Institut für SoftwareArchitektur

Fachhochschule Gießen Friedberg
University of Applied Sciences

Nils Asmussen

Ansätze zur Modellierung von Dynamik mit Alloy

28. September 2010

Technischer Bericht Nr. 1
des Instituts für SoftwareArchitektur

Inhaltsverzeichnis

1 Einleitung

Alloy ist eine leichtgewichtige Modellierungssprache, mit welcher in deklarativer Form Softwaresysteme beschrieben werden können [4]. Der dazugehörige *Alloy Analyzer*[1] ist in der Lage, gültige Instanzen eines Modells oder Gegenbeispiele zu Behauptungen zu finden. Er agiert dabei stets in einem endlichen Universum. D. h. im Gegensatz zu anderen Modellierungssprachen beweist man mit Alloy nicht die Korrektheit eines Systems, sondern lässt es automatisiert in Universen selbstgewählter Größe überprüfen. [3, S. 1] Daniel Jackson, der Urheber von Alloy, geht dabei von der „small scope hypothesis“ aus, die besagt, dass Fehler meistens kleine Gegenbeispiele haben. [1, S. 141]

In der vorliegenden Arbeit sollen drei Ansätze zur Modellierung von dynamischen Systemen – d. h. Systeme, die sich in Abhängigkeit der Zeit ändern – in Alloy beschrieben werden. Namentlich sind dies *Global State*, *Time Axis* und *Trace*. Die Techniken Global State und Time Axis wurden bereits in Daniel Jacksons Buch „Software Abstractions - Logic, Language, and Analysis“ erwähnt [1, S. 170-177], wenn auch nicht mit den genannten Namen. Der Ansatz Trace wurde ebenfalls kurz angesprochen [1, S. 177,178] und wurde etwas umfangreicher in „Abstract State Machines, Alloy, B and Z“ beschrieben [2, S. 105-117].

Aufgrund der verschiedenen Schwerpunkte der Ansätze, werden sie in dieser Arbeit in zwei Kategorien eingeteilt: *operationsfokussiert*, zu welcher Global State und Time Axis zählen, sowie *ablauffokussiert*, zu welcher Trace gehört. D. h. in ersterer Kategorie geht es vornehmlich um die Beschreibung der Operationen selbst, also diejenigen, die den Zustand des Systems verändern. In letzterer hingegen steht die Beschreibung des Ablaufes bzw. der Reihenfolge der Operationen im Vordergrund.

[1] http://alloy.mit.edu/alloy4/

2 Analyse der Beschreibungsstile

Die Beschreibung von Systemen, die dynamisch sind, d. h. ihren Zustand ändern, ist in Alloy nicht direkt umsetzbar. Denn die Welten in Alloy sind grundsätzlich unveränderlich. [1, S. 39] Insofern muss Dynamik auf eine andere Weise „simuliert“ werden. Im Folgenden sollen drei verschiedene Ansätze dafür anhand eines Beispiels beschrieben werden.

1 Beschreibung des Beispiels

Als Beispiel soll eine einfache Form eines Flughafens dienen. Es können Flüge erstellt werden und Passagiere können Flüge buchen sowie stornieren. Um das Beispiel übersichtlich zu halten und auf das Ziel dieser Arbeit auszurichten, ist das Modell auf die genannten Objekte beschränkt und verzichtet auch auf alle Eigenschaften, die in der Praxis relevant wären (Datum des Flugs, beteiligte Flughäfen, Anschlussflüge, Datum der Buchung, Daten des Passagiers, etc.).

2 Operationsfokussiert

Als Erstes sollen die operationsfokussierten Beschreibungsstile vorgestellt werden. Dafür werden zunächst die Signaturen und grundlegenden Prädikate für jeden Stil separat geschildert und anschließend die denkbaren Prüfungen, welche für beide Varianten identisch formuliert werden können (bei geeigneten Funktionen und Prädikaten, die die Unterschiede kapseln). Am Ende folgt ein Vergleich beider Stile.

2.1 Global State

Die Idee des Stils Global State besteht darin eine Zeit-Dimension in Form einer Signatur einzuführen, welche alle dynamischen Anteile des Modells besitzt. Nach Jackson [1, S. 175] wird dieser Ansatz in den meisten herkömmlichen Modellierungssprachen (z. B. Z und VDM) angewendet. Das Modell kann in diesem Stil wie folgt formuliert werden:

```
sig Passenger {}
sig Flight {}
sig Time {
  flights: set Flight,
  passengers: Flight -> Passenger
}
```

Die Signatur `Time` hat ein Feld `flights`, welches die existierenden Flüge (zu diesem Zeitpunkt) enthält. Des Weiteren ordnet das Feld `passengers` Passagiere Flügen zu.

Um einen geordneten zeitlichen Ablauf zu erreichen, wird das von Alloy mitgelieferte Hilfsmodul `util/ordering` verwendet:

```
open util/ordering[Time]
```

Dieses Modul definiert Funktionen und Prädikate, die es ermöglichen auf die Elemente des Modulparameters (`Time`) geordnet zuzugreifen, wie z. B. `first` oder `next`, welche das erste bzw. jeweils nächste Element der Folge liefern.

Zur einfacheren Formulierung der folgenden Prädikate und Behauptungen und um diese sowohl für Global State als auch für Time Axis benutzen zu können, werden Hilfsfunktionen bzw. Prädikate definiert:

```
fun flightPassengers [f: Flight, t: Time] : set Passenger {
  t.passengers[f]
}
pred flightBooked [f: Flight, p: Passenger, t: Time] {
  p in flightPassengers[f,t]
}
```

Die Funktion `flightPassengers` liefert alle Passagiere eines Fluges zu einem bestimmten Zeitpunkt. Das Prädikat `flightBooked` ist wahr, wenn der gegebene Passagier den Flug zu dem gewählten Zeitpunkt gebucht hat.

Die möglichen Operationen werden wie folgt definiert:

```
pred createFlight [f: Flight, t,t': Time] {
  f !in t.flights
  t'.flights = t.flights + f
  t'.passengers = t.passengers
}
pred bookFlight [f: Flight, p: Passenger, t,t': Time] {
  !flightBooked[f,p,t] and f in t.flights
  t'.passengers = t.passengers + (f -> p)
  t'.flights = t.flights
}
pred cancelFlight [f: Flight, p: Passenger, t,t': Time] {
  flightBooked[f,p,t] and f in t.flights
  t'.passengers = t.passengers - (f -> p)
  t'.flights = t.flights
}
```

Es existieren also drei Operationen: `createFlight` um einen Flug zu erstellen (d. h. als „existierend“ zu vermerken), `bookFlight` um als Passagier einen Flug zu buchen und `cancelFlight` um einen Flug wieder zu stornieren. Wie das Listing zeigt, werden alle Operationen in einem ähnlichen Stil formuliert:

- Alle sind Prädikate.
- Alle besitzen neben den für die Funktionalität notwendigen Parametern zwei Zeitpunkte: `t` und `t'`, wobei `t'` der Folgezeitpunkt ist.

- Jede Operationen hat „Vorbedingungen“[1], nimmt eine Veränderung vor und legt fest, dass alle anderen Felder des Zustands konstant bleiben sollen.

Am Beispiel `bookFlight` bedeutet dies, dass die Operation einen Flug und einen Passagier benötigt. Die Vorbedingung ist, dass der Passagier diesen Flug zum Zeitpunkt `t` noch nicht gebucht hat und der Flug zum Zeitpunkt `t` existiert. Anschließend legt das Prädikat fest, dass zum Zeitpunkt `t'` der Passagier `p` den Flug `f` gebucht haben soll. Die existierenden Flüge sollen von der Operation unberührt bleiben.

Um im ersten Zeitpunkt grundsätzlich ohne Flüge und ohne Buchungen zu starten, wird folgender Fakt formuliert:

```
fact {
  no first.flights
  no first.passengers
}
```

Auf diese Weise hat jede Instanz des Modells den gleichen Startzustand, welches die Formulierung und Überprüfung von Behauptungen erleichtert.

2.2 Time Axis

Analog zu Global State werden auch bei dem Stil Time Axis die Signaturen `Flight`, `Passenger` und `Time` definiert. Jedoch sind die Felder nicht `Time` zugeordnet, sondern `Flight`:

```
sig Time {}
sig Passenger {}
sig Flight {
  exists: set Time,
  passengers: Passenger -> Time
}
```

Die Zeitkomponente steht somit stets am Ende der Relationen, im Gegensatz zu Global State, bei dem sie die erste Komponente der Relationen bildet.

In diesem Modell wird ebenfalls das Modul `util/ordering` verwendet um die Elemente von `Time` zu ordnen. Des Weiteren sind `flightPassengers` und `flightBooked` wie folgt definiert:

```
fun flightPassengers [f: Flight, t: Time] : set Passenger {
  f.passengers.t
}
pred flightBooked [f: Flight, p: Passenger, t: Time] {
  p in flightPassengers[f,t]
}
```

Ein Unterschied lässt sich also lediglich in `flightPassengers` ausmachen: Entsprechend der anderen Reihenfolge bei den Relationen wird auch der `dot-join` anders angewandt.

[1] Alloy unterscheidet natürlich nicht zwischen als „Vorbedingungen“ bezeichenbaren und anderen Ausdrücken in einem Prädikat. Dies dient nur der Anschauung.

Die Operationen und der Fakt zur Festlegung der Startbedingungen im Stil Time Axis werden folgendermaßen formuliert:

```
pred createFlight [f: Flight, t,t': Time] {
  t !in f.exists
  exists.t' = exists.t + f
  passengers.t = passengers.t'
}
pred bookFlight [f: Flight, p: Passenger, t,t': Time] {
  !flightBooked[f,p,t] and t in f.exists
  passengers.t' = passengers.t + (f -> p)
  exists.t' = exists.t
}
pred cancelFlight [f: Flight, p: Passenger, t,t': Time] {
  flightBooked[f,p,t] and t in f.exists
  passengers.t' = passengers.t - (f -> p)
  exists.t' = exists.t
}

fact {
  no exists.first
  no passengers.first
}
```

Wie das Listing zeigt, hat sich ebenfalls nur die Reihenfolge der Operanden geändert.

2.3 Behauptungen und deren Prüfung

Das in den vorherigen Abschnitten in den Stilen Global State und Time Axis beschriebene Modell soll nun überprüft werden. Da es sich um operationsfokussierte Stile handelt, werden in erster Linie die Operationen selbst getestet. Dies kann für beide Stile in gleicher Weise durchgeführt werden, da die gleichen Namen gewählt und entsprechende Funktionen und Prädikate definiert wurden, die die Unterschiede verbergen.

2.3.1 Buchung und Stornierung

Als Erstes soll das Prädikat `bookFlight` überprüft werden. Dazu dient folgende Behauptung:

```
assert bookBooks {
  all t: Time, t': t.next, f: Flight, p: Passenger |
    bookFlight[f,p,t,t'] => {
      flightBooked[f,p,t']
      all p': flightPassengers[f,t] | flightBooked[f,p',t']
      all f': Flight - f | flightPassengers[f',t] = flightPassengers[f',t']
    }
}
```

Die Behauptung besagt, dass aus einer Buchung eines Passagiers für einen Flug folgt, dass

1. der Flug anschließend für diesen Passagier gebucht ist,
2. die vorherigen Buchungen dieses Fluges weiterhin bestehen und
3. alle anderen Flüge nicht von der Buchung betroffen sind.

Die Überprüfung von Stornierungen erfolgt analog. Eine weitere Behauptung stellt den Zusammenhang zwischen Buchungen und Stornierungen her:

```
assert cancelUndos {
  all t: Time, t': t.next, t'': t'.next, f: Flight, p: Passenger |
    (bookFlight[f,p,t,t'] and cancelFlight[f,p,t',t'']) =>
      flightPassengers[f,t] = flightPassengers[f,t'']
}
```

In dieser Behauptung wird also mit drei Zeitpunkten gearbeitet: Von `t` nach `t'` soll eine Buchung stattfinden und von `t'` nach `t''` eine Stornierung. Anschließend sollen die Passagiere dieses Flugs zum Zeitpunkt `t` identisch zu denen zum Zeitpunkt `t''` sein.

2.3.2 Erstellung von Flügen

An dieser Stelle soll ein anderer Ansatz gewählt werden um die Erstellung von Flügen bzw. die Konsequenzen daraus zu verifizieren. Die Frage, die sich stellt, ist, ob aus den Vorbedingungen der Operationen folgt, dass zu jedem Zeitpunkt nur Passagiere zu existierenden Flügen gebucht sind:

```
assert noPassengersInNonExistingFlights {
  all t: Time, f: Flight |
    t !in f.exists => no flightPassengers[f,t]
}
```

Lässt man den Alloy Analyzer mittels `check noPassengersInNonExistingFlights` ein Gegenbeispiel finden, erhält man ein möglicherweise überraschendes Ergebnis:

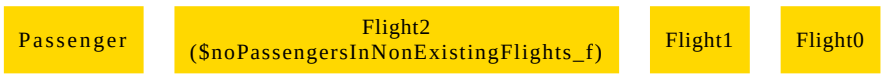

Abbildung 1: Projiziert auf Time0

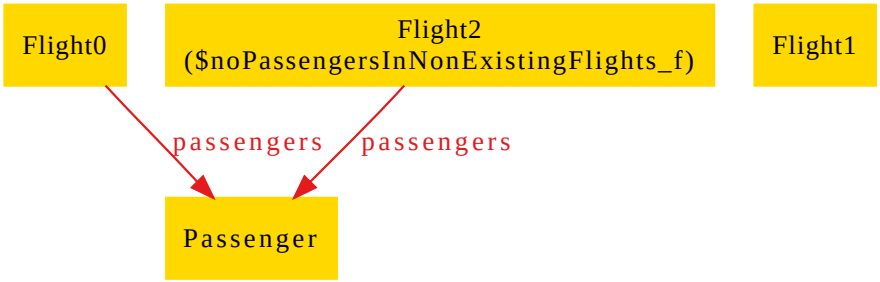

Abbildung 2: Projiziert auf Time1

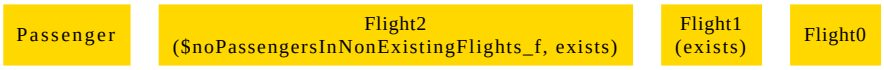

Abbildung 3: Projiziert auf Time2

D. h. zum Zeitpunkt `Time0` existieren weder Flüge noch Buchungen, in `Time1` zwei Buchungen (zu nicht existierenden Flügen) und in `Time2` drei Flüge ohne Buchungen. Der Grund für diese auf den ersten Blick evtl. unerwartete gültige Instanz des Modells ist, dass das Modell in keinster Weise festlegt, welcher Übergänge zwischen Zeitpunkten gültig sind. Es definiert lediglich Zeitpunkte und legt fest, wann ein Übergang eine Flugerstellung, Buchung oder Stornierung ist. Die Relationen zu den Zeitpunkten sind also beliebig.

Um zu erwingen, dass zwischen jeweils zwei Zeitpunkten ausschließlich die im Modell definierten Operationen angewendet werden, werden folgende Prädikate ergänzt:

```
pred opCreate [t,t': Time] {
  one f: Flight | createFlight[f,t,t']
}
pred opBook [t,t': Time] {
  one f: Flight, p: Passenger | bookFlight[f,p,t,t']
}
pred opCancel [t,t': Time] {
  one f: Flight, p: Passenger | cancelFlight[f,p,t,t']
}
pred oneOpPerTime [] {
  all t: Time, t': t.next |
    // min one
    (opCreate[t,t'] or opBook[t,t'] or opCancel[t,t']) and
    // max one
    !(opCreate[t,t'] and opBook[t,t']) and
    !(opCreate[t,t'] and opCancel[t,t']) and
    !(opBook[t,t'] and opCancel[t,t'])
}
```

Die ersten drei Prädikate sind wahr, wenn zwischen den gegebenen Zeitpunkten genau einmal die entsprechende Operation durchgeführt wird. Darauf aufbauend existiert das Prädikat `oneOpPerTime` um zu erwingen, dass stets **genau eine** Operation pro Zeitübergang angewandt wird.

Ändert man die obige Behauptung in

```
assert noPassengersInNonExistingFlights {
  oneOpPerTime[] =>
    (all t: Time, f: Flight |
      t !in f.exists => no flightPassengers[f,t])
}
```

wird das gewünschte Ergebnis erreicht, d. h. Alloy findet kein Gegenbeispiel im gewählten Scope.

2.4 Vergleich

Wie die Vorstellung der Stile Global State und Time Axis in den vorherigen Abschnitten gezeigt hat, existieren keine maßgeblichen Unterschiede zwischen den Vorgehensweisen. Beide führen eine Zeitdimension ein, legen mögliche Operationen fest und konzentrieren sich auf die Untersuchung der Operationen selbst. Global State sammelt alle dynamischen Anteile als Felder in der Zeitdimension, so dass die Zeit die erste Komponente der Relationen bildet. Time Axis hingegen fügt allen dynamischen Relationen als letzte Komponente die Zeit hinzu.

Sowohl Global State als auch Time Axis erlauben es leider nicht ein existierendes, statisches Modell lediglich zu benutzen und darauf aufbauend Dynamik hinzuzufügen. Im Falle von Global State müssen die dynamischen Anteile aus den bestehenden Signaturen in die Zeitdimension verschoben werden. Time Axis verlangt die Hinzufügung der Zeit als letzte Komponente der Relationen. Dies hat zur Folge, dass viele Ausdrücke umformuliert werden müssen.

Als Unterschied kann genannt werden, dass Time Axis eher die objektorientierte Herangehensweise ist, da alle dynamischen Eigenschaften zusammen mit den statischen bei den Objekten erhalten bleiben. Global State hingegen trennt zwischen statischen und dynamischen Anteilen des Modells und erleichtert die schrittweise Erweiterung des globalen Zustands. [1, S. 175] D. h. eine mögliche Sichtweise wäre, dass Global State vorzuziehen ist, wenn die Dynamik im Vordergrund steht bzw. schrittweise entwickelt werden soll und die statischen Anteile eher eine untergeordnete Rolle spielen. In anderen Fällen kann Time Axis als besser geeignet angesehen werden.

2.5 Eine weitere Variante?

Im Zuge der Betrachtung der verschiedenen Positionen der Zeitdimension in den Relationen stellt sich die Frage, ob neben der ersten und letzten Position nicht auch die Mitte in Erwägung gezogen werden sollte. Auf den ersten Blick erlaubt es sogar eine intuitivere Schreibweise:

```
sig Time {}
sig Passenger {}
sig Flight {
  passengers: Time -> Passenger
}

fun flightPassengers [f: Flight, t: Time] : set Passenger {
  f.passengers[t]
}
```

Dadurch stellt sich die Relation `passengers` als `Flight -> Time -> Passenger` dar, so dass `Time` sich in der Mitte befindet. Der Ausdruck `f.passengers[t]` kann auf diese Weise sehr intuitiv als „Die Passagiere des Flugs `f` zum Zeitpunkt `t`“ gelesen werden. D. h. es kann objektorientiert betrachtet werden: `f` ist das Objekt, `passengers` die Eigenschaft mit den Passagieren als assoziatives Array, dessen Schlüssel die Zeit und Wert eine Menge von Passagieren ist.

Dem aufmerksamen Leser mag aufgefallen sein, dass bei dem vorherigen Modell das Feld `exists` fehlt. Dieses wurde nicht ohne Grund weggelassen. Denn da es eine zweistellige Relation ist (`Flight -> Time`), existiert keine Möglichkeit die Zeit in der Mitte unterzubringen. Das hat zur Folge, dass dieser Modellierungsstil nicht konsequent durchgehalten werden kann, welches den ersten kleinen Nachteil bildet.

Ein viel größerer Nachteil ist jedoch, dass es in vielen Fällen sehr unbequem ist die Zeit in der Mitte der Relation zu platzieren. Versucht man beispielsweise die Operation `bookFlight` (bzw. einen Teil der zuvor definierten) zu formulieren, lässt sich dies nur umständlich vornehmen:

```
pred bookFlight [f: Flight, p: Passenger, t,t': Time] {
  // ...
  f.passengers[t'] = f.passengers[t] + p
  (Flight - f).passengers[t'] = (Flight - f).passengers[t]
  // ...
}
```

Der Stil erlaubt es nicht einen einfachen Join zu verwenden um den Wert der Relation zu einem bestimmten Zeitpunkt, d. h. alle Passagiere zu allen Flügen, zu erhalten [1, S. 175]. Aus diesem Grund muss zunächst der Passagier zu dem Flug `f` hinzugefügt werden und anschließend festgelegt werden, dass die Passagiere anderer Flüge davon unberührt sind.

Aufgrund der erwähnten Nachteile wird der Stil in dieser Arbeit nicht im vollem Umfang als dritte operationsfokussierte Variante beschrieben.

3 Ablauffokussiert

Im Vergleich zu Global State und Time Axis stellt Trace eine fundamental andere Herangehensweise bei der Analyse von dynamischen Systemen dar.

3.1 Das Modell

Jacob beschreibt diesen Stil in [2] um Sicherheitseigenschaften eines Systems anhand der Untersuchung der Folge von Interaktionen des Systems mit seiner Umwelt sicherzustellen. Dabei beschränkt er das Modell auf die Ereignisse, die eintreten können. Im Gegensatz zu den operationsfokussierten Stilen wird keine Zeitdimension eingeführt, so dass die grundlegenden Signaturen lediglich folgende sind:

```
sig Passenger {}
sig Flight {}
```

Darauf aufbauend werden die möglichen Ereignisse „Flug erstellen“, „Flug buchen“ und „Flug stornieren“ formuliert:

```
open util/ordering[Event]

abstract sig Event {
  flight: one Flight
```

```
}
abstract sig PassengerEvent extends Event {
  passenger: one Passenger
}
sig Booking, Cancellation extends PassengerEvent {}
sig CreateFlight extends Event {}
```

Wie das Listing zeigt, werden alle Ereignisse von `Event` abgeleitet. Zusätzlich wird wiederum das ordering-Modul herangezogen um die Ereignisse in eine definierte Reihenfolge zu bringen. Da bei allen Ereignissen in diesem Modell ein Flug beteiligt ist, wird dieser als Feld in `Event` integriert. Davon abgeleitet werden die Ereignisse, bei denen ein Passagier involviert ist, von dem Ereignis der Flugerstellung unterschieden.

Die Ereignisse bzw. Operationen werden im Gegensatz zu den operationsfokussierten Stilen bei der Verwendung von Trace nicht näher beschrieben. Anstelle dessen werden gültige Folgen von Ereignissen festgelegt und darauf basierend Behauptungen aufgestellt. Für deren Definition werden zunächst einige Hilfsfunktionen formuliert:

```
fun upto [e: Event] : set Event { prevs[e] + e }
fun bookings [e: Event] : set Event {
  {b: prevs[e] & Booking |
    b.passenger = e.passenger and b.flight = e.flight}
}
fun cancellations [e: Event] : set Event {
  {s: prevs[e] & Cancellation |
    s.passenger = e.passenger and s.flight = e.flight}
}
```

Die Funktion `upto` liefert mittels der von `util/ordering` zur Verfügung gestellten Funktion `prevs` alle vorherigen Ereignisse inklusive des „aktuellen“, d. h. `e`. Die anderen Funktionen geben die Menge der bisherigen Buchungen bzw. Stornierungen, die den selben Passagier und Flug wie in `e` betreffen, zurück.

Darauf aufbauend werden vier Prädikate formuliert, mit denen sich die gültigen Folgen von Ereignissen beschränken lassen:

```
pred passengerEventNeedsFlight [] {
  all e: PassengerEvent |
    (some c: upto[e] & CreateFlight | e.flight = c.flight)
}
pred noMultipleCreates [] {
  all disj e,e': CreateFlight |
    e.flight != e'.flight
}
pred cancellationNeedsBooking [] {
  all e: Cancellation |
    #bookings[e] > #cancellations[e]
}
pred noMultipleBookings [] {
  all e: Booking |
    #bookings[e] - #cancellations[e] = 0
}
```

Das erste Prädikat wird eingesetzt um zu erwingen, dass vor Buchungen und Stornierungen stets der beteiligte Flug erstellt wurde. Das Prädikat `noMultipleCreates` sorgt dafür, dass kein Flug mehrfach erstellt werden kann. Das nächste Prädikat definiert, dass die Anzahl der Buchungen eines Passagiers und Flugs stets größer als die Anzahl der Stornierungen sein muss. Anders formuliert: Es soll nicht möglich sein einen Flug zu stornieren, der entweder noch gar nicht gebucht war oder bereits storniert wurde. Das letzte Prädikat legt fest, dass kein Flug von dem selben Passagier mehrfach gebucht werden kann. Es soll jedoch erlaubt sein, einen Flug zu buchen, wieder zu stornieren und anschließend erneut zu buchen.

3.2 Behauptungen und deren Prüfung

Zur Demonstration möglicher Prüfungen sollen nun einige beispielhafte Behauptungen aufgestellt werden:

```
assert notMoreCancellationsThanBookings {
  cancellationNeedsBooking[] =>
    all e: Event |
      (let pfx = upto[e] | #(pfx & Booking) >= #(pfx & Cancellation))
}
assert cancellationHasBookingAndFlight {
  cancellationNeedsBooking[] and passengerEventNeedsFlight[] =>
    all e: Cancellation {
      (some b: prevs[e] & Booking |
        b.passenger = e.passenger and b.flight = e.flight)
      (some b: prevs[e] & CreateFlight |
        b.flight = e.flight)
    }
}
```

Die erste Behauptung ist, dass es zu keinem Zeitpunkt mehr Stornierungen als Buchungen geben kann, wenn das Prädikat `cancellationNeedsBooking` wahr ist. Des Weiteren wird behauptet, dass jede Stornierung sowohl eine vorherige Buchung dieses Flugs als auch die vorhergehende Erstellung des betroffenen Fluges benötigt. Dies soll der Fall sein, wenn `cancellationNeedsBooking` und `passengerEventNeedsFlight` gilt. Eine Überprüfung der Behauptungen im Scope von 6 liefert das erwartete Ergebnis:

```
Executing "Check notMoreCancellationsThanBookings for 6"
Solver=minisat(jni) Bitwidth=4 MaxSeq=6 SkolemDepth=1 Symmetry=20
3678 vars. 186 primary vars. 9813 clauses. 150ms.
No counterexample found. Assertion may be valid. 11795ms.

Executing "Check cancellationHasBookingAndFlight for 6"
Solver=minisat(jni) Bitwidth=4 MaxSeq=6 SkolemDepth=1 Symmetry=20
4439 vars. 222 primary vars. 11514 clauses. 195ms.
No counterexample found. Assertion may be valid. 235ms.
```

3.3 Visualisierung

Nachfolgend soll die Visualisierung gültiger Instanzen, d. h. Folgen von Ereignissen, geschildert werden, da ohne weitere Maßnahmen die Folge im *Alloy Visualizer* gar nicht erkennbar ist:

Abbildung 4: Visualisierung ohne Maßnahmen

Jacob wendet folgenden Trick an um die Folge sichtbar zu machen [2, S. 107]:

```
fun nxt []: Event -> Event { ordering/next }
```

Die Definition dieser Funktion veranlasst den Visualizer die `next`-Relation anzuzeigen, so dass sichtbar ist, in welcher Reihenfolge die Ereignisse auftreten:

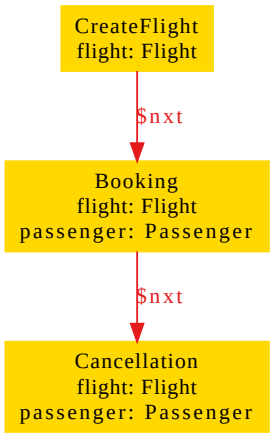

Abbildung 5: Visualisierung mit `nxt`

Die gültigen Folgen bei Berücksichtigung aller Prädikate können schließlich mit folgender Anweisung betrachtet werden:

```
run {
  passengerEventNeedsFlight[]
  noMultipleCreates[]
  cancellationNeedsBooking[]
  noMultipleBookings[]
} for 5
```

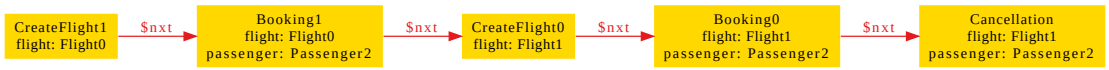

Abbildung 6: Visualisierung mit allen Prädikaten

3 Fazit

Die Vorstellung der Modellierungsstile Global State und Time Axis hat gezeigt, dass keine großen Unterschiede zwischen ihnen existieren. Beide konzentrieren sich auf die Beschreibung der Operationen selbst und platzieren lediglich die dynamischen Anteile der Modelle an unterschiedlichen Stellen. Im Gegensatz dazu beschreibt der Stil Trace nicht die Operationen, sondern gültige Folgen von Ereignissen. D. h. Modellierungsstile des Typs operationsfokussiert beschreiben Dynamik in Modellen aus einer anderen Perspektive als ablauffokussierte. Aus diesem Grund dürfte es bei der Überprüfung von Dynamik vorteilhaft sein, das System mit einem Stil aus jeder Kategorien zu beschreiben.

Wie bei dem Vergleich der operationsfokussierten Stile bereits erwähnt, erlauben es diese nicht ein existierendes statisches Modell zu benutzen um Dynamik hinzuzufügen und zu untersuchen, da das statische Modell verändert werden muss. Der Stil Trace hingegen ergänzt keine Zeitkomponente in Signaturen bzw. verschiebt keine Felder in eine neue Signatur, sondern erweitert das Modell lediglich um Ereignisse und untersucht Folgen selbiger. Aus diesem Grund wäre es bei dem gewählten Beispiel möglich, eine statische Beschreibung des Flughafens vorzunehmen und diese in einem anderen Modul zu verwenden um Dynamik mittels Trace zu untersuchen.

Beide Beschreibungsweisen erlauben eine gute Visualisierung. Bei den operationsfokussierten projiziert man üblicherweise auf die Zeitkomponente um verfolgen zu können, welche Operationen in welchem „Zeit-Übergang“ stattfinden. Bei den ablauffokussierten erlaubt der im vorherigen Abschnitt erwähnte Trick eine anschauliche Darstellung der Ereignisfolge. Bei beiden Typen wird die Visualisierung jedoch schnell unübersichtlich, wenn viele Elemente oder Zeitpunkte involviert sind. Dies kann vor allem dadurch ein Nachteil sein, dass interessante Ereignisfolgen bzw. Operationsfolgen häufig relativ lang sind (wie im Falle von Trace in [2, S. 106] erwähnt).

Literaturverzeichnis

[1] Jackson, Daniel: *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006

[2] Jacob, Jeremy L.: Trace Specifications in Alloy. In: *Abstract State Machines, Alloy, B and Z*. Springer Verlag, 2010, S. 105–117

[3] Renz, Burkhardt ; Asmussen, Nils: *Kurze Einführung in Alloy*. http://homepages.fh-giessen.de/~hg11260/mat/alloy-intro.pdf, Abruf: 04.08.2010

[4] Seater, Rob ; Dennis, Greg ; Berre, Daniel L. ; Chang, Felix: *Alloy Tutorial*. http://alloy.mit.edu/alloy4/tutorial4/, Abruf: 04.08.2010

[5] Zoth, Jochen: *Eine Fallstudie zur formalen Verifikation der Verträglichkeit voneinander abhängiger Konzepte*, Fachhochschule Gießen-Friedberg, Masterarbeit, 2010