Die Hard in Electrum

Im Film *Stirb langsam: Jetzt erst recht*, im Original *Die Hard with a Vengeance*, müssen John McClane, gespielt von Bruce Willis, und Zeus Carver, gespielt von Samuel L. Jackson, eine Bombe entschärfen. Der Zeitzünder kann deaktiviert werden, indem man ein Gefäß mit exakt 4 Gallonen Wasser auf den Auslöser stellt. Die Bombe befindet sich an einem Brunnen, an dem die beiden Akteure zwei Gefäße vorfinden. Das eine fasst genau 3 Gallonen, das andere genau 5 Gallonen. Beide Gefäße können aus dem Brunnen mit Wasser gefüllt werden. Den beiden Akteuren gelingt es recht schnell das Rätsel zu lösen und sie können den Zeitzünder stoppen. Die Szene im Film ist aber so rasant, dass die genaue Lösung des Rätsels sich einem nicht unmittelbar erschließt.

Leslie Lamport verwendet dieses Umfüllrästel in einem exzellenten, unbedingt sehenswerten Video seines Video-Kurses zu TLA$^+$, um zu zeigen, wie man Spezifikation in TLA$^+$ schreibt und sie mit dem Modelchecker TLC überprüft.

Die Spezifikationssprache Alloy mit ihrem interaktiven Werkzeug, dem *Alloy Analyzer* erlaubt es auch Systeme zu beschreiben und Modelle zu finden, oder auch Gegenbeispiele, wenn bestimmte erwartete Prädikate nicht erfüllt sind. Electrum ist eine Erweiterung von Alloy um Ausdrücke temporaler Logik. In Alloy muss man Zustandsübergange durch explizite zusätzliche Typen etwa `Time` spezifizieren. In Electrum ist dies eleganter möglich. Man kann Typen und Relationen in Electrum als `var` kennzeichnen. Solche Elemente können zu unterschiedlichen Zeitpunkten variieren. Dabei wird der Beistrich `'` verwendet, um einen Ausdruck im Folgezustand zu benennen.

Das Beispiel des Umfüllrätsel aus *Die Hard* wird uns helfen, Electrum zu verstehen und anwenden zu können.

Wir wollen uns zuerst die Lösung von Leslie Lamport in TLA$^+$ kurz vergegenwärtigen, um sie dann in die Sprache von Electrum recht direkt zu übertragen. Das Ergebnis ist valide, aber nicht in dem Stil, in dem man die Spezifikation und die Ermittlung in Electrum „idiomatisch“ machen würde. Deshalb verbessern wir die erste Lösung schrittweise zu einer eleganteren Fassung.

Die Hard in TLA$^+$

Die Spezifikation

Im Modul `diehard` verwenden wir das Modul `Integers`. Zwei Variablen `small` und `big` geben den Füllstand der beiden Gefäße an.

```
------------------------------ MODULE diehard ------------------------------

EXTENDS Integers

VARIABLES small, big
```

TLA$^+$ hat keine Typen, was bedeutet, dass unsere beiden Variablen Werte beliebiger Typen haben könnten. Deshalb gibt man in TLA$^+$ typischerweise ein Prädikat vor, das die Typinformation spezifiziert und in jedem Zustand eines Modells der Spezifikation invariant sein sollte.

In unserem Falle gibt das Prädikat `TypeOK` an, dass die beiden Variablen ganzzahlige Werte in den Bereichen $0 \ldots 3$ bzw. $0 \ldots 5$ haben.

```
TypeOK == /\ small \in 0..3
          /\ big   \in 0..5
```

Das Denkmodell von TLA$^+$ geht davon aus, dass wir ein dynamisches System spezifizieren, das einen Anfangszustand hat, der durch das Prädikat `Init` spezifiziert wird. Ferner spezifiziert man ein Prädikat `Next`, das die Übergänge zwischen den Zuständen festlegt.

```
Init == /\ big   = 0
        /\ small = 0
```

Im Anfangszustand sind beide Gefäße leer. Für die Übergänge zwischen den Zuständen müssen wir alle dabei auftretenden Möglichkeiten berücksichtigen, also z.B. dass ein Gefäß gefüllt wird oder geleert wird oder in ein anderen Gefäß umgefüllt wird. Für einen Zustandsübergang ist jede dieser Möglichkeiten denkbar und erlaubt. Wir arbeiten also auf folgendes Prädikat zu:

```
Next == \/ FillSmall
        \/ FillBig
        \/ EmptySmall
        \/ EmptyBig
        \/ SmallToBig
        \/ BigToSmall
```

Allerdings können wir es erst formulieren, wenn wir die einzelnen Prädikate der Disjunktion spezifiziert haben.

In jedem der folgenden Prädikate gibt das Beistrich an, dass eine Variable in einem Folgezustand gemeint ist.

Das Füllen und das Leeren der beiden Gefäße ist leicht zu spezifizieren. Man muss nur darauf achten, dass man nicht nur definiert, was sich ändert, sondern auch was gleich bleibt. Man nennt dies auch die *frame condition*.

```
FillSmall == /\ small' = 3
             /\ big'   = big

FillBig == /\ big'   = 5
           /\ small' = small

EmptySmall == /\ small' = 0
              /\ big'   = big

EmptyBig == /\ big'   = 0
            /\ small' = small
```

Interessanter ist die Definition der Übergänge, wenn Wasser aus den beiden Gefäßen umgefüllt wird, denn dann muss man nachprüfen, wieviel in das zweite Gefäß passt und wieviel im ersten Gefäß übrig bleibt.

```
SmallToBig == IF big + small =< 5
               THEN /\ big'   = big + small
                    /\ small' = 0
               ELSE /\ big'   = 5
                    /\ small' = small - (5 - big)

BigToSmall == IF big + small =< 3
               THEN /\ big'   = 0
                    /\ small' = big + small
               ELSE /\ big'   = small - (3 - big)
                    /\ small' = 3
```

Insgesamt bekommen wir also die folgende Spezifikation:

```
------------------------------ MODULE diehard ------------------------------

EXTENDS Integers

VARIABLES small, big

TypeOK == /\ small \in 0..3
          /\ big   \in 0..5

Init == /\ big   = 0
        /\ small = 0


FillSmall == /\ small' = 3
             /\ big'   = big

FillBig == /\ big'   = 5
           /\ small' = small

EmptySmall == /\ small' = 0
              /\ big'   = big
```

```
EmptyBig == /\ big'   = 0
            /\ small' = small

SmallToBig == IF big + small =< 5
              THEN /\ big'   = big + small
                   /\ small' = 0
              ELSE /\ big'   = 5
                   /\ small' = small - (5 - big)

BigToSmall == IF big + small =< 3
              THEN /\ big'   = 0
                   /\ small' = big + small
              ELSE /\ big'   = small - (3 - big)
                   /\ small' = 3

 Next == \/ FillSmall
         \/ FillBig
         \/ EmptySmall
         \/ EmptyBig
         \/ SmallToBig
         \/ BigToSmall
================================================================================
```

Leslie Lamport, der Entwickler von LaTeX, hat zu TLA$^+$ natürlich einen *pretty printer* geschrieben, der Spezifikationen via LaTeX produzieren kann. Man erhält in unserem Beispiel die in Abbildung 1 abgebildete Spezifikation.

Model Checking

Die Spezifikation selbst gibt uns noch keine Lösung des Rätsel. In der TLA$^+$ Toolbox kann TLC, der Model Checker von TLA$^+$, aufgerufen werden. Dazu erzeugt man ein neues Modell. In unserem Fall muss man gar nichts weiter tun. Denn in dieser einfachen Spezifikation kommen gar keine Mengen von Daten vor, die wir für ein Modell vorgeben müssten. Auch das Prädikat für den Initialzustand heißt bei uns wie vorgegeben `Init` und die Übergangsrelation `Next`. Also können wir zunächst mal die Prüfung der Typkonformität machen. Dazu tragen wir im Feld Invariants das Prädikat `TypeOK` ein und lassen den Model Checker prüfen, ob es in allen Zuständen erfüllt ist.

Nun können wir das Rätsel lösen. Der Trick besteht darin, dass wir ein Gegenbeispiel provozieren. Wir definieren die Invariante `big /= 4`, die aussagt, dass niemals 4 Gallonen Wasser im größeren Gefäß sein können. Der Model Checker überprüft nun diese Invariante und wenn sie *nicht* erfüllt ist, gibt er einen Ablauf an, der zu dem Zustand führt, in dem 4 Gallonen Wasser im Gefäß sind. Das ist aber dann genau die Lösung des Rätsels.

Der Error-Trace ergibt eine Folge von Aktionen, die Bruce Willis eingefallen ist, um die 4 Gallonen abzumessen wie in Abbildung 2 angegeben (dabei ist der obere Wert die Menge im größeren Gefäß).

MODULE *diehard*

EXTENDS *Integers*

VARIABLES *small*, *big*

$$
\begin{aligned}
TypeOK \triangleq\ & \land small \in 0 \mathinner{..} 3 \\
& \land big \in 0 \mathinner{..} 5
\end{aligned}
$$

$$
\begin{aligned}
Init \triangleq\ & \land big = 0 \\
& \land small = 0
\end{aligned}
$$

$$
\begin{aligned}
FillSmall \triangleq\ & \land small' = 3 \\
& \land big' = big
\end{aligned}
$$

$$
\begin{aligned}
FillBig \triangleq\ & \land big' = 5 \\
& \land small' = small
\end{aligned}
$$

$$
\begin{aligned}
EmptySmall \triangleq\ & \land small' = 0 \\
& \land big' = big
\end{aligned}
$$

$$
\begin{aligned}
EmptyBig \triangleq\ & \land big' = 0 \\
& \land small' = small
\end{aligned}
$$

$$
\begin{aligned}
SmallToBig \triangleq\ & \text{IF } big + small \leq 5 \\
& \text{THEN } \land big' = big + small \\
& \qquad\ \land small' = 0 \\
& \text{ELSE } \land big' = 5 \\
& \qquad\ \land small' = small - (5 - big)
\end{aligned}
$$

$$
\begin{aligned}
BigToSmall \triangleq\ & \text{IF } big + small \leq 3 \\
& \text{THEN } \land big' = 0 \\
& \qquad\ \land small' = big + small \\
& \text{ELSE } \land big' = small - (3 - big) \\
& \qquad\ \land small' = 3
\end{aligned}
$$

$$
\begin{aligned}
Next \triangleq\ & \lor FillSmall \\
& \lor FillBig \\
& \lor EmptySmall \\
& \lor EmptyBig \\
& \lor SmallToBig \\
& \lor BigToSmall
\end{aligned}
$$

* Modification History
* Last modified *Fri Jun* 25 14:11:32 *CEST* 2021 by *br*
* Created *Fri Jun* 25 13:54:46 *CEST* 2021 by *br*

Abbildung 1: Das Modul *diehard*

$$
\begin{bmatrix} 0 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 5 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 2 \\ 3 \end{bmatrix} \rightarrow \begin{bmatrix} 2 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 \\ 2 \end{bmatrix} \rightarrow \begin{bmatrix} 5 \\ 2 \end{bmatrix} \rightarrow \begin{bmatrix} 4 \\ 3 \end{bmatrix}
$$

Abbildung 2: Des Rätsels Lösung

Portierung der Spezifikation in TLA$^+$ nach Electrum

In diesem Abschnitt wollen wir die Spezifikation des Rätsels und seine Lösung aus TLA$^+$ möglichst direkt in eine Spezifikation in Electrum übertragen. Es wird sich zeigen, dass sie ganz ähnlich zum Original ist, jedoch nicht entsprechend eines „idiomatischen“ Stils in Electrum. Deshalb werden wir im Anschluss eine „schönere“ Version entwickeln.

Unsere Spezifikation beginnt durch die Angabe des Namens des Moduls und dem Importieren des Moduls `util/integer`.

```
module mfc/diehard

open util/integer
```

Bei der Verwendung des Moduls `util/integer` muss man Vorsicht walten lassen. Alloy hat den eingebauten Typ `Int`, der ganze Zahlen „emuliert“. Per Default verwendet Alloy Zweierkomplement-Repräsentation mit 4 Bits, d.h. den Bereich $-8 \ldots 7$. Für unser Rätsel reicht das nicht aus, weshalb wir diesen Bereich werden erweitern müssen. Außerdem ist `Int` wie jeder andere Typ in Alloy eine *Menge* und der Operator `+` ist die Mengenvereinigung. Man kann dies leicht im *Evaluator* des *Alloy Analyzer*s ausprobieren und wird sehen, dass das Ergebnis von $2+3$ die Menge ist, die die beiden Zahlen enthält, und somit ist $1 + 1 = 1$ wahr! Für arithmetische Operatoren muss man also Funktionen wie `add` verwenden, also `add[1, 1]` oder `1.add[1]`. Beide Ausdrücke ergeben die einelementige Menge, die die Zahl 2 enthält.

Wir definieren nun eine Signatur mit zwei Relationen, die die beiden Gefäße repräsentieren werden.

```
one sig State {
  var Small: Int,
  var Big: Int
}
```

`State` repräsentiert den Zustand in dem wir jeweils eine Relation zu genau einer Zahl für das kleine und das große Gefäß vorsehen. `State` selbst wird mit `one` in jedem Modell genau einmal instantiiert.

Leslie Lamport hat in seiner Spezifikation ein Prädikat für die Typüberprüfung, die im Model Checker als Invariante überprüft wird. Wir können dies in Electrum in folgender Weise nachbauen:

```
assert typeOK {
  always {
    State.Small >= 0 and State.Small <= 3
    State.Big >= 0 and State.Big <= 5 }
}
```

Die angegebenen Integritätsbedingungen sollen in jedem Zustand gelten, `TypeOK` ist also eine Invariante.

Nun definieren wir die Prädikate `init`, `fillSmall`, `emptySmall` sowie `fillBig` und `emptyBig` ganz analog zu Lamports Vorlage:

```
pred init {
  State.Small = 0
  State.Big = 0
}

pred fillSmall {
  State.Small' = 3
  State.Big' = State.Big
}

pred emptySmall {
  State.Small' = 0
  State.Big' = State.Big
}

pred fillBig {
  State.Big' = 5
  State.Small' = State.Small
}

pred emptyBig {
  State.Big' = 0
  State.Small' = State.Small
}
```

Bei den Prädikaten, die das Umfüllen von Wasser zwischen den Gefäßen spezifizieren, müssen wir die Funktionen für Integers von Alloy verwenden:

```
pred smallToBig {
  let amount = add[State.Big, State.Small] |
    amount <= 5 implies
      (State.Small' = 0 and State.Big' = amount)
    else
      (State.Big' = 5 and State.Small' = sub[State.Small, sub[5, State.Big]])
}

pred bigToSmall {
  let amount = add[State.Big, State.Small] |
    amount <= 3 implies
      (State.Big' = 0 and State.Small' = amount)
    else
      (State.Small' = 3 and State.Big' = sub[State.Small, sub[3, State.Big]])
}
```

Nun können wir den kompletten Ablauf durch ein `fact` in Alloy formulieren, dies entspricht der Festlegung des Initialzustands und der Übergangsrelation `Next` in TLA$^+$:

```
fact traces {
  init
  always {
    fillSmall or fillBig or
```

```
    emptySmall or emptyBig or
    smallToBig or bigToSmall
  }
}
```

Wir wollen nun die Typüberprüfung durchführen und danach das Rätsel lösen durch ein Gegenbeispiel zur Annahme, dass das große Gefäß niemals 4 Gallonen Wasser enthält.

```
check typeOK for 5 int, 7 steps

assert BruceWillDie {
  always not State.Big = 4
}

check BruceWillDie for 5 int
```

Ein Unterschied zur Typprüfung in TLA$^+$ ist zu beachten: Wir haben in Electrum die Überprüfung für Abläufe mit maximal 7 Schritten gemacht. In unserem Beispiel können wir dabei davon ausgehen, dass damit schon alle auftretenden Möglichkeiten einmal überprüft wurden, im Allgemeinen ist dies aber nicht der Fall. Electrum führt *bounded model checking* durch.

Doch nun zur Lösung des Rätsel: Wie in der Vorlage von Leslie Lamport formulieren wir eine Annahme, die zu einem Gegenbeispiel führt und dieses Gegenbeispiel ist dann genau ein Ablauf, der zeigt, wie man 4 Gallonen Wasser in das größere Gefäß füllen kann.

Wenn wir `check BruceWillDie for 5 int` ausführen, dann erhalten wir folgende Meldung

```
Executing "Check BruceWillDie for 5 int"
   Solver=sat4j Steps=1..10 Bitwidth=5 MaxSeq=4 SkolemDepth=4 Symmetry=20 Mode=batch
   1..7 steps. 48869 vars. 1848 primary vars. 162126 clauses. 633ms.
   Counterexample found. Assertion is invalid. 445ms.
```

Auf `Counterexample` kann man klicken und das Gegenbeispiel wird angezeigt. In Abb. 3 werden die ersten zwei Zustände des Ablaufs dargestellt. Die erste Aktion ist also das Füllen des großen Gefäßes. Man kann im *Alloy Analyzer* mit dem Button, der mit $\rightarrow$ beschriftet ist den Ablauf durchlaufen und erhält damit die Lösung.

Zum Abschluss dieses Abschnitts hier die vollständige Spezifikation in Electrum:

```
module mfc/diehard

open util/integer

one sig State {
  var Small: Int,
  var Big: Int
}
```

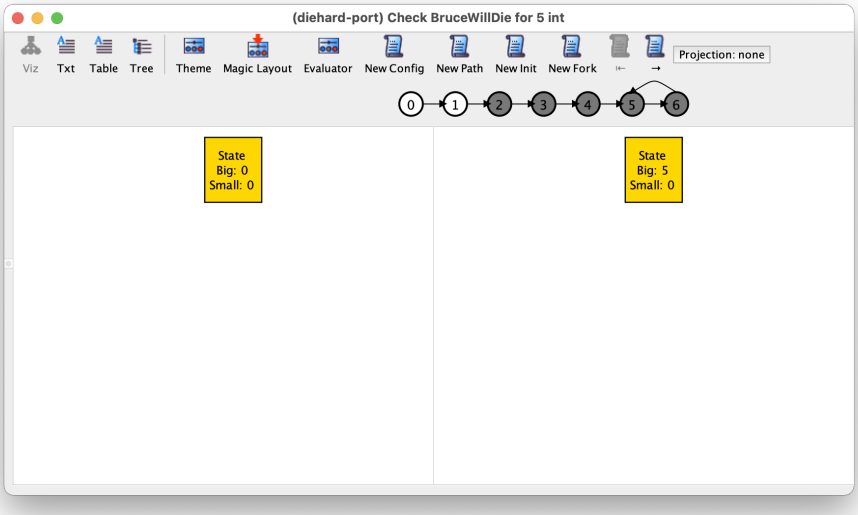

Abbildung 3: Gegenbeispiel – des Rätsels Lösung

```
assert typeOK {
  always {
    State.Small >= 0 and State.Small <= 3
    State.Big >= 0 and State.Big <= 5 }
}

pred init {
  State.Small = 0
  State.Big = 0
}

pred fillSmall {
  State.Small' = 3
  State.Big' = State.Big
}

pred emptySmall {
  State.Small' = 0
  State.Big' = State.Big
}

pred fillBig {
  State.Big' = 5
  State.Small' = State.Small
}

pred emptyBig {
```

```
  State.Big' = 0
  State.Small' = State.Small
}

pred smallToBig {
  let amount = add[State.Big, State.Small] |
    amount <= 5 implies
      (State.Small' = 0 and State.Big' = amount)
    else
      (State.Big' = 5 and State.Small' = sub[State.Small, sub[5, State.Big]])
}

pred bigToSmall {
  let amount = add[State.Big, State.Small] |
    amount <= 3 implies
      (State.Big' = 0 and State.Small' = amount)
    else
      (State.Small' = 3 and State.Big' = sub[State.Small, sub[3, State.Big]])
}

fact traces {
  init
  always {
    fillSmall or fillBig or
    emptySmall or emptyBig or
    smallToBig or bigToSmall
  }
}

check typeOK for 5 int, 7 steps

assert BruceWillDie {
  always not State.Big = 4
}

check BruceWillDie for 5 int
```

Im folgenden Abschnitt wollen wir die Spezifikation etwas eleganter machen.

Die Hard in Electrum

Die direkte Übertragung der Spezifikation von Leslie Lamport lässt sich in Electrum in mancher Hinsicht verbessern. Tatsächlich hat Leslie Lamport in seinem Hyperbook eine Lösung, die der folgenden sehr ähnelt. Peter Kriens und David Chemouil haben mir einige Tipps für diese verbesserte Version gegeben.

Unmittelbar fällt auf, dass die Definition der Prädikate für das Füllen und Leeren der Gefäße bis auf die Menge des Wassers identisch sind. Diese

Redundanz verschwindet, wenn man Gefäße selbst als Typ verwendet und ihnen eine maximale Füllmenge `capacity` zusätzlich zum tatsächlichen Füllstand `gallons` gibt.

```
module mfc/diehard

open util/integer

abstract sig Jug {
  var gallons : Int,
  capacity : Int
}

one sig Jug3 extends Jug {}{ capacity = 3 }
one sig Jug5 extends Jug {}{ capacity = 5 }
```

Nun kann man zum Beispiel für das Füllen folgendes Prädikat nehmen:

```
pred fill[j: Jug] {
  j.gallons' = j.capacity
  all unchanged : Jug-j | unchanged.gallons' = unchanged.gallons
}
```

Man kann aber mit der *frame condition* noch eleganter umgehen, indem man den Operator `++` verwendet:

```
pred fill[j: Jug] {
  gallons' = gallons ++ j->j.capacity
}

pred empty[j: Jug] {
  gallons' = gallons ++ j->0
}
```

`gallons` ist eine Relation `Jug` × `Int`, tatsächlich sogar eine Funktion und `j->j.capacity` ein Tupel. Der Operator `++` überschreibt das vorhandene Tupel für `j` mit dem neuen.

Was das Umfüllen angeht, hatten wir eine Fallunterscheidung, ob alles oder nur ein Teil des Wassers in einem Gefäß in das andere gefüllt wird. Wir können aber auch einfach berechnen, wieviel es ist: Nämlich alles oder soviel wie Platz ist, welche Zahl immer kleiner ist. Also:

```
pred pour[from, to: Jug] {
  let amount = min[to.capacity.minus[to.gallons] + from.gallons] {
    from.gallons' = from.gallons.minus[amount]
    to.gallons' = to.gallons.plus[amount]
  }
}
```

Man beachte hier, dass der Operator `+` die Vereinigung ist und `min` als Argument eine Menge von Zahlen erwartet.

Nun ist es einfach, das System vollständig zu definieren. Wir brauchen noch den Initialzustand und die Übergangsrelation:

```
pred init {
  all j: Jug | j.gallons = 0
}

fact traces {
  init
  always {
    one j: Jug | fill[j] or empty[j] or pour[j, Jug-j]
  }
}
```

Da wir in Electrum ja Typen haben, können wir auf die Typprüfung verzichten. Der Code ist so angelegt, dass wir uns leicht überzeugen, dass niemals das Fassungsvermögen eines Gefäßes überschritten wird.

Außerdem müssen wir für die Lösung des Rätsels gar keinen Widerspruchsbeweis machen, weil uns Electrum ja einen Ablauf „finden“ kann, der nicht nur bestimmte strukturelle Eigenschaften hat, sondern auch temporale Formeln erfüllt. Wir suchen also nach einem Ablauf, in dem schließlich 4 Gallonen Wasser im größeren Gefäß sind:

```
run {
  eventually Jug5.gallons = 4
}
```

Wir kommen hier auch mit dem Scope von 4 für die Bits der Integers aus, weil wir keine Zahlen größer 5 mehr verwenden.

Mit einem geeignet eingestellten visuellen *Theme* sehen die letzten beiden Zustände der Lösung aus wie iun Abb. 4 abgebildet.

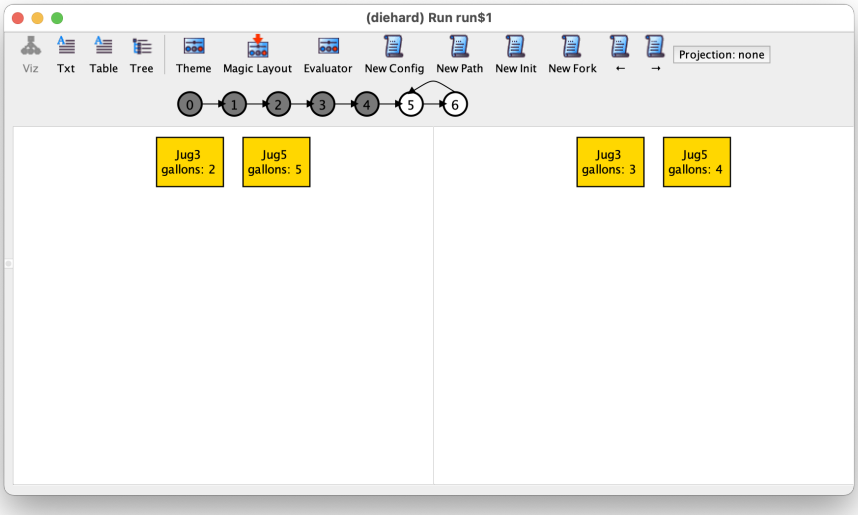

Abbildung 4: Die letzten beiden Zustände der Lösung

Hier nochmals die komplette Spezifikation:

```
module mfc/diehard

open util/integer

abstract sig Jug {
  var gallons : Int,
  capacity : Int
}

one sig Jug3 extends Jug {}{ capacity = 3 }
one sig Jug5 extends Jug {}{ capacity = 5 }

pred fill[j: Jug] {
  gallons' = gallons ++ j->j.capacity
}

pred empty[j: Jug] {
  gallons' = gallons ++ j->0
}

pred pour[from, to: Jug] {
  let amount = min[to.capacity.minus[to.gallons] + from.gallons] {
    from.gallons' = from.gallons.minus[amount]
    to.gallons' = to.gallons.plus[amount]
  }
}

pred init {
  all j: Jug | j.gallons = 0
}

fact traces {
  init
  always {
    one j: Jug | fill[j] or empty[j] or pour[j, Jug-j]
  }
}

run {
  eventually Jug5.gallons = 4
}
```