Willkommen zur Vorlesung
Softwarekonstruktion
im Wintersemester 2011/2012

Prof. Dr. Jan Jürjens

TU Dortmund, Fakultät Informatik, Lehrstuhl XIV

04. Algebraische Spezifikation

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

Themen der Vorlesung

Kapitel 1: Intro mit Vorstellung der Professur und Gliederung der Vorlesung

Kapitel 2: Allgemeine Prinzipien des SW-Engineering

Kapitel 3: Spezifikation im Allgemeinen

Kapitel 4: Algebraische Spezifikation

Kapitel 5: Petrinetze

Kapitel 6: Modellgetriebene SW-Entwicklung

Kapitel 7: Object Constraint Language (OCL)

Kapitel 8: Testen im Allgemeinen, Kontrollflussorientierte Testverfahren, Datenflussorientierte Testverfahren

Agenda

Kap. 04: Algebraische Spezifikation

- Das Spezifikationsproblem
- Vollständigkeit der Algebraischen Spezifikation
- Signatur und Algebra
- Homomorphismen
- Terme
- Algebraische Spezifikation

Motivation

In diesem Kapitel beschäftigen wir uns mit der *algebraischen Spezifikation* als *Grundlage* für die *Spezifikation* des *Verhaltens einzelner Softwaremodule*.

Warum ?

- *Verhalten einzelner Softwaremodule*: Wie in Kap. 02 diskutiert, sind heutige Softwaresysteme von erheblicher Komplexität, die i.A. nur durch Verwendung modularer Herangehensweisen beherrscht werden kann.
- *Spezifikation*: Wie in Kap. 03 diskutiert, ist es daher wichtig, das Verhalten einzelner Softwaremodule spezifizieren zu können, um eine modulare Softwareentwicklung und Qualitätssicherung zu ermöglichen.
- *Grundlage*: In der Praxis gibt es verschiedene Ansätze, das Verhalten einzelner Softwaremodule zu spezifizieren, z.B. UML (insbes. Object Constraint Language OCL), Java / C assertions, Java Markup Language (JML). Um diese zu lernen, ist es effizienter, zunächst ihre gemeinsam zugrundeliegenden Konzepte zu lernen, und anschließend die Ansätze selber (z.B. OCL im Kap. 07).

In einigen Anwendungen kommt algebraische Spezifikation auch direkt zum Einsatz (Beispiel Verifikation von kryptographischen Protokollen: Spezifikation kryptographischer Eigenschaften als algebraische Gleichungen; vgl. später und MGSE).

Algebraische Spezifikation

- Algebraische Spezifikation ...
  - … betrachtet abstrakte Datentypen als algebraische Strukturen, d.h. als Mengen mit darauf definierten Operationen, für die bestimmte Axiome gelten,
  - … dient zur formalen Spezifikation des Verhaltens von Modulen / Modulschnittstellen (Unterschied zur Spezifikation des Verhaltens eines gesamten Software-Systems),
  - … dient zur Verifikation der Implementierung gegen die Spezifikation und der Verifikation der Spezifikation gegenüber den zu erfüllenden Anforderungen.
  - Durch das Zusammensetzen solcher Module kommt man zu Software-Systemen, deren Verhalten durch die Spezifikation der Module und durch die Konstruktion der Module zu einem Software-System definiert ist.
  - Die Konstruktion von Software-Systemen aus Modulen ist Gegenstand des Entwurfs.

Algebraische Spezifikation: Vorgehen

Nach [Som10] besteht der Prozess zur Erstellung einer vollständigen algebraischen Spezifikation aus den folgenden Schritten:

1) **Strukturieren der Spezifikation:** Zuerst müssen alle Schnittstellen eines Systems identifiziert werden. Das heißt also, es muss herausgefunden werden, welche Teile miteinander kommunizieren. Dazu wird informell beschrieben, welche Operationen notwendig sind.
2) **Benennen der Spezifikation:** Alle abstrakten Datentypen einer Spezifikation werden mit einem Namen versehen (z.B. Liste, Stack, Queue, Bool usw.) und es wird entschieden, ob sie generische Parameter benötigen.
3) **Auswahl der Operationen:** Hierbei wird die Bezeichnung aller Operationen festgelegt.
4) **Informelles Spezifizieren der Operation:** Die identifizierten Operationen sowie deren Auswirkungen auf das System werden beschrieben.
5) **Definieren der Syntax:** Für jede identifizierte Operation wird die Syntax formal beschrieben. Dazu werden alle Operationen zusammen mit ihren Parametern definiert. Die Syntax der gesamten Spezifikation ergibt sich aus der Zusammensetzung aller abstrakten Datentypen.
6) **Definieren der Semantik:** Die Semantik der Operationen wird definiert, indem beschrieben wird, welche Bedingungen (Axiome) für verschiedene Kombinationen von Operationen immer zutreffen müssen.

[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 einem Syntax- und einem Semantikteil:

- Syntax: Menge von Vereinbarungen von Zugriffsoperationen
  - Operationsname: Definitionsbereich → Wertebereich
- Semantik: Menge von Gleichungen, die Beziehungen zwischen den Eingabe- und Rückgabewerten der verschiedenen 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

Beispiel Syntax und Semantik: Kellerverwaltung

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

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).

Spezifikation des Datentyps String: Schritt 5

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 nur die Syntax fest, aber was bedeuten die Operationen?

- Die Semantik der Operationen wird durch **Gleichungen** (engl. Equations) beschreiben.
- Diese Gleichungen formulieren Eigenschaften, die von der Realisierung der Operationen nicht verletzt werden dürfen. Die Gleichungen heißen deshalb auch **Axiome**.
- Alle späteren Realisierungen der Operationen, die die Axiome nicht verletzen, sind spezifikationskonform ! „Lücken“ in den Axiomen (unvollständige Axiomatisierung) sind deshalb fast zwangsläufig die Ursache für spätere Missverständnisse.
- (Informelle) **Definition**: Wir bezeichnen eine algebraische Spezifikation als **unvollständig**, wenn sich nicht alle (intuitiv) als wahr angenommenen Aussagen auch anhand der Axiome ableiten lassen.

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

Semantik

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: 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).

Spezifikation des Datentyps String: Vollständigkeit der Axiome I

LEHRSTUHL 14
SOFTWARE ENGINEERING

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.

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 der Induktionsbeweis die o.g. Gleichung ?
Antwort: Die Strings, die durch new und add erzeugt werden.

Generierende Operationen: Definition

Der letzte Beweis basiert auf der Annahme, dass alle Strings durch die Operationen `new` und `add` erzeugt werden (denn nur für diese gilt der Beweis).

Zur Ermittlung der erforderlichen Axiome und zur Sicherstellung der Beweiskraft von Induktionsbeweisen wird eine erzeugende Menge von Operationen identifiziert.

Definition: Eine Menge *O* von Operationen heisst „**generierend**“ für eine Menge *X*, wenn alle Elemente in *X* durch sukzessive Anwendung der Operationen erzeugt werden können.

Bemerkung: Insbesondere kann die Menge *O* null-stellige Operationen (= Konstanten) enthalten.

Generierende Operationen: Beispiel

Frage:

- Was ist eine generierende Menge von Operationen der Booleschen Algebra ?

Antwort: zum Beispiel {false, not}

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

Antwort: zum Beispiel {0, succ}

Generierende Operationen: Algebraische Spezifikation

Im Rahmen der algebraischen 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 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 den Axiomen ableiten.

=> Die algebraische Spezifikation ist unvollständig, weil sich nicht alle (intuitiv) als wahr angenommenen Aussagen beweisen lassen.

Vervollständigung einer algebraischen Spezifikation

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 ?

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)~~

(usw...)

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

Drittes Beispiel: Texteditor

Operationen:

- Erzeugen einer neuen Datei („file“): `newF`
- Testen, ob eine Datei leer ist: `isEmptyF`
- Anfügen einer Zeichenkette an eine Datei: `addF`
- Einfügen einer Zeichenkette an einer bestimmten 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)) =

$$\text{equalF}(f_1,f_2) \wedge \text{equalC}(c_1,c_2)$$

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

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

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

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) = ?

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) $\wedge$ 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]

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

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

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: $type: F \rightarrow S^* \times S$
  wobei $S^0=\{\emptyset\}$ und $S^*=\bigcup_{n\in\mathbb{N}} S^n$
- Für $type(f)=(s_1,...,s_n,s)$ schreiben wir $f: s_1,...,s_n \rightarrow s$
- *type(f)* bezeichnet man auch als *Signatur* der Operation *f*.
- *Konstanten* sind die Abbildungen aus *F*, deren Vorbereich die leere Menge ist.

Algebraische Signatur: Beispiel

$\Sigma_{Test} = (S, F)$ mit

$S = \{Nat, Bool\}$

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

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

Algebren

Eine Algebra interpretiert eine Signatur Σ, indem sie die 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 $f: s_1,...,s_n \rightarrow s$ sei $f_A: A_{s1} \times ... \times A_{sn} \rightarrow A_s$ 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.
- Die Beschreibung der Operation f in der Algebra A umfasst ihre Signatur type(f) und ihre Semantik f_A.
- Die Semantik von Operationen wird axiomatisch definiert.

Σ-Algebra: Beispiel

Signatur Σ_{Test}	Σ_{Test}-Algebra A
Nat	$A_{Nat} =_{def} \mathbb{N}$
Bool	$A_{Bool} =_{def} \{true, false\}$
zero:$\rightarrow$ Nat	$zero_A =_{def} 0$
one: $\rightarrow$ Nat	$one_A =_{def} 1$
succ : Nat $\rightarrow$ Nat	$\forall n \in \mathbb{N}.\ succ_A(n) =_{def} n + 1$
add : Nat x Nat $\rightarrow$Nat	$\forall n,m \in \mathbb{N}.\ add_A(n,m) =_{def} n + m$
equal : Nat x Nat $\rightarrow$ 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: Die Axiome definieren jeweils eindeutig eine Funktion (müsste genaugenommen jeweils bewiesen werden).

Σ-Algebra: Beispiel (Fortsetzung)

Signatur Σ_{Test}	Σ_{Test}-Algebra A
equalBool : Bool x Bool $\rightarrow$ 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:$\rightarrow$ Bool	$T_A =_{def}$ true
F:$\rightarrow$ 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)

Wiederverwendbare Software

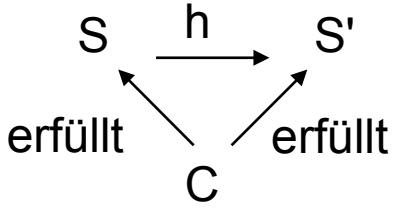
Wie wir in Kap. 2 gesehen haben, ist ein wichtiges Ziel ist die Wiederverwendbarkeit von Software-Komponenten.

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

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

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

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



Homo-/Isomorphismen

Homomorphie

- Homòs (gr.): gleichwertig
- Abbildung einer Algebra A in/auf eine Algebra B unter Bewahrung der Operationen von A.

Isomorphie

- Ìsos (gr.): gleich
- Gegenseitige Austauschbarkeit zweier Algebren unter Bewahrung der Operationen („Gleichheit bis auf Umbenennen / Umordnen“)

Homomorphismus: Definition

Sei $\Sigma = (S,F)$ eine Signatur; A, B zwei Σ-Algebren.

Ein Σ-Homomorphismus $h: A \rightarrow B$ ist eine Familie von Abbildungen $h = (h_s: A_s \rightarrow B_s)$ mit

- $h_s(c_A) = c_B$ für alle Konstanten c der Signatur
- $h_s(f_A(a_1,.., a_n)) = f_B(h_{s1}(a_1),..., h_{sn}(a_n))$ für alle Funktionssymbole f der Signatur und für alle Elemente a_i der Trägermengen des Definitionsbereichs von f_A

Homomorphismus: Beispiel

Σ-Algebra $A = (A_s, f_A)$

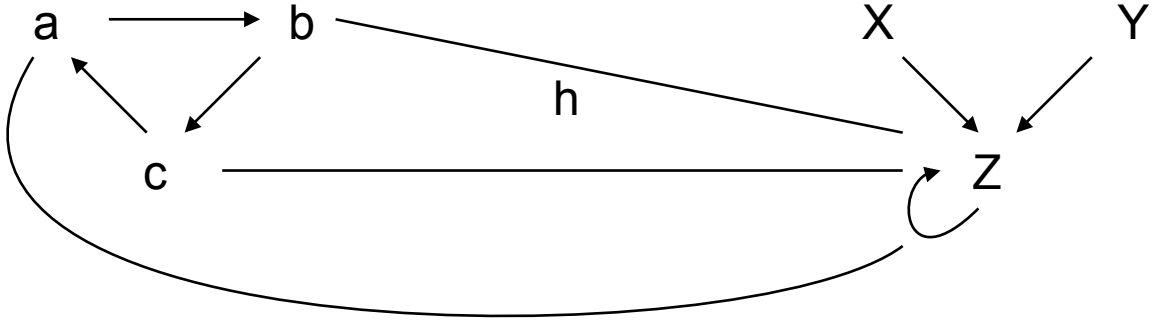
$A_s = \{a,b,c\}$

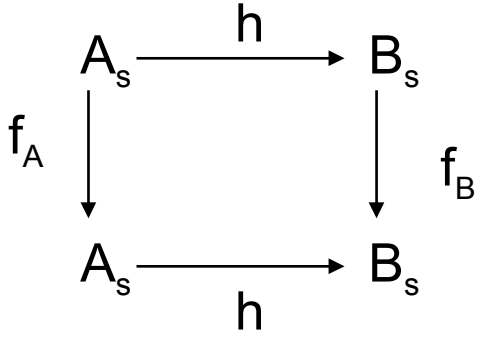
$f_A = \{a \to b, b \to c, c \to a\}$

Σ-Algebra $B = (B_s, f_B)$

$B_s = \{X,Y,Z\}$

$f_B = \{X \to Z, Y \to Z, Z \to Z\}$





h Homomorphismus-Abbildung

- Wird a auf X abgebildet, so muss auch das Bild von a (nämlich b) auf das Bild von X (nämlich Z) abgebildet werden, usw.
- Damit gibt es nur einen Homomorphismus, nämlich den, der a,b,c auf Z abbildet.

Homomorphismus: Beispiel Nat

Gegeben sei die Signatur $\Sigma_{Nat'} = (S, F)$ mit

- S = {Nat, Bool}
- F = {one, succ, add, T}

- A sei die bereits diskutierte Σ_{Test}-Algebra (die gleichzeitig eine $\Sigma_{Nat'}$-Algebra ist)
- A2 sei eine weitere $\Sigma_{Nat'}$-Algebra
  - Trägermenge: die geraden natürlichen Zahlen (also {0, 2, 4, 6, ...})
  - Die Konstantenfunktion one aus Σ_{Test} wird als Konstantenfunktion 2 interpretiert.

Homomorphismus: Beispiel Nat

Signatur $\Sigma_{Nat'}$	$\Sigma_{Nat'}$-Algebra A	$\Sigma_{Nat'}$-Algebra A2
Nat	$A_{Nat} =_{def} \mathbb{N}$	$A2_{Nat} =_{def} \{0,2,4,...\}$
Bool	$A_{Bool} =_{def} \{true, false\}$	$A2_{Bool} =_{def} \{true, false\}$
one: $\rightarrow$ Nat	$one_A =_{def} 1$	$one_{A-2} =_{def} 2$
succ : Nat $\rightarrow$ Nat	$succ_A (n) =_{def} n + 1$	$succ_{A-2} (n) =_{def} n + 2$
add : Nat x Nat $\rightarrow$ Nat	$add_A (n,m) =_{def} n + m$	$add_{A-2} (n,m) =_{def} n + m$
T: $\rightarrow$ Bool	$T_A =_{def}$ true	$T_{A-2} =_{def}$ true

A und A2 sind $\Sigma_{Nat'}$-homomorph, wenn ein $\Sigma_{Nat'}$-Homomorphismus existiert. Frage: Beispiel ? Was ist zu beweisen ?

$h_{Nat} : A_{Nat} \rightarrow A2_{Nat}$, $n \rightarrow 2n$

$h_{Bool}: A_{Bool} \rightarrow A2_{Bool}$, $b \rightarrow b$

Homomorphismus: Beispiel Nat

Ein Homomorphismus h: A $\rightarrow$ A2 besteht aus den Abbildungen $h_{Nat} : A_{Nat} \rightarrow A2_{Nat}$ und $h_{Bool} : A_{Bool} \rightarrow A2_{Bool}$, d.h. $h = (h_{Nat}, h_{Bool})$ mit

- $h_{Nat}\ (one_A) = one_{A\text{-}2}$ $\quad [\Leftrightarrow h_{Nat}\ (1) = 2]$
- $h_{Bool}\ (T_A) = T_{A\text{-}2}$ $\quad [\Leftrightarrow h_{Bool}\ (true) = true]$
- $\forall\ x,y \in A_{Nat} . h_{Nat}\ (add_A(x,y)) = add_{A\text{-}2}(h_{Nat}(x), h_{Nat}(y))$

  Beispielhaft einsetzen:
  - $x=1,y=2 : h_{Nat}\ (add_A(1,2)) = add_{A\text{-}2}(h_{Nat}(1), h_{Nat}(2))$

    $\Leftrightarrow\ h_{Nat}\ (1+ 2) = add_{A\text{-}2}(2, 4)$

    $\Leftrightarrow\ 2 * 3 = 2 + 4$

  $\Rightarrow$ Überprüfung für alle Konstanten und für alle Operationssymbole !

Homomorphismus: Beispiel Nat

Vollständige Darstellung des Homomorphismus h

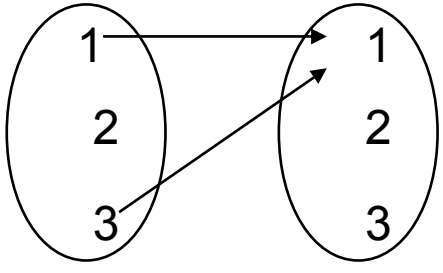
Signatur $\Sigma_{Nat'}$	$\Sigma_{Nat'}$-Algebra A	$\Sigma_{Nat'}$-Algebra A2	h: A $\rightarrow$ A2
Nat	$A_{Nat} =_{def} \mathbb{N}$	$A2_{Nat} =_{def} \{0,2,4,...\}$	h_{Nat}: {0$\rightarrow$0, 1$\rightarrow$2, 2$\rightarrow$4, ...}
Bool	$A_{Bool} =_{def}$ {true, false}	$A2_{Bool} =_{def}$ {true, false}	h_{Bool}: {true$\rightarrow$true, false$\rightarrow$false}
one: $\rightarrow$ Nat	$one_A =_{def} 1$	$one_{A-2} =_{def} 2$	$h_{Nat}(1) = 2$
succ: Nat $\rightarrow$ Nat	$succ_A(n) =_{def} n + 1$	$succ_{A-2}(n) =_{def} n + 2$	$h_{Nat} \circ succ_A = succ_{A-2} \circ h_{Nat}$
add: Nat x Nat $\rightarrow$ Nat	$add_A(n,m) =_{def} n + m$	$add_{A-2}(n,m) =_{def} n + m$	$h_{Nat} \circ add_A = add_{A-2} \circ h_{Nat}$
T:$\rightarrow$ Bool	$T_A =_{def}$ true	$T_{A-2} =_{def}$ true	h_{Bool} (true) = true

Spezielle Homomorphismen

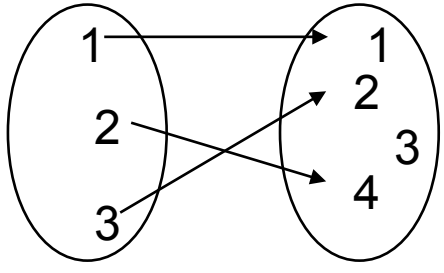
Definition: Sei $\Sigma = (S,F)$ eine Signatur.
Ein Σ-Homomorphismus $h: A \rightarrow B$ heißt

- injektiv, falls $\forall\, s \in S$: h_s injektiv
- surjektiv, falls $\forall\, s \in S$: h_s surjektiv
- bijektiv, falls $\forall\, s \in S$: h_s bijektiv

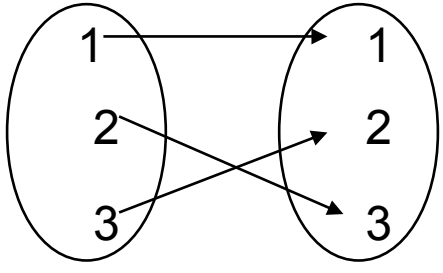
nicht injektiv,
nicht surjektiv



injektiv,
nicht surjektiv



bijektiv



Homomorphismen

Eigenschaften des untersuchten Homomorphismus h_{Nat} (mit h: n $\rightarrow$ 2n):

- Injektivität von h_{Nat} gegeben, denn h: n $\rightarrow$ 2n streng monoton
- Surjektivität von h_{Nat} gegeben, denn alle geraden Zahlen werden erreicht.
- Bijektivität von h_{Nat} somit auch gegeben.
  - Da beides auch für die Identität in Bool gilt, ist h= (h_{Nat}, h_{Bool}) ein bijektiver $\Sigma_{Nat'}$-Homomorphismus von der $\Sigma_{Nat'}$-Algebra A zur $\Sigma_{Nat'}$-Algebra A2.

Isomorphismen

Ein Σ-Isomorphismus ist ein bijektiver Σ-Homomorphismus (d.h. alle seine Funktionen sind bijektiv).

Idee des Isomorphie-Nachweises:

- Homomorphie zeigen
- Bijektivität aller Funktionen des Homomorphismus zeigen (eine pro Sorte), also Injektivität und Surjektivität dieser Funktionen

h_{Nat} ist somit ein $\Sigma_{Nat'}$-Isomorphismus von der $\Sigma_{Nat'}$-Algebra A zur $\Sigma_{Nat'}$-Algebra A2.

Terme: Überblick

Σ-Grundterm

- Signatur: Syntax
- Algebra: Semantik
- Terme: Wörter der Sprache
  - Beschreibt syntaktische Struktur auf Ebene der Signatur
  - Interpretation durch die jeweilige Algebra
    - z.B. succ(add(n1,n2))
  - Grundterme: Terme ohne Variablen
  - Terme: Terme mit (oder ohne) Variablen

Definition: Σ-Grundterm

Σ-Grundterm (Definition)

Sei $\Sigma = (S,F)$ eine Signatur.

Die Familie $T_\Sigma=(T(\Sigma,s))_{s \in S}$ der Mengen von Σ-Grundtermen

besteht für jedes $s \in S$ aus der kleinsten Teilmenge $T(\Sigma,s)$ von Wörtern über dem Alphabet $(F \cup \{„(“, „)“, „,“\})$, für die gilt:

Konstanten

- für jede Konstante $c: \rightarrow s$ aus der Menge F gilt $c \in T(\Sigma,s)$ und

Operations-symbole + zusammen-gesetzte Terme

- für jede Funktion $f: s_1...s_n \rightarrow s$ aus der Menge F und alle Terme $t_i \in T(\Sigma,s_i)$ (für alle $i \in \{1,...,n\}$) gilt $f(t_1,..,t_n) \in T(\Sigma,s)$.

Σ-Grundterm: Beispiel

Σ-Grundterm (Beispiel)

- Sei gegeben Σ_{Nat} = (S, F) aus vorherigem Beispiel
- $T(\Sigma_{Nat}, Nat)$
  - Stufe 0: Konstanten
  - Stufe 1: Operationen angewendet auf Stufe 0
  - Stufe 2: etc.
  - Stufe 3: ...
  - ....
- ...

Signatur $\Sigma_{Nat'}$
Nat
Bool
one: → Nat
succ: Nat → Nat
add: Nat x Nat → Nat T: → Bool

Σ-Grundterm: Beispiel

Σ-Grundterm (Beispiel)

- Sei gegeben Σ_{Nat} = (S, F) aus vorherigem Beispiel
- $T(\Sigma_{Nat}, Nat)$
  - Stufe 0: one
  - Stufe 1: succ(one), add(one,one)
  - Stufe 2: succ(succ(one)), succ(add(one,one)), add(succ(one),one), add(one,succ(one)), add(add(one,one),succ(one)),...
  - Stufe 3: succ(succ(succ(one))),...
  - ....
- Vollständige Beschreibung per Rekursion
  - $T(\Sigma_{Nat}, Nat) = \{one\} \cup \{succ(n) \mid n \in T(\Sigma_{Nat}, Nat)\}$
    $\cup \{add\ (n1,n2) \mid \{n1,n2\} \subseteq T(\Sigma_{Nat}, Nat)\}$

Signatur $\Sigma_{Nat'}$
Nat
Bool
one: $\rightarrow$ Nat
succ: Nat $\rightarrow$ Nat
add: Nat x Nat $\rightarrow$ Nat
T: $\rightarrow$ Bool

Auswertung von Σ-Grundtermen

Definition: Auswertung von Σ-Grundtermen

Sei $\Sigma = (S,F)$ eine Signatur und A eine Σ-Algebra.

Die Auswertung (Evaluation / Interpretation) eval(A)

der Terme zur Signatur Σ in A ist eine Familie von Abbildungen

- $eval(A) = (eval(A)_s : T(\Sigma,s) \rightarrow A_s)_{s \in S}$

Sie ist definiert für alle $s \in S$ durch:

Konstanten

- für jede Konstante c: $\rightarrow$ s aus der Menge F gilt:
  $eval(A)_s(c) = c_A$

Operations-symbole + zusammen-gesetzte Terme

- für jede Funktion f: $s_1 ... s_n \rightarrow s$ aus der Menge F und alle Terme $t_i \in T(\Sigma,s_i)$ (für alle $i \in \{1,...,n\}$) gilt:
  $eval(A)_s(f(t_1,..,t_n)) = f_A(eval(A)_{s1}(t_1),..., eval(A)_{sn}(t_n))$

Auswertung von Σ-Grundtermen: Beispiel

Beispiel für Auswertung von Σ-Grundtermen: Σ_{Nat}-Algebren A und A2

- $eval(A)_{Nat}$ (succ(add(one,one)))
  $= succ_A(eval(A)_{Nat}(add(one,one)))$ [$eval(A)_s(f(t_1,..,t_n)) = f_A(eval(A)_{s1}(t_1),..., eval(A)_{sn}(t_n))$]
  $= succ_A(add_A(eval(A)_{Nat}(one), eval(A)_{Nat}(one)))$ [„]
  $= succ_A(add_A(one_A, one_A))$ [$eval(A)_s(c) = c_A$]
  $= succ_A(add_A(1, 1))$
  $= succ_A(2)$
  $= 3$
- $eval(A2)_{Nat}$ (succ(add(one,one)))
  $= succ_{A2}(eval(A2)_{Nat}$ (add(one,one)))
  $= succ_{A2}(add_{A2}(eval(A2)_{Nat}(one), eval(A2)_{Nat}(one)))$
  $= succ_{A2}(add_{A2}(one_{A2}, one_{A2}))$
  $= succ_{A2}(add_{A2}(2, 2))$
  $= succ_{A2}(4)$
  $= 6$

Definition: Signatur mit Variablen

Signatur mit Variablen (Definition):

- Sei $\Sigma = (S,F)$ eine Signatur. Sei $X = (X_s)_{s \in S}$ eine Familie von Mengen, die durch S indiziert ist (dies wird als „S-sortiert“ oder „S-indiziert“ bezeichnet) und sodass für jedes $s \in S$ gilt: $X_s \cap F = \{\}$. Die Elemente der Mengen X_s werden dann als „Variablen“ bezeichnet.
- Dann heißt $\Sigma = (S,F,X)$ Signatur mit Variablen.

Definition: Allgemeine Terme

Sei $\Sigma = (S,F,X)$ eine Signatur mit Variablen und $X = (X_s)_{s \in S}$ eine Familie von Mengen. Die Familie $T_\Sigma(X)=(T(\Sigma,s)(X))_{s \in S}$ der Mengen von (allgemeinen) Σ-Termen besteht für jedes $s \in S$ aus der kleinsten Teilmenge $T(\Sigma,s)(X)$ von Wörtern über dem Alphabet $(F \cup X \cup \{„(“, „)“, „,“\})$, für die gilt:

Variablen

Konstanten

Operations-symbole + zusammen-gesetze Terme

- für jede Variable $x \in X_s$ gilt $x \in T(\Sigma,s)(X)$,
- für jede Konstante c: $\rightarrow$ s aus der Menge F gilt $c \in T(\Sigma,s)(X)$ und
- für jede Funktion f: $s_1...s_n \rightarrow$ s aus der Menge F und alle Terme $t_i \in T(\Sigma,s_i)(X)$ (für alle $i \in \{1,...,n\}$) gilt $f(t_1,..,t_n) \in T(\Sigma,s)(X)$.

Variablen spielen dieselbe syntaktische Rolle wie Konstantensymbole, unterscheiden sich jedoch in der Semantik:

- Konstanten haben in Σ-Algebra eine feste Bedeutung.
- Variablen müssen für eine Termauswertung erst belegt werden !

Auswertung von allgemeinen Σ-Termen

Definition: Auswertung von allgemeinen Σ-Termen:

Sei $\Sigma = (S,F,X)$ eine Signatur mit Variablen, A eine Σ-Algebra und ass: $X \rightarrow A$ eine Variablenbelegung („assignment“).
Die erweiterte Auswertung $xeval(ass):T_\Sigma(X) \rightarrow A$ der Terme zur Signatur Σ in A (bzgl. ass) ist eine Familie von Abbildungen

- $(xeval(ass)_s:T(\Sigma,s)(X) \rightarrow A_s)_{s \in S}$

Sie ist definiert für alle $s \in S$ durch:

Variablen
- für jede Variable $x \in X_s$ sei $xeval(ass)_s(x) =_{def} (ass)_s(x)$,

Konstanten
- für jede Konstante c: $\rightarrow$ s aus der Menge F sei $xeval(ass)_s(c) =_{def} c_A$ und

Operations-symbole + zusammen-gesetze Terme
- für jede Funktion f: $s_1 \ldots s_n \rightarrow$ s aus der Menge F und alle Terme $t_i \in T(\Sigma,s_i)(X)$ (für alle $i \in \{1,\ldots,n\}$) sei $xeval(ass)_s(f(t_1,..,t_n)) =_{def} f_A(xeval(ass)_{s1}(t_1),\ldots, eval(ass)_{sn}(t_n))$

Beispiel: Auswertung von allgemeinen Σ-Termen

Beispiel: Auswertung von allgemeinen Σ-Termen:

- Signatur Σ-Nat, Algebra A
- $X_{Nat} = \{n_1, n_2\}$
- Variablenbelegung $\nu: X \rightarrow A$
  - $\nu_{Nat}(n_1) = 1909$
  - $\nu_{Nat}(n_2) = 5$
- $xeval(\nu)_{Nat}\ (add(succ(0), succ(n_2))$ [$xeval(ass)_s(f(t_1,..,t_n)) =_{def} f_A(xeval(ass)_{s1}(t_1),..., eval(ass)_{sn}(t_n))$]

  $= add_A(xeval(\nu)_{Nat}(succ(0)), xeval(\nu)_{Nat}\ (succ(n_2)))$ […]

  $= add_A(succ_A(xeval(\nu)_{Nat}(0)), succ_A\ (xeval(\nu)_{Nat}\ (n_2)))$ [$xeval(ass)_s(c) =_{def} c_A$; $xeval(ass)_s\ (x) =_{def} (ass)_s\ (x)$]

  $= add_A(succ_A\ (0), succ_A(\nu_{Nat}\ (n_2)))$ [$\nu_{Nat}(n_2) = 5$]

  $= add_A(succ_A\ (0), succ_A(5))$ [Def. $succ_A$]

  $= add_A(1, 6)$ [Def. add_A]

  $= 7$

Algebraische Spezifikation

- Spezifikation (lat.): Auflistung
  - Auflistung syntaktisch formulierbarer Eigenschaften einer Algebra
- Signaturen repräsentieren die einfachste Form der Spezifikation
- Idee: Wie kann die Interpretation einer Signatur durch eine Algebra als Funktion dargestellt werden ?
  - Bildung von Formeln über den Termen einer Signatur
- Spezifikationen erweitern also Signaturen um Formeln, die mit Hilfe von Termen syntaktisch formuliert werden
  - Algebraische Gleichungslogik

Definition: ∑-Gleichung

Definition: ∑-Gleichung / Grundgleichung

- Sei A eine Algebra mit der Signatur ∑ =(S,F,X) und s ∈ S.
  Für t, t‘ ∈ T(Σ,s)(X) heißt

  $t =_s t'$ eine **∑-Gleichung**.

- $t =_s t'$ ist „gültig“ in einer Algebra A, falls für alle Variablenbelegungen die Auswertung der Terme identisch ist.
- $t =_s t'$ heißt **Grundgleichung**, falls weder t noch t‘ Variablen enthalten.

Beispiele:

- Signatur ∑-Nat, ∑-Algebra A
  - $eval(A)_{Nat}$ (zero) = $eval(A)_{Nat}$ (add(zero,zero)) gültig
  - $eval(A)_{Nat}$ (zero) = $eval(A)_{Nat}$ (one) nicht gültig
  - $xeval(ass)_{Nat}$ (add(n_1,n_2)) = $xeval(ass)_{Nat}$ (add(n_2,n_1)) gültig

Definition: Menge der wohldefinierten Formeln

Definition: Menge der wohldefinierten Formeln

Zu einer Algebra mit der Signatur Σ =(S,F,X) und einer S-sortierten Variablenmenge X wird die Menge der wohldefinierten Formeln über der Signatur Σ definiert als die Menge WFF(Σ), die die kleinste Menge mit den folgenden Eigenschaften ist:

- Jede Σ-Gleichung ist in WFF(Σ).
- Für alle Formeln G und H in WFF(Σ) sind auch die Formeln $\neg$G, (G $\wedge$ H) und (G $\vee$ H) enthalten in WFF(Σ).

Bemerkung: Man könnte auch die $\forall$- und $\exists$-Quantoren einbeziehen; zur Vereinfachung verzichten wir hier darauf.

Frage: Was ist mit G => H ?

Antwort: G => H ist äquivalent mit $\neg$G $\vee$ H.

Menge der wohldefinierten Formeln: Beispiel

Zur Signatur Σ_{Nat} enthält $WFF(\Sigma_{Nat})$ beispielsweise:

- jede Σ_{Nat}-Gleichung aus $T(\Sigma_{Nat},S)$, z.B. add(zero,n) = n
- Negationen, Disjunktion und Konjunktionen, z.B. $\neg$ (add(zero,n) = n)

Algebraische Spezifikation: Definition

Definition: Algebraische Spezifikation

Sei $\sum = (S,F,X)$ eine Signatur mit Variablen und E eine Menge von $\sum$-Formeln mit $E \subseteq WFF(\sum)$. Dann heißt $SP = <\sum,E>$ algebraische Spezifikation.

Beispiel:

Die algebraische Spezifikation Spec-Nat erweitert die Signatur $\sum_{Nat}$ um die $\sum_{Nat}$-Formeln (in diesem Fall alles Gleichungen):

- $equal_{Nat}(zero,zero) = true$
- $equal_{Nat}(succ_{Nat}(n_1), succ_{Nat}(n_2)) = equal_{Nat}(n_1,n_2)$
- $succ_{Nat}(n_1) = add_{Nat}(n_1,1)$
- $succ_{Nat}(zero) = one$

Definition: Modell

Sei SP=<∑,E> eine algebraische Spezifikation und A eine ∑-Algebra.

Dann heißt A Modell von SP (auch genannte eine „SP-Algebra“), wenn alle Formeln aus E in A erfüllt sind, d.h.:

- für jede Gleichung t = t‘ aus E und jede Variablenauswertung ass: X → A gilt: xeval(ass)(t) = xeval(ass)(t‘) und
- analog gelten alle Formeln ¬G, (G ∧ H) und (G ∨ H) aus E in A für jede Variablenauswertung ass: X → A.

Modelle sind also Algebren mit der gegebenen Signatur, die die gegebenen Formeln erfüllen.

Modell: Beispiel

Die bereits definierte Algebra A ist ein Modell der algebraischen Spezifikation Spec-Nat = (Σ_{Nat},E), wenn A alle Formeln aus E erfüllt.

Beispielhafte Überprüfung anhand des Beispielaxioms $equal_{Nat}(succ_{Nat}(n_1), succ_{Nat}(n_2)) = equal_{Nat}(n_1,n_2)$:

$$xeval(v)_{Nat}\left(equal\left(suc(n_1), suc(n_2)\right)\right) = xeval(v)_{Nat}\left(equal(n_1, n_2)\right)$$
$$\Leftrightarrow equal_A\left(xeval(v)_{Nat}\left(succ(n_1)\right), xeval(v)_{Nat}\left(succ(n_2)\right)\right)$$
$$= equal_A\left(xeval(v)_{Nat}(n_1), xeval(v)_{Nat}(n_2)\right)$$
$$\Leftrightarrow equal_A\left(succ_A\left(v_{Nat}(n_1)\right), succ_A\left(v_{Nat}(n_2)\right)\right) = equal_A\left(v_{Nat}(n_1), v_{Nat}(n_1)\right)$$

Letztere Gleichung gilt für jede Variablenauswertung ν_{Nat}, weil zwei natürliche Zahlen i, j genau dann gleich sind, wenn i+1=j+1.

Algebraische Spezifikation: Nicht-Formale 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.

Wünschenswerte Eigenschaften eines Modelles

Im allgemeinen kann eine algebraische Spezifikation mehrere Modelle haben. Besonders geeignet sind Modelle, die die folgenden beiden Eigenschaften erfüllen:

- Jedes Element des Modelles kann mittels Termen der Signatur bezeichnet werden (no-junk-Prinzip).
  - Eine Algebra, die Elemente in ihren Trägermengen hat, die wir mit Grundtermen nicht bezeichnen können, war mit der Signatur vermutlich nicht „gemeint“ (sonst wäre die Signatur entsprechend größer gewählt).
  - Alles was von der Signatur (mittels der Grundterme) nicht erfasst wird, wird als „junk“ bezeichnet.
- In dem Modell gelten keine Gleichungen, die nicht zwingend aus den Axiomen folgenden würden (no-confusion-Prinzip).
  - Das heißt, wir interpretieren die durch eine Signatur gegebene Spezifikation minimal. Sie beschreibt nichts „Überflüssiges“. Wäre eine Algebra „gemeint“ gewesen, in der weitere Gleichungen gelten, die nicht aus den Axiomen folgen, hätte diese in den Axiomen spezifiziert werden sollen.

Gute Nachricht: Für jede algebraische Spezifikation kann mittels der Grundterme und der gegebenen Axiome ein Modell konstruiert werden (das sogenannte „initiale Modell“), dass die o.g. Eigenschaften hat (lassen wir hier aus Zeitgründen aus).

Beispiel: Junk / confusion

Σ_{NatB} (ohne +)	Nat	INT	NATMOD3
Nat	$\mathbb{N}$	$\mathbb{Z}$	{0,1,2}
zero: → Nat succ: Nat → Nat	0 n → n+1	0 i → i+1	0 n → (n+1)mod3
(keine Axiome)	**No junk / confusion**	**Junk**	**Confusion**

- Die Standardalgebra Nat enthält weder „junk“ noch „confusion“.
- INT enthält „junk“, da nicht alle Elemente der Trägermenge $INT_{NAT} = Z$ von Grundtermen bezeichnet werden können.
  Alle negativen Zahlen bleiben syntaktisch unbenannt.
- NATMOD3 enthält keinen „junk“, ist aber „confus“, denn es gelten Gleichungen (z.B. zero() = succ(succ(succ(zero())))), die nicht aus den Axiomen folgen.

Widersprüchlichkeit und Redundanz

Definition: Eine algebraische Spezifikation ist **widersprüchlich**, wenn sich true = false beweisen lässt.

- **Beispiel**: Eine Spezifikation, die die Axiome c = c' und ¬(c = c') für zwei Konstanten c, c' enthält.

Definition: Eine algebraische Spezifikation ist redundant, wenn die Menge der beweisbaren Aussagen gleich bleibt auch dann, wenn ein Axiom gestrichen wird.

- **Beispiel**: Wenn die „String“-Algebra um das Axiom append (new(),s) = s erweitert wird, dann ist sie redundant, denn dieses Axiom ist ohnehin ableitbar und trägt somit nicht zu zusätzlichen beweisbaren Axiomen bei.

Beide Eigenschaften sind unerwünscht: Aus einer widersprüchlichen Spezifikation lässt sich jede Aussage ableiten (sie ist somit inhaltsleer), eine redundante Spezifikation verursacht unnötig viel Aufwand bei der Anwendung durch die überflüssigen Axiome.

Datenstrukturen vs. Modelle

Eine Datenstruktur besteht aus verschiedenen Datenbereichen und einer Menge von Operationen auf diesen Bereichen (z.B. INT, REAL, NAT, STRING, STACK, QUEUE).

Wie kann ich:

A) aus einer zu der Datenstruktur *D* gehörigen algebraischen Spezifikation *S* ein Modell *M* der Spezifikation *S* ableiten oder

B) zu einer gegebenen Algebra *A* entscheiden, ob sie ein „optimales“ Modell der Spezifikation *S* ist ?

A) Von Datenstruktur zum Modell

Von Datenstruktur *D* zur algebraischen Spezifikation *S*:

1. Aufstellung einer Signatur Σ
   - Sorten (Datenbereiche) und Operationen (Methoden, Funktionen)

Von algebraischer Spezifikation *S* zu Modell *M.* Aus der Signatur kann eine „Grundtermalgebra“ abgeleitet werden (initiale Semantik):

3. Formulieren von Σ-(Grund-)Termen
   - Ermöglichen die Bezeichnung der Elemente des Datenbereichs
     - Stellt unsere Signatur genügend syntaktische Ausdrücke bereit, um alle Elemente des Datenbereiches zu bezeichnen ? Wenn nein → Signatur erweitern.
   - Elemente der Grundtermalgebra sind die Grundterme; das Ergebnis der Anwendung einer Operation $f: s_1...s_n \rightarrow s$ auf die Werte $t_i \in T(\Sigma,s_i)$ (für $i \in \{1,...,n\}$) ist definiert als der entsprechende Term $f(t_1,..,t_n)$ in $T(\Sigma,s)$.

A) Von Datenstruktur zum Modell

3. Aufstellen von Gleichungen
   - Elemente der Datenbereiche lassen sich im Allgemeinen durch verschiedene Grundterme beschreiben (z.B. 2+1=1+2).
     - Alles, was nicht explizit als gleich definiert wird, wird verschieden interpretiert (wegen „no-confusion“-Prinzip).
     - Es muss definiert werden, welche Grundterme die gleichen Datenelemente bezeichnen.
   - Gesucht: eine Menge *E* von Gleichungen, um die Gleichheit von Elementen zu beschreiben,
     - die in der Datenstruktur gültig sein soll (Korrektheit von *E*) und
     - die Menge aller in der Datenstruktur gültigen Grundgleichungen generiert (Vollständigkeit von *E*)
     - und die idealerweise keine Redundanz enthält.

4. Ableiten des Modelles aus der Spezifikation
   - Aus der Spezifikation (Signatur + Gleichungen) kann eine „Quotienten-Termalgebra“ abgeleitet werden (initiale Semantik).
   - Elemente der Quotienten-Termalgebra sind die Mengen der Elemente in der Grundtermalgebra, die jeweils aufgrund der Menge *E* der Gleichungen als gleich definiert sind (die sogenannten Äquivalenzklassen).
   - Das Ergebnis der Anwendung einer Operation $f: s_1...s_n \rightarrow s$ auf die Äquivalenzklassen, die die Werte $t_i \in T(\Sigma,s_i)$ (für $i \in \{1,...,n\}$) enthalten, ist definiert als die Äquivalenzklasse, die den Term $f(t_1,..,t_n)$ enthält. (NB: Aus der Konstruktion einer Äquivalenzklasse folgt, dass diese Definition wohldefiniert ist.)

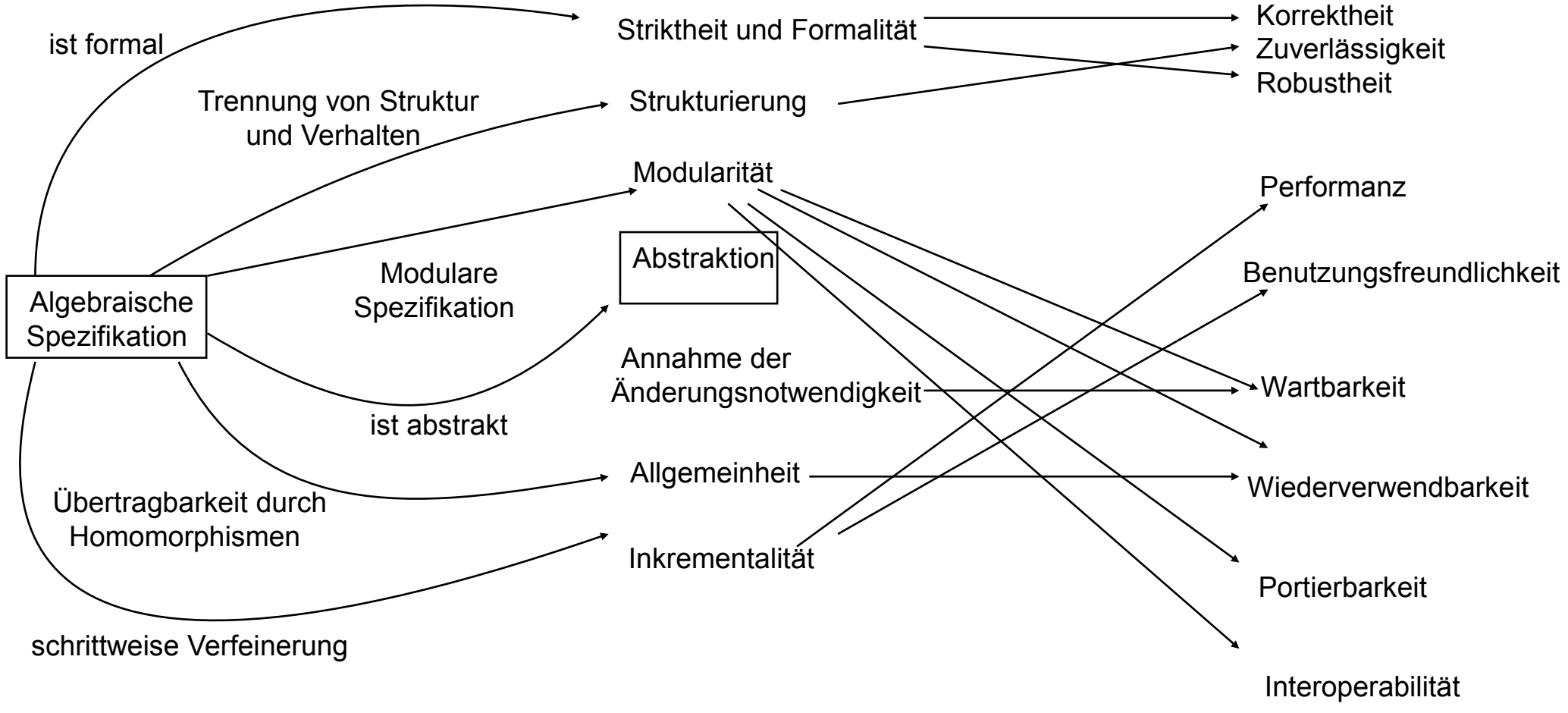
B) Gegebene Algebra als Modell

Wie stelle ich fest, ob eine gegebene Algebra A ein Modell der Spezifikation *S* ist (und zwar ein möglichst „optimales“, d.h. ohne „confusion“) ?

1. „No-junk“-Prinzip: Eine Algebra soll nur Elemente enthalten, die durch ∑-Terme bezeichnet werden können.
   - Lassen sich alle Elemente der Algebra *A* durch die Signatur bezeichnen?
     → evtl. *A* einschränken
2. Die Quotienten-Termalgebra ist idealerweise isomorph zu unserem Modell (Ausgangs-Algebra):
   - Wenn es einen surjektiven Homomorphismus von einer Quotienten-Termalgebra auf eine Algebra *X* gibt, dann erfüllt *X* die Spezifikation, aber erfüllt möglicherweise noch weitere Eigenschaften, die nicht aus der Spezifikation folgen („confusion“).
   - Wenn es einen solchen Homomorphismus gibt, der ausserdem injektiv ist, gibt es keine „confusion“.
   - Strukturelle Induktion als Werkzeug für den Beweis der Existenz einer solchen Abbildung.

Bezug zu Kap. 2: Prinzipien des SWE

Inwieweit werden die in Kap. 2 diskutierten Prinzipien durch Ansätze, die auf algebraischen Spezifikationen basieren, unterstützt ?



Diskussion: algebraische Spezifikation

Nachteile der algebraischen Spezifikation

- Konstruktion algebraischer Spezifikationen nicht trivial
- Fälle wie z.B. Grenzbedingungen können leicht übersehen werden
- Schwierig auf Konsistenz und Vollständigkeit zu prüfen
- Operationen mit mehreren Wertebereichen sind schwierig zu beschreiben

Diskussion: algebraische Spezifikation

Wegen der o.g. Nachteile hat sich die algebraische Spezifikation in der Praxis bislang nicht weitgehend durchgesetzt.

Aber:

- Wird in speziellen Einsatzgebieten (z.B. Einsatz von Kryptographie in Software) verwendet, wo sich der Aufwand lohnt.
  [vgl. Vorlesung MGSE im SS 2012]
- Bildet Grundlage für „benutzerfreundlichere“ Ansätze, die allgemein verwendet werden (z.B. Object Constraint Language (OCL) im Rahmen der UML, oder Java / C assertions). [OCL: Kap. 7 dieser Vorlesung]

Bildet Grundlagen 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]. [JML: nächste Woche]

Anwendung I: Einsatz von Kryptographie in Software

Exp: Menge der *Krypto-Terme*, die aus den Symbolen in den Mengen *Data*, *Keys*, *Var* gebildet werden unter Verwendung der Operationen:

- *_::_* (Konkatenation), *head(_)*, *tail(_)*,
- *(_)$^{-1}$* (K^{-1}: zum Verschlüsselungsschlüssel K gehöriger Entschlüsselungsschlüssel)
- *{ _ }_* ($\{M\}_K$: Verschlüsselung der Nachricht M mit Schlüssel *K*)
- *Dec_( )* ($Dec_K(C)$: Entschlüsselung der Daten C mit dem Schlüssel *K*)
- *Sign_( )* ($Sign_K(M)$: Signatur der Nachricht M mit dem Schlüssel *K*)
- *Ext_( )* ($Ext_K(S)$: Extrahieren der Signatur S mit dem Schlüssel *K*)

Anwendung I: Einsatz von Kryptographie in Software

… unter Berücksichtigung der folgenden Gleichungen:

- $\forall E,K.\ Dec_K^{-1}(\{E\}_K)=E$
- $\forall E,K.\ Ext_K(Sign_K^{-1}(E))=E$
- $\forall E_1,E_2.\ head(E_1::E_2)=E_1$
- $\forall E_1,E_2.\ tail(E_1::E_2)=E_2$
- Assoziativität für $::$.

Zur besseren Lesbarkeit, schreibe $E_1::E_2::E_3$ für $E_1::(E_2::E_3)$ und $fst(E_1::E_2)$ für $head(E_1::E_2)$ etc.

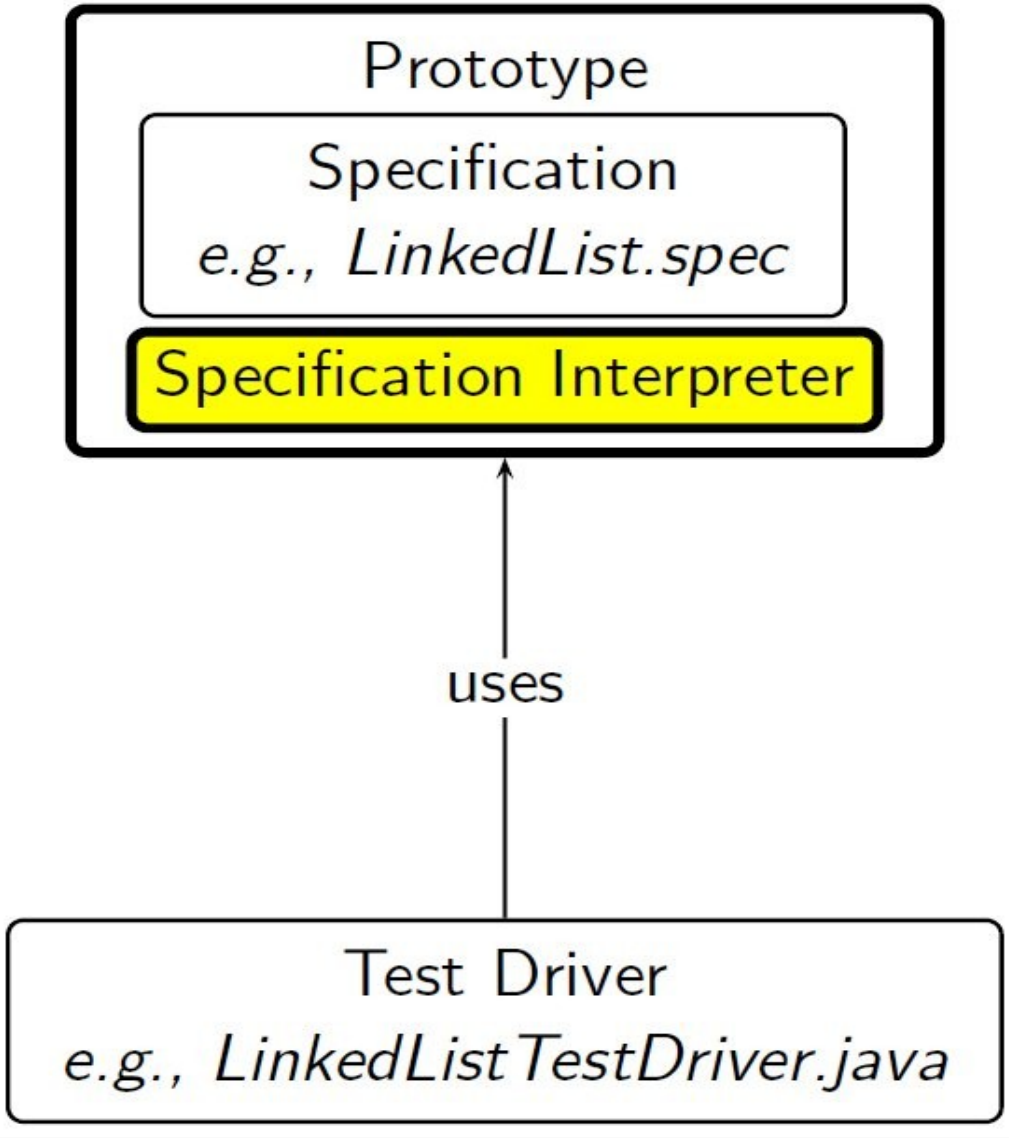
[NB: Die o.g. Gleichungen können bei Bedarf um weitere krypto-spezifische Eigenschaften erweitert werden (z.B. bei XOR).]

Anwendung II: 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

[HenDiw04]

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

Algebraic Signature of get?

$$LinkedList \times int \rightarrow LinkedList \times Object$$

Receiver object
Parameter
Return value
Updated receiver object

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

LEHRSTUHL 14
SOFTWARE ENGINEERING

[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*(. . ., 5).*state*, 7).*state* *retval* :
`i = s.pop();`	*state* : *pop*(*push*(. . ., 7).*state*).*state* *retval* : *pop*(*push*(. . ., 7).*state*).*retval*

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 haben wir folgende Themen behandelt:

- Das Spezifikationsproblem
- Vollständigkeit der Algebra
- Signatur und Algebra
- Homomorphismen
- Terme
- Algebraische Spezifikation

Wir haben somit algebraische Spezifikation als ein Beispiel für einen Ansatz für das in Kap. 3 behandelte Spezifikationsproblem betrachtet, das insbesondere grundlegende Konzepte für die Spezifikation des Verhaltens einzelner Softwaremodule bereitstellt. Wir haben außerdem kurz einige Beispiele für praktische Anwendungen dieses Ansatz betrachtet.

Ausblick

Nach einem kurzen Exkurs in eine weitere Anwendung dieser Konzepte (Java Modeling Language (JML), s. nächste Woche) werden wir uns im Kap. 5 mit einem zweiten Beispiel für einen allgemeinen Spezifikationsansatz beschäftigen (Petri-Netze als Grundlage für die Spezifikation der Interaktion von Softwaremodulen mit anderen Modulen bzw. der Umgebung), um uns anschließend eingehender mit der modell-basierten Entwicklung mit UML zu beschäftigen.

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