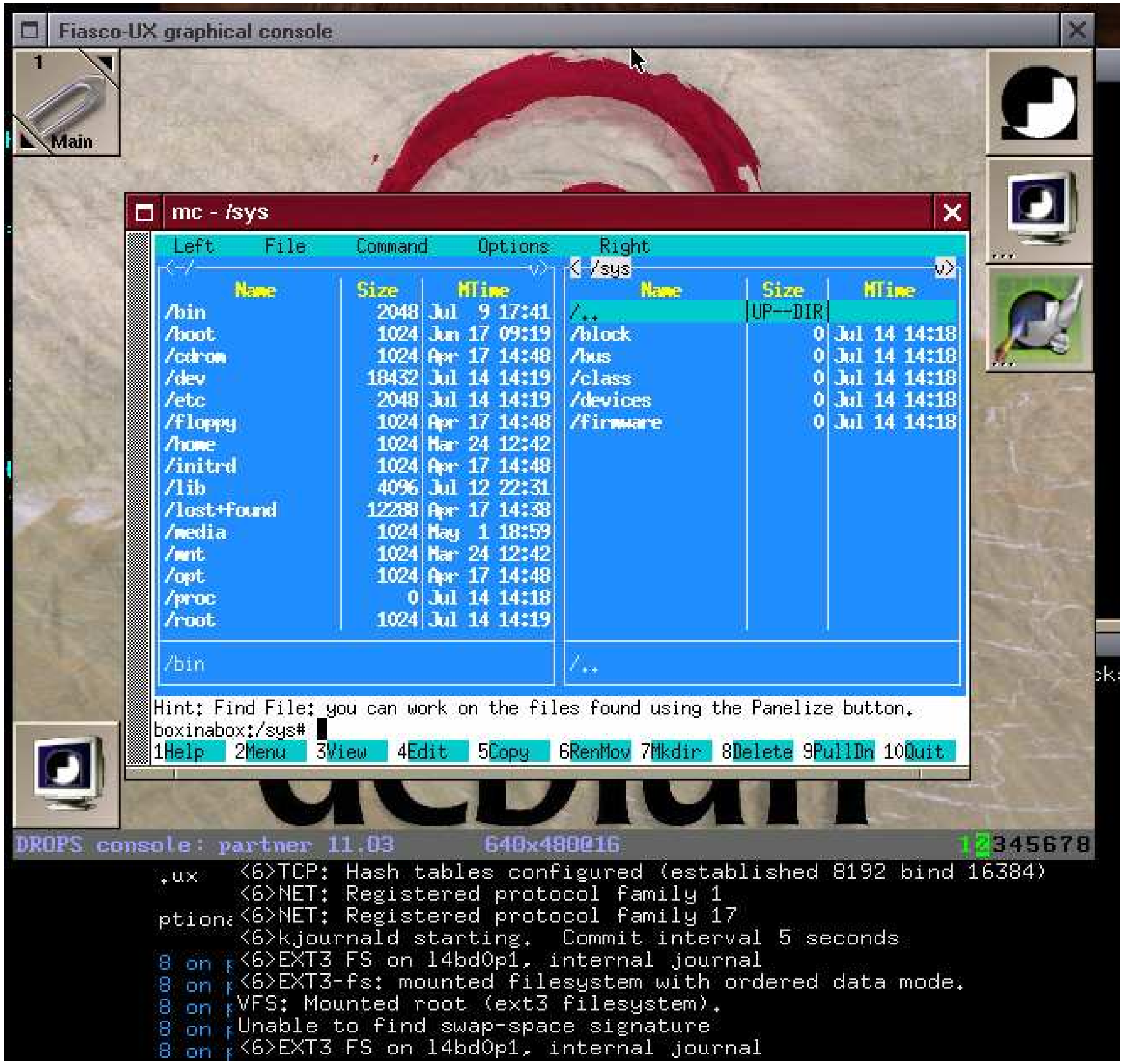
- Fehlerhafte Spezifikation
- Unvollständige Spezifikation
- Beliebig lange Verifikationskette: Compiler → Theorembeweiser → OS → CPU → ...
- Inkonsistente Semantik

Fiasco

- Echtzeit-Mikrokern, entwickelt im DROPS-Projekt an der TU-Dresden
- L4 kompatibel
- Erlaubt das parallele Ausführen mehrerer Linux-Instanzen ohne Verlust der Echtzeitfähigkeit
- Fast vollständig in C/C++ geschrieben
- Momentan ca. 20.000 Zeilen Code

VFiasco

- Verifikation bestimmter sicherheitsrelevanter Teile von Fiasco
- Entwicklung von Methoden zur formalen Verifikation von C/C++
- Besondere Herausforderungen:
  - Typumwandlungen
  - Sprungbefehle
  - Abrupte Termination
  - Zeigerarithmetik



Fiasco in Aktion

Fahrplan

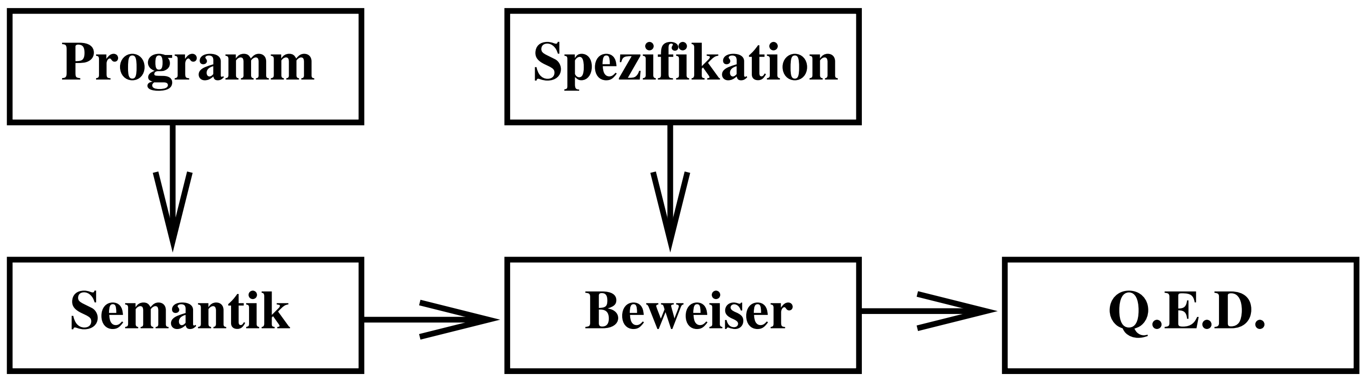
I. Semantiken

II. Im VFiasco-Projekt entwickelte C/C++ Semantik

III. Verifikation mit PVS am Beispiel *Duff's Device*

Semantik

- *Bedeutungslehre*
- Dient zur Formalisierung von Programmen
- Dazu ist ein mathematisches Modell notwendig
- Programmkonstrukte werden in dieses mathematische Modell „übersetzt“
- Ermöglicht das Führen mathematischer Beweise



Axiomatische Semantik

- beschreibt den Zustand eines Programms vor und nach der Ausführung mit Hilfe logischer Formeln (*Assertions*)
- Programmkonstrukte verändern diese Assertions
- Bestimmte Assertions gelten vor und nach Ausführung des Programms:

$$\{Assertions\ vor\ Ausf.\}\ \texttt{Programm}\ \{Assertions\ nach\ Ausf.\}$$

Operationale Semantik

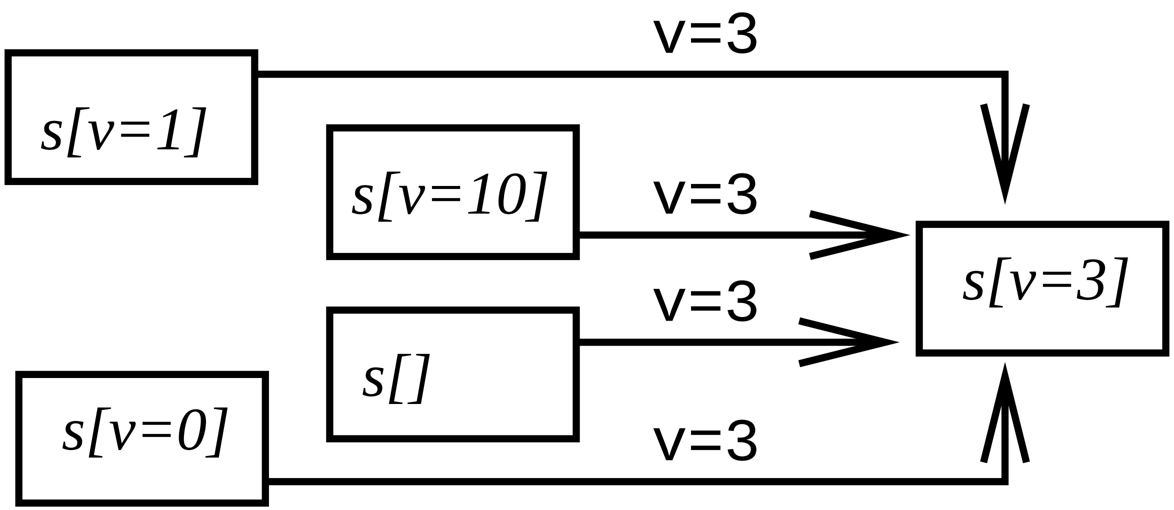
- Semantische Domäne ist ein Transitionssystem
- Transitionen haben die Form:

$$\langle Zustand,\ (Teil-)Programm\rangle \longrightarrow \langle NFZustand,\ (Teil-)Programm\rangle$$

Denotationelle Semantik für C/C++ (1)

- Vorgestellte Semantik ist eine *denotationelle Semantik*
- Programm befindet sich vor und nach Ausführung in einem Zustand
- Zustand beinhaltet alle Variablenbelegungen, Typen etc.
- Programmkonstrukte manipulieren diesen Zustand
- In denotationeller Semantik: jedes Programmkonstrukt ist eine Funktion *State* $\longrightarrow$ *State*

Beispiel:



Denotationelle Semantik für C/C++ (2)

- Problematisch in C/C++:
  - `goto`-Sprünge
  - `setjmp`/longjmp
  - Abruptes Verlassen von Schleifenkörpern mit `break` oder `return`
  - Durchfallendes `switch`
- Diese *Abnormalitäten* müssen durch die Semantik repräsentiert werden

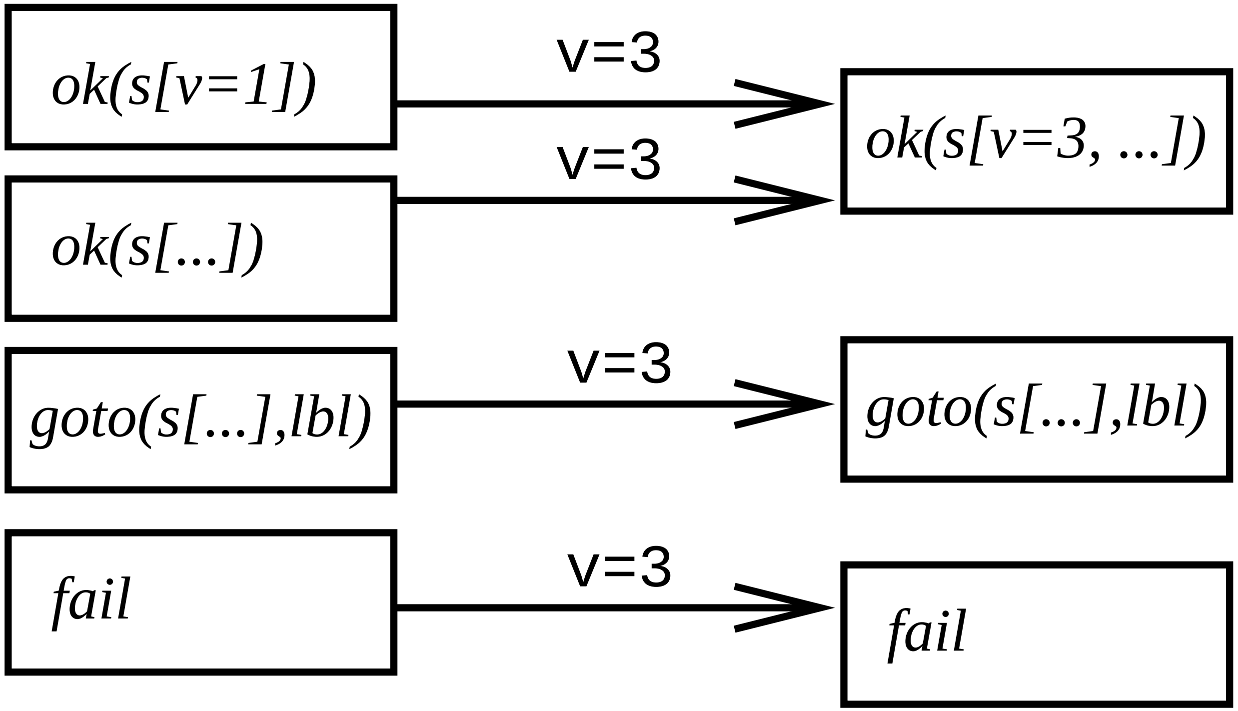
Denotationelle Semantik für C/C++ (3)

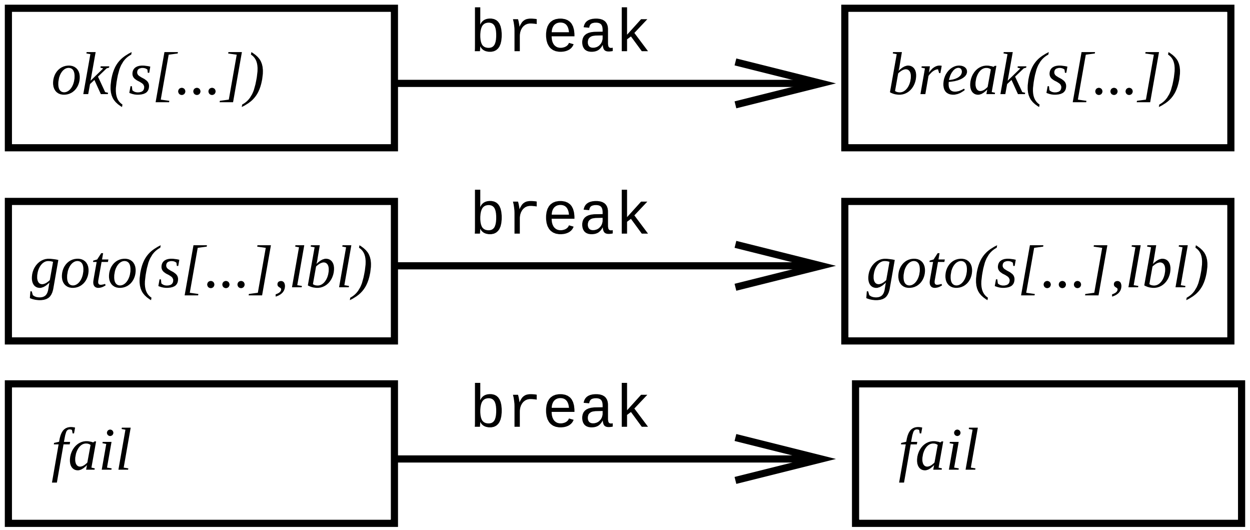
- Im Nachfolgenden bezeichnet
  - S die Menge der Zustände
  - $\mathbb{Z}$ die Menge der ganzen Zahlen
  - $\mathbb{L}$ die Menge der Sprungmarken (Labels)
  - $\mathbf{1}$ die einelementige Menge
- Abnormalitäten werden durch komplexe Zustände repräsentiert:

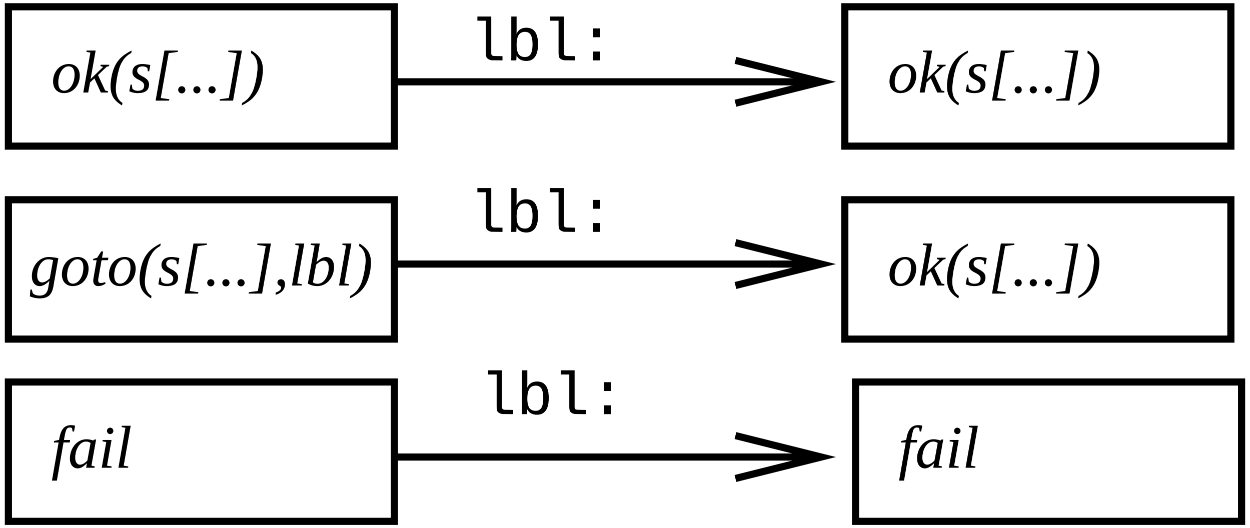
$$Result \quad = \quad \overset{ok:}{S} \uplus \overset{break:}{S} \uplus \overset{case:}{(S \times \mathbb{Z})} \uplus \overset{default:}{S} \uplus \overset{goto:}{(S \times \mathbb{L})} \uplus \overset{fail:}{\mathbf{1}} \uplus \overset{hang:}{\mathbf{1}}$$

ok	Resultat einer normalen Termination einer Anweisung mit Nachfolgezustand
break	Resultat einer `break`-Anweisung, kapselt Zustand vor Ausführung des Breaks
case	Resultat einer `switch`-Anweisung, kapselt Zustand und evaluierte Switch-Expression
goto	Resultat einer `goto`-Anweisung, kapselt Zustand und Sprungmarke
fail, hang	Programmabsturz

- Jedes Programmkonstrukt ist eine Funktion *Result* $\longrightarrow$ *Result* (*Semantische Domäne*)


ok(s[v=1])
ok(s[...])
goto(s[...],lbl)
fail
v=3
v=3
v=3
v=3
ok(s[v=3, ...])
goto(s[...],lbl)
fail


ok(s[...])
break
break(s[...])
goto(s[...],lbl)
break
goto(s[...],lbl)
fail
break
fail


ok(s[...])
lbl:
ok(s[...])
goto(s[...],lbl)
lbl:
ok(s[...])
fail
lbl:
fail

Formale Interpretation von Programmkonstrukten (1)

- $[\![-]\!]$ ist eine Funktion, die eine Anweisung in die Sematische Domäne übersetzt

$$[\![v = expr]\!] = \left| \begin{array}{lll} ok(s) & \mapsto & \begin{cases} ok(s[v=i]) & \text{falls } [\![expr]\!](s) = ok(i) \\ fail & \text{sonst} \end{cases} \\ x & \mapsto & x \end{array} \right.$$

$$[\![st_1; st_2]\!] = \left| \; x \mapsto [\![st_2]\!]\big([\![st_1]\!](x)\big) \right.$$

$$[\![\texttt{if } b \texttt{ then } st;]\!] = \left| \begin{array}{lll} ok(s) & \mapsto & \begin{cases} [\![st]\!](ok(s)) & \text{falls } [\![b]\!](s) = true \\ ok(s) & \text{sonst} \end{cases} \\ x & \mapsto & [\![st]\!](x) \qquad \text{/* falls } st \text{ ein Label enthält */} \end{array} \right.$$

- *expressions* haben hier mal keine Seiteneffekte (der Einfachheit halber)

Formale Interpretation von Programmkonstrukten (2)

- `switch` ist etwas komplizierter ...
- ... aber kein Problem solange man nicht auf Papier rechnen muss

$$\mathcal{L} \quad : \quad \textit{statement} \longrightarrow \mathcal{P}(\mathbb{Z})$$

$$\mathcal{L}(st) \quad = \quad \text{Menge aller case-Labels in } st$$

$$sw(bo, x) \quad : \quad (\mathit{Result} \to \mathit{Result}) \times \mathit{Result} \longrightarrow \mathit{Result}$$

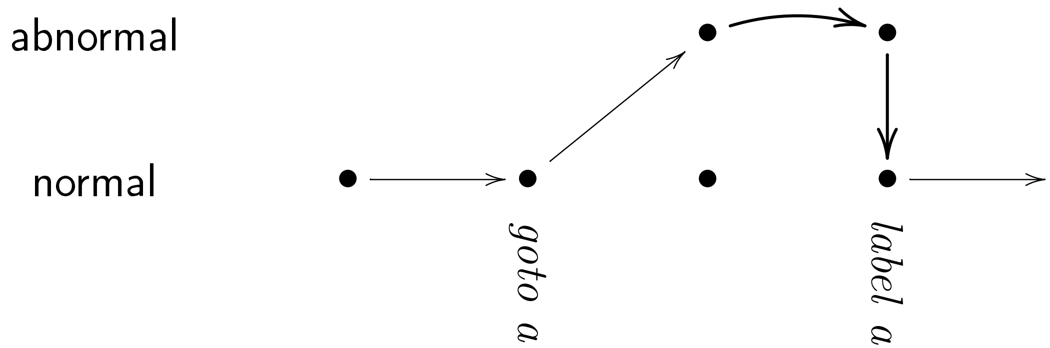
$$sw(bo, x) \quad = \quad \begin{cases} ok(s) & \text{falls } bo(x) = \mathit{break}(s) \\ ok(s) & \text{falls } bo(x) = \mathit{default}(s) \\ bo(x) & \text{sonst} \end{cases}$$

$$[\![\texttt{switch}(ix)\ st]\!] \quad = \quad \left| \begin{array}{lcl} \mathit{break}(s) & \mapsto & \mathit{break}(s) \\ \mathit{default}(s) & \mapsto & \mathit{default}(s) \\ \mathit{case}(s,i) & \mapsto & \mathit{case}(s,i) \\ ok(s) & \mapsto & \begin{cases} sw([\![st]\!], \mathit{case}(s,i)) & \text{falls } [\![ix]\!] = ok(i) \ \wedge \ i \in \mathcal{L}(st) \\ sw([\![st]\!], \mathit{default}(s)) & \text{falls } [\![ix]\!] = ok(i) \ \wedge \ i \notin \mathcal{L}(st) \\ \mathit{fail} & \text{sonst} \end{cases} \\ x & \mapsto & \begin{cases} ok(s) & \text{falls } [\![st]\!](x) = \mathit{break}(s) \\ [\![st]\!](x) & \text{sonst} \end{cases} \end{array} \right.$$

Formale Interpretation von Programmkonstrukten (3)

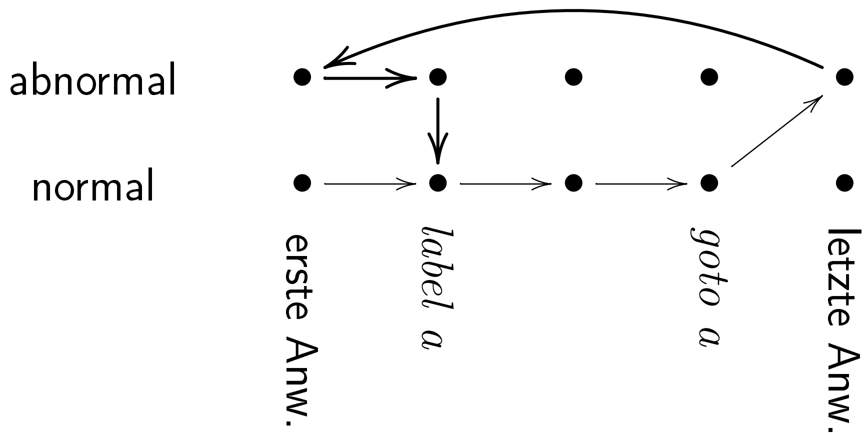
$$[\![\texttt{goto(}\ lexpr\ \texttt{)}]\!] = \left|\begin{array}{lcl} ok(s) & \mapsto & goto(s,l) \\ x & \mapsto & x \end{array}\right.$$

$$[\![l\ :]\!] = \left|\begin{array}{lcl} goto(s,l) & \mapsto & ok(s) \\ x & \mapsto & x \end{array}\right.$$



Rückwärts springen

- Sprungmarken in C/C++ haben eingeschränkten Sichtbarkeitsbereich
- `goto`-Anweisungen die rückwärts springen können zur Nichttermination führen
- Rückwärtssprünge können durch eine Schleife, welche um den gesamten Funktionskörper liegt, behandelt werden:



Ein Korrektheitsbeweis für Duff's Device

```
void copy(char * to, char * from, int count)
{
  int rounds= count / 8;
  switch(count % 8)
  {
    case 0: while(rounds-- > 0){ *to++ = *from++;
    case 7:                       *to++ = *from++;
    case 6:                       *to++ = *from++;
    case 5:                       *to++ = *from++;
    case 4:                       *to++ = *from++;
    case 3:                       *to++ = *from++;
    case 2:                       *to++ = *from++;
    case 1:                       *to++ = *from++;
    }
  }
}
```

„Disgusting, no? But it compiles and runs just fine. I feel a combination of pride and revulsion at this discovery. If no one's thought of it before, I think I'll name it after myself.“ (Tom Duff)

```
duff(source, dest : posnat, count : nat) :
                    [Result[State, Unit] -> Result[State,Unit]] =
  write_int(rounds, div(const_int(count), const_int(8))) ##
  write_int(i, const_int(0)) ##
  int_switch_stm(
    rem( const_int(count), const_int(8)),
      int_case(0) ##
              gwhile_stm(
                 const_int(0) < post_decr_const(rounds),
                    skip_res ##
                    write_int_array(dest, read_int(i),
                          read_int_array(source, read_int(i))) ##
                      write_int(i, read_int(i) ++ const_int(1)) ##
      int_case(7) ##  write_int_array(dest, read_int(i),
                          read_int_array(source, read_int(i))) ##
                      write_int(i, read_int(i) ++ const_int(1)) ##
              .............................................
      int_case(1) ##  write_int_array(dest, read_int(i),
                          read_int_array(source, read_int(i))) ##
                      write_int(i, read_int(i) ++ const_int(1))

              )  % end gwhile_stm
  ) % end int_switch_stm
```

Spezifikation von Duff's Device

- Nach Durchlauf der Funktion `copy` sollen folgende Eigenschaften ausgehend von einem „vernünftigen“ Anfangszustand universell gelten:
  - Die Funktion terminiert ohne Laufzeitfehler
  - Vom Quellarray wurden `count` Zellen korrekt in den Zielarray kopiert
  - Die Zellen des Quellarrays bleiben unverändert
  - Es werden nur die benötigten Zellen des Zielarrays verändert
  - `rounds` hat den Integer-Wert -1

Spezifikation von Duff's Device in PVS

```
duff_total : Theorem Forall(source, dest : posnat, count : nat) :
    duff_var_ok(source, dest, count, s) Implies
      duff(source, dest, count)(ok(s,unit)) =
      ok(s WITH [
          `vars := Lambda(j : posnat) :
              IF j = rounds Then int(-1)
              Elsif j = i Then int(count)
              Elsif cell_in_array(s, dest)(j) And
                      index_from_cell(s, dest, j) < count
                  Then
                      s`vars(get_array(s`vars(source))
                            `fields(index_from_cell(s, dest, j)))
              Else s`vars(j)
              Endif
         ], unit)
```

Zusammenfassung

- Formale Verifikation von Software, auch wenn sie in C/C++ geschrieben ist, ist möglich
- Die gezeigte Semantik erlaubt es nahezu alle Sprachmittel von C/C++ zu formalisieren
- Ein erfolgreich geführter Beweis ist unumstösslich, d. h. dass die verifizierte Software *garantiert* in jedem Falle die spezifizierte Eigenschaft erfüllt

Vielen Dank.