Die Türme von Hanoi in TLA$^+$

Burkhardt Renz

01.05.2020

1 Das Spiel

Das Spiel „Türme von Hanoi“ besteht aus drei gleich großen Stäben, den Türmen. Auf die Türmen können mehrere gelochte Scheiben gelegt werden, die alle verschieden groß sind. Zu Beginn liegen alle Scheiben auf dem linken Stab, und zwar der Größe nach geordnet, die kleinste Scheibe oben.

Ziel des Spiels ist es, den kompletten Scheiben-Stapel vom linken Turm auf den rechten Turm zu versetzen. Dabei darf bei jedem Zug die oberste Scheibe eines Turms auf einen anderen Turm befördert werden, allerdings so, dass jeder Turm immer der Größe nach geordnet ist.

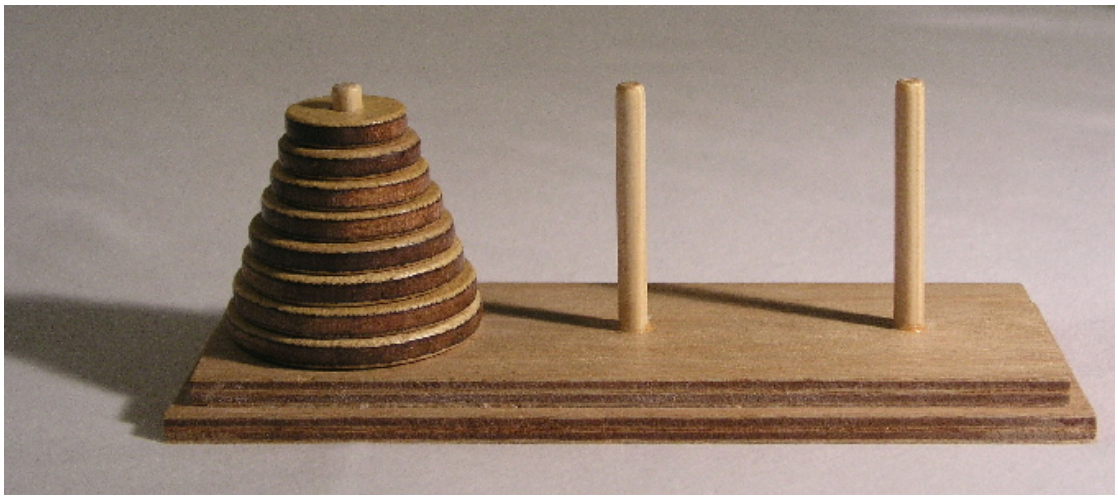

Abbildung 1: Türme von Hanoi, from https://commons.wikimedia.org/wiki/File:Tower_of_Hanoi.jpeg

2 Die Spezifikation des Spiels in TLA$^+$

Die Grundidee von TLA$^+$ besteht darin, ein System als eine diskrete Folge von Zuständen zu sehen. Es gibt einen Initialzustand und dann Folgezustände. Das System hat ferner Variablen, die in jedem Zustand bestimmte Werte haben. Bei jedem Zustandsübergang ändern sich die Werte von Variablen, andere können aber auch gleich bleiben.

Das Konzept in den Worten von Leslie Lamport:

> "… we represent the execution of a system as a sequence of states. Formally, we define a *behavior* to be a sequence of states, where a state is an assignment of values to variables. We specify a system by specifying a set of possible behaviors—the ones representing a correct execution of the system." [1, S. 15]

Die Spezifikation des Systems besteht nun darin, dass man zwei logische Formeln beschreibt und dann die Konjunktion dieser beiden Formeln bildet:

- Die erste Formel, üblicherweise `INIT` genannt, drückt aus, was im Initialzustand gelten muss.
- Die zweite Formel, üblicherweise `NEXT` genannt, gibt an, welche Bedingungen für einen Zustandswechsel gelten. Dazu muss man natürlich die Möglichkeit haben, den Wert einer Variablen *nach* dem Zustandsübergang zu formulieren. Dazu wird eine (in der Logik nicht unübliche) Konvention verwendet: Ist `v` der Wert der Variable `v`, dann ist `v'` ihr Wert nach dem Zustandsübergang.

Spezifikationen in TLA$^+$ werden in Module aufgeteilt. Für unsere Spezifikation der „Türme von Hanoi" erzeugen wir ein neues Modul mit dem Namen `hanoi1`. Die Inhalte des Moduls werden zwischen seinem Kopf und seinem Fuß geschrieben:

Listing 1: Modul-Kopf und -Fuß

```
---------------------------- MODULE hanoi1 ----------------------------
...
============================================================================
```

Dabei müssen mindestens vier der Zeichen - bzw. = aufeinander folgen.

TLA$^+$ hat eine Bibliothek von Standard-Modulen, die man wie natürlich auch selbst geschriebene Module verwenden kann. Wir verwenden `Integers` und `Sequences`.

Das Modul `Integers` enthält die üblichen numerischen und Vergleichsoperatoren für ganzzahlige Werte. Das Modul `Sequences` spezifiziert Operatoren für endliche Sequenzen.[1] Sequenzen werden wir für die Türme einsetzen und die Scheiben repräsentieren wir durch Zahlen.

[1] In [1, Kap. 18, S. 344ff] werden die Standard-Module im Detail beschrieben. Einen kurzen Überblick findet man im TLA$^+$ Cheat Sheet, das man aus der TLA$^+$ Toolbox aufrufen kann.

Listing 2: Modul verwendet Standard-Module

```
---------------------------- MODULE hanoi1 ----------------------------

EXTENDS Integers, Sequences
...
=============================================================================
```

`EXTENDS` führt dazu, dass alle Deklarationen, Definition, Annahmen und Theoreme der genannten Module „eingeblendet“ werden und verwendet werden können, so als ob sie hier selbst spezifiziert worden wären.

Das Datenmodell:

- Wir nehmen drei Variablen für jeden der Türme links, in der Mitte und rechts. Wir nennen sie `l, m, r`.
- Jeder Turm ist eine Sequenz, die die Scheiben enthält. Man schreibt Sequenzen in TLA$^+$ in der Form `<< ... >>`.
- Die Scheiben werden durch die Zahlen 1 bis 4 repräsentiert, die zugleich für die Größe der Scheiben stehen. Im Initalzustand sind alle Scheiben der Größe nach geordnet auf dem linken Turm.

Listing 3: Das Datenmodell und zugleich der Initalzustand

```
VARIABLES l, m, r

Init == /\ l = << 1, 2, 3, 4>>
        /\ m = << >>
        /\ r = << >>
```

Es folgen zwei Definitionen, die später die Spezifikation vereinfachen. Das Prädikat `Empty(seq)` ist wahr, wenn die Sequenz `seq` leer ist. Der Ausdruck `Cons(elem,seq)` steht für die Sequenz, die `elem` als Kopf und `seq` als Rest hat. In der Definition ist `<< elem >>` die Sequenz, die als erstes und einziges Element `elem` enthält, und `\o` ist der Operator im Modul `Sequences`, der die Konkatenation der beiden Sequenzen bildet.

Listing 4: Hilfs-Definitionen

```
Empty(seq) == Len(seq) <= 0

Cons(elem, seq) == << elem >> \o seq
```

Ein erlaubter Zustandsübergang im Spiel „Die Türme von Hanoi“ ist ein Zug, der den Regeln entspricht. Die Spezifikation eines Zugs wird in zwei Definitionen aufgeteilt. Zuerst das Prädikat `CanMove(t1, t2)`, das für zwei Türme angibt, ob es nach den Regeln möglich wäre, die oberste

Scheibe von `t1` auf den Turm `t2` zu legen. Die zweite Definition `Move(t1, t2, t3)` spezifiziert den eigentlichen Zug.

Listing 5: Spezifikation eines Zugs

```
CanMove(t1, t2) ==
  /\ ~ Empty(t1)
  /\ Empty(t2) \/ (~ Empty(t2) /\ Head(t1) < Head(t2))

Move(t1, t2, t3) ==
  /\ CanMove(t1, t2)
  /\ t1' = Tail(t1) /\ t2' = Cons(Head(t1), t2) /\ t3' = t3
```

Wann darf man eine Scheibe von `t1` nach `t2` verschieben? Die erste notwendige Bedingung dafür ist, dass `t1` nicht leer ist. Die zweite Bedingung lautet: `t2` ist leer oder die oberste Scheibe von `t1` ist kleiner als die oberste Scheibe von `t2`.

TLA$^+$ hat eine spezielle Notation für Konjunktionen und Disjunktionen, in dem man die Operatoren genau untereinander schreibt. Dadurch kann man Klammern sparen.

Wie sieht nun ein Zug aus? Er ist nur möglich, wenn `t1` und `t2` das Prädikat `CanMove` erfüllen. Ferner bleibt von `t1` nur der Rest ohne Kopf, während `t2` nun den Kopf von `t1` als Kopf hat. Der Zustandsübergang wird dadurch ausgedrückt, dass die Werte `t1', t2'` der Variablen `t1, t2` im Folgezustand spezifiziert wird. Man beachte, dass man außerdem ausdrücken muss, dass alles andere im Folgezustand unverändert bleibt. Hier also, dass der dritte Turm `t3` im Folgezustand seinen Wert behält.[2]

Die Bedingung `Next` besteht nun einfach in allen möglichen Zügen. D.h. wir bilden die Disjunktion der Züge für die sechs Permutationen der drei Türme. Die vollständige Spezifikation `Spec` wird dann aus `Init` und `Next` gebildet.

Listing 6: Züge und die vollständige Spezifikation

```
Next ==
  \/ Move(l, m, r) \/ Move(l, r, m)
  \/ Move(m, l, r) \/ Move(m, r, l)
  \/ Move(r, l, m) \/ Move(r, m, l)
(* Das sind alle möglichen Zustandswechsel nach Init *)

Spec == Init /\ [][Next]_<<l, m, r>>
```

[2] Dass nicht nur formuliert werden muss, was sich ändert, sondern auch was alles gleich bleibt, wird gerne vergessen. Insbesondere mit einem Hintergrund von Programmiersprachen neigt man dazu anzunehmen, dass gewissermaßen implizit ohnehin sich nur das „ändern“ kann, was man in einer Funktion formuliert. In logischen Sprachen ist diese Annahme nicht richtig. Ein Zustandsübergang muss *vollständig* formuliert werden. Die nötigen Aussagen darüber, was sich nicht ändert, wird auch als *frame condition* bezeichnet. TLA$^+$ hat einen speziellen Operator `UNCHANGED e`, mit dem man festlegt, dass sich die Variable `e` beim Zustandsübergang nicht ändert.

In der Definition von `Spec` kommt der temporale Operator `[]` vor, dessen Semantik so definiert ist: Für eine Formel ϕ bedeutet $\Box\,\phi$, dass sie in *allen* Zuständen gültig ist.

Darüber hinaus hat `Next` ein Subskript `<<l, m, r>>`. Dies bedeutet in TLA$^+$, dass auch Übergänge erlaubt sind, in denen sich die genannten Variablen überhaupt nicht ändern.[3] Dies wird generell in Spezifikationen in TLA$^+$ so gemacht. Das spezifizierte System kann dann gewissermaßen immer weiter laufen. Zweitens aber erlaubt es die Komposition von Spezifikationen. Wenn in unserer Spezifikation Übergänge erlaubt sind, bei denen sich unsere Variablen nicht ändern, könnte eine andere Spezifikation der Komposition ihre Variablen während solcher Schritte ändern.

3 Lösung des Spiels mit Hilfe des TLC Model Checkers

Für TLA$^+$ gibt es einen Model Checker, mit dem Aussagen über Modelle, die die Spezifikation erfüllen, überprüft werden können.

Den Model Checker TLC kann man verwenden, um eine Lösung für das Spiel zu erzeugen. Dazu formulieren wir zuerst das Ziel `Goal` das Spiels. Wenn wir nun die logische Negation des Spiels bilden und sie als Invariante des Modells betrachten, dann würde das bedeuten, dass es niemals einen Zustand geben kann, in dem das Ziel erfüllt ist. Dies ist gerade das Gegenteil einer Lösung. Wir lassen den Model Checker also einen *Widerspruchsbeweis* führen. Er soll feststellen, dass es *doch* möglich ist, in einen Zustand zu kommen, in dem alle Scheiben sortiert auf dem rechten Turm liegen. Findet er einen solchen Zustand und alle Schritte zu diesem Zustand, dann haben wir eine Lösung für die „Türme von Hanoi“.

Listing 7: Ziel des Spiels und seine Negation

```
Goal == r = <<1, 2, 3, 4>>

Check == \lnot Goal
```

Gesagt, getan: Im Menü TLC Model Checker>New Model der TLA$^+$ Toolbox erzeugen wir ein Modell. Es ist dann als behavior spec bereits unsere Spezifikation `Spec` eingetragen. Man muss jetzt noch die zu prüfende Invariante im Feld Invariants eintragen, also `Check`.

Dann starten wir den Model Checker mit dem Menüpunkt TLC Model Checker>Run Model.

Und in der Tat: Der Model Checker meldet einen Fehler – Invariant Check is violated – und gibt die Folge der Zustände aus, die zur Verletzung der Invariante geführt hat – den Error-Trace. In Abbildung 2 wird ein Ausschnitt der Ausgabe des Model Checkers dargestellt.

Der TLC Model Checker hat die Lösung in Tabelle 1 gefunden:

[3] Leslie Lamport nennt dies *stuttering steps* [1, S. 17]

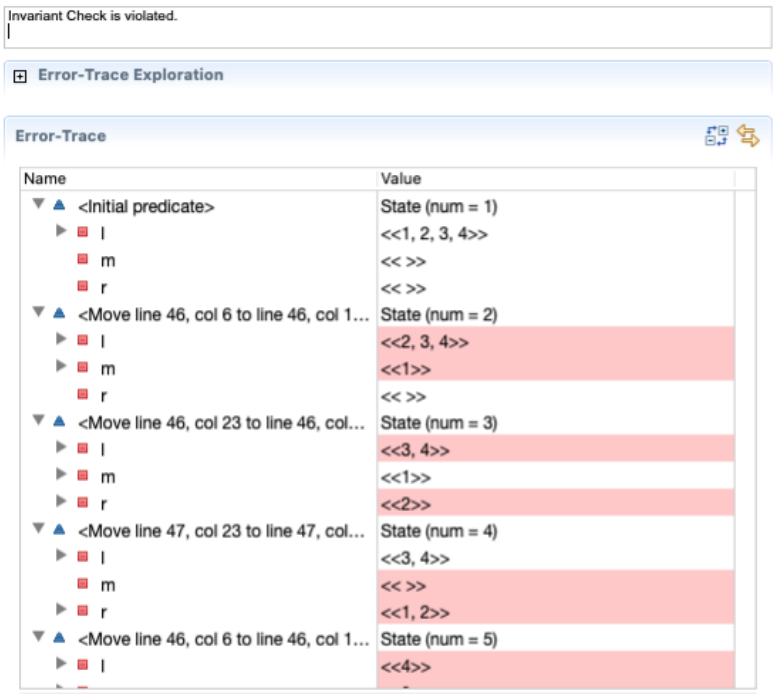

Abbildung 2: Error-Trace, d.h. Lösung des Spiels

Tabelle 1: Lösung des Spiels

Zustand	links	mitte	rechts
1	⟨1, 2, 3, 4⟩	⟨ ⟩	⟨ ⟩
2	⟨2, 3, 4⟩	⟨1⟩	⟨ ⟩
3	⟨3, 4⟩	⟨1⟩	⟨2⟩
4	⟨3, 4⟩	⟨ ⟩	⟨1, 2⟩
5	⟨4⟩	⟨3⟩	⟨1, 2⟩
6	⟨1, 4⟩	⟨3⟩	⟨2⟩
7	⟨1, 4⟩	⟨2, 3⟩	⟨ ⟩
8	⟨4⟩	⟨1, 2, 3⟩	⟨ ⟩
9	⟨ ⟩	⟨1, 2, 3⟩	⟨4⟩
10	⟨ ⟩	⟨2, 3⟩	⟨1, 4⟩
11	⟨2⟩	⟨3⟩	⟨1, 4⟩
12	⟨1, 2⟩	⟨3⟩	⟨4⟩
13	⟨1, 2⟩	⟨ ⟩	⟨3, 4⟩
14	⟨2⟩	⟨1⟩	⟨3, 4⟩
15	⟨ ⟩	⟨1⟩	⟨2, 3, 4⟩
16	⟨ ⟩	⟨ ⟩	⟨1, 2, 3, 4⟩

4 Die Spezifikation des Spiels mit $^+$CAL

$^+$CAL ist eine Sprache zur Formulierung von Algorithmen, die Leslie Lamport 2009 vorgestellt hat. Er nennt folgende wichtigen Merkmale von $^+$CAL:

> "PlusCal combines five important features: simple conventional program constructs, extremely powerful expressions, nondeterminism, a convenient way to describe the grain of atomicity, and model checking." [2]

Ein Algorithmus in $^+$CAL wird in TLA$^+$ übersetzt und kann dann mit allen Werkzeugen von TLA$^+$ verwendet werden. Es handelt sich also um eine Sprache, in der eine Spezifikation eines diskreten Systems mit einem Initialzustand und den Zustandsübergängen durch einen Algorithmus gegeben wird. Dies passiert konzeptionell dadurch, dass ein Programmzähler (*program counter, pc*) zusammen mit den Werten der Variablen den Ablauf des Algorithmus in Zustände gliedert.

Im Folgenden wird „Türme von Hanoi" in $^+$CAL formuliert. Den dargestellten Algorithmus habe ich von Hillel Wayne [3, Abschnitt TLA+>Tuples and Structures] übernommen.

Die Beschreibung des Algorithmus in $^+$CAL wird in die TLA$^+$-Datei als Kommentar eingefügt:

Listing 8: Der Rahmen für den Algorithmus in $^+$CAL

```
------------------------------ MODULE hanoi2 ------------------------------
(* Zweite Variante der Türme von Hanoi mit PlusCal,
   basierend auf einer Lösung von Hillel Wayne *)

EXTENDS TLC, Integers, Sequences

(* --algorithm hanoi {

...

}
*)
=============================================================================
```

Es gibt zwei Möglichkeiten für die Syntax: den „p-Stil", der an Pascal angelehnt ist, und den „c-Stil", der an C angelehnt ist. Ich verwende den c-Stil.

Das Datenmodell ist gegenüber der Spezifikation in TLA$^+$ etwas abgewandelt. Die drei Türme sind weiter Sequenzen, deren Element die Scheiben sind, die als Zahlen repräsentiert werden. Die drei Türme werden in dieser Spezifikation selbst eine Sequenz. Der linke Turm ist der erste in der Sequenz usw.

Ferner definieren wir die Menge `pos`. Der Operator `DOMAIN` ist der Definitionsbereich einer Funktion. In TLA$^+$ ist eine Sequenz konzeptionell eine Funktion von $\{1, 2, \ldots, n\}$ (n die Länge der Sequenz) in die Menge der möglichen Werte der Sequenz. Da die Variable `towers` drei Elemente hat, ist `pos` gerade die Menge $\{1, 2, 3\}$. `Cons` kennen wir schon.

Listing 9: Die Türme und zwei Definitionen

```
(* --algorithm hanoi {

variable towers = << <<1, 2, 3, 4>>, << >>, << >> >>;
define {pos == DOMAIN towers
        Cons(elem, seq) == << elem >> \o seq}
...
}
*)
```

Nach diesen Vorbereitung können wir in einem Block den eigentlichen Algorithmus formulieren, so dass sich Folgendes ergibt:

Listing 10: Der Hanoi-Algorithmus in $^+$CAL

```
1  (* --algorithm hanoi {
2
3  variable towers = << <<1, 2, 3, 4>>, << >>, << >> >>;
4  define {pos == DOMAIN towers
5          Cons(elem, seq) == << elem >> \o seq}
6
7  { while (TRUE) {
8      assert towers[3] /= <<1, 2, 3, 4>>;
9      with (from \in {i1 \in pos: towers[i1] /= << >>},
10             to \in   {i2 \in pos: \/ towers[i2] = <<>>
11                                   \/ Head(towers[from]) < Head(towers[i2])}){
12         towers[from] := Tail(towers[from]) ||
13         towers[to] := Cons(Head(towers[from]), towers[to]);
14     }
15   }
16 }
17 }*)
```

Der Algorithmus läuft in einer While-Schleife (Zeile 7) bis die Bedingung von Zeile 8 nicht mehr erfüllt ist. Dies ist die Art und Weise, wie der Widerspruchsbeweis in $^+$CAL angelegt wird: Der Schleifendurchgang, in dem alle Scheiben auf dem rechten Turm liegen wird die Assertion verletzen und somit zu einer Fehlermeldung führen. Die Verwendung des Standard-Moduls `TLC` ist notwendig, denn dort ist `assert` definiert.

In den Zeilen 9 bis 14 wird eine `with`-Anweisung verwendet. Diese Anweisung ist nicht-deterministisch: Der nachfolgende Block wird für eine Wahl aus allen möglichen Kombinationen der Werte `from` und `to` ausgeführt, die für das `with` ermittelt werden. Man beachte, dass sich der Wert der Variable `towers` bei einem Zustandsübergang durch den Funktionskörper des `with` ändert, d.h. die möglichen Werte von `from` und `to` werden jeweils neu berechnet.

Welche Kombinationen sind das? `from` sind die Positionen, an denen der Turm nicht leer ist. Denn von einem leeren Turm kann man keine Scheiben nehmen. `to` sind alle Positionen, bei

denen der Turm leer ist oder die oberste Scheibe des Turms an der Position `from` ist kleiner als die oberste Scheibe des Turms an dieser Position. Die Wahl der Positionen `from` und `to` ergibt also die Türme, bei denen ein Zug möglich ist.

Was wird nun für diese Kombinationen im Körper des `with` gemacht? Vom Turm an der Position `from` wird die oberste Scheibe heruntergenommen und sie wird auf den Turm an der Position `to` gelegt. In $^+$CAL wird das Symbol `:=` für eine „Zuweisung“ verwendet. Tatsächlich steht dahinter, dass die Variable auf der linken Seite der Zuweisung im nächsten Zustand den Wert hat, der auf der rechten Seite definiert wird. Eine Besonderheit ist außerdem `||`. Durch `||` werden mehrere Zuweisungen zu einer Anweisung zusammengefasst. Das bedeutet, dass zuerst alle rechten Seiten der Anweisungen berechnet werden und erst dann die Zuweisungen erfolgen.[4]

5 Die Lösung des Spiels auf Basis des Hanoi-Algorithmus

Wie bereits erwähnt wird ein Algorithmus in $^+$CAL zu einer Spezifikation in TLA$^+$ übersetzt. In der TLA$^+$-Toolbox löst man die Übersetzung durch den Menübefehl File>Translate PlusCal Algorithm. In unserem Beispiel wird dann unter den Algorithmus Folgendes eingefügt:

Listing 11: Der übersetzte Hanoi-Algorithmus

```
1  \* BEGIN TRANSLATION
2  VARIABLE towers
3
4  (* define statement *)
5  pos == DOMAIN towers
6  Cons(elem, seq) == << elem >> \o seq
7
8
9  vars == << towers >>
10
11 Init == (* Global variables *)
12         /\ towers = << <<1, 2, 3, 4>>, << >>, << >> >>
13
14 Next == /\ Assert(towers[3] /= <<1, 2, 3, 4>>,
15                   "Failure of assertion at line 14, column 5.")
16         /\ \E from \in {i1 \in pos: towers[i1] /= << >>}:
17              \E to \in {i2 \in pos: \/ towers[i2] = <<>>
18                                     \/ Head(towers[from]) < Head(towers[i2])}:
19                towers' = [towers EXCEPT ![from] = Tail(towers[from]),
20                                         ![to] = Cons(Head(towers[from]), towers[to])]
21
22 Spec == Init /\ [][Next]_vars
23
24 \* END TRANSLATION
```

[4] Hat man zwei Variablen `u` und `v`, dann kann man das Vertauschen der Werte in $^+$CAL durch die Anweisung `{ u := v || v := u }` festlegen.

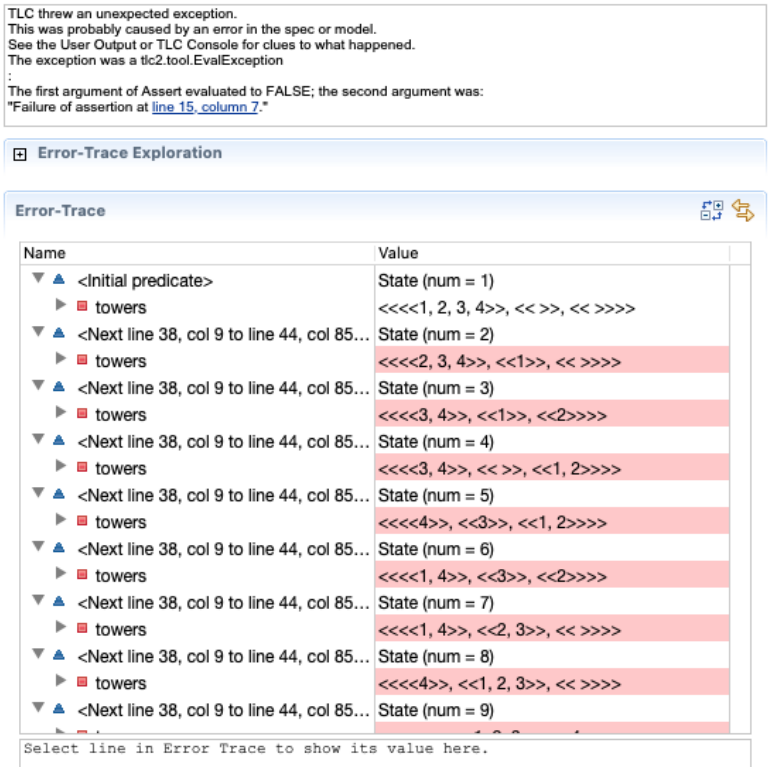

Abbildung 3: Error-Trace, d.h. Lösung des Spiels

Zu erläutern ist der Existenz-Quantor `\E` in Zeile 16 und 17. Er bedeutet zum Beispiel für Zeile 16: Die Aussage wird wahr, wenn es einen Wert in der dort angegebenen Menge gibt. Wir haben also hier eine Formel, die man so lesen muss: „Es existiert ein `from` in ..., für das ein `to` in ... existiert, so dass `towers' = ...` gilt."

Außerdem haben wir bisher `EXCEPT` nicht gesehen. Es ist eine andere Möglichkeit, um die *frame condition* auszudrücken: Alles bleibt gleich außer dem, was in `EXCEPT` angegeben wird. Dabei steht `!` für die neue Sequenz, also `towers'` in diesem Fall.

Nach der Übersetzung haben wir eine TLA$^+$-Spezifikation und wir können wieder ein Modell bilden und es mit dem TLC Model Checker überprüfen. Eine Invariante müssen wir nicht angeben, weil ja das `Assert` Teil von `Next` ist.

Das Ergebnis des Model Checkers wird in Abbildung 3 dargestellt. Und in der Tat ergibt sich exakt die Lösung aus Tabelle 1.

6 Anhang

6.1 Spezifikation der „Türme von Hanoi“ in TLA$^+$

MODULE hanoi1

EXTENDS Integers, Sequences

VARIABLES l, m, r

$$
\begin{aligned}
&\text{Init} \triangleq \land\, l = \langle 1, 2, 3, 4\rangle \\
&\qquad\quad \land\, m = \langle\rangle \\
&\qquad\quad \land\, r = \langle\rangle
\end{aligned}
$$

$$\text{Empty}(seq) \triangleq \text{Len}(seq) \leq 0$$

$$\text{Cons}(elem, seq) \triangleq \langle elem\rangle \circ seq$$

$$
\begin{aligned}
&\text{CanMove}(t1, t2) \triangleq \\
&\quad \land\, \neg\text{Empty}(t1) \\
&\quad \land\, \text{Empty}(t2) \lor (\neg\text{Empty}(t2) \land \text{Head}(t1) < \text{Head}(t2))
\end{aligned}
$$

$$
\begin{aligned}
&\text{Move}(t1, t2, t3) \triangleq \\
&\quad \land\, \text{CanMove}(t1, t2) \\
&\quad \land\, t1' = \text{Tail}(t1) \land t2' = \text{Cons}(\text{Head}(t1), t2) \land t3' = t3
\end{aligned}
$$

$$
\begin{aligned}
&\text{Next} \triangleq \\
&\quad \lor\, \text{Move}(l, m, r) \lor \text{Move}(l, r, m) \\
&\quad \lor\, \text{Move}(m, l, r) \lor \text{Move}(m, r, l) \\
&\quad \lor\, \text{Move}(r, l, m) \lor \text{Move}(r, m, l)
\end{aligned}
$$

$$\text{Spec} \triangleq \text{Init} \land \Box[\text{Next}]_{\langle l, m, r\rangle}$$

$$\text{Goal} \triangleq r = \langle 1, 2, 3, 4\rangle$$

$$\text{Check} \triangleq \neg\text{Goal}$$

Der kommentierte Quelltext befindet sich auf https://esb-dev.github.io/mat/hanoi1.tla

6.2 Spezifikation der „Türme von Hanoi“ in $^+$CAL

MODULE hanoi2

EXTENDS TLC, Integers, Sequences

```
--algorithm hanoi{
variable towers = ⟨⟨1, 2, 3, 4⟩, ⟨⟩, ⟨⟩⟩;
define { pos ≜ DOMAIN towers
         Cons(elem, seq) ≜ ⟨elem⟩ ∘ seq }
{
  while ( TRUE ) {
    assert towers[3] ≠ ⟨1, 2, 3, 4⟩;
    with ( from ∈ {i1 ∈ pos : towers[i1] ≠ ⟨⟩},
           to ∈  {i2 ∈ pos : ∨ towers[i2] = ⟨⟩
                             ∨ Head(towers[from]) < Head(towers[i2])} )
     { towers[from] := Tail(towers[from]) ||
       towers[to] := Cons(Head(towers[from]), towers[to]);
     }
  }
} }
```

BEGIN TRANSLATION

VARIABLE towers

define statement

pos ≜ DOMAIN towers

Cons(elem, seq) ≜ ⟨elem⟩ ∘ seq

vars ≜ ⟨towers⟩

Init ≜ Global variables

∧ towers = ⟨⟨1, 2, 3, 4⟩, ⟨⟩, ⟨⟩⟩

Next ≜ ∧ Assert(towers[3] ≠ ⟨1, 2, 3, 4⟩,

$$\text{“Failure of assertion at line 15, column 7.”})$$

$$\land \exists\, \text{from} \in \{\text{i1} \in \text{pos} : \text{towers}[\text{i1}] \neq \langle\rangle\} :$$

$$\exists\, \text{to} \in \{\text{i2} \in \text{pos} : \lor \text{towers}[\text{i2}] = \langle\rangle$$

$$\lor \text{Head}(\text{towers}[\text{from}]) < \text{Head}(\text{towers}[\text{i2}])\} :$$

$$\text{towers}' = [\text{towers EXCEPT } ![\text{from}] = \text{Tail}(\text{towers}[\text{from}]),$$

$$![\text{to}] = \text{Cons}(\text{Head}(\text{towers}[\text{from}]), \text{towers}[\text{to}])]$$

$$\text{Spec} \triangleq \text{Init} \land \Box[\text{Next}]_{\text{vars}}$$

END TRANSLATION

Der kommentierte Quelltext befindet sich auf https://esb-dev.github.io/mat/hanoi2.tla

Literatur

[1] Leslie Lamport: *Specifying Systems – The TLA$^+$ Language and Tools for Hardware and Software Engineers*, Boston: Addison-Wesley, 2003.

[2] Leslie Lamport: *The PlusCal Algorithm Language*, Microsoft Research: https://www.microsoft.com/en-us/research/uploads/prod/2016/12/The-PlusCal-Algorithm-Language.pdf, 2009.

[3] Hillel Wayne: *Learn TLA+*, Tutorial https://learntla.com/introduction/, 2. April 2020.