KV Software Engineering
Prof. Dr. Martin Glinz

Kapitel 4

Formale Verifikation

Universität Zürich
Institut für Informatik

4.1 Grundlagen

4.2 Verifikation auf axiomatischer Basis

4.3 Verifikation von Eigenschaften, Model Checking

4.4 Stärken und Schwächen formaler Verifikation

Begriffe

Verifikation (verification) – (1) Der Prozess der Beurteilung eines Systems oder einer Komponente mit dem Ziel, festzustellen, ob die Resultate einer gegebenen Entwicklungsphase den Vorgaben für diese Phase entsprechen, (2) Der formale Beweis der Korrektheit eines Programms. (IEEE 610.12)

- Mit formaler Verifikation ist die Bedeutung (2) gemeint
- Formale Verifikation ist der Beweis, dass ein Programm P alle in einem Vorgabedokument S geforderten Eigenschaften aufweist
  - S ist die Spezifikation
  - die Verifikation erfolgt mit den Mitteln der mathematischen Logik
  - formale Verifikation erfordert eine formale Definition von Syntax und Semantik der für P und S verwendeten Sprachen.

Syntax und Semantik von Sprachen

- Syntax – Regelwerk für die Konstruktion orthographisch und grammatisch korrekter Sätze einer Sprache

- Syntaxdefinition formaler Sprachen
  - konstruktiv durch Übersetzer und gleichzeitig informal mit Text für Benutzer
  - durch formale Definition einer Grammatik (zum Beispiel Backus-Naur-Form (BNF)) für Übersetzer und Benutzer

- Semantik – Festlegung der Bedeutungen der syntaktisch korrekten Sätze einer Sprache

Semantikdefinition formaler Sprachen

- Übersetzersemantik: Bedeutungen durch Übersetzer (Compiler) bestimmt
- Operationale Semantik: Bedeutungen durch Abläufe in einer abstrakten Maschine (einem Automaten) bestimmt
- Denotationelle Semantik: Bedeutungen durch Funktion f: E → A, definiert, welche den Eingangszustand E auf den Ausgangszustand A abbildet.
- Axiomatische Semantik: Bedeutungen durch Axiome in Form von Voraussetzungen und Ergebniszusicherungen bestimmt

4.1 Grundlagen

4.2 Verifikation auf axiomatischer Basis

4.3 Verifikation von Eigenschaften, Model Checking

4.4 Stärken und Schwächen formaler Verifikation

Ansatz

Zu beweisen:

(i) Wenn P terminiert, so transformiert P jeden Zustand, in dem V gilt, in einen Zustand, in dem N gilt. Schreibweise: {V} P {N}

(ii) P terminiert (Hoare, 1969)

- P zu verifizierendes Programm
- {V} Menge der Voraussetzungen (Vorbedingungen, Preconditions), deren Erfüllung vor Ausführung von P garantiert, dass die Ergebniszusicherungen gelten
- {N} Menge der Ergebniszusicherungen (Nachbedingungen, Postconditions), die nach Ausführung von P gelten
- N und V sind Ausdrücke der Prädikatenlogik
- {V} und {N} bilden zusammen die Spezifikation von P

Beispiel 4.1

Gegeben sei die Spezifikation

$V \equiv x, y \in \text{INTEGER} \wedge x = X, y = Y$
$N \equiv x = \max(0, Y)$

Beweise: das Programm

```
P ≡  x := y;
     if x < 0 then x := 0
```

erfüllt die Spezifikation, d.h. {V} P {N}

Hinweis: x = X bedeutet, dass die Variable x den Wert X hat

Beweisidee

- Das zu verifizierende Programm wird in Schritten verifiziert:
- Zu beweisen sei $\{V\}\ P\ \{N\}$ mit $P \equiv s_1; s_2; ...; s_n$ (s_i: Anweisungen)

  Dann werden Bedingungen Q_i bestimmt mit

  $V = Q_0$, $N = Q_n$ und

  $\{Q_0\}\ s_1\ \{Q_1\}\ s_2\ \{Q_2\}\ s_3\ ...\ \{Q_{n-1}\}\ s_n\ \{Q_n\}$

  Nun gilt:

  Aus $\{Q_{i-1}\}\ s_i\ \{Q_i\}$ für alle $1 \le i \le n$ folgt $\{V\}\ P\ \{N\}$

- Jeder Einzelschritt wird bewiesen, indem man ihn aus der Semantikdefinition der verwendeten Programmiersprache herleitet.
- Beweise dieser Art setzen eine axiomatische Semantik für die verwendete Programmiersprache voraus.

Axiomatische Semantik von Programmiersprachen

- Die Semantik wird durch ein Axiom {V} s {N} für jede ausführbare Anweisung s der verwendeten Programmiersprache definiert.
  (Hoare, 1969)

- In den Axiomen sollen die Ergebniszusicherungen unter möglichst schwachen Voraussetzungen gelten: Gesucht ist die schwächste Voraussetzung (weakest precondition) V, für die {V} s {N} bewiesen werden kann.
  Schreibweise: V ≡ wp (s, N)

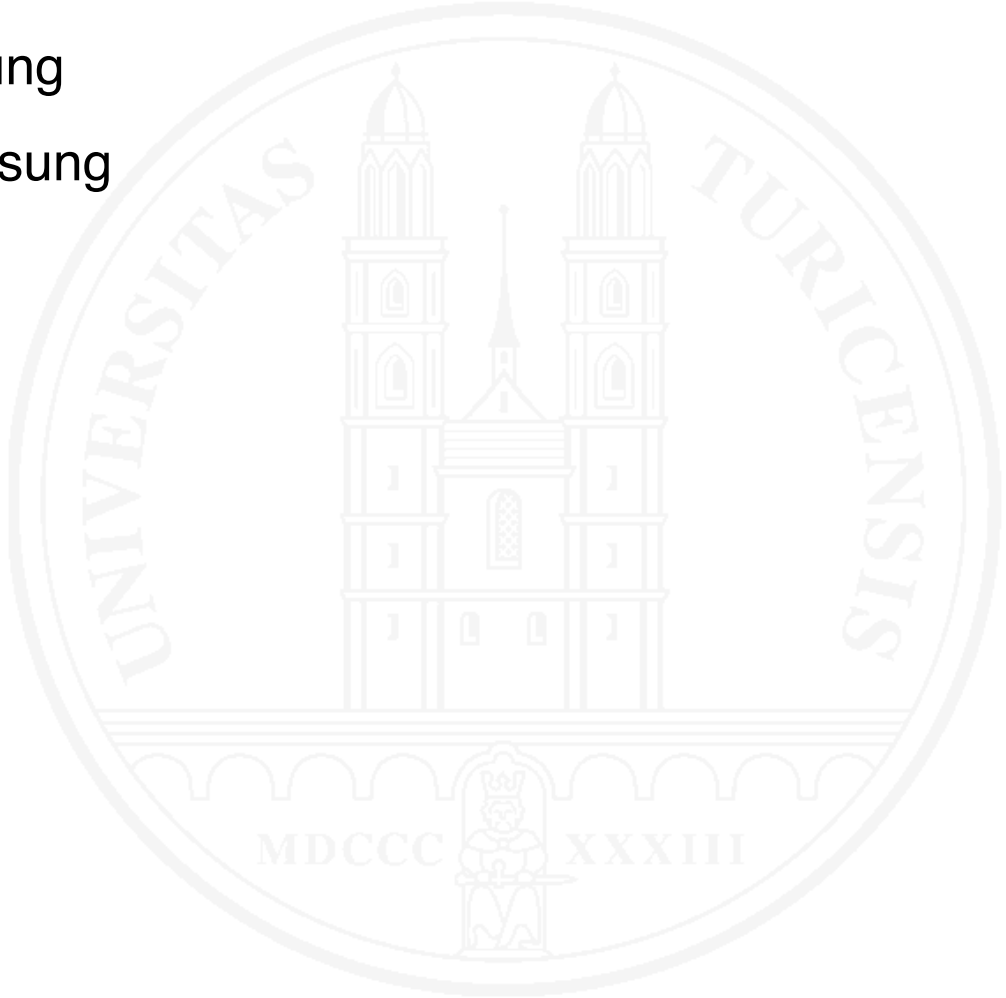
➪ Definiere wp(s, N) für jede ausführbare Anweisung der verwendeten Programmiersprache (Dijkstra, 1976; Gries, 1981)

Axiomatische Semantik von Programmiersprachen – 2

- Die Semantik ist nur für syntaktisch korrekte Konstrukte definiert.
- Es wird daher im Folgenden immer vorausgesetzt, dass die zu verifizierenden Programme oder Programmteile syntaktisch korrekt sind.
- Bei Sprachen mit Typprüfung, z.B. PASCAL oder Java, gehört die Typverträglichkeit aller Ausdrücke und Zuweisungen ebenfalls zur Syntax.

Ausgewählte Axiome für die Sprache PASCAL

- Wertzuweisung
- Leere Anweisung
- Sequenz
- Alternative
- Iteration



Wertzuweisung x := E

Sei x eine Variable, E ein Ausdruck vom gleichen Typ wie x und Q(x) die Ergebniszusicherung von x := E

Dann ist die Semantik von x := E definiert durch

$$wp(x:=E, Q(x)) \equiv E \text{ ist berechenbar} \wedge Q(E)$$

In Worten:
Wenn E berechnet werden kann, so ergibt sich die schwächste Voraussetzung, indem alle Vorkommen von x in der Ergebniszusicherung durch E ersetzt werden.

Hinweis: Situationen, in denen E nicht berechnet werden kann, ergeben sich beispielsweise, wenn E eine Division durch Null enthält.

Mini-Übung 4.1

Seien a, b Zahlen gleichen Typs und seien gegeben

Wertzuweisung	b := a+1
Ergebniszusicherung	b > a

Berechnen Sie die schwächste Voraussetzung.

Leere Anweisung ε

$wp(\varepsilon, Q) \equiv Q$

Die leere Anweisung wird u.a. benötigt, um IF-Anweisungen ohne ELSE-Zweig zu verifizieren.

Sequenz s1 ; s2

$wp(s1;s2, Q) \equiv wp(s1, wp(s2, Q))$

In Worten:
Die schwächste Voraussetzung der zweiten Anweisung ist gleich der Ergebniszusicherung N_{s1} der ersten Anweisung . Die schwächste Voraussetzung der Sequenz ist damit die schwächste Bedingung, die bei Ausführung von s1 die Bedingung N_{s1} liefert.

Mini-Übung 4.2

Seien x,y Zahlen gleichen Typs und seien gegeben

Anweisungssequenz $y := x;$
$x := 0$

Ergebniszusicherung $y = X \wedge x \leq 0$

Berechnen Sie die schwächste Voraussetzung.

Alternative if b then s1 else s2

$$wp(\text{if } b \text{ then } s1 \text{ else } s2, Q) \equiv (b \wedge wp(s1, Q)) \vee (\neg b \wedge wp(s2, Q))$$

In Worten:
Ist b wahr, so hat die Alternative die Semantik der Anweisung s1, andernfalls diejenige der Anweisung s2.

Mini-Übung 4.3

Seien x, y, z vergleichbare Objekte gleichen Typs und seien gegeben

Alternative if x < y then z := y else z := x

Ergebniszusicherung z = max (X, Y)

Berechnen Sie die schwächste Voraussetzung.

Iteration while b do s

- Durch rekursive Definition zurückführbar auf Semantik einer Folge von Anweisungen:

  while b do s ≡ if b then begin s ; while b do s end

➪ führt zu Rekursion in der Definition von wp ➪ schwierig

- Besser: Verifikation über eine Schleifeninvariante
- Liefert allerdings nicht notwendig die schwächste Voraussetzung

➪ Invariante so wählen, dass möglichst schwache Voraussetzungen genügen

Verifikation mit Schleifeninvarianten

- Eine Schleifeninvariante Inv ist ein logischer Ausdruck, der bei jeder Prüfung der Schleifenbedingung b wahr ist (vgl. Kapitel 2)
- Für jeden Schleifendurchlauf gilt $\{Inv \wedge b\}\ s\ \{Inv\}$
- Für den letzten Schleifendurchlauf gilt zusätzlich $\{Inv \wedge b\}\ s_e\ \{Inv \wedge \neg b\}$
- Durch geeignete Wahl von Voraussetzungen V und einer Invarianten Inv ist zu beweisen:
  - $V \Rightarrow Inv$ sowie $\{Inv \wedge b\}\ s_e\ \{Inv \wedge \neg b\} \Rightarrow N$
  - Die Schleife terminiert
- Nach den Regeln für die Sequenz (siehe oben) ist damit
- $\{V\}$ while b do s $\{N\}$ bewiesen.

Mini-Übung 4.4

Gegeben sei die Iteration

```
while i ≤ n do begin
    p := p * s;
    i := i + 1
end
```

mit der Ergebniszusicherung {p = S**N}

Bestimmen Sie eine Voraussetzung, für die das Programm korrekt ist

Mini-Übung 4.5

a) Was passiert, wenn man im Beispiel aus Mini-Übung 4.4 für die Schleifeninitialisierung wie üblich i := 1 wählt?

b) Was passiert, wenn man im Beispiel aus Mini-Übung 4.4 die Voraussetzung abschwächt zu $(p = S) \wedge (i = 2) \wedge (n > 0)$?

c) Was können Sie aus dem Ergebnis von b) schließen?

Verifikation des Programms aus Beispiel 4.1

Seien x, y vom Typ INTEGER

Beweise:

{x = X, y = Y}

x := y;
if x < 0 then x := 0

{x = max (0, Y)}

1. Bestimme Q_3 = wp (x := 0, (x = max (0, Y))
= (0 = max (0, Y))
= (Y ≤ 0)

2. Bestimme Q_2 = wp (ε , (x = max (0, Y))
= (x = max (0, Y))

Verifikation des Programms aus Beispiel 4.1 – 2

3. Bestimme Q_1 = wp (if $x < 0$ then $x := 0$, $(x = \max(0, Y))$

$= ((x < 0) \wedge (Y \leq 0) \vee (x \geq 0) \wedge (x = \max(0, Y)))$

4. Bestimme Q_0 = wp $(x := y, Q_1)$

$= ((y < 0) \wedge (Y \leq 0) \vee (y \geq 0) \wedge (y = \max(0, Y)))$

5. Zeige: $(x = X, y = Y) \Rightarrow Q_0$

$y = Y \Rightarrow (y < 0) \wedge (y = Y) \vee (y \geq 0) \wedge (y = Y)$

$\Rightarrow (y < 0) \wedge (Y \leq 0) \vee (y \geq 0) \wedge (y = \max(0, Y))$

6. Zeige, dass das Programm terminiert

Das Programm enthält weder Schleifen noch Rekursion

➪ das Programm terminiert

4.1 Grundlagen

4.2 Verifikation auf axiomatischer Basis

4.3 Verifikation von Eigenschaften, Model Checking

4.4 Stärken und Schwächen formaler Verifikation

Das Problem

- Die klassische Programmverifikation beweist, dass ein Programm P die gegebene Spezifikation S erfüllt, d.h. logisch gesehen $P \vdash S$
- Ein solcher Beweis ist aufwendig und nicht automatisierbar
- Häufig interessiert man sich primär dafür, die Gültigkeit einzelner, wichtiger Eigenschaften zu beweisen, zum Beispiel
  - Sicherheitseigenschaften (Unerreichbarkeit verbotener/gefährlicher Zustände)
  - Lebendigkeitseigenschaften (Verhinderung von Verklemmung und Verhungern)

Beweis von Eigenschaften

- Wird das zu untersuchende System als Modell *M* und die zu beweisende Eigenschaft als Formel Φ in einer gegebenen Logik (in der Regel Prädikatenlogik oder temporale Logik) aufgefasst, so ist zu beweisen, dass das Modell *M* die Formel Φ erfüllt, d.h. $M \models \Phi$
- Ist *M* eine Implementierung und Φ ein Teil der Spezifikation, so bedeutet dies eine partielle Programmverifikation
- Ist *M* eine Spezifikation und Φ eine Eigenschaft, so wird bewiesen, dass die Spezifikation die Eigenschaft Φ hat

Model Checking

Unter Model Checking versteht man eine automatisierte, partielle Verifikation eines Programms oder einer Spezifikation

- Als Basis dient temporale Logik. Temporale Logik ist im Prinzip Prädikatenlogik mit zwei weiteren Operatoren:
  - □ „immer gilt“
  - ◇ „irgendwann in der Zukunft gilt“
- Model Checking geschieht, indem man mit Hilfe geeigneter Software
  - sämtliche möglichen Variablenbelegungen des gegebenen Modells berechnet
  - die Gültigkeit der zu verifizierenden Eigenschaft Φ für jede Belegung testet

Mini-Übung 4.6

Gegeben sei das Problem des wechselseitigen Ausschlusses von zwei Prozessen p_1 und p_2. Es gebe einen kritischen Bereich c, in dem sich höchstens ein Prozess gleichzeitig befinden darf.

Sei $c_i \equiv p_i$ befindet sich im kritischen Bereich
und $t_i \equiv p_i$ versucht, den kritischen Bereich zu betreten

a) Formulieren Sie den Ausschluss als Formel in temporaler Logik

b) Formulieren Sie die Eigenschaft, dass jeder Prozess, welcher versucht, den kritischen Bereich zu betreten, ihn auch irgendwann betritt.

c) Von welcher Art sind die Eigenschaften a) und b)?

Model Checking – 2

- Vorteil: automatisierte Verifikation
- Nachteil: Die Zustandsräume realer Programme sind für Model Checking zu groß
- ➪ Vereinfachung notwendig, z.B. nur drei Werte für eine Integer-Variable: -3, 0, 3
- Model Checking liefert dann keinen Beweis mehr, sondern nur noch Evidenz
- Wird von einem Beweisverfahren zu einem automatisierten Testverfahren

4.1 Grundlagen

4.2 Verifikation auf axiomatischer Basis

4.3 Verifikation von Eigenschaften, Model Checking

4.4 Stärken und Schwächen formaler Verifikation

Stärken formaler Verifikation

- Ein formal verifiziertes Programm erfüllt nachweislich seine Spezifikation

➪ ein Test erübrigt sich (zumindest theoretisch)

➪ das Programm liefert für alle spezifizierten Daten korrekte Ergebnisse – Fehlfunktionen in Sonderfällen, die beim Implementieren übersehen wurden, gibt es nicht

Schwächen formaler Verifikation

- Viele Beweise ignorieren reale Randbedingungen:

  Die Verifikation ist aufwendig
  - Sehr viel Handarbeit
  - Beweisersoftware unterstützt, ersetzt aber die Handarbeit nicht
  - Automatisierte Verfahren (z.B. Model Checking) verifizieren nur partiell
- Es gibt keine Garantie gegen fehlerhafte Beweise. In der Literatur sind Beispiele fehlerhafter Programmbeweise dokumentiert (z.B. Gerhart und Yelowitz 1976)
- ➪ Der Test wird nicht überflüssig
- Verifikation hilft nicht gegen Spezifikationsfehler

Mini-Übung 4.7

In Mini-Übung 4.1 wurde bewiesen

{a, b sind Zahlen gleichen Typs}
b := a+1
{b > a}

Angenommen, a und b sind vom Typ REAL und intern mit einer Mantisse von 24 Bit und einem Exponenten von 6 Bit repräsentiert.

Wie verhält sich das Programm für $a \geq 2^{23}$?

Fazit

- Eine generelle, vollständige formale Verifikation von Software ist mit den heutigen Mitteln weder machbar noch wirtschaftlich
- Partielle, automatisierte Verifikation (Model Checking) wird in der Praxis teilweise verwendet und dient insbesondere zur Verifikation sicherheitskritischer Eigenschaften von Software
- Die Erstellung und Prüfung einer vollständigen formalen Spezifikation bleibt bei großer Software ein ungelöstes Problem
- Formale Verifikation ist angezeigt
  - für eher kurze, aber sehr wichtige Algorithmen
  - in Spezialfällen, z.B. für Kommunikationsprotokolle
  - für sicherheitskritische Eigenschaften von Software

Literatur

Dijkstra, E.W. (1976). *A Discipline of Programming*. Englewood Cliffs: Prentice Hall.

Gerhart, S.L., L. Yelowitz (1976). Observations on Fallibility in Applications of Modern Programming Methodologies. *IEEE Transactions on Software Engineering*, **SE-2**, 3 (Sept. 1976). 195-207.

Gries, D. (1981). *The Science of Programming*. Berlin, etc.: Springer.

Hoare, C.A.R. (1969). An Axiomatic Basis for Computer Programming. *Communications of the ACM* **12**, 10 (Oct. 1969). 576-583.

Huth, M.R.A., M.D. Ryan (2000). *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge: Cambridge University Press.

IEEE (1990). *Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12-1990. IEEE Computer Society Press.

