Kurze Einführung in Alloy

von Burkhardt Renz und Nils Asmussen

Was ist Alloy?

Alloy[1] ist eine Sprache, in der man mit den Mitteln der Prädikatenlogik Eigenschaften eines Modells definieren kann. Dabei gibt man nicht ein konkretes Modell vor, sondern spezifiziert nur die gewünschten Eigenschaften, die ein Modell haben soll, das der Spezifikation entspricht.

Der *Alloy Analyzer* arbeitet als *model finder*, d.h. er konstruiert ein Modell, das dieser Spezifikation entspricht, sofern dies möglich ist. Es wird also die Spezifikation *simuliert*. Der Alloy Analyzer findet endliche Modelle, indem er stets innerhalb eines endlichen Universums agiert. Das Universum ist in der Regel in paarweise disjunkte Teilmengen, die *Typen* in Alloy, zerlegt und für jeden dieser Typen kann man angeben, wieviele Elemente ein Modell maximal haben soll.

In der Softwaretechnik wendet man Alloy also so an, dass man die grundlegenden Abstraktionen einer Software in Alloy formuliert; man bildet ein *Mikromodell* der Software, besser zu nennen: eine *logische Spezifikation*.

Nun kann man mit Alloy *Welten simulieren*, die dieser Spezifikation entsprechen – und wird wahrscheinlich einige Überraschungen erleben. Die Arbeit mit Alloy zeigt nämlich, dass man häufig nicht wirklich präzise spezifiziert hat, was man meint. Dies führt dann dazu, dass der Alloy Analyzer *Welten* produziert, die man nicht erwartet hätte.

Man kann auch gewisse Eigenschaften formulieren, die man auf der Basis der Spezifikation erwarten würde. Der Alloy Analyzer versucht dann ein Gegenbeispiel zu finden; man verwendet so Alloy zur *Verifikation*. Wenn Alloy kein Gegenbeispiel findet, bedeutet das nicht, dass es keines gibt, weil die Prädikatenlogik nicht entscheidbar ist und Alloy ja nur beschränkte Universen überprüft. Daniel Jackson, der Alloy entwickelt hat, geht von der *small scope hypothesis* aus, die besagt, dass Fehler oft kleine Gegenbeispiele haben. Und in der Tat macht ja jeder Softwareentwickler die Erfahrung, dass ein einmal erkannter Fehler in einer Software meist schon an kleinen Beispielen nachvollziehbar ist. Und dies gilt auch (und insbesondere) für Designfehler.

Die folgende kurze Einführung in Alloy referiert aus dem Buch *Software Abstractions – Logic, Language, and Analysis* von Daniel Jackson (MIT Press, 2012) und verwendet Beispiele aus dem Tutorial zu Alloy 4 von

[1] Alloy bedeutet auf deutsch *Legierung*. Der Name kommt wohl dadurch zustande, dass Daniel Jackson diese Sprache und ihre Verwendung durch die Kombination zweier Konzepte entwickelt hat: der Spezifikationssprache *Z* einerseits und der Verifikation von Software durch *Model Checking* andererseits.

Greg Dennis und Rob Seater, zu finden unter http://alloy.lcs.mit.edu/alloy/tutorials/online/.

Die Logik von Alloy

Atome und Relationen

Die „Dinge“ oder „Entitäten“ oder „Objekte“ in Alloy sind *Elemente* eines Universums, die in der Logik atomar sind, d.h. unveränderliche Elemente des Universums.

In der Regel gibt man in einem Alloy-Modell das Universum *nicht* vor, sondern definiert die Eigenschaften der Struktur – Alloy erstellt dann ein Universum mit n Elementen, das diese Eigenschaften hat oder ein Gegenbeispiel, das zeigt, dass es keine Struktur mit n Elementen geben kann, die diese Eigenschaften hat.

Verwendet man Alloy dazu, objektorientierte Systeme zu modellieren und zu analysieren, muss man beachten, dass die Elemente in Alloy, die man als „Objekte“ auffasst, tatsächlich Elemente einer Menge im mathematischen Sinne sind, also unveränderbar. Will man etwa Veränderungen durch Funktionen an solchen „Objekten“ modellieren, muss man demzufolge eine Dimension „Zustand“ in das Modell einführen.

In Alloy gibt es *Relationen* über dem Universum – und nur solche, d.h. jeder Wert in Alloy ist eine Relation. Es ist Daniel Jackson bei der Konstruktion der Logik und der Sprache von Alloy ein wichtiges Anliegen, dass die Konstrukte möglichst einheitlich sind, dass Alloy dem Prinzip der *konzeptionellen Integrität* folgt.

In folgenden Beispielen werden Tupel mit runden Klammern geschrieben, Relationen, also Mengen von Tupeln, mit geschweiften Klammern.

Skalare sind also in Alloy unäre Relationen mit einem Element.
Beispiele:

```
myName   = {(N1)}
yourName = {(N2)}
myBook   = {(B0)}
```

Mengen sind in Alloy unäre Relationen.
Beispiele:

```
Name = {(N0), (N1), (N2)}
Addr = {(A0), (A1), (A2)}
Book = {(B0), (B1)}
```

Relationen stellen Beziehungen zwischen Atomen dar. Es handelt sich um Mengen von geordneten Tupeln gleicher Länge m mit Werten aus dem Universum. Der *Grad* der Relation, auch die Arität genannt, ist die Zahl der Werte in ihren Tupeln, also m.

Beispiele:

```
names = {(B0, N0), (B0, N1), (B1, N2)}
addrs = {(B0, N0, A0),
         (B0, N1, A1),
         (B1, N1, A2),
         (B1, N2, A2)}
```

In diesen Beispielen ist `names` eine binäre Relation, die angibt, welche Namen in welchem Adressbuch verzeichnet sind, und `addrs` ist eine ternäre Relation, die angibt, welche Adressen zu den Namen in den Adressbüchern verzeichnet sind.

Man kann sich jede Relation auch als ein Prädikat denken: für alle Tupel in der Relation ist das Prädikat `true`, für alle anderen `false`.

Die obigen Beispiele haben die Eigenschaft, dass die Relationen über disjunkte Teilmengen des Universums gebildet werden: die Namen, die Bücher, die Adressen. Dies ist ein typisches Strukturierungsmerkmal für die Spezifikationen der Modelle, die wir in Alloy formulieren. *Signaturen* (siehe unten) zerlegen das Universum in paarweise disjunkte Teilmengen von Elementen.

Konstanten

In Alloy gibt es folgende vorgegebene Konstanten:

- `none`, die leere Menge von Tupeln
- `univ`, das Universum
- `iden`, die Identitätsrelation

Angenommen wir haben zwei disjunkte Teilmengen `Name` und `Addr` in unserem Universum; dann haben die Konstanten folgende Werte:

```
Name = {(N0), (N1), (N2)}
Addr = {(A0), (A1)}

none = {}
univ = {(N0), (N1), (N2), (A0), (A1)}
iden = {(N0, N0), (N1, N1), (N2, N2), (A0, A0), (A1, A1)}
```

Mengenoperatoren

Mengenoperatoren kann man anwenden auf Relationen desselben Grades. Alloy hat folgende Mengenoperatoren:

- `+`, Vereinigung
- `&`, Schnittmenge
- `-`, Differenz
- `in`, Teilmenge
- `=`, Gleichheit

Genau genommen sind `in` und `=` Prädikate, die wir zu den definierten Symbolen der Logik hinzunehmen.

Beispiel:

```
Name  = {(N0), (N1), (N2)}
Alias = {(N1), (N2)}
Group = {(N0)}
RecentlyUsed = {(N0), (N2)}

Alias + Group = {(N0), (N1), (N2)}
Alias & RecentlyUsed = {(N2)}
Name - RecentlyUsed = {(N1)}
RecentlyUsed in Alias = false
RecentlyUsed in Name = true
Name = Group + Alias = true
```

Produkt und Komposition

Das kartesische Produkt zweier Relationen wird in Alloy mit dem Operator `->` geschrieben.

Beispiel:

```
Name = {(N0), (N1)}
Addr = {(A0), (A1)}
Book = {(B0)}

Name->Addr = {(N0, A0), (N0, A1),
              (N1, A0), (N1, A1)}
Book->Name->Addr =
             {(B0, N0, A0), (B0, N0, A1),
              (B0, N1, A0), (B0, N1, A1)}
```

Alloy kennt neben dem kartesischen Produkt die relationale Komposition, die man als eine spezielle Form des Verbunds, den *dot join*, auffassen kann:

Sind R und S Relationen des Grades n bzw. m (beide > 1), dann ist

$$R.S = \{(r_1, r_2, \ldots, r_{n-1}, s_2, s_3, \ldots, s_m)/ \\ (r_1, \ldots, r_n) \in R \text{ und } (s_1, \ldots, s_m) \in S \text{ mit } r_n = s_1\}$$

Wenn R und S rechtseindeutige binäre Relationen, also Funktionen, sind, dann ist $R.S$ gerade die Komposition der Funktionen, daher die Notation in Alloy.

Beispiel:

```
Book = {(B0)}
Name = {(N0), (N1), (N2)}
Addr = {(A0), (A1), (A2)}

addrs = {(B0, N0, A0), (B0, N1, A0), (B0, N2, A2)}
```

```
Book.addrs = {(N0, A0), (N1, A0), (N2, A2)}
```

In Alloy kann man den *dot join* auch syntaktisch auf eine weitere Weise darstellen; man spricht dann vom *box join*:

`R.S = S[R]` , `a.b.c[d] = d.(a.b.c)`

Beispiel: (Relationen wie oben)

```
myName = {(N1)}

Book.addrs = {(N0, A0), (N1, A0), (N2, A2)}
Book.addrs[myName] = {(A0)}
```

Alloy bietet beide Notationen für die Komposition an, damit man Ausdrücke bilden kann, die aussehen wie in objektorientierten Sprachen. In obigem Beispiel können wir uns vorstellen, dass `Book` für eine Klasse steht, die ein Feld `addrs` hat, das ein durch Namen indiziertes assoziatives Array ist.

`Book.addrs[myName]` kann man dann so lesen, dass man vom Objekt `{(B0)}` zunächst das Objekt `addrs` erhält und dann am Index `myName` die Adresse. Die *dot*-Notation steht dann für den Punkt-Operator mit dem man auf Felder eines Objekts zugreift und die *box*-Notation für den indizierten Zugriff in einem Array.

Operatoren für binäre Relationen

Für binäre Relationen gibt es drei unäre Operatoren:

- `~`, die Transposition,
- `^`, den transitiven Abschluss und
- `*`, den reflexiven transitiven Abschluss.

Die Transposition `~R` einer Relation `R` ist die Relation, die durch das Vertauschen der Reihenfolge der Atome in allen Tupeln entsteht.

Eine binäre Relation R ist transitiv, wenn aus $(r, s) \in R$ und $(s, t) \in R$ stets folgt, dass $(r, t) \in R$ gilt. Der transitive Abschluss einer binären Relation R ist die kleinste Relation, die R enthält und transitiv ist.

Eine binäre Relation R ist reflexiv, wenn für jedes Atom r das Tupel (r, r) in R ist. Der reflexive transitive Abschluss einer binären Relation R ist die kleinste Relation, die R enthält und reflexiv und transitiv ist.

Beispiel:

```
Node = {(N0), (N1), (N2), (N3)}
next = {(N0, N1), (N1, N2), (N2, N3)}

~next = {(N1, N0), (N2, N1), (N3, N2)}
^next = {(N0, N1), (N0, N2), (N0, N3),
         (N1, N2), (N1, N3), (N2, N3)}
*next = {(N0, N0), (N0, N1), (N0, N2), (N0, N3),
         (N1, N1), (N1, N2), (N1, N3),
         (N2, N2), (N2, N3), (N3, N3)}
```

Restriktion und Überschreiben

Die Definitionsmenge (*domain*) einer Relation ist die Menge der Atome der ersten Spalte, die Zielmenge (*range*) der Relation die Menge der Atome der letzten Spalte. Diese Sprechweise verallgemeinert diese Begriffe, wie man sie von Funktionen kennt.

Relational ausgedrückt ist die Definitionsmenge die Projektion auf die erste Spalte, die Zielmenge die Projektion auf die letzte Spalte.

Die Restriktionsoperatoren in Alloy werden verwendet, um Relationen auf eine bestimmte Definitionsmenge oder Zielmenge einzuschränken:

Ist `s` eine Menge und `r` eine Relation, dann ist `s <: r` die Menge der Tupel in `r`, die mit einem Element aus `s` beginnen. Und `r :> s` ist die Menge der Tupel in `r`, die mit einem Element aus `s` enden.

Beispiel: (`addrs` enthält Namen zugeordnet zu Adressen, aber auch zu anderen Namen, also ein Adressbuch mit Aliasen und indirekter Zuordnung der Adressen)

```
Name = {(N0), (N1), (N2)}
Alias = {(N0), (N1)}
Addr = {(A0)}
addrs = {(N0, N1), (N1, N2), (N2, A0)}

Alias <: addrs = {(N0, N1), (N1, N2)}
addrs :> Alias = {(N0, N1)}
addrs :> Addr  = {(N2, A0)}
```

Der Operator `++` (*override*) ist eine spezielle Form der Vereinigung, bei der aber Elemente, die an der ersten Spalte identisch sind, ersetzt werden. Man kann den Operator folgendermaßen definieren:

```
p ++ q = p - (domain[q] <: p) + q
```

Beispiel:

```
homeAddrs = {(N0, A1), (N1, A2), (N2, A3)}
workAddrs = {(N0, A0), (N1, A4)}

homeAddrs ++ workAddrs =
              {(N0, A0), (N1,A4), (N2, A3)}
```

Logische Operatoren

Alloy kennt folgende logischen Operatoren, die in der Syntax jeweils in einer Langform und einer Kurzform möglich sind:

- `not` bzw. `!`, Negation
- `and` bzw. `&&`, Konjunktion
- `or`, bzw. `||`, Disjunktion
- `implies`, bzw. `=>`, Implikation

- `else`, Alternative
- `iff`, bzw. `<=>`, Äquivalenz

Quantoren

Man kann in Alloy Quantoren für Variablen einsetzen. Z.B. bedeutet `all x: e | F`, dass die Formel `F` für alle `x` in `e` gilt.

Es gibt in Alloy folgende Quantoren:

- `all x : e | F`, `F` gilt für alle `x` in `e`
- `some x : e | F`, `F` gilt für mindestens ein `x` in `e`
- `no x : e | F`, `F` gilt für kein `x` in `e`
- `lone x : e | F`, `F` gilt für höchstens ein `x` in `e`
- `one x : e | F`, `F` gilt für genau ein `x` in `e`

Man kann die Quantoren auch direkt auf Ausdrücke anwenden:

- `some e`, `e` hat mindestens ein Tupel
- `no e`, `e` hat kein Tupel
- `lone e`, `e` hat höchstens ein Tupel
- `one e`, `e` hat genau ein Tupel

Deklarationen

Mengen definiert man in Alloy mit `set`, `one`, `lone` oder `some`.

Beispiele:

- `RecentlyUsed: set Name`
  RecentlyUsed ist eine Teilmenge der Menge Name
- `senderAddress: Addr`
  senderAddress ist ein Singleton (eine einelementige Teilmenge) von Addr (Wird keine Mächtigkeit angegeben, gilt automatisch `one`)
- `senderName: lone name`
  senderName ist eine leere oder einelementige Teilmenge von Name
- `receiverAddress: some Addr`
  receiverAddress ist eine nicht-leere Teilmenge von Addr.

Man kann Mengen oder Relationen auch dadurch definieren, dass man angibt, welche Elemente sie enthalten sollen:

```
{x1:e1, x2:e2, ..., xn:en | F}
```

Beispiele für das weiter oben definierte Adressbuch:

- `{n: Name | no n.^addrs & Addr}`
  Menge der Namen, die keine Adresse haben, auch nicht indirekt.
- `{n: Name, a: Address | n -> a in ^addrs}`
  Relation, die Namen die Adresse zuordnet

Relationen kann man mit einschränkenden Bedingungen über die Multiplizität der beteiligten Atome definieren:

Beispiele:

- `workAddrs: Name -> lone Addr`
  Die Relation workAddrs besteht aus Namen und Adressen, wobei jeder Name höchstens eine Adresse hat.
- `homeAddrs: Name -> one Addr`
  Zu jedem Namen gehört genau eine Adresse
- `members: Name lone -> some Addr`
  Zu einem Namen gehört mindestens eine Adresse und eine Adresse gehört zu höchstens einem Namen

let

Wenn man komplexe Ausdrücke in Alloy formulieren möchte, kann es vorkommen, dass sich ein Teilausdruck wiederholt. In dieser Situation kann man mit `let x = e | A` in `A` jedes Vorkommen des Ausdrucks `e` durch `x` ausdrücken.

Beispiel:

Man kann statt

```
all n: Name |
  (some n.workAddrs implies n.addrs = n.workAddrs
                      else n.addrs = n.homeAddrs)
```

auch schreiben:

```
all n: Name |
  let w = n.workAddrs, a = n.addrs |
    (some w implies a = w else a = n.homeAddrs)
```

Kardinalitäten

Man kann in Alloy die Kardinalität von Mengen und Relationen explizit angeben oder verlangen:

- `#r`, die Zahl der Tupel in `r`
- `0, 1, ...`, Literale für ganze Zahlen
- `+, -`, Addition und Subtraktion
- `=, <, >, <=, >=`, Vergleichsoperatoren
- `sum x: e | ie`, Summe aller Berechnungen `ie` für alle Tupel aus `e`.

Beispiele:

- `all b: Bag | #b.marbles <= 3`
  Alle Beutel enthalten höchstens 3 Murmeln

- `#Marble = sum b: Bag | #b.marbles`
  Die Summe der Murmeln in den Beuteln ist gleich der Gesamtsumme der Murmeln, d.h. jede Murmel ist in einem Beutel.

Ausdrucksmöglichkeiten in Alloy

Alloy ist syntaktisch so gebaut, dass man viele Möglichkeiten hat, einen Sachverhalt auszudrücken. Im Grunde ist die Sprache so gemacht, dass man sowohl Ausdrücke der Prädikatenlogik schreiben kann als auch äquivalente Ausdrücke im relationalen Kalkül.

Dies kann man an folgendem Beispiel sehen:
Die Aussage sei: „everybody loves a winner“.
Diese Aussage kann man in der Prädikatenlogik so ausdrücken:

$$\forall w\, Winner(w) \rightarrow \forall p\, loves(p, w)$$

Dieselbe Aussage liest sich als relationaler Ausdruck so:

$$Person \times Winner \subseteq loves$$

In Alloy kann man die Aussage so ausdrücken:

- `all p: Person, w: Winner | p -> w in loves`
- `Person -> Winner in loves`
- `all p: Person | Winner in p.loves`

Die Sprache von Alloy

Aufbau der Quellen, Module

Quellen in Alloy werden in Module organisiert. Jedes Modul erhält optional einen Modulnamen durch die Anweisung `module <modul-name>`, was die erste Anweisung in einer Quelle sein muss. Dadurch wird ein Namensraum festgelegt und dieses Modul kann von anderen Modulen verwendet werden.

Will man in einem Modul ein anderes verwenden, gibt man die Anweisung `open <modul-pfad>/<modul-name>`. Dadurch können Namenskonflikte entstehen, weshalb man dem Modul bei der Verwendung auch einen neuen Namen geben kann, z.B. `open util/boolean as b`. Deklarationen aus diesem Modul werden dann mit `b/<name>` angesprochen, wenn `<name>` ein Bezeichner im Modul ist.

Es gibt auch *parametrisierte* Module, in denen bestimmte Typen durch eigene ersetzt werden können. Dies ist besonders praktisch für generische Datenstrukturen.

In der Auslieferung von Alloy werden Module mitgeliefert, die oft nützlich sein können; siehe http://alloy.lcs.mit.edu/alloy/documentation/quickguide/

`index.html` Abschnitt „Built-in Utility Modules“. Darunter befinden sich auch parametrisierte Module, etwa `graph`, bei deren Nutzung eigene Signaturen als Knoten verwendet werden können.

Kommentare kann man im C-Stil machen, d.h. alles von `/*` bis `*/` ist ein Kommentar, oder auch im C++-Stil, d.h. `//` bis zum Zeilenende ist ein Kommentar oder auch im SQL-Stil, d.h. `--` bis zum Zeilenende ist ein Kommentar.

Signaturen und Felder

In Alloy definiert man *Signaturen*, die das Universum für die Modelle in Alloy strukturieren. Im Kern wird das Universum zerlegt in paarweise disjunkte Teilmengen, die durch die Signaturen definiert werden. Man kann die Definition von Signaturen auch so sehen, dass *Typen* eingeführt werden und das Universum aus verschiedenen Typen besteht, die jeweils eine eigene Menge von Elementen darstellen.

Beispiele für die Definition von Signaturen:

- `sig A{}`
  eine Menge von Atomen namens A
- `sig A{}`
  `sig B{}`
  zwei disjunkte Mengen A und B
- `sig A, B{}`
  zwei disjunkte Mengen A und B
- `sig B extends A{}`
  die Menge B ist eine Teilmenge von A
- `sig B extends A{}`
  `sig C extends A{}`
  die Mengen B und C sind disjunkte Teilmengen von A
- `sig B, C extends A{}`
  die Mengen B und C sind disjunkte Teilmengen von A
- `abstract sig A{}`
  `sig B extends A{}`
  `sig C extends A{}`
  die Menge A ist partitioniert in die disjunkten Teilmengen B und C, d.h. jedes Element von A ist entweder in B oder C.
- `sig B in A{}`
  B ist eine Teilmenge von A, aber nicht notwendigerweise disjunkt zu anderen Teilmengen
- `sig C in A + B{}`
  C ist eine Teilmenge der Vereinigung von A und B

- ```
  one sig A{}
  lone sig B{}
  some sig C{}
  ```
  A ist ein Singleton, besteht also aus genau einem Element, B ist ein Singleton oder leer und C ist eine nicht-leere Menge.

Signaturen können *Felder* haben, wodurch Relationen zwischen Mengen definiert werden[2]

Beispiele:

- ```
  sig A{
    field: e
  }
  ```
  field ist eine binäre Relation mit dem Definitionsbereich A und dem Zielbereich bestimmt durch den Ausdruck e, d.h. $field \subseteq A \times e$

- ```
  sig A{
    f1: one e1,
    f2: lone e2,
    f3: some e3,
    f4: set e4
  }
  ```
  Hiermit werden 4 binäre Relationen definiert, wobei in f1 zu jedem Element in A genau eines aus e1 zugeordnet wird, bei f2 höchstens eines aus e2, bei f3 mindestens eins aus e3 und bei f4 beliebig viele aus e4. Gibt man keine Multiplizität an, ist `one` gemeint.

- ```
  sig Book{
    names: set Name,
    addrs: names -> Addr
  }
  ```
  Hiermit wird eine binäre Relation names definiert, die jedem Book b eine Mengen von Namen zu ordnet, sowie eine ternäre Relation zwischen Book, Name und Addr, mit der den Namen in b Adressen zugeordnet werden.

- ```
  abstract sig Person{
    father: lone Man,
  ```

[2]Diese Eigenschaft der Sprache Alloy mutet zunächst merkwürdig an, denn in einer Relation ist ja normalerweise nicht eine beteiligte Menge gegenüber der anderen ausgezeichnet, wie es hier im Unterschied von Signatur und Feld ist. Der Grund liegt darin, dass Jackson seiner Sprache einen „objektorientierten Geschmack“ geben wollte.
Diese Eigenschaft von Alloy erlaubt es Objektmodelle aus der objektorientierten Welt recht einfach in Alloy zu übertragen und vereinfacht damit den Einstieg in die Technik der Simulation und Verifikation mit Alloy.
Andererseits sind Atome und Relation in Alloy *keine* Objekte und eine Signatur definiert auch keine Klasse. Denn Atome sind Werte und damit unveränderlich. Spätestens wenn man anfängt, Zustände und Zustandsänderungen in Alloy zu beschreiben, muss man diesen Unterschied verstehen und kann durch die „OO-Schreibweise“ sogar in die Irre geführt werden.
  ```

```
  mother: lone Woman
}
sig Man extends Person{
  wife: lone Woman
}
sig Woman extends Person{
  husband: lone Man
}
```

Jede Person ist entweder ein Mann oder eine Frau. Jede Person hat höchstens einen Vater und höchstens eine Mutter, die Männer oder Frauen sind. Jeder Mann ist höchstens mit einer Frau verheiratet und jede Frau mit höchstens einem Mann.

Fakten

Fakten sind Bedingungen, die in jedem Modell gelten müssen, das der Alloy Analyzer erzeugen soll.

Beispiel:

```
fact {
  no p: Person |
    p in p.^(mother + father)
  wife = ~husband
}
```

Damit wird ausgedrückt, dass eine Person niemals ihr eigener Vorfahre sein kann, und dass der Ehemann einer Frau eben diese Frau zur Ehefrau hat.

Funktionen und Prädikate

Funktionen sind parametrisierte Ausdrücke, die Relationen zum Ergebnis haben.

Beispiel:

```
sig Name, Addr{}
sig Book{
  addr: Name -> Addr
}

fun lookup[b: Book, n: Name] : set Addr {
  b.addr[n]
  // oder: n.(b.addr)
}

fact everyNameMapped {
  all b: Book, n: Name | some lookup[b, n]
}
```

Die Funktion `lookup` hat zum Ergebnis die Menge der Adressen zu einem Namen in einem Adressbuch. Im Fakt wird diese Funktion verwendet, um festzulegen, dass in allen Bücher für alle Namen mindestens eine Adresse vorhanden ist.

Prädikate sind parametrisierte Formeln, die einen Wahrheitswert haben. Beispiel:

```
sig Name, Addr{}
sig Book {
  addr: Name -> Addr
}

pred contains[b: Book, n: Name, d: Addr] {
  n-> d in b.addr
}

fact everyNameMapped {
  all b: Book, n: Name |
    some d: Addr | contains[b, n, a]
}
```

Das Prädikat `contains` ist wahr, wenn das Paar von Name und Adresse im Adressbuch vorkommt. Im Fakt wird dieses Prädikat verwendet, um festzulegen, dass in allen Büchern für jeden Namen mindestens eine Adresse vorhanden ist.

Funktionen und Prädikate können auch in „objektorientierter“ Manier verwendet werden, wenn der erste Parameter ein Skalar ist. Eine Funktion

```
fun grandpas[p: Person] : set Person {
  p.(mother + father).father
}
```

könnte auch so geschrieben werden

```
fun Person.grandpas : set Person {
  this.(mother + father).father
}
```

und wenn p eine Person ist, kann man in beiden Fällen `p.grandpas[]` schreiben, was `grandpas[p]` gleichkommt.

Zusicherungen und Überprüfungen

Mit dem Schlüsselwort `assert` kann man in Alloy Aussagen formulieren, bei denen man davon ausgeht, dass sie in der gegebenen Spezifikation erfüllt sind. Mit `check` veranlasst man den Alloy Analyzer nach einem Gegenbeispiel zu der Aussage zu suchen.

Beispiel:

```
fact {
```

```
  no p: Person | p in p.^(mother + father)
wife = ~husband
}

assert noSelfFather {
  no m: Man | m = m.father
}

check noSelfFather for 4
```

In diesem Beispiel wird das Faktum formuliert, dass niemand sein eigener Abkömmling ist. Die Aussage `noselfFather` gibt an, dass kein Man sein eigener Vater sein kann. Die Anweisung `check` ist ein Befehl an den Alloy Analyzer für jeweils bis zu 4 Elemente jeder Signatur zu überprüfen, ob die Annahme zutrifft oder ob es ein Gegenbeispiel gibt. Die Zahl bei `check` nennt man den *scope* der Analyse. Wird er nicht angegeben, untersucht der Alloy Analyzer „Welten“ mit je maximal 3 Elementen jedes Typs.

Man kann auch verschiedene Anzahlen für die einzelnen Signaturen vorgeben:

```
abstract sig Person{}
sig Man extends Person{}
sig Woman extends Person{}
sig Grandpa extends Man{}
...
check a
check a for 4
check a for 4 but 3 Woman
check a for 4 but 3 Man, 5 Woman
check a for 4 Person
check a for 4 Person, 3 Woman
check a for 3 Man, 4 Woman
check a for 3 Man, 4 Woman, 2 Grandpa
```

Simulationen

Der Alloy Analyzer kann nicht nur Annahmen überprüfen, sondern auch „Beispielwelten“ produzieren; er wird dann zur *Simulation* eingesetzt.

Will man etwa einfach mal eine „Welt“ für die Spezifikation sehen, schreibt man das Kommando `run{}` und führt es aus. Dies ist insbesondere bei der Entwicklung von Spezifikationen in Alloy eine große Hilfe. Denn oftmals schreibt man Signaturen, Fakten oder Prädikate und vergisst dabei ganz simple Fälle. Regelmäßiges Prüfen der Spezifikation durch Simulation kann zeigen, dass man das eine oder andere noch berücksichtigen muss.

Man kann `run` mit einem Prädikat aufrufen, dann wird ein Modell erzeugt, das dieses Prädikat erfüllt. Oder man kann `run{...}` mit einer Formel aufrufen, die zwischen den Klammern steht; eventuell auch mit einer leeren Formel.

Der Alloy Analyzer

Einführung

Der Alloy Analyzer ist eine Java-Anwendung, die als Open-Source unter der MIT-Lizenz zur Verfügung gestellt wird. Sie kann unter http://alloy.lcs.mit.edu/alloy/download.html heruntergeladen werden.

Die Anwendung besteht aus einem Texteditor, mit welchem die Modelle formuliert werden können, und dem *Visualizer*, der Instanzen des Modells graphisch darstellt.

Entwicklung eines Modells

Um die Funktionsweise zu demonstrieren und einen Einblick in die Möglichkeiten zu geben, soll in diesem Abschnitt Schritt für Schritt ein Modell in Alloy entwickelt werden.

Unser Ziel soll es sein ein Adressbuch zu modellieren, welches zu jedem Namen mehrere Adressen speichern und die Namen in Gruppen einteilen kann. Gruppen sollen dabei auch Untergruppen erlauben. Dies formulieren wir zunächst – etwas naiv – folgendermaßen in Alloy:

```
sig Addr {}
abstract sig Entry {}
sig Name extends Entry {
  addr: set Addr
}
sig Group extends Entry {
  sub: set Entry
}
sig Book {
  entries: set Entry
}
```

D.h. ein Adressbuch besteht aus Einträgen, wobei jeder Eintrag entweder ein Name ist, welchem Adressen zugeordnet sind, oder eine Gruppe, die wiederum Einträge enthält (Gruppen oder Namen).

Lässt man den Alloy Analyzer mit `run{}` gültige Instanzen finden, erhält man als Ergebnis Abb. 1. Es sind fünf Elemente als orange gefüllte Rechtecke zu sehen, die jeweils den Namen der Signatur haben. Falls es mehrere Elemente einer Signatur gibt, hängt der Visualizer eine Zahl, beginnend mit 0, an den Namen. Die Relationen sind als Pfeile dargestellt. D.h. *Group* ist über die Relation *sub* auf *Name* zugeordnet und *Name* ist über *addr* auf *Addr* zugeordnet. Es existieren also zwei Adressbücher, jeweils ohne Einträge und eine Gruppe, die *Name* enthält. Diesem ist nur eine Addresse zugeordnet: *Addr*.

Nun stellt sich die Frage: Entspricht dies einem gültigen Modell in dem Sinne, wie man es sich bei der anfänglichen, recht unpräzisen Formulierung

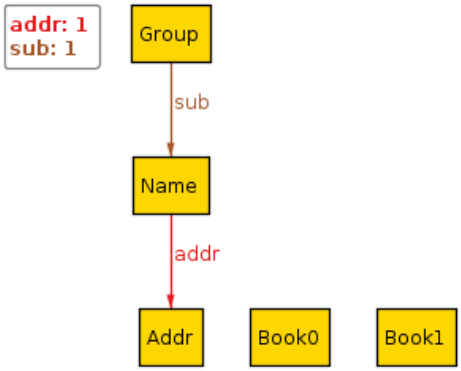

Abbildung 1: Gültige Instanz bei erstem Ansatz

des Vorhabens vorgestellt hat? Nein. Denn es macht keinen Sinn, Gruppen zu erlauben, die zu keinem Adressbuch gehören.

Demzufolge wollen wir ein Prädikat[3] hinzufügen, um dies zu verhindern. Wir formulieren dies gleich allgemein für Namen und Gruppen, d.h. für Einträge. Für das Prädikat wäre es hilfreich, wenn wir eine Funktion hätten, die uns alle Einträge in einem Adressbuch liefert. Versucht man die Funktion zu schreiben, stellt man jedoch fest, dass es sich relativ kompliziert gestaltet. Warum ist das so?

Der Grund ist, dass wir zu Beginn eine ungeeignete Beschreibung gewählt haben. Denn sowohl die Signatur *Book* als auch *Group* enthält eine Menge von Einträgen. Viel geschickter wäre es, wenn *Book* lediglich eine (Wurzel-)Gruppe enthalten würde und nur *Group* eine Menge von Einträgen. Auf diese Weise verhindern wir Spezialfälle bei der weiteren Spezifikation des Modells. Diese Tatsache hätten wir vermutlich auch bei der Umsetzung des Adressbuchs in Software festgestellt (womöglich mit erheblich mehr Aufwand bei der darauf folgenden Umstellung).

Unser überarbeitetes Modell sieht folgendermaßen aus:

```
sig Addr {}
abstract sig Entry {}
sig Name extends Entry {
  addr: set Addr
}
sig Group extends Entry {
  entries: set Entry
}
sig Book {
  root: one Group
}

fun bookEntries [b: Book] : set Entry {
```

[3] Man könnte dies auch als Fakt formulieren, jedoch ist es oft von Vorteil Prädikate zu verwenden, da man diese flexibel an- und abschalten kann.

```
  b.root + b.root.^entries
}
fun bookAddresses [b: Book] : set Addr {
  bookEntries[b].addr
}

pred noFreeElems [] {
  all e: Entry | (some b: Book | e in bookEntries[b])
  all a: Addr | (some b: Book | a in bookAddresses[b])
}
```

Die Funktion *bookEntries* liefert also zu einem Adressbuch alle dazugehören Einträge (sowohl Gruppen als auch Namen). Dafür verwenden wir den Operator ^ für den transitiven Abschluss um auch Gruppen und Namen in Untergruppen mit einzubeziehen. Desweiteren gibt es die Funktion *bookAddresses*, die alle Adressen in einem Adressbuch liefert und dafür *bookEntries* verwendet. Um keine „freistehenden“ Namen, Gruppen und Adressen zuzulassen, existiert das Prädikat *noFreeElems*, auf das wir in Zukunft bei der Generierung von Welten in Form von `run{ noFreeElems[] }` zurückgreifen werden.

Die erste generierte Welt ist in Abb. 2 dargestellt.

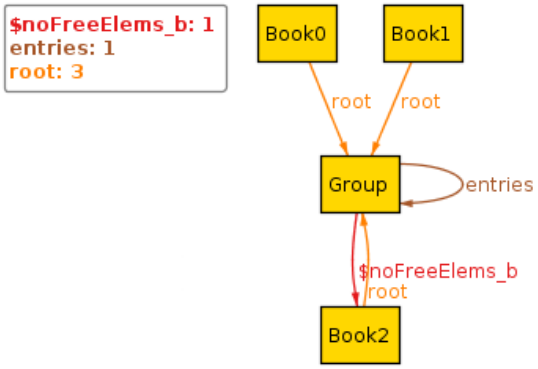

Abbildung 2: Überarbeitete Version

Wie in der Abbildung zu sehen ist, stellt der Visualizer auch Prädikate graphisch dar. In diesem Fall gilt das Prädikat für *Group*, da es ein Adressbuch *b* (*Book2*) gibt, welchem sie zugeordnet ist.

Desweiteren werden zwei ungewollte Szenarios deutlich:

1. Laut Modell ist es zulässig, dass mehrere Adressbücher die gleiche Gruppe haben
2. Ebenfalls kann eine Gruppe seine eigene Untergruppe sein

Diese verhindern wir mit einem Fakt, da derartige Modelle nie möglich sein sollen:

```
fact {
  all g: Group | g not in g.^entries
```

```
  all disj b,b': Book | no (bookEntries[b] & bookEntries[b'])
}
```

Die erste Zeile besagt, dass keine Gruppe in ihren eigenen Untergruppen sein kann, d.h. die Relation *azyklisch* ist (um Situationen wie *Group0* in *Group1* in *Group0* zu verhindern, wird wieder der transitive Abschluss gebildet). Die zweite legt fest, dass Einträge nie in mehreren Adressbüchern sein können, so dass der Schnitt der Einträge von verschiedenen Adressbüchern immer leer ist.

Eine weitere Weltgenerierung mit dem genannten Fakt ergibt Abb. 3.

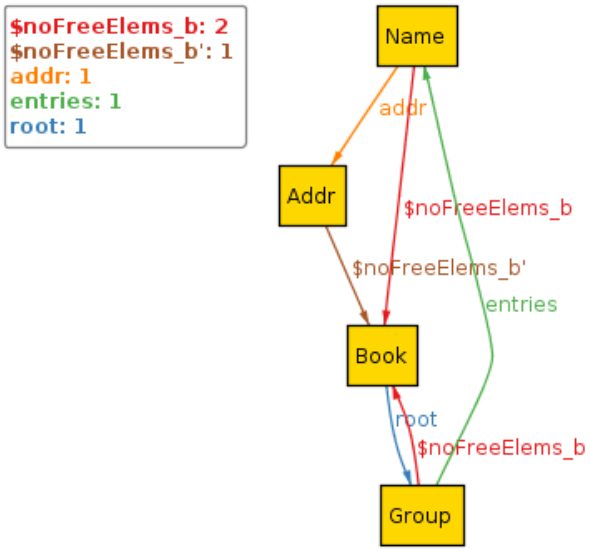

Abbildung 3: Nochmals überarbeitete Version

An dieser Stelle wird auch deutlich, dass die graphische Darstellung schnell unübersichtlich wird. Um dem entgegenzuwirken bietet der Visualizer die Möglichkeit das sogenannte *Theme* anzupassen. In diesem Fall ist es zweckdienlich, die Relationen aus dem Prädikat *noFreeElems* nicht anzuzeigen. Dies lässt sich im Visualizer einstellen, wie in Abb. 4 gezeigt.

Überprüfung des Modells

Um sicher zu stellen, dass die zuvor formulierten Funktionen *bookEntries* und *bookAddresses* auch richtig sind, können wir den im Visualizer integrierten *Evaluator* verwenden. Dieser bietet die Möglichkeit das Ergebnis von beliebigen Ausdrücken in Alloy in einer konkreten Welt anzeigen zu lassen. Wir lassen dazu den Alloy Analyzer mittels

```
run {
  noFreeElems and #Book >= 2 and #Group >= 3 and #Addr >= 3
} for 4
```

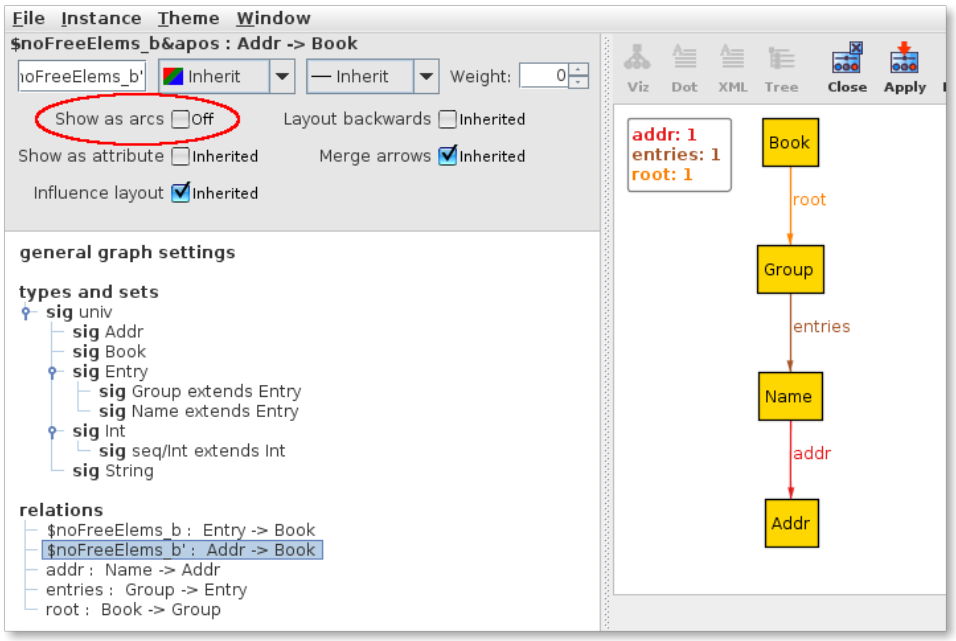

Abbildung 4: Anpassung des Themes

eine hinreichend komplizierte Welt erzeugen, in welcher wir die Funktionen gut untersuchen können. Im Evaluator lassen wir uns das Ergebnis der beiden Funktionen darstellen, um uns von deren Korrektheit zu überzeugen. Der Evaluator zeigt uns das Ergebnis wie in Abb. 5.

Eine weitere Möglichkeit das Modell zu überprüfen, ist den Alloy Analyzer anzuweisen zu einer Behauptung ein Gegenbeispiel zu finden. Auf diese Weise können wir z.B. überprüfen ob der Fakt

```
all disj b,b': Book | no (bookEntries[b] & bookEntries[b'])
```

wirklich ausschließt, dass kein Eintrag in mehreren Adressbüchern sein kann. Dazu formulieren wir folgende Behauptung und lassen sie von dem Alloy Analyzer überprüfen:

```
assert noEntryInMultipleBooks {
  no e: Entry, b,b': Book |
    b != b' and e in bookEntries[b] and e in bookEntries[b']
}
check noEntryInMultipleBooks for 8
```

Um möglichst sicher sein zu können[4], dass es wirklich kein Gegenbeispiel gibt, lassen wir es in Welten mit bis zu 8 Elementen pro Signatur validieren. Der Alloy Analyzer liefert uns das in Abb. 6 dargestellte Ergebnis.

[4]Der Alloy Analyzer betrachtet stets nur endliche Welten, so dass es nicht ausgeschlossen ist, dass z.B. in diesem Fall ein Gegenbeispiel mit mehr als 8 Elementen pro Signatur existiert.

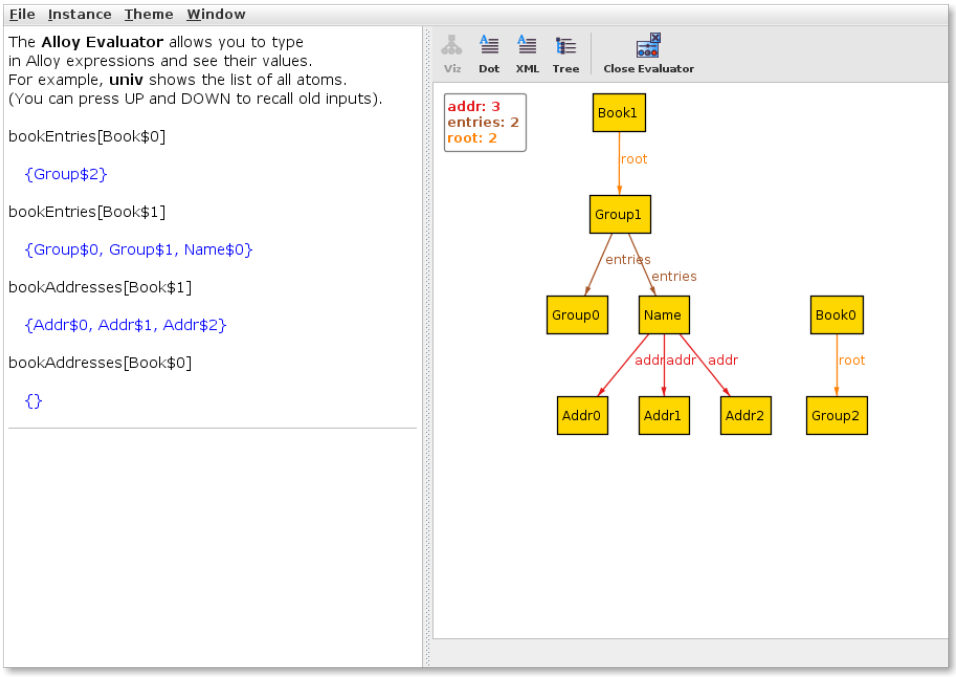

Abbildung 5: Überprüfung der Funktionen im Evaluator

```
Executing "Check noEntryInMultipleBooks for 8"
  Solver=minisat(jni) Bitwidth=4 MaxSeq=7 SkolemDepth=1 Symmetry=20
  5092 vars. 248 primary vars. 12053 clauses. 156ms.
  No counterexample found. Assertion may be valid. 4ms.
```

Abbildung 6: Finden eines Gegenbeispiels

Fazit

Das hier entwickelte Beispiel hat gezeigt, wie man ein einfaches Modell in Alloy entwickeln kann und wie man dabei den Alloy Analyzer, Visualizer und Evaluator sinnvoll einsetzen kann. Neben den geschilderten Möglichkeiten gibt es natürlich noch viele andere. Des weiteren lässt das entwickelte Modell noch weitere, vermutlich nicht erwünschte Instanzen zu. Es sei dem Leser überlassen, Alloy und den Alloy Analyzer weiter zu erforschen und das Modell fertigzustellen.