LEHRSTUHL 14
SOFTWARE ENGINEERING

Vorlesung (WS 2013/14)

Softwarekonstruktion

Prof. Dr. Jan Jürjens

TU Dortmund, Fakultät Informatik, Lehrstuhl XIV

Teil 3.3: Algebraische Spezifikation

v. 17.01.2014

[inkl. Beiträge von Prof. Volker Gruhn und Dr. Johannes Henkel]

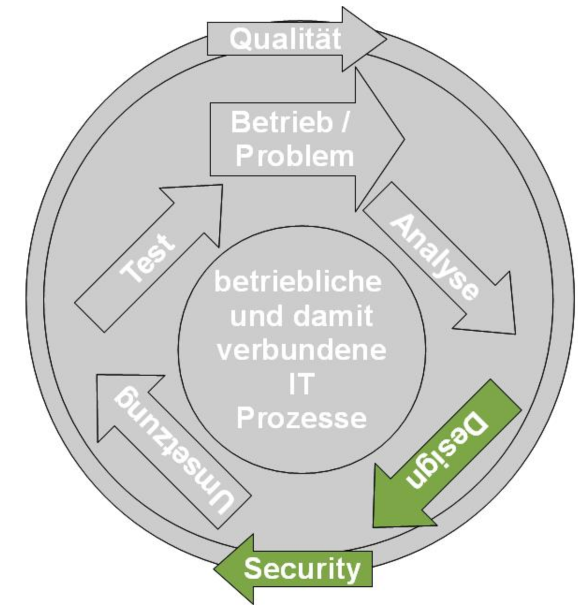
Literatur:

J. Ludewig, H. Lichter: Software Engineering - Grundlagen, Menschen, Prozesse, Techniken, dpunkt.verlag, 3. Auflage, 2013
https://www2.swc.rwth-aachen.de/se_buch . Ab 33 EUR (e-book).
Unibibliothek (2 Exemplare, 2. Auflage):
http://www.ub.tu-dortmund.de/katalog/titel/1283989 .
(Bei Engpässen kann eine **Kopiervorlage** der relevanten Ausschnitte zur Verfügung gestellt werden.)

- **Kapitel 5**

Einordnung Algebraische Spezifikation

- Modellgetriebene SW-Entwicklung
  - OCL
  - Modellbasierte Softwareentwicklung
  - EMF
- Qualitätsmanagement
- Softwareverifikation
  - Algebraische Spezifikation



3.3 Algebraische Spezifikation

- Das Spezifikationsproblem
- Vollständigkeit der Algebraischen Spezifikation
- Signatur und Algebra
- Algebraische Spezifikation: Diskussion

Motivation

In diesem Kapitel: ***Algebraische Spezifikation*** als *Grundlage* für *Spezifikation* des *Verhaltens einzelner Softwaremodule*.

Warum ?

- ***Verhalten einzelner Softwaremodule*:** heutige Softwaresysteme sind von erheblicher Komplexität. → Durch Verwendung modularer Herangehensweisen beherrschbar.
- ***Spezifikation*:** Spezifikation Verhalten einzelner Softwaremodule wichtig.

  → Modulare Softwareentwicklung und Qualitätssicherung ermöglichen.
- ***Grundlage*:** Praxis: Verschiedene Ansätze zur Spezifikation des Verhalten einzelner Softwaremodule, z.B. UML, Java / C assertions, Java Markup Language (JML). **In diesem Abschnitt:** Formale Grundlagen dazu betrachten.

Direkter Einsatz in einigen Anwendungen. (Beispiel Verifikation von kryptographischen Protokollen: Spezifikation kryptographischer Eigenschaften als algebraische Gleichungen; vgl. später).

Algebraische Spezifikation

- Betrachtet **abstrakte Datentypen** als algebraische Strukturen.

  → Mengen mit darauf definierten Operationen, für die bestimmte Axiome gelten.
- Dient zur **formalen Spezifikation des Verhaltens** von Modulen/ Schnittstellen.
- Dient zur **Verifikation der Implementierung** gegen Spezifikation und Verifikation der Spezifikation gegenüber zu erfüllenden Anforderungen.
- Durch **Zusammensetzen solcher Module** kommt man zu **Software-Systemen**. → Definition deren Verhalten durch Spezifikation der Module und Konstruktion der Module zum Software-System.
- **Konstruktion von Software-Systemen** aus Modulen: Gegenstand des Entwurfs.

Algebraische Spezifikation
Vorgehen

Nach [Som10] besteht Erstellungsprozess aus folgenden Schritten:

1) **Strukturieren der Spezifikation:** Alle Schnittstellen eines Systems identifizieren.
   → Miteinander kommunizierende Teile herausfinden
   → Notwendige Operationen informell beschreiben.
2) **Benennen der Spezifikation:** Alle abstrakten Datentypen einer Spezifikation mit Namen versehen (z.B. Liste) und entscheiden, ob sie generische Parameter benötigen.
3) **Auswahl der Operationen:** Bezeichnung aller Operationen festlegen.
4) **Informelles Spezifizieren der Operation:** Identifizierte Operationen sowie deren Auswirkungen auf System beschreiben.
5) **Definieren der Syntax:** Für jede identifizierte Operation Syntax formal beschreiben.
   → Alle Operationen zusammen mit ihren Parametern definieren. Syntax gesamter Spezifikation: Zusammensetzung aller abstrakten Datentypen.
6) **Definieren der Semantik:** Definition der Semantik der Operationen: Zutreffende Bedingungen (Axiome) für verschiedene Kombinationen von Operationen beschreiben.

[Som10] Sommerville, Ian: Software engineering, ch. 27: Formal Specification. Addison-Wesley Longman Publishing Co., Inc., 2010.
http://www.cs.st-andrews.ac.uk/~ifs/Books/SE9/WebChapters/PDF/Ch_27_Formal_spec.pdf

Zu Schritten 5) und 6)
Syntax und Semantik

Algebraische Spezifikation eines abstrakten Datentyps besteht aus Syntax- und Semantikteil:

- **Syntax:** Menge von Vereinbarungen von Zugriffsoperationen:
  - Operationsname: Definitionsbereich → Wertebereich.
- **Semantik:** Menge von Gleichungen, die Beziehungen zwischen Eingabe- und Rückgabewerten verschiedener Operationen in Form von Axiomen beschreiben.

Bemerkung: Algebraische Spezifikation bietet zunächst kein Konzept, interne Zustandsänderungen direkt zu beschreiben, sondern allenfalls indirekt, z.B. indem für jede Operation, für die Zustandsänderungen modelliert werden soll, ein Eingabe- und Rückgabewert vom Type „State“ (o.ä.) definiert wird (1). Es gibt allerdings Erweiterungen, die internen Zustand als Modellierungskonzept beinhalten (z.B. Abstract State Machines (2) oder Algebraic State Machines (3)).

(1) vgl. http://www.ldl.jaist.ac.jp/jaist-fssv2010/studentsSlides/zhang_slides.pdf , http://www-plan.cs.colorado.edu/henkel/pubs.html

(2) http://en.wikipedia.org/wiki/Abstract_state_machines

(3) http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.140.8498&rep=rep1&type=pdf

Zu Schritten 5) und 6)
Syntax und Semantik

algebra X

introduces sorts X,Y,Z

operations

op1 : X → X

op2 : X → Y

constraints op1, op2 **so that for all**

x,y : X, z: Y

…

Syntax

Semantik

algebra stack-of-nat **introduces sorts** nat, bool, stack-of-nat

operations

create: → stack-of-nat

push: stack-of-nat, nat → stack-of-nat

pop: stack-of-nat → stack-of-nat

top: stack-of-nat → nat

isempty: stack-of-nat → bool

(push, pop: verändernde Operationen; top, isempty: inspizierende Operationen)

constraints create, push, pop, top, isempty

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

isempty (create()) = true

isempty (push (st, n)) = false

pop (create()) = create()

pop (push (st, n)) = st

top (create()) = 0

top (push (st, n)) = n

Beispiel für gesamtes Vorgehen

Spezifikation des Datentyps String

Schritt 1) - 4):

1) 3) 2)

- Erzeugen (*new*). Sorte *String* wird dafür gebraucht.
- Verketten (*append*).
- Zeichen anhängen (*add*). Sorte *Char* wird außerdem gebraucht.
- Länge ermitteln (*length*). Sorte *Nat* wird außerdem gebraucht.

  NB: Wir definieren hier die Sorte *Nat* als die natürlichen Zahlen inklusive 0.
- Ermitteln, ob leer (*isEmpty*). Sorte *Bool* wird außerdem gebraucht.
- Gleichheit zweier Zeichenketten (*equal*)

Schritt 4) hier implizit (es ist klar, was z.B. „verketten“ für Auswirkungen hat).

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

Spezifikation des Datentyps String

algebra StringSpec legt Syntax fest, aber was bedeuten Operationen?

- Beschreibung der **Semantik** der Operationen durch **Gleichungen**.

  → Formulieren Eigenschaften, die von Realisierung der Operationen nicht verletzt werden dürfen. Gleichungen heißen deshalb **Axiome**.

- Spätere Realisierungen der Operationen, die Axiome nicht verletzen: **spezifikationskonform** !

- „Lücken“ in Axiomen: Ursache für spätere Missverständnisse.

- (Informelle) **Definition**: Wenn sich nicht alle als wahr angenommenen Aussagen auch anhand Axiome ableiten lassen. → Algebraische Spezifikation **unvollständig.**

Spezifikation des Datentyps String
Schritt 6

algebra StringSpec **introduces sorts** String, Char, Nat, Bool

operations ....

constraints new, append, add, length, isEmpty, equal

so that for all s,s1,s2: String, c: Char

isEmpty(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)

Semantik

Frage: Sind diese Gleichungen (intuitiv) „korrekt“ ? Sind sie „vollständig“ ?

Spezifikation des Datentyps String
Schritt 6

„Korrekt“ ? => Die o.g. Gleichungen sind intuitiv korrekt...

… allerdings nur, wenn bestimmte Annahmen an die Intuition erfüllt sind (die im Rahmen von Schritt 4 des Vorgehens bereits vorab informell spezifiziert worden sein sollten). Zum Beispiel:

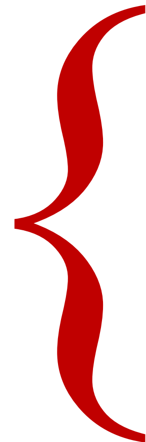
- Das Axiom „append (s1, add(s2,c)) = add (append(s1,s2),c)“ setzt voraus, dass die intendierten Implementierungen von append(s1,s2) und add (s,c) konsistent bezüglich der Reihenfolgen von s1 und s2 bzw s und c sein sollen (z.B. append(s1,s2)=[s1::s2] und add(s,c)=[s::c] oder append(s1,s2)=[s2::s1] und add(s,c)=[c::s], aber nicht append(s1,s2)=[s1::s2] und add(c,s)=[c::s], wobei [s1::s2] die Konkatenation von s1 gefolgt von s2 ist).
- Das Axiom „isEmpty(add (s,c)) = false“ setzt voraus, dass es kein „leeres“ Zeichen geben soll.

Vollständig ? Nein. Zum Beispiel: Die Gleichungen implizieren nicht:

equal(s1,s2) = false (für s1#s2 mit length(s1)=length(s2))
obwohl dies intuitiv gelten sollte.

Anmerkung: Kommutativität von equal folgt allerdings schon aus den Folien der vorigen Folie (Beweis per Induktion).

3.3 Algebraische Spezifikation



Das Spezifikationsproblem

Vollständigkeit der Algebraischen Spezifikation

Signatur und Algebra

Algebraische Spezifikation: Diskussion

Frage:

Gilt

(i) append (new(), add (new(), c)) = add (new(), c) ??

Frage:

Gilt

(i) append (new(), add (new(), c)) = add (new(), c) ??

Start vom Axiom:

(ii) append (s1, add(s2,c)) = add (append (s1,s2),c)

Frage:

Gilt

(i) append (new(), add (new(), c)) = add (new(), c) ??

Start vom Axiom:

(ii) append (s1, add(s2,c)) = add (append (s1,s2),c)

Wir ersetzen hier s1 und s2 durch new():

(iii) append (new(), add (new(),c)) = add (append (new(), new()),c)

Frage:

Gilt

(i) append (new(), add (new(), c)) = add (new(), c) ??

Start vom Axiom:

(ii) append (s1, add(s2,c)) = add (append (s1,s2),c)

Wir ersetzen hier s1 und s2 durch new():

(iii) append (new(), add (new(),c)) = add (append (new(), new()),c)

Im Axiom append (s, new()) = s ersetzen wir s durch new() und erhalten:

(iv) append (new(), new()) = new()

Spezifikation des Datentyps String

Vollständigkeit der Axiome I

Frage:

Gilt

(i) append (new(), add (new(), c)) = add (new(), c) ??

Start vom Axiom:

(ii) append (s1, add(s2,c)) = add (append (s1,s2),c)

Wir ersetzen hier s1 und s2 durch new():

(iii) append (new(), add (new(),c)) = add (append (new(), new()),c)

Im Axiom append (s, new()) = s ersetzen wir s durch new() und erhalten:

(iv) append (new(), new()) = new()

In (iii) setzen wir (iv) ein und erhalten:

(v) append (new(), add (new(),c)) = add (new(),c)

womit (i) gezeigt ist.

Frage: Gilt (i) append (new(), s) = s ??

Frage: Gilt (i) append (new(), s) = s ??

- Induktionsanfang

  Offensichtlich gilt (i) für s=new(), denn append(new(),new()) = new() gilt nach Axiom append(s,new()) = s

Frage: Gilt (i) append (new(), s) = s ??

- Induktionsanfang

  Offensichtlich gilt (i) für s=new(), denn append(new(),new()) = new() gilt nach Axiom append(s,new()) = s

- Induktionsannahme:

  Wir wählen ein beliebiges s' sodass s=add(s',c) und nehmen an, dass (i) für s' gilt.

Frage: Gilt (i) append (new(), s) = s ??

- Induktionsanfang

  Offensichtlich gilt (i) für s=new(), denn append(new(),new()) = new() gilt nach Axiom append(s,new()) = s

- Induktionsannahme:

  Wir wählen ein beliebiges s' sodass s=add(s',c) und nehmen an, dass (i) für s' gilt.

- Nachweis des Axioms

  Damit gilt:

append (new(),s) =	Induktionsannahme
append (new(), add(s',c)) =	Axiom append (s1, add(s2,c)) = add (append(s1,s2),c)
add (append (new(),s'),c) =	Induktionsannahme
add (s',c) =	Induktionsannahme
s	

Spezifikation des Datentyps String
Vollständigkeit der Axiome II

Frage: Gilt (i) append (new(), s) = s ??

- Induktionsanfang

  Offensichtlich gilt (i) für s=new(), denn append(new(),new()) = new() gilt nach Axiom append(s,new()) = s

- Induktionsannahme:

  Wir wählen ein beliebiges s' sodass s=add(s',c) und nehmen an, dass (i) für s' gilt.

- Nachweis des Axioms

  Damit gilt:

append (new(),s) =	Induktionsannahme
append (new(), add(s',c)) =	Axiom append (s1, add(s2,c)) = add (append(s1,s2),c)
add (append (new(),s'),c) =	Induktionsannahme
add (s',c) =	Induktionsannahme
s	

Frage: Für welche Elemente in String beweist Induktionsbeweis o.g. Gleichung ?

Frage: Gilt (i) append (new(), s) = s ??

- Induktionsanfang

  Offensichtlich gilt (i) für s=new(), denn append(new(),new()) = new() gilt nach Axiom append(s,new()) = s

- Induktionsannahme:

  Wir wählen ein beliebiges s' sodass s=add(s',c) und nehmen an, dass (i) für s' gilt.

- Nachweis des Axioms

  Damit gilt:

append (new(),s) =	Induktionsannahme
append (new(), add(s',c)) =	Axiom append (s1, add(s2,c)) = add (append(s1,s2),c)
add (append (new(),s'),c) =	Induktionsannahme
add (s',c) =	Induktionsannahme
s	

Frage: Für welche Elemente in String beweist Induktionsbeweis o.g. Gleichung ?
Antwort: Strings, die durch new und add erzeugt werden.

Generierende Operationen
Definition

LEHRSTUHL 14
SOFTWARE ENGINEERING

- Letzter Beweis basiert auf **Annahme:** Alle Strings sind durch Operationen *new* und *add* erzeugbar (nur für diese gilt der Beweis).
- **Identifikation erzeugender Menge** von Operationen. →
  Zur Ermittlung erforderlicher Axiome und Sicherstellung der Beweiskraft von Induktionsbeweisen.
- **Definition**: Menge O von Operationen heisst „**generierend**" für Menge X, wenn alle Elemente in X durch sukzessive Anwendung der Operationen erzeugt werden können.
- **Bemerkung:** Insbesondere kann Menge O null-stellige Operationen (= Konstanten) enthalten.

Generierende Operationen
Beispiel

Frage:

- Was ist generierende Menge von Operationen der Booleschen Algebra ?

- Was ist generierende Menge von Operationen der Algebra der positiven, ganzen Zahlen ?

Generierende Operationen
Beispiel

Frage:

- Was ist generierende Menge von Operationen der Booleschen Algebra ?

Antwort: zum Beispiel {false, not}

- Was ist generierende Menge von Operationen der Algebra der positiven, ganzen Zahlen ?

Generierende Operationen

Beispiel

Frage:

- Was ist generierende Menge von Operationen der Booleschen Algebra ?

Antwort: zum Beispiel {false, not}

- Was ist generierende Menge von Operationen der Algebra der positiven, ganzen Zahlen ?

Antwort: zum Beispiel {zero, succ}

Im Rahmen algebraischer Spezifikation können wir definieren, dass Typen von bestimmten Operationen generiert werden:

constraints <*operations*> **so that**
<*Type*> **generated by** [<*generating ops*>]
for all <*variables*>: <*Type*> ...

Beispiel:

constraints new, append, add, length, isEmpty, equal **so that**
String **generated by** [new,add,Char]
for all s,s1,s2: String, c: Char

Bemerkung: Hier wird Char als Menge von Atomen (generierende Konstanten) vorausgesetzt.

„Vollständige“ Algebraische Spezifikation

Erweiterung algebraischer Spezifikation *String* um Konstanten 'a‘, 'b' von Typ Char (formal: nullstellige Operation a: () → 'a').

isEmpty(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)

Frage:

equal (add (s, 'a'), add (s, 'b')) = false ?

„Vollständige“ Algebraische Spezifikation

Erweiterung der algebraischen Spezifikation *String* um Konstanten 'a', 'b' von Typ Char (formal: nullstellige Operation a: () → 'a').

```
isEmpty(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)
```

Frage:

equal (add (s, 'a'), add (s, 'b')) = false ?

- Sollte intuitiv gelten, lässt sich aber nicht aus Axiomen ableiten.

→ **Algebraische Spezifikation: Unvollständig**, weil nicht alle (intuitiv) als wahr angenommene Aussagen beweisbar.

Vervollständigung von *String* durch zusätzliche Operation

equalC : Char, Char → Bool

mit den Axiomen:

equalC ('a','a') = true

equalC ('a','b') = false

equalC ('b','a') = false

equalC ('b','b') = true

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)

Vollständigkeit der Algebra

Frage:

equal (add (s,'a'), add (s,'b'))
= false ?

isEmpty(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)~~

equalC ('a','a') = true (usw...)
equalC ('a','b') = false
equal (add (s1,c1), add (s2,c2))
= equal (s1,s2) ∧ equalC (c1,c2)

Vollständigkeit der Algebra

Frage:

equal (add (s,'a'), add (s,'b'))
= false ?

Antwort:

equal (add (s,'a'), add (s,'b'))

```
isEmpty(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)~~

```
equalC ('a','a') = true        (usw...)
equalC ('a','b') = false
equal (add (s1,c1), add (s2,c2))
        = equal (s1,s2) ∧ equalC (c1,c2)
```

Vollständigkeit der Algebra

Frage:

equal (add (s,'a'), add (s,'b'))
= false ?

Antwort:

equal (add (s,'a'), add (s,'b'))
= equal (s,s) ∧ equalC ('a','b')

isEmpty(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)~~

equalC ('a','a') = true (usw...)
equalC ('a','b') = false
equal (add (s1,c1), add (s2,c2))
= equal (s1,s2) ∧ equalC (c1,c2)

Vollständigkeit der Algebra

Frage:

equal (add (s,'a'), add (s,'b'))
= false ?

Antwort:

equal (add (s,'a'), add (s,'b'))
= equal (s,s) ∧ equalC ('a','b')
= true ∧ false = false

isEmpty(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)~~

equalC ('a','a') = true (usw...)
equalC ('a','b') = false
equal (add (s1,c1), add (s2,c2))
= equal (s1,s2) ∧ equalC (c1,c2)

Operationen:

- **Erzeugen** einer neuen Datei („file“): *newF.*
- Testen, ob Datei **leer** ist: *isEmptyF.*
- **Anfügen** einer Zeichenkette an eine Datei: *addF.*
- **Einfügen** einer Zeichenkette an bestimmter Position einer Datei: *insertF.*
- **Hintereinanderhängen** zweier Dateien: *appendF*.

Beispiel: Texteditor

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

lenghtF: Text → Nat

equalF: Text, Text → Bool

addFC: Text, Char → Text

Operationen aus String implizit weiter verfügbar. Mit Suffix „S“ versehen um Verwechslung zu vermeiden.

Hilfsoperation für addF.

Beispiel: Texteditor

constraints *newF, isEmptyF, addF, appendF, insertF*
so that for all *[f, f_1, f_2: Text; s: String; c: Char; cursor: Nat]*

isEmptyF (newF()) = true

isEmptyF (addFC (f,c)) = false

addF (f, newS()) = f

addF (f, addS(s,c)) = addFC (addF(f,s), c)

lengthF (newF()) = 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)

…

Beispiel: Texteditor

…

equalF (newF(), newF()) = true

equalF (newF(), addFC(f,c)) = false

equalF (addFC(f,c), newF()) = false

equalF (addFC(f1,c1), addFC(f2,c2)) $= equalF(f_1,f_2) \wedge equalC(c_1,c_2)$

insertF (f, cursor, newS()) = f

$[(equalF\ (f, appendF(f_1,f_2)) \wedge (lengthF(f_1) = cursor))$

$\Rightarrow equalF\ (insertF\ (f, cursor+1, s), appendF\ (addF(f_1,s), f_2))]$

Beispiel: Texteditor

…

equalF (newF(), newF()) = true

equalF (newF(), addFC(f,c)) = false

equalF (addFC(f,c), newF()) = false

equalF (addFC(f1,c1), addFC(f2,c2)) = $equalF(f_1,f_2) \wedge equalC(c_1,c_2)$

insertF (f, cursor, newS()) = f (*)

$[(equalF\ (f, appendF(f_1,f_2)) \wedge (lengthF(f_1) = cursor))$

$\Rightarrow equalF\ (insertF\ (f, cursor+1, s), appendF\ (addF(f_1,s), f_2))]$

Frage: *insertF (newF(),0,s) =* ?

Beispiel: Texteditor

…

equalF (newF(), newF()) = true

equalF (newF(), addFC(f,c)) = false

equalF (addFC(f,c), newF()) = false

equalF (addFC(f1,c1), addFC(f2,c2)) = $equalF(f_1,f_2) \land equalC(c_1,c_2)$

insertF (f, cursor, newS()) = f (*)

$[(equalF\ (f,\ appendF(f_1,f_2)) \land (lengthF(f_1) = cursor))$

$\Rightarrow equalF\ (insertF\ (f,\ cursor+1,\ s),\ appendF\ (addF(f_1,s),\ f_2))]$

Frage: *insertF (newF(),0,s)* = **?**

Antwort:

- *insertF (newF(),0,newS()) = newF()*

Beispiel: Texteditor

…

equalF (newF(), newF()) = true

equalF (newF(), addFC(f,c)) = false

equalF (addFC(f,c), newF()) = false

equalF (addFC(f1,c1), addFC(f2,c2)) $= equalF(f_1,f_2) \wedge equalC(c_1,c_2)$

insertF (f, cursor, newS()) = f (*)

$[(equalF(f, appendF(f_1,f_2)) \wedge (lengthF(f_1) = cursor))$

$\Rightarrow equalF(insertF(f, cursor+1, s), appendF(addF(f_1,s), f_2))]$

Frage: *insertF (newF(),0,s) =* **?**

Antwort:

- *insertF (newF(),0,newS()) = newF()*
- *unbestimmt für s#newS():*
  für cursor = 0 gilt nur Gleichung (*)

Beispiel: Texteditor

…

equalF (newF(), newF()) = true

equalF (newF(), addFC(f,c)) = false

equalF (addFC(f,c), newF()) = false

equalF (addFC(f1,c1), addFC(f2,c2)) = equalF(f_1,f_2) ∧ equalC(c_1,c_2)

insertF (f, cursor, newS()) = f (*)

[(equalF (f, appendF(f_1,f_2)) ∧ (lengthF(f_1) = cursor))

⇒ equalF (insertF (f, cursor+1, s), appendF (addF(f_1,s), f_2))]

Frage: *insertF (newF(),1,s) =* ?

Beispiel: Texteditor

…

equalF (newF(), newF()) = true

equalF (newF(), addFC(f,c)) = false

equalF (addFC(f,c), newF()) = false

equalF (addFC(f1,c1), addFC(f2,c2)) = $equalF(f_1,f_2) \land equalC(c_1,c_2)$

insertF (f, cursor, newS()) = f (*)

$[(equalF\ (f, appendF(f_1,f_2)) \land (lengthF(f_1) = cursor))$

$\Rightarrow equalF\ (insertF\ (f, cursor+1, s), appendF\ (addF(f_1,s), f_2))]$

Antwort: *insertF (newF(),1,s)*

= appendF (addF(newF(),s), newF())

Beispiel: Texteditor

…

equalF (newF(), newF()) = true

equalF (newF(), addFC(f,c)) = false

equalF (addFC(f,c), newF()) = false

equalF (addFC(f1,c1), addFC(f2,c2)) = equalF$(f_1,f_2) \wedge$ *equalC*(c_1,c_2)

insertF (f, cursor, newS()) = f (*)

$[($*equalF (f, appendF*$(f_1,f_2)) \wedge ($*lengthF*(f_1) *= cursor*$))$

$\Rightarrow$ *equalF (insertF (f, cursor+1, s), appendF (addF*$(f_1$*,s),* $f_2))]$

Antwort: *insertF (newF(),1,s)*

= appendF (addF(newF(),s), newF())

= addF (newF(),s) *[appendF (f, newF()) = f]*

Beispiel: Texteditor

…

equalF (newF(), newF()) = true

equalF (newF(), addFC(f,c)) = false

equalF (addFC(f,c), newF()) = false

equalF (addFC(f1,c1), addFC(f2,c2)) = $equalF(f_1,f_2) \land equalC(c_1,c_2)$

insertF (f, cursor, newS()) = f (*)

$[(equalF(f, appendF(f_1,f_2)) \land (lengthF(f_1) = cursor))$

$\Rightarrow equalF(insertF(f, cursor+1, s), appendF(addF(f_1,s), f_2))]$

Antwort: *insertF (newF(),1,s)*

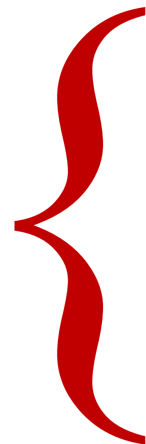
= appendF (addF(newF(),s), newF())

= addF (newF(),s) [appendF (f, newF()) = f]

= addFC (addF(newF(),s'),c) für s = addS (s',c)

= … [addF (f, addS(s,c)) = addFC (addF(f,s), c)]

3.3
Algebraische Spezifikation



- Das Spezifikationsproblem
- Vollständigkeit der Algebraischen Spezifikation
- **Signatur und Algebra**
- Algebraische Spezifikation: Diskussion

Definition „Algebraische Signatur“

$\Sigma = (S,F)$ heißt **algebraische Signatur**

- *S* eine Menge von **Sorten (Typen).**
- *F* eine Menge von **Operationssymbolen**.
- Auf *F* ist eine **Abbildung** definiert: $type: F \to S^* \times S$
  wobei $S^0=\{\emptyset\}$ und $S^*=\bigcup_{n\in\mathbb{N}} S^n$
- Für $type(f)=(s_1,\ldots,s_n,s)$ schreiben wir $f: s_1,\ldots,s_n \to s$.
- *type(f)* bezeichnet man auch als ***Signatur*** der Operation *f*.
- ***Konstanten:*** Abbildungen aus *F*, deren Vorbereich leere Menge ist.

Algebraische Signatur
Beispiel

Σ_{Test} = (S, F) mit

S = {*Nat*, *Bool*}

F = {*zero, one , succ, add, equal, equalBool, T, F*}

zero:		→ *Nat*	[d.h.type(zero)=(Ø,Nat)]
one:		→ *Nat*	
succ :	*Nat*	→ *Nat*	
add:	*Nat x Nat*	→ *Nat*	
equal :	*Nat x Nat*	→ *Bool*	[d.h. type(equal)=(Nat,Nat,Bool)]
equalBool :	*Bool x Bool*	→ *Bool*	
T :		→ *Bool*	
F:		→ *Bool*	

Algebren

Interpretiert eine Signatur Σ, indem sie Symbole in Σ mit **konkreten Mengen und Abbildungen „füllt“.**

Σ-Algebra (Definition):

- Sei $\Sigma = (S,F)$ eine Signatur.
- Für alle $s \in S$ sei A_s eine Menge.
- Für alle $f \in F$ mit $f: s_1,\ldots,s_n \rightarrow s$ sei $f_A: A_{s1} \times \ldots \times A_{sn} \rightarrow A_s$ eine Abbildung.
- Dann ist das Paar $A = ((A_s)_{s \in S}, (f_A)_{f \in F})$ eine Σ-Algebra.
- A_s bezeichnet man auch als **Trägermenge** von A zu s, f_A die Interpretation der Operation f.
- Beschreibung der Operation f in Algebra A umfasst ihre Signatur type(f) und ihre Semantik f_A.
- **Semantik von Operationen: Axiomatisch.**

Σ-Algebra
Beispiel

Signatur Σ_{Test}	Σ_{Test}-Algebra A
Nat	$A_{Nat} =_{def} \mathbb{N}$
Bool	$A_{Bool} =_{def} \{true, false\}$
zero: $\to$ Nat	$zero_A =_{def} 0$
one: $\to$ Nat	$one_A =_{def} 1$
succ : Nat $\to$ Nat	$\forall n \in \mathbb{N}.\ succ_A(n) =_{def} n + 1$
add : Nat x Nat $\to$ Nat	$\forall n,m \in \mathbb{N}.\ add_A(n,m) =_{def} n + m$
equal : Nat x Nat $\to$ Bool	$equal_A\ (zero_A, zero_A) =_{def} true$ $\forall n,m \in \mathbb{N}.\ equal_A(succ_A(n),succ_A(m)) =_{def} equal_A(n,m)$ $\forall n,m \in \mathbb{N}.\ equal_A\ (n,m) =_{def} equal_A\ (m,n)$ $\forall m \in \mathbb{N}.\ equal_A\ (zero_A, add_A\ (one_A,m)) =_{def} false$

Anmerkung: Axiome definieren eindeutig eine Funktion (müsste genaugenommen jeweils bewiesen werden).

Σ-Algebra
Beispiel (Fortsetzung)

Signatur Σ_{Test}	Σ_{Test}-Algebra A
equalBool : Bool x Bool → Bool	$equalBool_A$ (true,true) $=_{def}$ true $equalBool_A$ (false,false) $=_{def}$ true $equalBool_A$ (true, false) $=_{def}$ false $equalBool_A$ (false, true) $=_{def}$ false
T:→ Bool	$T_A =_{def}$ true
F:→ Bool	$F_A =_{def}$ false

Signatur und Algebra
Beispiele

- Beispiele mit genau **einer Trägermenge:**
  - Σ_{Nat}-Algebra B (A_{Nat} mit den relevanten Operationen und Axiomen aus Σ_{Test}).
  - Gruppen, Ringe, Körper, Verbände.
- Beispiele mit **mehreren Trägermengen:**
  - Σ_{Test}-Algebra A.
  - Vektorräume (Vektor und Skalare als Träger).

Teil 1: **Wichtiges Ziel:** Wiederverwendbarkeit von Software-Komponenten.

Frage: Wie können Ansätze, die auf algebraischer Spezifikation basieren, dieses Ziel unterstützen ?

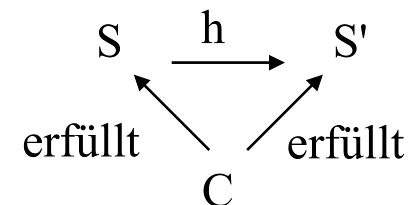
Teil 1: **Wichtiges Ziel:** Wiederverwendbarkeit von Software-Komponenten.

Frage: Wie können Ansätze, die auf algebraischer Spezifikation basieren, dieses Ziel unterstützen ?

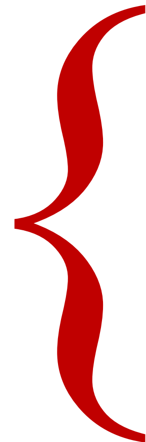
Antwort: Für gegebene algebraische Spezifikation S einer Software-Komponente kann ich eine Alt-Komponente C wiederverwenden, sofern sie Spezifikation S erfüllt.

Frage: Was tun, wenn Alt-Komponente C gegen (etwas) abweichenden algebraischen Spezifikation S' entwickelt wurde ?

Antwort: Definiere Abbildung h zwischen algebraischen Spezifikationen S und S', die wesentlichen gewünschten Eigenschaften bewahrt (genannt „Homomorphismus“ bzw. „Isomorphismus“).



3.3
Algebraische Spezifikation



Das Spezifikationsproblem
Vollständigkeit der Algebraischen Spezifikation
Signatur und Algebra
Algebraische Spezifikation: Diskussion

Algebraische Spezifikation: Übersicht

LEHRSTUHL 14
SOFTWARE ENGINEERING

Algebraische Signatur ∑

- Sorten S
- Operationssymbole F
  - Operationssymbole bilden abstrakt von Sorten auf Sorten ab
  - Konstanten bilden auf Sorten ab.
- ggf. Variablen X

Algebra

- Signatur ∑
- Mengen A
  - Konkrete Ausprägungen der Sorten
- Operationen f
  - konkrete Definition der Operationssymbole

Term t

- Verknüpfungen von Konstanten und Operationen (→ Grundterme) sowie ggf. Variablen (→ Allgemeine Terme)
- Entsprechend den Regeln der Algebra

Gleichung e

- Verknüpfung von zwei Termen
- Gültig, wenn beide Terme (ggf. für alle Variablenbelegungen) identisch ausgewertet werden

algebraische Spezifikation SP

- Signatur ∑ und wohldefinierte Formeln E (Gleichungen und ggf. logische Verknüpfungen)

Modell

- algebraische Spezifikation SP und Algebra A, mit „gültigen“ Gleichungen.

Diskussion: Anwendung von algebraischer Spezifikation

Algebraische Spezifikation:

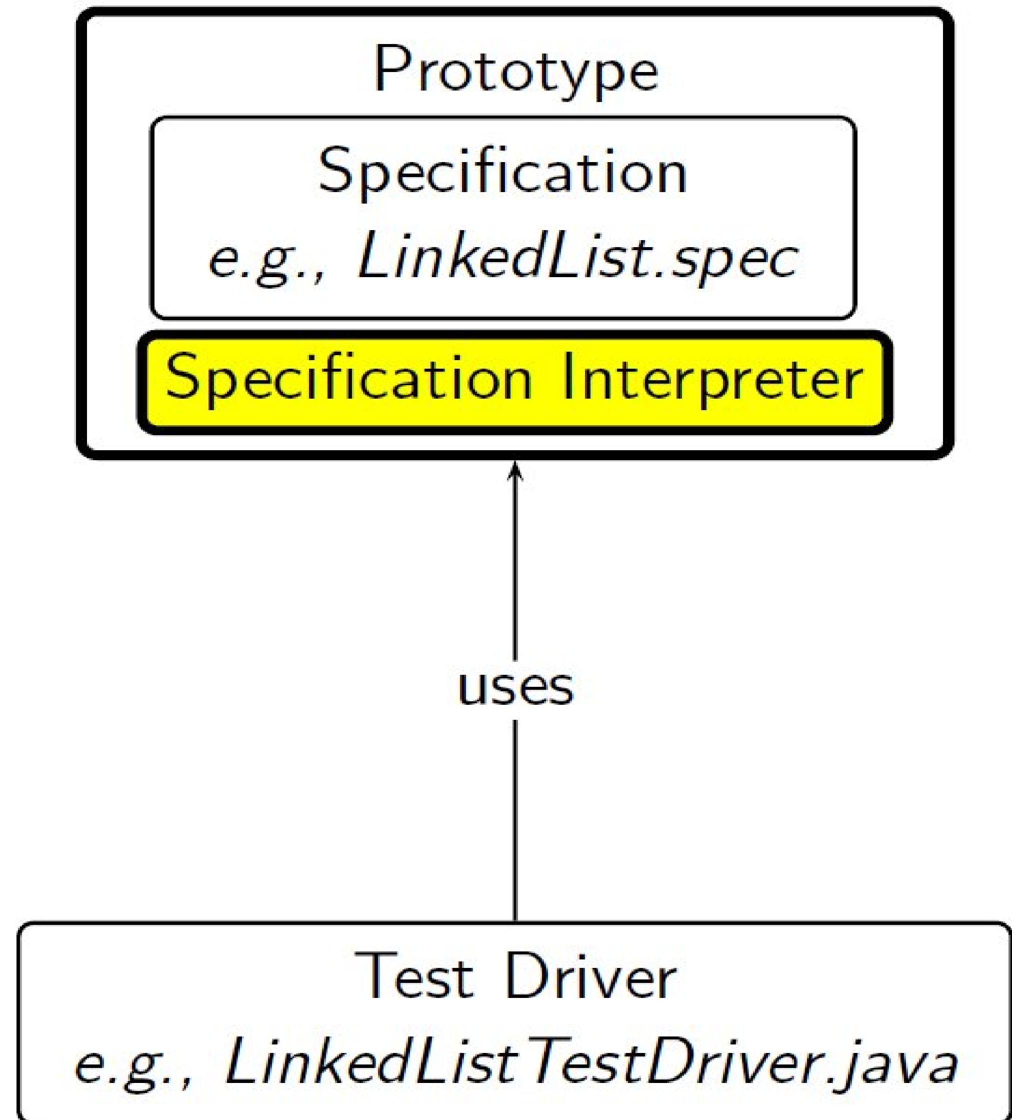
- **Verwendung in speziellen Einsatzgebieten** (z.B. Einsatz von Kryptographie in Software).
- **Grundlage für „benutzerfreundlichere“ Ansätze**, die allgemein verwendet werden (z.B. Object Constraint Language (OCL) im Rahmen der UML, oder Java / C assertions). [OCL: Teil 3.0 dieser Vorlesung]
- Grundlage für **allgemein verwendete Software-Entwicklungs-Prinzipien** wie „Design-by-contract“ (vgl. Programmiersprache Eiffel oder Java Markup Language (JML)) oder Software-Verifikationswerkzeuge wie VCC.
[http://research.microsoft.com/en-us/projects/vcc]

Anwendung: Werkzeug für algebraische Spezifikation von Java-Programmen

Johannes Henkel and Amer Diwan: A Tool for Writing and Debugging Algebraic Specications, International Conference on Software Engineering (ICSE) 2004

(http://www-plan.cs.colorado.edu/henkel/pubs.html)

Zum Beispiel für **spezifikations-basiertes Testen.**



Beispiel: Von Java zu algebraischer Spezifikation

LEHRSTUHL 14 SOFTWARE ENGINEERING

[HenDiw04]

```
class LinkedList {
        Object get(int index){...}
}
```

Algebraic Signature of get?

Retrieving the 5th element of LinkedList ℓ:

get(ℓ,5).state — updated state of ℓ after executing ℓ.get(5)
get(ℓ,5).retval — return value of ℓ.get(5)

Beispiel: Von Java zu algebraischer Spezifikation

[HenDiw04]

original client	algebraic term
`s = new IntStack();`	$state : NewIntStack().state$ $retval :$
`s.push(5);`	$state : push(NewIntStack().state, 5).state$ $retval :$
`s.push(7);`	$state : push(push(\ldots, 5).state, 7).state$ $retval :$
`i = s.pop();`	$state : pop(push(\ldots, 7).state).state$ $retval : pop(push(\ldots, 7).state).retval$

Weitere Anwendungen:
Algebraische Spezifikation und Testen

- **ASTOOT** [Doong, Frankl 1994]:
  http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.23.2625
- **Black/White** [Chen et al. 1998]:
  http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.541
- **TACCLE** [Chen, Tse, Chen 2001]:
  http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.2484
- **Korat** [Boyapati, Kurshid, Marinov 2002]:
  http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.122.9788

Zusammenfassung

- **In diesem Kapitel:**
  - Spezifikationsproblem.
  - Vollständigkeit der Algebra.
  - Signatur und Algebra.
  - Algebraische Spezifikation.
- **Betrachtung:**
  - algebraischer Spezifikation als **Grundlage für Constraint-Spezifikationsansätze** wie OCL.

    → Stellt **grundlegende Konzepte für Spezifikation** des Verhaltens einzelner Softwaremodule bereit.
  - einiger **Beispiele** für praktische Anwendungen dieses Ansatz.

Literatur

- Ehrig, Mahr: Fundamentals of Algebraic Specification 1: Equations and Initial Semantics, Springer Verlag, 1985
- Ehrig, Mahr: Fundamentals of Algebraic Specification 2: Module Specifications and Constraints, Springer Verlag, 1990
- Ehrig, Mahr, Cornelius et.al.: Mathematisch-strukturelle Grundlagen der Informatik, Springer Verlag, 1999
- Ehrich, Gogolla, Lipeck: Algebraische Spezifikation abstrakter Datentypen, Teubner Verlag, 1989
- Sommerville, Ian: Software Engineering, Addison-Wesley Longman, 2010. Kapitel 27: Formal Specification. http://www.cs.st-andrews.ac.uk/~ifs/Books/SE9/WebChapters/PDF/Ch_27_Formal_spec.pdf