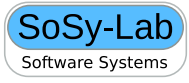
Formale Spezifikation und Verifikation

Wintersemester 2024

Prof. Dr. Gidon Ernst
gidon.ernst@lmu.de

Software and Computational Systems Lab
Ludwig-Maximilians-Universität München, Germany

November 29, 2024



Verifikation von Objektorientierten Programmen

Agenda

- Spezifikation von Klassen
  - Externe *logische* Datensicht (mit Mengen, Sequenzen, ...)
  - Kontrakte für Methoden
- Verifikation von Klassen
  - Klasseninvarianten
  - Vererbungsprinzip

Verifikation von Objektorientierten Programmen

Verifikation: Klasseninvarianten, Vererbung

Beispiel: Implementierung von `List` (Auszug)

class List
  model $xs = \langle\rangle$

  method `add`(e)
    ensures $xs = \mathbf{old}(xs) \circ \langle e \rangle$

Beispiel: Implementierung von `List` (Auszug)

class List
  model $xs = \langle\rangle$

  method `add`(e)
    ensures $xs = \mathbf{old}(xs) \circ \langle e \rangle$

class ArrayList **extends** List
  model $xs = \langle\rangle$

  state `size` $= 0$
  state `data`: array

  method `add`(e)
    ensures $xs = \mathbf{old}(xs) \circ \langle e \rangle$
  begin
    `data`[`size`] $= e$
    `size` $=$ `size` $+ 1$
  end

Beispiel: Implementierung von `List` (Auszug)

class List
  model $xs = \langle\rangle$

  method `add`(e)
    ensures $xs = \mathbf{old}(xs) \circ \langle e \rangle$

class ArrayList **extends** List
  model $xs = \langle\rangle$

  state `size` $= 0$
  state `data`: array

  method `add`(e)
    ensures $xs = \mathbf{old}(xs) \circ \langle e \rangle$
  begin
    `data`[`size`] $= e$
    `size` $=$ `size` $+ 1$
  end

✓ Übernahme der Spezifikation (wegen **extends**)

Beispiel: Implementierung von `List` (Auszug)

class List
  model $xs = \langle\rangle$

  method `add`(e)
    ensures $xs = \mathbf{old}(xs) \circ \langle e \rangle$

class ArrayList **extends** List
  model $xs = \langle\rangle$

  state `size` $= 0$
  state `data`: array

  method `add`(e)
    ensures $xs = \mathbf{old}(xs) \circ \langle e \rangle$
  begin
    `data`[`size`] $= e$
    `size` $=$ `size` $+ 1$
  end

- ✓ Übernahme der Spezifikation (wegen **extends**)
- ✓ Hinzufügen von Attributen und Implementierung

Beispiel: Implementierung von `List` (Auszug)

class List
  model $xs = \langle\rangle$

  method `add`(e)
    ensures $xs = \mathbf{old}(xs) \circ \langle e \rangle$

class ArrayList **extends** List
  model $xs = \langle\rangle$

  state `size` $= 0$
  state `data`: array

  method `add`(e)
    ensures $xs = \mathbf{old}(xs) \circ \langle e \rangle$
  begin
    `data`[`size`] $= e$
    `size` $=$ `size` $+ 1$
  end

- ✓ Übernahme der Spezifikation (wegen **extends**)
- ✓ Hinzufügen von Attributen und Implementierung
- ▶ Zum Pausieren und Nachdenken: ist ArrayList korrekt? Warum (nicht)?

Welche Bedingungen müssen für ArrayList gelten?

Überlegungen

- Wie genau repräsentiert das externe Modell xs den internen Zustand `data`, `size` der Klasse?
- Wann ist eine Methode korrekt implementiert? ($\approx$ Regel zur Verifikation von Prozeduren)

class ArrayList **extends** List

model $xs = \langle\rangle$

state `size` $= 0$

state `data`: array

method `add`(e)

ensures $xs = \mathbf{old}(xs) \circ \langle e \rangle$

begin

`data`[`size`] $= e$

`size` $=$ `size` $+ 1$

end

Welche Bedingungen müssen für ArrayList gelten?

Überlegungen

- Wie genau repräsentiert das externe Modell xs den internen Zustand `data`, `size` der Klasse?
- Wann ist eine Methode korrekt implementiert? ($\approx$ Regel zur Verifikation von Prozeduren)

$\rightarrow$ Klasseninvarianten

class ArrayList **extends** List

model $xs = \langle\rangle$

state `size` $= 0$

state `data`: array

method `add`(e)

ensures $xs = \mathbf{old}(xs) \circ \langle e \rangle$

begin

`data`[`size`] $= e$

`size` $=$ `size` $+ 1$

end

Klasseninvarianten

Gegeben:

- Modellvariable(n) $\vec{m} = m_1, \ldots, m_n$
- Klassenattribute $\vec{\mathsf{a}} = \mathsf{a}_1, \ldots, \mathsf{a}_n$

Eine *Klasseninvariante* $R(\vec{m}, \vec{\mathsf{a}})$ ist eine Relation die

- den Zusammenhang zwischen der externen und internen Sicht beschreibt

Klasseninvarianten

Gegeben:

- Modellvariable(n) $\vec{m} = m_1, \ldots, m_n$
- Klassenattribute $\vec{\mathsf{a}} = \mathsf{a}_1, \ldots, \mathsf{a}_n$

Eine *Klasseninvariante* $R(\vec{m}, \vec{\mathsf{a}})$ ist eine Relation die

- den Zusammenhang zwischen der externen und internen Sicht beschreibt
  Beispiel: $xs = \langle \mathtt{data}[0], \ldots, \mathtt{data}[\mathtt{size} - 1] \rangle$

Klasseninvarianten

Gegeben:

- Modellvariable(n) $\vec{m} = m_1, \ldots, m_n$
- Klassenattribute $\vec{\mathsf{a}} = \mathsf{a}_1, \ldots, \mathsf{a}_n$

Eine *Klasseninvariante* $R(\vec{m}, \vec{\mathsf{a}})$ ist eine Relation die

- den Zusammenhang zwischen der externen und internen Sicht beschreibt
  Beispiel: $xs = \langle \mathtt{data}[0], \ldots, \mathtt{data}[\mathtt{size} - 1] \rangle$
- zusätzliche Konsistenzbedingungen über den internen Zustand $\vec{\mathsf{a}}$ aufstellt

Klasseninvarianten

Gegeben:

- Modellvariable(n) $\vec{m} = m_1, \ldots, m_n$
- Klassenattribute $\vec{\mathsf{a}} = \mathsf{a}_1, \ldots, \mathsf{a}_n$

Eine *Klasseninvariante* $R(\vec{m}, \vec{\mathsf{a}})$ ist eine Relation die

- den Zusammenhang zwischen der externen und internen Sicht beschreibt
  Beispiel: $xs = \langle \mathtt{data}[0], \ldots, \mathtt{data}[\mathtt{size} - 1] \rangle$
- zusätzliche Konsistenzbedingungen über den internen Zustand $\vec{\mathsf{a}}$ aufstellt
  Beispiel: $\mathtt{size} \leq \mathtt{data.length}$
  Beispiel: verkettete Liste hat keine Zyklen, internes Array ist sortiert

Klasseninvarianten

Gegeben:

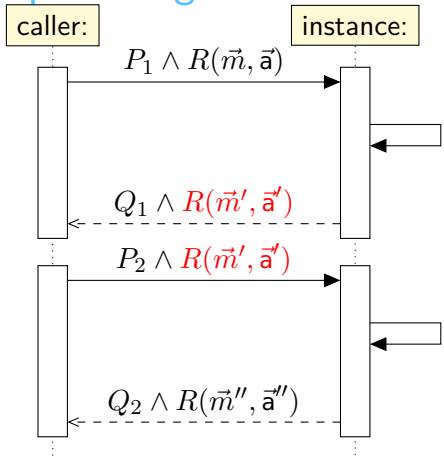
- Modellvariable(n) $\vec{m} = m_1, \ldots, m_n$
- Klassenattribute $\vec{\mathsf{a}} = \mathsf{a}_1, \ldots, \mathsf{a}_n$

Eine *Klasseninvariante* $R(\vec{m}, \vec{\mathsf{a}})$ ist eine Relation die

- den Zusammenhang zwischen der externen und internen Sicht beschreibt
  Beispiel: $xs = \langle \mathtt{data}[0], \ldots, \mathtt{data}[\mathtt{size} - 1] \rangle$
- zusätzliche Konsistenzbedingungen über den internen Zustand $\vec{\mathsf{a}}$ aufstellt
  Beispiel: $\mathtt{size} \leq \mathtt{data.length}$
  Beispiel: verkettete Liste hat keine Zyklen, internes Array ist sortiert

Definition: Relation $R(\vec{m}, \vec{\mathsf{a}})$ ist eine (korrekte) Klasseninvariante wenn $\exists \vec{m}.\ R(\vec{m}, \vec{\mathsf{a}})$ in allen Zuständen möglicher Ausführungen gilt für alle Instanzen (Objekte) der Klasse, für die nicht gerade eine Methode ausgeführt wird.

Visualisierung als Sequenzdiagramm



- Zusammenhang zwischen Objektzustand $\vec{a}$ und logischem Modell $\vec{m}$ werden vom Aufrufer von einem Aufruf zum nächsten erhalten
- Voraussetzung: kein direkter Zugriff auf $\vec{a}$ von Außen (private Attribute)
- Zwischendurch darf die Verifikation des Aufrufers die Veränderung von $\vec{m}$ gemäß den Nachbedingungen “beobachten”

Ansatz zur Verifikation

Theorem: $R(\vec{m}, \vec{a})$ Klasseninvariante falls

- $\exists \vec{m}.\ R(\vec{m}, \vec{a})$ gilt nach Ausführen des Konstruktors
- $\exists \vec{m}.\ R(\vec{m}, \vec{a})$ zusätzliche (implizite) Vor- und Nachbedingung für Methoden

In Dafny muss man die Veränderung für $\vec{m}$ konkret angeben und auch die Klasseninvariante(n) explizit als Vor-/Nachbedingung hinzfügen

Ansatz zur Verifikation

Theorem: $R(\vec{m}, \vec{\mathsf{a}})$ Klasseninvariante falls

- $\exists \vec{m}.\ R(\vec{m}, \vec{\mathsf{a}})$ gilt nach Ausführen des Konstruktors
- $\exists \vec{m}.\ R(\vec{m}, \vec{\mathsf{a}})$ zusätzliche (implizite) Vor- und Nachbedingung für Methoden

In Dafny muss man die Veränderung für $\vec{m}$ konkret angeben und auch die Klasseninvariante(n) explizit als Vor-/Nachbedingung hinzfügen

Konkret: für Methode mit Parametern $\vec{\mathsf{x}}$ Vorbedingung P, Nachbedingung Q, Implementierung $code$ müssen wir das Hoare Tripel beweisen

$\{P \wedge \vec{x}^{\mathsf{pre}} = \vec{\mathsf{x}} \wedge R(\vec{m}^{\mathsf{pre}}, \vec{\mathsf{a}})\}$

$\vdots$

$code$

$\vdots$

$\{\exists \vec{m}.\ Q[\vec{\mathsf{x}} \mapsto \vec{x}^{\mathsf{pre}}, \mathbf{old}(\vec{m}) \mapsto \vec{m}^{\mathsf{pre}}] \wedge R(\vec{m}, \vec{\mathsf{a}})\}$

- Logische Variablen $\vec{x}^{\mathsf{pre}}, \vec{m}^{\mathsf{pre}}$ verweisen auf Startwerte
- Wir müssen *angeben*, was der neue Wert von $\vec{m}$ sein soll (durch Instanziierung des $\exists$-Quantors in der Nachbedingung)

Beispiel: Verifikation der Klasse ArrayList

Gegeben:

$R(xs, \texttt{data}, \texttt{size}) = \texttt{size} \leq \texttt{data.length} \wedge xs = \langle \texttt{data}[0], \ldots, \texttt{data}[\texttt{size} - 1] \rangle$

Beispiel: Verifikation der Klasse ArrayList

Gegeben:
$R(xs, \texttt{data}, \texttt{size}) = \texttt{size} \leq \texttt{data.length} \wedge xs = \langle \texttt{data}[0], \ldots, \texttt{data}[\texttt{size} - 1] \rangle$

Überprüfung der Initialisierung

- $\exists xs.\ R(xs, \texttt{data}, 0)$
  - ✓ $0 \leq \texttt{data.length}$
  - ✓ $\langle \ldots, \texttt{data}[0 - 1] \rangle = \langle\rangle$ (Konvention der informellen Notation für Sequenzen)
- Daher: wähle $xs = \langle\rangle$

Beispiel: Verifikation der Klasse ArrayList

Gegeben:
$R(xs, \texttt{data}, \texttt{size}) = \texttt{size} \leq \texttt{data.length} \wedge xs = \langle \texttt{data}[0], \ldots, \texttt{data}[\texttt{size} - 1] \rangle$

Überprüfung der Methode $\texttt{add}(e)$ mit

- Vorbedingung $\quad \texttt{true} \wedge R(xs^{\texttt{pre}}, \texttt{data}, \texttt{size})$
- Nachbedingung $\quad \exists xs.\ xs^{\texttt{pre}} \circ e = xs \wedge R(xs, \texttt{data}, \texttt{size})$

(analog zur Prozedurregel, Arrays wie in bei Schleifeninvarianten)

Beispiel: Verifikation der Klasse ArrayList

Gegeben:
$R(xs, \texttt{data}, \texttt{size}) = \texttt{size} \leq \texttt{data.length} \wedge xs = \langle \texttt{data}[0], \ldots, \texttt{data}[\texttt{size} - 1] \rangle$

Überprüfung der Methode $\texttt{add}(e)$ mit

- Vorbedingung $\quad \texttt{true} \wedge R(xs^{\texttt{pre}}, \texttt{data}, \texttt{size})$
- Nachbedingung $\quad \exists xs.\ xs^{\texttt{pre}} \circ e = xs \wedge R(xs, \texttt{data}, \texttt{size})$

(analog zur Prozedurregel, Arrays wie in bei Schleifeninvarianten)

Beobachtungen

- ✗ Falls zufällig $\texttt{size} = \texttt{data.length}$ dann $\texttt{data}[\texttt{size}]$ ungültiger Index
- ✗ $\texttt{size} \leq \texttt{data.length} \not\Longrightarrow \texttt{size} + 1 \leq \texttt{data.length}$

Beispiel: Verifikation der Klasse ArrayList

Gegeben:
$R(xs, \texttt{data}, \texttt{size}) = \texttt{size} \leq \texttt{data.length} \wedge xs = \langle \texttt{data}[0], \ldots, \texttt{data}[\texttt{size} - 1] \rangle$

Überprüfung der Methode $\texttt{add}(e)$ mit

- Vorbedingung $\quad \texttt{true} \wedge R(xs^{\texttt{pre}}, \texttt{data}, \texttt{size})$
- Nachbedingung $\quad \exists xs.\ xs^{\texttt{pre}} \circ e = xs \wedge R(xs, \texttt{data}, \texttt{size})$

(analog zur Prozedurregel, Arrays wie in bei Schleifeninvarianten)

Beobachtungen

- ✗ Falls zufällig $\texttt{size} = \texttt{data.length}$ dann $\texttt{data}[\texttt{size}]$ ungültiger Index
- ✗ $\texttt{size} \leq \texttt{data.length} \not\Longleftrightarrow \texttt{size} + 1 \leq \texttt{data.length}$
- Wahl von $\exists xs$ bereits durch $xs^{\texttt{pre}} \circ e = xs$ bestimmt
- ✓ $\mathbf{old}(\langle \texttt{data}[0], \ldots, \texttt{data}[\texttt{size} - 1] \rangle) \circ \langle e \rangle = \langle \texttt{data}[0], \ldots, \texttt{data}[\texttt{size} - 1] \rangle$

Vererbung

Komplexes Thema!

Möglichkeiten für das Erben von Interfaces/Klassen

- Erweiterung des logischen Modells
- Hinzunahme neuer Invarianten

Prinzip für das Implementieren/Überschreiben von Methoden in Subklassen

- Vorbedingungen dürfen abgeschwächt werden
- Nachbedingungen dürfen verstärkt werden

(analog zur Konsequenzregel)

Diskussion

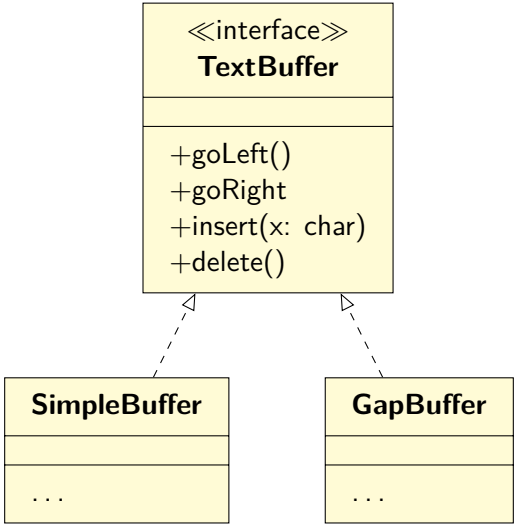
Finden korrekter Klasseninvarianten

- Analog zu Schleifeninvarianten: erfordert Kreativität!
- Automatisierung schwierig aber in manchen Fällen möglich
  (Gegenstand meiner aktuellen Forschung!)

Modularität & Kapselung von Klasseninvarianten

- Irrelevant für den Aufrufer! Dieser "sieht" nur die logische Sicht
- Alleinige Verantwortung der implementierenden Klasse (in deren Verifikation)

Case Study: Datenstruktur für einen Texteditor



Einfache Implementierung

```
class SimpleBuffer {
    String contents;
    int position;

    void goLeft()  { if(0 < position) position--; }
    void goRight() { if(position < contents.length()) position++; }

    void insert(char x) {
        String first  = contents.substring(0, position);
        String second = contents.substring(position);
        contents = first + x + second;
        position ++;
    }

    void delete() {
        // similar
    }
}
```

Visualisierung der einfachen Implementierung

Effizienter Gap Buffer

```
char[] buf;
// invariant 0 <= left <= right <= buf.length
int left, right;

void goLeft() {
  if(0 < left) {
    left  -= 1;
    right -= 1;
    buf[right] = buf[left];
  }
}

void goRight() {
  if(0 < left) {
    buf[left] = buf[right];
    left  += 1;
    right += 1;
  }
}

void insert(char x) {
  if(left < right) {
      buf[left] = x;
      left ++;
  }
}

void delete() {
  if(0 < left) {
      left --
  }
}
```

Visualisierung des Gap Buffers

Spezifikation und Korrektheitsbeweis

Zusammenfassung

Klasseninvarianten

- beschreiben den Zusammenhang zwischen logischer Außensicht und dem internen Zustand
- können zusätzliche Eigenschaften über die internen Datenstrukturen ausdrücken, diese werden evtl. zur Verifikation benötigt werden
- müssen vom Konstruktor etabliert werden
- müssen über alle Methoden erhalten bleiben

License