TLA$^+$ statt Pseudocode

Jennifer Moritz
(Matrikelnummer: 70420751)

Eingereichte Abschlussarbeit

zur Erlangung des Grades

Bachelor of Science (B. Sc.)

im Studiengang

Logistik und Informationsmanagement (LIM)

an der

Karl-Scharfenberg-Fakultät

der Ostfalia Hochschule für angewandte Wissenschaften

Erster Prüfer: Herr Prof. Dr. Jetzke

Zweiter Prüfer: Dipl.-Math. Jürgen Böttcher

Eingereicht am: 08.03.2018

Inhaltsverzeichnis

1 Einleitung

Die Bachelorarbeit befasst sich mit der von Leslie Lamport entwickelten Spezifikationssprache TLA$^+$, mit der Algorithmen unter Verwendung einfacher Mathematik formell beschrieben werden können. Sie richtet sich an alle Mitarbeiter eines IT- Unternehmens oder einer IT-Abteilung eines beliebigen Unternehmens, die anhand von Kundenwünschen und Kundenanforderungen Programme oder Software entwickeln.

Etabliert ist hierbei die Verwendung von Pseudocodes zur Beschreibung von Algorithmen. Sowohl Algorithmen und Pseudocodes, als auch die Spezifikationssprache TLA$^+$ sind im Bereich der theoretischen Informatik angesiedelt. Die theoretische Informatik „befasst sich mit Informations- und Codierungstheorie, formalen Sprachen, Automatentheorie, Algorithmen, Berechenbarkeit, Datenstrukturen und mathematischen Methoden“ [2] und „entwickelt mathematische Modelle von Computern, Systemen und Prozessen, sowie Hilfsmittel zu ihrer präzisen Beschreibung.“ [1] Unter Pseudocode versteht man im Allgemeinen die Beschreibung eines Algorithmus in einer formalisierten, natürlichen Sprache, die dabei beispielsweise auf Notationen aus höheren Programmiersprachen oder der Mathematik zurückgreift: Die Vorgehensweise eines Algorithmus „lässt sich etwas formaler in einem sogenannten Pseudocode formulieren, welcher von der textuellen Beschreibung abstrahiert, um eine dem Rechner verständliche Form zu erreichen. Vergleichbar zu einer Programmiersprache […] basiert auch diese Schreibweise auf eindeutigen Schlüsselwörtern, z. B. „Setze“ oder „Solange“. Der Vorteil von Pseudocode besteht darin, dass er sich sowohl leicht aus einer textuellen Beschreibung ableiten, aber auch vergleichsweise einfach in eine rechnerverständliche Form als Programm überführen lässt.“ [1]

Leslie Lamport erläutert in seinem Vortrag “The PlusCal Algorithm Language” [3] im Rahmen des vierten *Heidelberg Laureate Forum* jedoch die Nachteile von Pseudocodes zur Beschreibung von Algorithmen. Sie sind zum einen nicht präzise genug, zum anderen sind Pseudocodes nicht überprüfbar. Dementsprechend eignen sie sich nicht, um Algorithmen zu beschreiben. Aber auch Programmiersprachen sind seiner Meinung nach ungeeignet. Aus diesem Grund entwickelte Lamport die Spezifikationssprache TLA$^+$. Das Ziel dieser Arbeit ist es aufzuzeigen, wie TLA$^+$ funktioniert und wie Algorithmen in der Spezifikationssprache dargestellt werden können. In den nächsten Kapiteln werden daher die Bedeutung von TLA$^+$, sowie wichtige inhaltliche und technische Grundlagen erläutert und die Funktionsweise von

TLA$^+$ an zwei einfachen Beispielen erklärt, bevor die Arbeit zusammengefasst und mit einem Fazit abgeschlossen wird.

2 Die Bedeutung von TLA$^+$

Die überwiegende Anzahl von Programmierproblemen entsteht in der realen Welt. Die klassische Vorgehensweise bei der Lösung solcher Probleme lässt sich nach Heinrich Müller und Frank Weichert in drei Phasen unterteilen:

1. Modellierung
2. Darstellung
3. Realisierung

In der Phase der Modellierung erfolgt mit Hilfe von Abstraktion die Analyse des Problems. Im Anschluss an die Problemanalyse wird eine funktionale Spezifikation erstellt, die beschreibt, was der Algorithmus genau machen soll. Mit Hilfe dieser funktionalen Spezifikation wird der Entwurf des Algorithmus erstellt, der festlegt, wie das Ganze umzusetzen ist. In dieser Phase findet auch die Spezifikationssprache TLA$^+$ von Leslie Lamport Verwendung.

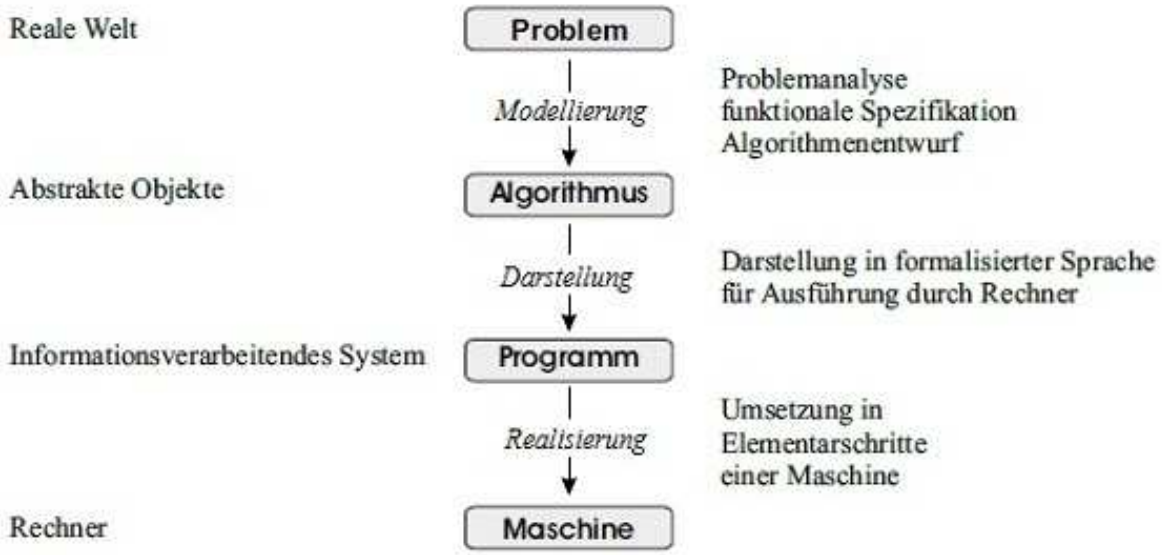


Abbildung 1: Vorgehensweise zur Lösung von Problemen in der Informatik [1]

Die Darstellung ist der Übergang vom Algorithmus zu einem Programm. In dieser zweiten Phase wird der Algorithmus in einer formalen Sprache, beispielsweise in der Programmiersprache C, beschrieben. In der letzten Phase der Realisierung wird das Programm in die Elementarschritte einer Maschine umgesetzt, so dass dieses auf einem Rechner ausgeführt werden kann. [1]

Ein weit verbreitetes und allgegenwärtiges Problem ist beispielweise das Kürzeste- Wege-Problem. Es beschreibt das Problem in einem Netzwerk den kürzesten, schnellsten oder kostengünstigsten Weg zu finden. Eine praktische Anwendung aus der Logistik ist die Routenplanung in Straßennetzen. Der meistbenutzte Algorithmus, um kürzeste Wege zu berechnen, ist der Algorithmus von Dijkstra. [7] Ein Algorithmus ist definiert als „eine endliche Vorschrift zur eindeutigen Überführung von Eingabegrößen zu Ausgabegrößen in einer endlichen Zahl von Schritten.“ [1] Gemäß dieser Definition sind die Zubereitungsanweisungen in einem Rezept zum Backen eines Kuchens ein Praxisbeispiel für einen Algorithmus aus dem Alltag. Im Fall des Dijkstra- Algorithmus ist die Eingabegröße ein Graph mit nicht- negativen Kantenkosten, mit dessen Hilfe das Problem modelliert werden kann. Die Ausgabegröße ist ein kostenminimaler Weg, wobei die Kosten in den Fall mit Zeit, Geld oder Distanz beschrieben werden können. [7] Ein solcher Graph kann wie folgt aussehen.

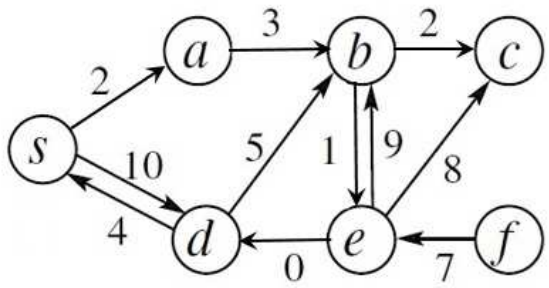


Abbildung 2: Beispiel für einen Graphen vor der Anwendung des Dijkstra- Algorithmus

In der logistischen Praxis handelt es sich bei den Knoten a bis f möglicherweise um Städte, in denen potenzielle Kunden eines Unternehmen sitzen, die bei einer erfolgten Bestellung beliefert werden sollen. Bei den Kunden mit einem Sitz in der Stadt bzw. Knoten f handelt es sich beispielsweise um Kunden aus dem Ausland, die von einer Belieferung ausgeschlossen sind. Das ausliefernde Unternehmen hat seinen Sitz in der Stadt s, die gleichzeitig auch der Startknoten ist. Das Unternehmensziel ist eine Auslieferung auf dem kürzesten Wege,

weswegen die Berechnung der Route vom Startknoten s zum Zielknoten mit dem Algorithmus von Dijkstra erfolgen soll. Die Straßen sind durch gerichtete Kanten in Form von Pfeilen dargestellt. Diese dürfen nur in die entsprechende Richtung befahren werden. Die Kantenkosten werden durch die Zahlen an den Kanten dargestellt. In diesem Beispiel wird angenommen, dass es sich dabei um die Weglänge handelt.

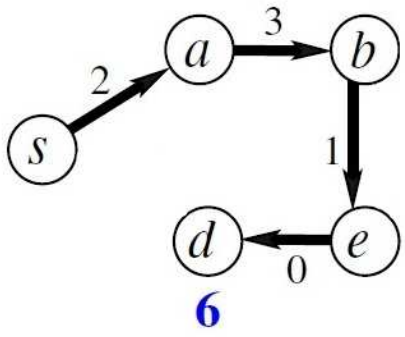
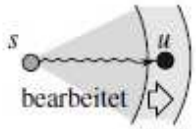


Abbildung 3: Beispiel für einen Graphen nach der Anwendung des Dijkstra- Algorithmus

Erfolgt eine Bestellung von einem Kunden in der Stadt d, ergibt sich nach der Anwendung des Dijkstra- Algorithmus ausgehend vom Startknoten s ein kürzester Weg zu dem Zielknoten d, dargestellt durch die schwarz hervorgehobenen Kanten. Bei der blauen Knotenzahl handelt es sich um die so genannte Kürzeste-Wege-Distanz. Die Kürzeste-Wege-Distanz drückt aus, wie hoch die Gesamtkosten vom Startknoten s zu dem Zielknoten d sind. Der kürzeste Weg vom Startknoten s zu dem Zielknoten d verläuft über die Knoten a, b und e und hat eine Weglänge von sechs.

Die beiden vorherigen Abbildungen zeigen jeweils eine graphische Darstellung des Eingabe- und Ausgabegraphen, mit dem das Problem modelliert wird. Die genaue Vorgehensweise des Dijkstra- Algorithmus bleibt dabei unberücksichtigt. Diese wird im Folgenden durch Pseudocode beschrieben. Das Problem bei der Erstellung und Verwendung von Pseudocodes ist, dass es keine einheitlichen Normen und Notierungen dafür gibt. Ein und derselbe Algorithmus kann daher durch eine Vielzahl von unterschiedlichen Pseudocodes beschrieben werden. In dem Buch „Algorithmen und Datenstrukturen. Die Grundwerkzeuge“ von Martin Dietzfelbinger, Kurt Mehlhorn und Peter Sanders findet sich daher der folgende Pseudocode für den Dijkstra- Algorithmus. [7]

```
Function Dijkstra(s : NodeId) : NodeArray×NodeArray          // gibt (d,parent) zurück
    d = ⟨∞,...,∞⟩ : NodeArray of ℝ∪{∞}                        // Schätzdistanzen
    parent = ⟨⊥,...,⊥⟩ : NodeArray of NodeId
    parent[s] := s                                            // Schleife signalisiert Wurzel
    Q : NodePQ                                                // unbearbeitete gefundene Knoten
    d[s] := 0;   Q.insert(s)
    while Q ≠ ∅ do
        u := Q.deleteMin                                      // es gilt d[u] = μ(u)
        foreach edge e = (u,v) ∈ E do
            if d[u] + c(e) < d[v] then                        // relaxiere
                d[v] := d[u] + c(e)
                parent[v] := u                                // aktualisiere Baum
                if v ∈ Q then Q.decreaseKey(v)
                else Q.insert(v)
    return (d,parent)
```



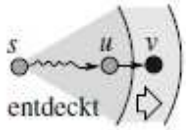


Abbildung 4: Dijkstra- Pseudocode nach Dietzfelbinger, Mehlhorn und Sanders [7]

In einem anderen Buch mit dem Titel „Algorithmen und Datenstrukturen“ von Thomas Ottmann und Peter Widmayer ist der Dijkstra- Algorithmus durch den folgenden Pseudocode beschrieben. [9]

```
procedure shortestpath ((V,E) : Graph; s : Knoten);
{berechnet zum Graphen mit Knotenmenge V und Kantenmenge E
 und zu s ∈ V für jeden Knoten v ∈ V die
 Länge d(v) des kürzesten Weges von s nach v}
begin
   {Initialisierung}
   for all v ∈ V\{s} do d(v) := ∞;
   d(s) := 0;  S := ∅;
   Initialisiere V\S := V als Priority Queue, geordnet nach Distanzen
      d(v), v ∈ V;
   {anfangs ist also s minimales Element in V\S; es folgen alle
    übrigen Elemente von V in beliebiger Reihenfolge}
   {vergrößern von S nach dem Prinzip: Knoten mit kürzester Distanz
    von s zuerst}
   while V\S ≠ ∅ do
      begin
         v := min(V\S);  deletemin(V\S);
         S := S∪{v};
         for all w ∈ N(v)\S do
            if d(v) + l(vw) < d(w)
               then decreasekey(w, d(v) + l(vw))
      end
end
```

Abbildung 5: Dijkstra- Pseudocode nach Ottmann und Widmayer [9]

Sogar in der Online- Enzyklopädie Wikipedia findet sich eine eigene Version des Dijkstra-Pseudocodes wieder. [12]

```
1  Funktion Dijkstra(Graph, Startknoten):
2      initialisiere(Graph,Startknoten,abstand[],vorgänger[],Q)
3      solange Q nicht leer:                          // Der eigentliche Algorithmus
4          u:= Knoten in Q mit kleinstem Wert in abstand[]
5          entferne u aus Q                                  // für u ist der kürzeste Weg nun bestimmt
6          für jeden Nachbarn v von u:
7              falls v in Q:                              // falls noch nicht berechnet
8                  distanz_update(u,v,abstand[],vorgänger[])   // prüfe Abstand vom Startknoten zu v
9      return vorgänger[]
```

```
1  Methode initialisiere(Graph,Startknoten,abstand[],vorgänger[],Q):
2      für jeden Knoten v in Graph:
3          abstand[v]:= unendlich
4          vorgänger[v]:= null
5      abstand[Startknoten]:= 0
6      Q:= Die Menge aller Knoten in Graph
```

```
1  Methode distanz_update(u,v,abstand[],vorgänger[]):
2      alternativ:= abstand[u] + abstand_zwischen(u, v)   // Weglänge vom Startknoten nach v über u
3      falls alternativ < abstand[v]:
4          abstand[v]:= alternativ
5          vorgänger[v]:= u
```

Abbildung 6: Dijkstra- Pseudocode nach Wikipedia [12]

Auf den ersten Blick lassen sich bereits essentielle Unterschiede in den einzelnen Pseudocodes entdecken. Der Pseudocode nach Dietzfelbinger, Mehlhorn und Sanders in Abbildung 4 nutzt eine Funktion, an die die Knoten-ID des Startknotens und ein Array von Knoten übergeben werden, welches die Knoten des Graphen enthält. Der Pseudocode nach Ottmann und Widmayer in Abbildung 5 hingegen verwendet eine Prozedur zur Berechnung des kürzesten Weges, an die der Startknoten s und ein Graph, bestehend aus einer Menge von Knoten V und einer Menge von Kanten E, übergeben werden. Der in Wikipedia eingestellte Pseudocode des Dijkstra- Algorithmus in Abbildung 6 nutzt zwar ebenfalls eine Funktion, an die der Graph und der Startknoten übergeben werden, die Initialisierung und das Update der Distanzen wurden jedoch jeweils in eine eigene Methode ausgelagert.

Unberücksichtigt bleiben in allen Pseudocodes zudem die folgenden Problemfälle, die nicht dem Idealfall entsprechen, aber in der Praxis durchaus vorkommen können.

1. Der Graph ist nicht zusammenhängend. Gemäß der Graphentheorie ist ein Graph „genau dann zusammenhängend, wenn alle Paare von verschiedenen Knoten des Graphen durch

mindestens eine Kantenfolge verbunden sind.“ [2] In dem nicht zusammenhängenden Fall ist entsprechend nicht jeder Knoten von jedem anderen Knoten aus erreichbar.

2. Der Startknoten liegt nicht im Graphen. Dies kann z.B. der Fall sein, wenn die Knoten eines Graphen als ein Array von Objekten eingegeben werden und der Startknoten separat davon als ein eigenes Objekt, das sich nicht in dem Array befindet.

3. Der Zielknoten ist nicht erreichbar, etwa wenn der Graph nicht zusammenhängend ist und sich der Startknoten und der Zielknoten in jeweils unterschiedlichen Zusammenhangskomponenten des Graphen befinden.

Tritt eines dieser Szenarien ein und gibt es entsprechend keinen kürzesten Weg, wird im besten Fall eine Fehlermeldung ausgegeben oder das Programm stürzt bei dem Versuch der Berechnung ab. Im schlimmsten Fall erfolgt eine fehlerhafte Berechnung, die nicht als solche erkannt wird.

Zudem geht aus keinem der drei Pseudocodes hervor, in welcher Form die Eingabe zu erfolgen hat, damit die Berechnungen korrekt durchgeführt werden können. Auch die Ausgabeform des kürzesten Weges ist nicht definiert. Im Fall des Pseudocodes nach Ottmann und Widmayer in Abbildung 5 geht es sogar so weit, dass bei einer späteren Implementierung mit Hilfe einer Prozedur keine Rückgabe des kürzesten Weges an das Hauptprogramm erfolgt. Eine Prozedur ist nämlich mit einer void- Funktion, wie sie beispielsweise in der Programmiersprache C verwendet werden kann, vergleichbar, die keinen Rückgabewert hat.

Neben der mangelnden Normierung bringen Pseudocodes zwei weitere Nachteile mit sich: Bei der Verwendung von natürlichen Sprachen zur Beschreibung von Algorithmen ist weder die Syntax des Pseudocodes, noch die Semantik automatisch überprüfbar. Das stellte auch Lamport im Jahr 2004 fest, als er den Begriff „quicksort“ googelt und die ersten zehn Algorithmen testet, die er in den Suchergebnissen findet. Vier Stück davon waren in Pseudocode geschrieben. Diese vier waren falsch. Die anderen sechs waren nur richtig, weil sie in ausführbarem Code geschrieben und somit bereits getestet waren. Dass Pseudocodes weiterhin nicht präzise genug sind, stellte er anhand seiner im Jahr 2001 veröffentlichten Abhandlung „Paxos Made Simple“ fest, in der er den Paxos- Algorithmus informell erklärt. Ein von den Lesern falsch interpretierter Satz aus dieser Arbeit führte mehrfach zu falschen Implementierungen des Algorithmus. Nach Lamport eignen sich Pseudocodes dementsprechend nicht, um Algorithmen

zu beschreiben. Aber auch Programmiersprachen sind aufgrund ihrer unnötigen und verwirrenden Komplexität mit Objekten, Vererbung, Typen usw. seiner Meinung nach ungeeignet. Denn trotz ihrer Komplexität sind diese zur Algorithmus- Beschreibung nicht ausdrucksstark genug. [3]

Als Antwort auf diese Probleme entwickelte Lamport die Spezifikationssprache TLA$^+$. Diese Bachelorarbeit soll aufzeigen, wie TLA$^+$ funktioniert und wie Algorithmen in der Spezifikationssprache dargestellt werden können. Das Ziel dabei ist es herauszufinden, ob sich TLA$^+$ anstatt Pseudocode zur Beschreibung von Algorithmen eignet.

3 Allgemeines

3.1 Was ist TLA+?

TLA$^+$ wurde von Leslie Lamport entwickelt, einem US- amerikanischen Mathematiker, Informatiker und Programmierer, der für seine grundlegende Forschung im Bereich der Theorie und Praxis verteilter und nebenläufiger Systeme im Jahr 2013 den A. M. Turing Award erhielt. Der Award gilt als höchste Auszeichnung in der Informatik und ist vergleichbar mit dem Nobelpreis. Er wird seit 1966 jährlich von der *Association for Computing Machinery*, der ersten wissenschaftlichen Gesellschaft für Informatik, verliehen. Die Preisträger zeichnen sich allesamt durch ihre besonderen Beiträge zur nachhaltigen Entwicklung der Informatik aus. Lamport ist weiterhin bekannt für sein Anfang der Achtzigerjahre entwickeltes Softwarepaket LaTeX, das eine Reihe von Makros bereitstellt, um die Benutzung des Textsatzsystems TeX von Donald Knuth zu vereinfachen. LaTeX zählt heute zu einer der populärsten Methoden bei der Verwendung von TeX und wird dementsprechend überwiegend genutzt, wenn TeX zu Einsatz kommt. [6]

Lamport selber bezeichnet sich als Theoretiker, der die Bedürfnisse von Programmierern und Entwicklern versteht und mit seiner Forschung dementsprechend praktische Ziele verfolgt. Im Jahr 2002 veröffentlichte er sein Buch „Specifying Systems - The TLA$^+$ Language and Tools for Hardware and Software Engineers“. TLA$^+$ ist eine Spezifikationssprache zur high-level Modellierung von digitalen Systemen und stellt diverse Tools bereit, um diese Modelle zu überprüfen. Nach Lamport umfassen digitale Systeme Algorithmen, Programme und

Computersysteme. High-level bedeutet in dem Fall auf der Ebene des Entwurfs solcher Systeme, über der Code- Ebene, d.h. noch bevor die erste Zeile Code geschrieben wird. Die Spezifikationssprache wird verwendet, um kritische Teile von digitalen System zu modellieren. Durch Abstraktion, d.h. durch den Prozess der Vereinfachung durch das Entfernen irrelevanter Details, verschwinden zum einen weniger kritische Teile und zum anderen unnötige Implementierungsfeinheiten. [4]

3.2 Wie funktioniert TLA+?

Einfache Mathematik und temporale Logik bilden die Basis von TLA$^+$. Temporale Logik, auch Zeitlogik genannt, ist eine Erweiterung der klassischen Logik, mit der zeitliche Abläufe mit Hilfe von Vorher- nachher- Beziehungen zwischen Zeitpunkten erfasst werden können. Die grundlegende Abstraktion, der TLA$^+$ unterliegt, funktioniert nach dem folgenden Prinzip: Die Ausführung eines digitalen Systems wird repräsentiert durch eine Abfolge eigenständiger Schritte. Die drei wichtigsten Merkmale sind hierbei

1. Eigenständigkeit

   Ein digitales System ist ein physisches System, dessen ständige Entwicklung abstrakt durch eine Abfolge eigenständiger Ereignisse dargestellt werden kann. Ein digitales System ist beispielweise das eines Uhrwerkes. Obwohl sich alle Teile der realen Uhr im physischen System kontinuierlich bewegen, kann diese durch eine abstrakte Uhr beschrieben werden. Ein weiteres Beispiel für ein sich ständig weiterentwickelndes physisches System ist ein Computer. Die Ausführung eines Programms kann im digitalen System abstrakt, bestehend aus eigenständigen Schritten, dargestellt werden.

2. Abfolge

   Ein weiterer Grund, warum Leslie Lamport TLA$^+$ entwickelte, war die Beschreibung von nebenläufigen Algorithmen, die mit Pseudocode nahezu unmöglich darzustellen sind. [3] Es scheint zunächst befremdlich ein nebenläufiges System, in dem mehrere Dinge zur gleichen Zeit passieren, als eine Abfolge von Schritten zu beschreiben. Es ist dennoch möglich und noch dazu einfach und funktional.

3. Schritt

   In TLA$^+$ wird ein Schritt beschrieben als ein Zustandsübergang, d.h. die Veränderung von einem Zustand zum nächsten. Eine Ausführung ist nichts anderes als eine Abfolge von Schritten, auch Verhalten genannt. Digitale Systeme werden entsprechend modelliert als ein Zustand, der sich in eigenständigen Schritten verändert. TLA$^+$ beschreibt dabei einen Zustand als Wertzuweisung zu Variablen.

Um alle möglichen Verhalten eines digitalen Systems in TLA$^+$ zu beschreiben, werden so genannte *state machines*, auf Deutsch auch Zustandsmaschinen genannt, verwendet. Sie werden beschrieben durch

1. alle möglichen Anfangszustände
2. den Folgezustand, der auf jeden gegebenen Zustand folgen kann.

Da ein Zustand eine Zuweisung von Werten zu Variablen ist, muss zudem beschrieben werden,

1. was die Variablen sind
2. was die möglichen Anfangswerte dieser Variablen sind
3. wie die Beziehung zwischen den Werten der Variablen im aktuellen Zustand und der möglichen Werte im Folgezustand ist.

Oftmals gibt es zusätzlich einen *control state* oder Kontrollzustand, der spezifiziert, welche Anweisung als nächstes ausgeführt werden soll. Repräsentiert wird der Kontrollzustand durch eine zusätzliche Variable, z.B. kurz *pc* für *program control*.

Die Verwendung von *state maschines* macht Sinn. Diese sind nämlich einfacher als Programme, da alle Teile von einem Zustand als Variablenwerte ausgedrückt werden. Bei einem Programm werden verschiedene Teile eines Zustandes oftmals unterschiedlich ausgedrückt, weil sie auch unterschiedlich implementiert werden. Mit TLA$^+$ werden solche Implementierungsdetails eliminiert und eine einfache Abstraktion geschaffen, um Algorithmen und Systeme zu spezifizieren. Aus einer TLA$^+$ Spezifikation kann jedoch nicht automatisch Code generiert werden. [4]

3.3 Wie kann man TLA+ erlernen?

Leslie Lamport stellt auf seiner Homepage, die unter dem Link http://www.lamport.org/ zu erreichen ist, zum einen das TLA$^+$ Hyperbook und zum anderen einen TLA$^+$ Video Course zur Verfügung. Beides bezeichnet er als unvollendete Werke, die laufend überarbeitet werden. Diese Arbeit bezieht sich auf die Hyperbook- Version vom 20. August 2015 und die Video Course- Version vom 11. Dezember 2017.

3.3.1 Das TLA$^+$ Hyperbook

Das TLA$^+$ Hyperbook ist ein Hyperlink- Buch im pdf- Format, das am besten digital gelesen werden sollte. Es besteht aus einer Sammlung an pdf- Dateien. Die Root- Datei ist *start.pdf*, die anderen Dateien befinden sich alle in dem Ordner *hyper-tla*. Das Ganze steht als zip- Archiv zum Download auf Lamports Webseite zur Verfügung.

Das Hyperbook selber besteht aus insgesamt zwanzig Kapiteln und beinhaltet zwei Übungen, die erste „Principles of Concurrent Computing“ in Kapitel 7 und Kapitel 8, die zweite „Specification of Concurrent Systems“ in Kapitel 9. Beide basieren auf TLA$^+$, sowie den sechs vorangegangenen Kapiteln. Die Benutzung der TLA$^+$ Toolbox wird von beiden Übungstiteln vorausgesetzt. Der Principles- Übungstitel verwendet darüber hinaus hauptsächlich die PlusCal Algorithmussprache, um Algorithmen zu beschreiben und später automatisch in TLA$^+$ zu übersetzen. Weiterhin beinhaltet das TLA$^+$ Hyperbook in den Kapiteln 10, 11 und 12 einen Übungstitel zum Thema TLA$^+$ Proofs, der erklärt, wie das TLA$^+$ proof system, kurz TLAPS, zu verwenden ist. Danach folgen weitere Kapitel mit Grundlagen zur verwendeten Mathematik, sowie in TLA$^+$ häufig verwendete Konstrukte, die jedoch meist in den vorherigen Kapiteln an thematisch passender Stelle verlinkt wurden und bei Bedarf zum besseren Verständnis gelesen werden können.

Das Hyperbook wird laufend überarbeitet, es ist ein „work in progress“. Unvollständige Stellen sind entsprechend gekennzeichnet. Das Ziel des Hyperbooks ist es den Leser zu schulen, wie man die TLA$^+$ Toolbox verwendet, um Spezifikationen und Algorithmen zu schreiben und zu überprüfen.

3.3.2 Der TLA+ Video Course

Der TLA+ Video Course besteht aus einer Reihe von Video- Vorträgen von Leslie Lamport und steht ebenfalls auf seiner Webseite zur Verfügung. Die einzelnen Videos können entweder online angeschaut oder heruntergeladen werden. Da diese dafür bestimmt sind in Verbindung mit der jeweiligen Webseite betrachtet zu werden, gibt es spezielle Download- Anweisungen. Zusammen mit den Videos wird eine html- Datei heruntergeladen, über die das Video gestartet werden sollte. Nur so ist sichergestellt, dass alle Funktionen, wie z.B. Links in dem Video oder die Einstellung der Wiedergabegeschwindigkeit, aktiv genutzt werden können.

Insgesamt gibt es neun Vorträge: Vortrag 1 bis 7 besteht aus jeweils einem Video, während die Vorträge 8 und 9 aus jeweils zwei Videos bestehen. Zu jedem Video gibt es darüber hinaus ein Skript in Form einer pdf- Datei, so dass sich beispielsweise auch Hörgeschädigte die Videos ohne Einschränkungen anschauen können. Das Ziel des TLA+ Video Course ist es, Programmierer und Softwareentwickler zu schulen, eigene TLA+ Spezifikationen zu schreiben. Ein grundlegendes Verständnis von Programmierkonzepten wird dabei vorausgesetzt. Einige Kenntnisse in einfacher Mathematik können hilfreich sein, sind aber kein Muss.

4 Inhaltliche Grundlagen

In seinem TLA+ Hyperbook [10] geht Leslie Lamport zunächst auf ein paar inhaltliche Grundlagen ein und erklärt mit Nebenläufigkeit, Modellierung, Spezifikation und Spezifikationssprachen wichtige Begriffe, die essentiell für das Verständnis von TLA+ sind.

4.1 Nebenläufigkeit

Der Begriff „Concurrent computation“ bedeutet gleichzeitiges oder auch nebenläufiges Rechnen, wobei letzteres die in der Informatik gebräuchlichere Übersetzung ist. Im Falle der Nebenläufigkeit werden verschiedene Prozesse oder Rechenschritte gleichzeitig ausgeführt. Diese müssen aber nicht unbedingt etwas miteinander zu tun haben; sie können kausal unabhängig voneinander und somit auch unabhängig voneinander ausführbar sein. Bei der

Ausführung nebenläufiger Prozesse können diese zueinander in Konkurrenz stehen, beispielsweise bei der Belegung und Benutzung von Rechner- Prozessoren. Ein Anwendungsbeispiel aus der Praxis sind genetische Algorithmen. „Parallel computation“, also paralleles Rechnen, ist eine spezielle Form der Nebenläufigkeit. Im Falle der Parallelität wird z.B. ein Programm in einzelne Einheiten, die auch Prozesse genannt werden, zerlegt. Diese werden gleichzeitig und parallel ausgeführt. Unterteilt wird das Ganze noch in „echt parallel“ und „quasiparallel“. Echt parallele Prozesse werden auf mehreren Prozessoren ausgeführt, während quasiparallele Prozesse auf einem Prozessor ausgeführt werden. [11] Das Ziel des parallelen Rechens ist die Verkürzung der Rechenzeit. Als Anwendungsbeispiel aus der Praxis seien hier Abstandsberechnungen in Graphen genannt. Aber auch die im späteren Verlauf dieser Arbeit als Beispiel aufgegriffene Fakultät einer natürliche Zahl lässt sich auf diese Weise berechnen, denn

$$n! = \prod_{i=1}^{n} i = \prod_{i=1}^{\frac{n}{2}} i * \prod_{i=\frac{n}{2}+1}^{n} i$$

Im Gegensatz zu nebenläufigem Rechnen ist paralleles Rechnen vom Grundsatz her vermeidbar, da die einzelnen Rechenschritte, die zur Beschleunigung der Ausführung parallel aufgeführt werden, theoretisch auch nacheinander ausgeführt werden können. [10]

4.2 Modellierung

„Modeling Computation“ bedeutet übersetzt so viel wie „Rechnen modellieren“. Unter Rechnen ist laut Lamport all das zu verstehen, was ein Computer macht, oder in der heutigen Zeit besser noch, was ein digitales System macht. Der Unterschied besteht darin, dass das Rechnen in einem digitalen System aus einer Vielzahl eigenständiger Ereignisse besteht. Wenn im Folgenden nunmehr der Begriff System verwendet wird, ist damit ein digitales System gemeint.

Abstrakte Systeme sind Abstraktionen von digitalen Systemen, die man erhält, indem man davon ausgeht, dass ihre Berechnungen aus bestimmten eigenständigen Ereignissen bestehen. Je weniger einzelne Ereignisse, desto einfacher ist die Abstraktion. Je mehr einzelne Ereignisse, desto genauer lässt sich ein System beschreiben. Der Grad der Abstraktion ist abhängig von

dem angedachten Verwendungszweck und sollte so einfach wie möglich und so präzise wie nötig gewählt werden. Die Darstellung einer Abstraktion eines Systems ist das Modell eines Systems. Es gibt etliche Möglichkeiten, um ein System zu modellieren. In dem von TLA$^+$ verwendeten Standardmodell wird ein abstraktes System durch eine Sammlung von Verhalten beschrieben. Jedes Verhalten stellt dabei eine mögliche Ausführung des Systems dar und ist definiert durch eine Abfolge von Zuständen. Ein Zustand ist eine Zuweisung von Werten zu Variablen. Der Übergang von einem Zustand zum nächsten ist ein Ereignis, auch Schritt genannt. Der im Folgenden verwendete Ausdruck Systemmodell oder Modell eines Systems ist gleichzusetzen mit dem Modell einer Abstraktion eines Systems.

4.3 Spezifikation

Der von Lamport sowohl im Hyperbook, als auch in seinem Buch „Specifying Systems“ verwendete Begriff „Specification“ heißt übersetzt Spezifikation und ist eine Beschreibung des Modells eines Systems. Im Folgenden wird dafür der Ausdruck Systemspezifikation verwendet. Eine Systemspezifikation ist also die Spezifikation eines Modells einer Abstraktion eines digitalen Systems.

Eine Spezifikation ist wie eine Blaupause, d.h. der Entwurf eines Gebäudes. So wie niemand einfach so ein großes Gebäude bauen würde, ohne vorher entsprechende Pläne anzufertigen, so sollte niemand ein komplexes System bauen, ohne es zuvor zu spezifizieren. Systemspezifikationen können zwar auch im Nachhinein angefertigt werden, um zu verstehen, was das System tut, möglicherweise um Fehler ausfindig zu machen oder das System zu modifizieren, am hilfreichsten sind sie allerdings im Vorfeld. Ähnlich wie Blaupausen, die den Bau des Gebäudes anleiten.

Es kann zwischen formalen und informalen Spezifikationen unterschieden werden. Eine formale Spezifikation ist in einer präzise definierten, formalen Sprache geschrieben und hat Ähnlichkeit mit einer detaillierten Blaupause. Eine informale, d.h. entsprechend formlose, Spezifikation ist dagegen vergleichbar mit einer groben Design- Skizze, ausreichend für kleinere System- Konstruktionen. Der größte Vorteil einer formalen Spezifikation, wie sie mit TLA$^+$ erstellt wird, ist, dass man hierauf Tools zur Fehlersuche anwenden kann.

4.4 Spezifikationssprachen

4.4.1 TLA+

Ein weit verbreiteter Glaube ist nach Lamport, dass eine Systemspezifikation am zweckdienlichsten ist, wenn sie in einer Sprache geschrieben wird, die der Sprache ähnelt, in der das System später implementiert wird. Konkret bedeutet das, dass die Spezifikationssprache, in der ein Programm spezifiziert wird, Ähnlichkeiten mit der jeweiligen Programmiersprache aufweisen sollte. Nach dieser Argumentation müssten, wenn ein Gebäude aus Ziegeln gebaut wird, die Blaupausen ebenfalls aus Ziegeln bestehen.

Eine Spezifikationssprache ist dazu da, um Modelle von Systemen zu beschreiben. Die meisten Wissenschaftler und Ingenieure haben sich auf die Mathematik als geläufige Sprache festgelegt, um System- Modelle zu beschreiben. Die Sprache der Mathematik ist einfach und ausdruckstark und eignet sich somit hervorragend, um digitale Systeme zu beschreiben. Um Spezifikationen zu schreiben, braucht es lediglich ziemlich grundlegende Mathematik: Mengen, Funktionen und einfache Logik.

Obwohl die Mathematik simpel ist, sind Fehler beim Schreiben von mathematischen Formeln nicht auszuschließen. Lamport schreibt dazu in seinem Hyperbook: „It is almost as hard to get a formula right the first time as it is to write a program that works the first time you run it.” [10] Um diese mit entsprechenden Tools zu überprüfen, müssen die mathematischen Spezifikationen formal sein. Da es keine allgemein akzeptierte formale Sprache zum Schreiben solcher mathematischen Spezifikationen gibt, wurde TLA$^+$ entwickelt. TLA$^+$ enthält aber nicht nur Notationen und Konzepte aus der Mathematik. Zwei wesentliche Dinge, die TLA$^+$ von grundlegender Mathematik unterscheidet sind im Folgenden erklärt.

1. In TLA$^+$ gibt es eine Notation für lange Formeln. Konjunktionen und Disjunktionen können als Aufzählung geschrieben werden, was das Lesen der Formeln vereinfacht. Die folgende mathematische Schreibweise mit Konjunktion und Disjunktion hat die gleiche Bedeutung, wie die Schreibweise in TLA$^+$ als Aufzählung.

   Mathematische Schreibweise:

$$((b = 0) \wedge (b' = 1)) \vee ((b = 1) \wedge (b' = 0))$$

Schreibweise in TLA$^+$:

$$\begin{aligned} \vee \wedge\, b &= 0 \\ \wedge\, b' &= 1 \\ \vee \wedge\, b &= 1 \\ \wedge\, b' &= 0 \end{aligned}$$

2. Weiterhin gibt es eine Notation für lange Spezifikationen. Die Ideen dazu stammen aus den Konzepten von Programmiersprachen. Zum einen müssen Variablen explizit deklariert werden. Die Deklaration kann dabei helfen Fehler zu finden. Zum anderen kann eine Spezifikation auf mehrere separate Module aufgeteilt werden. Auf diese Weise lassen sich auch komplexe Spezifikationen leichter handhaben.

Zudem bringt TLA$^+$ laut Lamport [10] zwei weitere Nachteile mit sich:

1. Für viele Algorithmen sind informale Spezifikationen, die als Pseudocode geschrieben werden, leichter verständlich als mathematische.
2. Die meisten Menschen sind es nicht gewohnt mathematische Spezifikationen zu lesen. Sie bevorzugen welche, die mehr wie Programme aussehen.

Mit PlusCal hat Lamport daher eine weitere Sprache entwickelt, um formale Spezifikationen für Algorithmen zu schreiben.

4.4.2 PlusCal

Lamport bezeichnet PlusCal als *algorithm language* oder auf Deutsch Algorithmussprache. Eine Algorithmussprache ist dafür da, um Algorithmen zu schreiben, so wie eine Programmiersprache dazu da ist, um Programme zu schreiben. PlusCal hat zwei unterschiedliche Syntaxen. Die C-Syntax ist den C- basierten Programmiersprachen ähnlich, hier werden beispielweise Klammern verwendet, um den Beginn und das Ende von Blöcken zu markieren. Die P-Syntax verwendet stattdessen Begriffe wie begin und end. Aufgrund dessen ähnelt PlusCal einer sehr einfachen Programmiersprache, außer dass jeder TLA$^+$ Ausdruck als ein Ausdruck in einem PlusCal- Algorithmus verwendet werden kann. Diese Tatsache macht PlusCal deutlich ausdrucksstärker als jede Programmiersprache. Ein Algorithmus, der in

PlusCal geschrieben wurde, kann zudem in eine TLA$^+$ Spezifikation übersetzt und somit mit den entsprechenden TLA$^+$ Tools überprüft werden. PlusCal wurde entwickelt, um die TLA$^+$ Übersetzung eines Algorithmus einfacher und leicht verständlich zu machen. Die PlusCal-Version einer Spezifikation wird daher oftmals von Menschen bevorzugt, die sich mit der Mathematik weniger wohl fühlen.

5 Technische Grundlagen

Alle TLA$^+$ Tools können über eine Konsole, wie die Windows- Eingabeaufforderung, genutzt werden. Für den praktischen Einsatz von TLA$^+$ hat Leslie Lamport jedoch die TLA$^+$ Toolbox entworfen, über die sich alle TLA$^+$ Tools zentral und einfach bedienen lassen. Bei diesen Tools handelt es sich standardmäßig um den *TLA$^+$ Syntactic Analyzer*, den *TLC model checker*, den *PlusCal translator* und den *TLATeX Typesetter*.

5.1 Die TLA$^+$ Toolbox

Die TLA$^+$ Toolbox ist eine integrierte Entwicklungsumgebung, kurz IDE von englisch *integrated development environment*, für die TLA$^+$ Tools. Das Programm unterstützt den Benutzer in der Anwendung der TLA$^+$ Tools und kann unter anderem dazu verwendet werden, um TLA$^+$ Spezifikationen mit Hilfe eines Texteditors zu erstellen und zu bearbeiten und diese in einer formatierten Ausgabe anzuzeigen und zu speichern. Ein Texteditor oder kurz Editor ist ein Programm zum Erstellen und Bearbeiten von Texten, häufig Quellcodes. Im Gegensatz zu Textverarbeitungsprogrammen, wie Microsoft Word, speichern sie den Text als reinen ASCII- oder UTF-8- Text ab, d.h. ohne Formatierungen. Der Standard- Texteditor unter dem Betriebssystem Windows ist der Microsoft Editor, ein weiteres Beispiel ist die frei verfügbare Software Notepad++. Des Weiteren können über die Toolbox alle Tools, wie der *TLC model checker*, der *PlusCal translator* oder das *TLA$^+$ proof system* ausgeführt werden.

Die TLA$^+$ Toolbox arbeitet mit Ansichten. Eine Ansicht befindet sich immer in einem separaten Tab. Diese können mit einem Mausklick auf das X geschlossen oder per Drag-and-Drop verschoben werden. Ein Stern vor dem Dateinamen im Tab signalisiert nicht gespeicherte

Änderungen in der entsprechenden Ansicht. Die Ansichten sind in Ordnern gruppiert. Die einzelnen Fenster, in denen sich die Ordner befinden, können durch einen Mausklick auf das Mini- oder Maximierungssymbol ein- oder ausgeblendet werden.

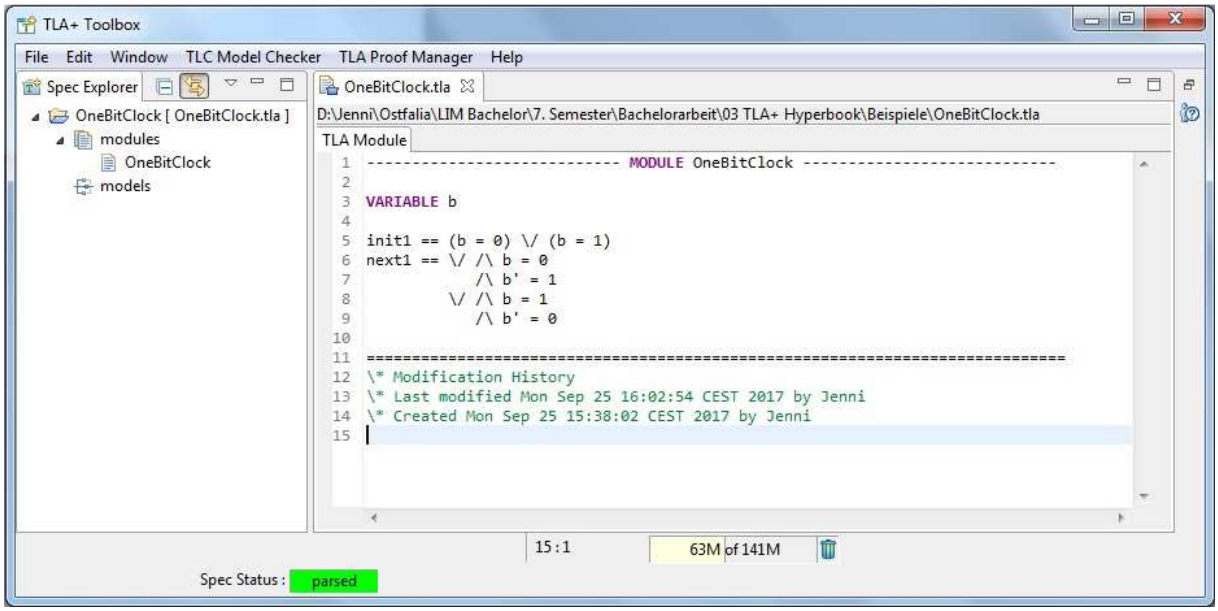


Abbildung 7: Die TLA+ Toolbox

Unter dem Menüpunkt *File - Preferences* können verschiedene Benutzereinstellungen vorgenommen werden: Hier finden sich unter anderem die Update- Einstellungen und verschiedene Möglichkeiten, um den Editor anzupassen. Es können beispielsweise voreingestellte graphische Benutzeroberflächen ausgewählt oder Farben und Schriftarten geändert werden, sowie diverse Optionen, wie z.B. *Show line numbers*, aktiviert oder deaktiviert oder Abkürzungen für Tastaturbelegungen eingesehen oder geändert werden. Unter dem Menüpunkt *Help* stehen zudem zahlreiche Anleitungen und Hilfen zur Verfügung.

Die TLA Toolbox kann über die Homepage von Leslie Lamport unter folgenden Links heruntergeladen werden:

- Installationsanweisungen:
  http://lamport.azurewebsites.net/tla/toolbox.html#downloading
- Download:
  https://github.com/tlaplus/tlaplus/releases/tag/v1.5.3

Es gibt Versionen für Linux, Mac und Windows. Eine klassische Installation ist nicht erforderlich. Die Zip- Datei muss lediglich entpackt und die Anwendung *toolbox.exe* ausgeführt werden. Einzige Voraussetzung für die Lauffähigkeit ist eine installierte Java Runtime Environment, Version 1.8. [8]

5.2 Die TLA+ Tools

Alle TLA+ Tools sind über die TLA+ Toolbox bedienbar und werden mit dem Download automatisch heruntergeladen. Eine Ausnahme bildet hier TLAPS, das *TLA+ proof system*, das bei Bedarf separat auf der TLAPS Webseite heruntergeladen werden kann und in dieser Arbeit nicht weiter besprochen wird.

5.2.1 Der TLA+ Syntactic Analyzer

Der *TLA+ Syntactic Analyzer*, kurz *SANY*, ist ein Programm zur Analyse von TLA+ Spezifikationen, mit dem TLA+ Spezifikationen geparst und auf Fehler überprüft werden. Parsen bedeutet nichts anderes, als die Syntax einer formalen Sprache zu analysieren. Mit Hilfe des *TLA+ Syntactic Analyzers* können zwei unterschiedliche Arten von Fehlern aufgedeckt werden:

1. Syntaktische Fehler
2. Semantische Fehler

Im ersten Fall führt ein Fehler zu einer grammatikalisch inkorrekten Spezifikation gemäß der TLA+ Syntax. Der Ausdruck „semantischer Fehler“ im zweiten Fall ist irreführend, denn er bedeutet nicht, dass die Spezifikation den falschen Sinn hat bzw. inhaltlich falsch ist, sondern dass bestimmte Rechtmäßigkeitsbedingungen von TLA+ verletzt sind. Das kann z.B. heißen, dass ein in der Spezifikation verwendeter Operator nicht definiert ist. Alle Fehler, die der *TLA+ Syntactic Analyzer* findet, sind entsprechend Fehler, die die TLA+ Spezifikation von vornherein ungültig machen, d.h. eine inhaltliche Prüfung auf Richtigkeit ist gar nicht erst notwendig, da die Spezifikation ungültig ist.

Ausgeführt wird der *TLA*$^+$ *Syntactic Analyzer* in der TLA$^+$ Toolbox beim Speichern der Spezifikation. Die Überprüfung einer Spezifikation erfolgt sequenziell und startet am Dateianfang. *SANY* meldet Fehler, wenn und genau dann, wenn er einen Punkt in der Spezifikation erreicht, an dem es beim Fortfahren unmöglich wird eine grammatikalisch korrekte Spezifikation zu erstellen. Um die Fehlerquelle für den Anwender besser lokalisierbar zu machen, wird eine Fehlermeldung mit Zeilenangabe ausgegeben. Der *TLA*$^+$ *Syntactic Analyzer* stoppt, wenn er den ersten syntaktischen Fehler nach obiger Definition gefunden hat. Umgekehrt können in einem Durchlauf mehrere semantische Fehler festgestellt werden. [5]

5.2.2 Der TLC model checker

Der *TLC model checker*, kurz *TLC*, ist ein Programm, mit dessen Hilfe inhaltliche Fehler in TLA$^+$ Spezifikationen gefunden werden können. Er dient als eine Art Modell- Prüfer und Simulator für TLA$^+$ Spezifikationen. Der effektivste Weg, um Fehler in TLA$^+$ Spezifikationen ausfindig zu machen, führt über den Versuch zu verifizieren, dass die Spezifikation alle Eigenschaften erfüllt, die sie erfüllen soll. Dies geschieht im *TLC model checker*, indem Verhalten erzeugt werden, die die Spezifikation erfüllen.

Es gibt zwei verschiedene Modi, in denen *TLC* verwendet werden kann:

1. im *model checking*- Modus
2. im *simulation*- Modus

Im ersten und voreingestellten Fall versucht der *TLC model checker* alle möglichen, erreichbaren Zustände zu finden, d.h. alle Zustände, die in allen Verhalten verkommen können, die die Spezifikation erfüllen. Im zweiten Fall des Simulationsmodus erstellt *TLC* willkürlich Verhalten, ohne dabei alle erreichbaren Zustände zu überprüfen. [5]

5.2.3 Der PlusCal translator

Der *PlusCal translator* ist ein Übersetzer, der von der Algorithmussprache PlusCal in die Spezifikationssprache TLA$^+$ übersetzt. Zu vergleichen ist der *PlusCal translator* daher am ehesten mit einem Compiler. Ein Compiler sorgt üblicherweise für die korrekte Übersetzung

des Quellcodes, der in einer Programmiersprache geschrieben ist, in Maschinensprache, die von einem Computer ausgeführt werden kann.

5.2.4 Der TLATeX Typesetter

Der *TLATeX Typesetter* ist ein Java Programm, mit dem TLA$^+$ Spezifikationen gesetzt werden können. Es ruft das von Lamport entwickelte Textsatzsystem *LaTeX* auf, welches den formatierten Schriftsatz erstellt. Eine Voraussetzung zur Verwendung von *TLATeX* ist dementsprechend die Installation von LaTeX und pdflatex, ein Programm, welches in den meisten Versionen von LaTeX enthalten ist und eine Konvertierung der LaTeX- Datei mit der Endung .dvi in eine pdf- Datei erlaubt. Eine beliebte Quelle ist MiKTeX, das unter dem folgenden Link kostenlos herunter geladen werden kann.

- https://miktex.org/download

Für das Betriebssystem Windows 7 mit 64 Bit ist die Version mit dem Namen *basic-miktex-2.9.6361-x64.exe* auszuwählen Zur Installation genügt ein Doppelklick auf die herunter geladene Anwendung und das Befolgen der vorgegebenen Installationsschritte.

Als Eingabedatei dient eine vollständige TLA$^+$ Spezifikation. Die Formatierung erfolgt automatisch auf die folgende Art. Sinnvolle Ausrichtungen bleiben dabei erhalten. Auch zusätzliche Leerzeichen werden in der Ausgabedabei zurückgegeben. Allerdings behandelt der *TLATeX Typesetter* kein Leerzeichen und ein Leerzeichen auf die gleiche Weise. [5]

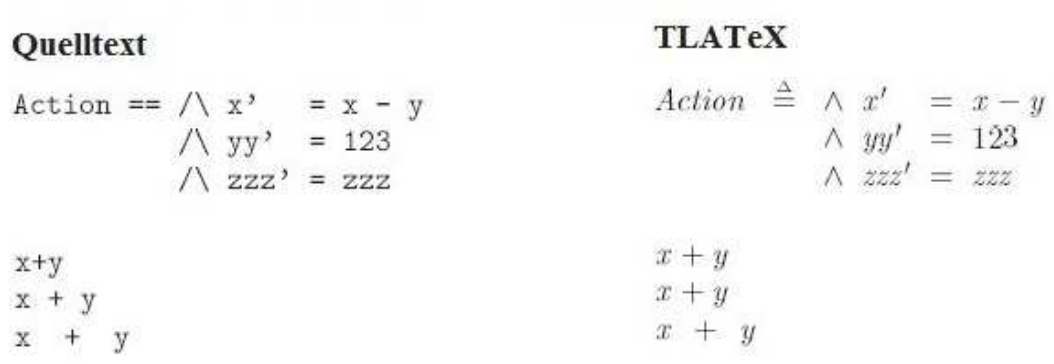


Abbildung 8: Beispiele für die Formatierung durch den TLATeX Typesetter

6 Die Erstellung einer TLA+ Spezifikation am Beispiel der Ein-Bit-Uhr

In seinem TLA+ Hyperbook [10] verwendet Lamport mit der Ein-Bit-Uhr ein einfaches Beispiel, um die Erstellung einer TLA+ Spezifikation in der TLA+ Toolbox zu erklären. Es handelt sich dabei um die einfachste Uhr überhaupt, denn sie zeigt als „Zeiten“ lediglich eine Null oder eine Eins an. Solche Uhren steuern z.B. Computer. Eine echte Uhr würde mit einer annähernd konstanten Geschwindigkeit ticken. Die Ein-Bit-Uhr als Rechengerät wechselt lediglich zwischen den zwei Zuständen, in denen sie entweder eine Null oder eine Eins anzeigt. Da es in diesem einfachen Beispiel keine Nebenläufigkeit zu geben scheint –die Uhr macht eine Sache auf einmal, während ein System eine beliebige Anzahl an Sachen auf einmal machen kann– wird von der Annahme ausgegangen, dass die Eins ein einfacher Spezialfall von einer beliebigen Zahl ist.

6.1 Das Verhalten der Ein-Bit-Uhr

Symbol	Bedeutung
=	Gleichheitsbeziehung
→	Zustandsübergang

Tabelle 1: Im Folgenden verwendete Symbole

Das Verhalten der Uhr wird im Standardmodell beschrieben. Die Uhr selber wird dabei mit einer einzigen Variable b modelliert, die sozusagen das Ziffernblatt darstellt. Die Wertzuweisung $b = 0$ repräsentiert die Anzeige der Zahl Null, die Wertzuweisung $b = 1$ die Anzeige der Zahl Eins auf dem Ziffernblatt. Bei den Wertzuweisungen handelt es sich jeweils um einen Zustand.

Davon ausgehend, dass die Uhr eine Null im Display anzeigt, kann das Verhalten der Uhr wie folgt beschrieben werden:

$$b = 0 \rightarrow b = 1 \rightarrow b = 0 \rightarrow b = 1 \rightarrow \ldots$$

Beginnt die Anzeige mit der Eins, sieht das Verhalten folgendermaßen aus:

$$b = 1 \rightarrow b = 0 \rightarrow b = 1 \rightarrow b = 0 \rightarrow \ldots$$

Die drei Punkte „…“ am Ende bedeuten, dass die Uhr aus diese Weise für immer läuft. Diese beiden Verhalten der Uhr sind die einzig möglichen für eine Ein-Bit-Uhr. Mit „Verhalten der Uhr“ ist korrekterweise das „Verhalten im Standardmodell einer Abstraktion einer realen Uhr“ gemeint. Die Übergänge zwischen den Zuständen und somit auch die dazwischenliegenden Werte, werden im Modell als eigenständige Schritte bzw. Zustandsänderung beschrieben.

6.2 Die Beschreibung des Verhaltens

Symbol	Bedeutung	TLA+
=	Gleichheitsbeziehung	=
∧	Konjunktion bzw. Und	/\
∨	Disjunktion bzw. Oder	\/

Tabelle 2: Im Folgenden verwendete Symbole

Um ein Rechengerät zu beschreiben, müssen alle seine möglichen Verhalten beschrieben werden. Allerdings ist es, vor allem bei komplexen Geräten, nicht immer möglich alle möglichen Verhalten aufzuführen. Es gibt praktische Sprachen, um Rechengeräte zu beschreiben, die zwei Dinge spezifizieren:

1. die möglichen Startzustände
2. die möglichen Schritte, d.h. Übergänge von einem Zustand in einen anderen.

Beides wird mit Hilfe der Mathematik beschrieben. Dazu werden die booleschen Operatoren Und, auch Konjunktion genannt, dargestellt durch ein ∧, und Oder, auch Disjunktion genannt, dargestellt durch ein ∨, benutzt. Es handelt sich dabei um binäre Operatoren, die einen Wahrheitswert zurückgeben, in dem booleschen Fall entweder „wahr“ oder „falsch“. Am Beispiel der Ein-Bit-Uhr sieht die Formel für die möglichen Startzustände, die auch *initial predicate* oder Anfangsaussage genannt wird, folgendermaßen aus:

$$(b = 0) \vee (b = 1)$$

Damit wird die Behauptung beschrieben, dass der Startwert der Variable b gleich 0 oder 1 ist. Mathematisch korrekt kann der Startwert der Variable in dem Fall auch gleich 0 und 1 sein. Bei der Ein-Bit-Uhr ist aber davon auszugehen, dass der Startwert entweder den Wert 0 oder den Wert 1 annehmen kann, da es sich um ein vereinfachtes Beispiel handelt.

Um die möglichen Zustandsübergänge zu beschreiben, wird eine mathematische Formel benötigt, die sich auf die Werte von b in zwei Zuständen bezieht, sprich auf den ersten Zustand in diesem Schritt und den nächsten Zustand in diesem Schritt. Hierfür wird die Variable b' eingeführt. b ist also der Wert von b in dem ersten Zustand, b' entsprechend der Wert im nächsten Zustand. In dem Uhren- Beispiel gibt es zwei mögliche Schritte - einen, bei dem b = 0 und b' = 1 ist und einen, bei dem b = 1 und b' = 0 ist. Die folgende Formel, die *next state action* oder Nächster-Zustand-Aktion genannt wird, beschreibt die möglichen Schritte:

$$((b = 0) \wedge (b' = 1)) \vee ((b = 1) \wedge (b' = 0))$$

Da diese Schreibweise mit den vielen runden Klammern schnell unübersichtlich wird, erlaubt TLA$^+$ diese Formeln als Aufzählung zu schreiben. Die booleschen Operatoren dienen dabei als eine Art Aufzählungszeichen.

$$\begin{aligned} &\vee \wedge b = 0 \\ &\quad \wedge b' = 1 \\ &\vee \wedge b = 1 \\ &\quad \wedge b' = 0 \end{aligned}$$

6.3 Das Schreiben der Spezifikation

Symbol	Bedeutung	TLA+
=	Gleichheitsbeziehung	=
∧	Konjunktion bzw. Und	/\
∨	Disjunktion bzw. Oder	\/
≜	Definition	==

Tabelle 3: Im Folgenden verwendete Symbole

Nachdem in der TLA$^+$ Toolbox über die Menüschritte *File – Open Spec – Add New Spec* eine neue Spezifikation mit dem Namen *OneBitClock* für das Uhren- Beispiel angelegt wurde, öffnet sich der Editor. Die erste Linie in Zeile 1 ist die Modul- Eröffnung, die letzte Linie in Zeile 6 ist das Modul- Ende:

```
TLA Module
1  ---------------------------- MODULE OneBitClock ----------------------------
2
3
4
5
6  ==============================================================================
7  \* Modification History
8  \* Last modified Tue Jan 23 17:31:01 CET 2018 by Jenni
9  \* Created Mon Sep 25 15:38:02 CEST 2017 by Jenni
10
```

Abbildung 9: Eine leere Spezifikation in der TLA+ Toolbox

Die Linien müssen jeweils mindestens aus 3 Zeichen bestehen und können eine beliebige Länge haben. Nur der Text, der zwischen den beiden Linien steht, wird berücksichtig. Text vor der Eröffnung oder nach dem Ende des Moduls wird ignoriert.

Als nächstes werden der Anfangsaussage und der Nächster-Zustand-Aktion Namen zugeordnet. Diese können z.B. $init1$ und $next1$ sein. Eine Nummerierung ist sinnvoll, wenn es jeweils mehrere Aussagen und Aktionen gibt. Irgendwo in der Spezifikation müssen $init1$ und $next1$ zudem definiert werden. Die Definition erfolgt mit Hilfe eines Entspricht- Zeichens $\triangleq$, das nach dem *American Standard Code for Information Interchange*, kurz ASCII, in TLA$^+$ als == dargestellt wird. Da jedes Symbol, das in dem Modul vorkommt, entweder ein TLA$^+$ Operator oder deklariert sein muss, werden am besten gleich zu Beginn des Moduls die Variablen deklariert. Dies geschieht mit der Deklaration VARIABLE und dem Variablennamen. Für die Ein-Bit-Uhr sieht die Spezifikation nun folgendermaßen aus:

```
TLA Module
1  ---------------------------- MODULE OneBitClock ----------------------------
2
3  VARIABLE b
4
5  init1 == (b = 0) \/ (b = 1)
6  next1 == \/ /\ b = 0
7             /\ b' = 1
8          \/ /\ b = 1
9             /\ b' = 0
10
11 ==============================================================================
```

Abbildung 10: Die Spezifikation der Ein-Bit-Uhr in der TLA+ Toolbox

Wenn von Spezifikation gesprochen wird, ist nun entweder

1. das vollständige Modul gemeint oder
2. die Anfangsaussage und die Nächster-Zustand-Aktion.

Um Verwirrungen zu vermeiden, sollte im ersten Fall eher von Modul als von Spezifikation gesprochen werden. Im zweiten Fall sollte der Ausdruck Verhaltensspezifikation verwendet werden.

6.4 Die formatierte Ausgabe der Spezifikation

Es ist möglich die ASCII Version der Spezifikation bzw. des Moduls, das gerade bearbeitet wird, in der Toolbox in einer formatierten Ausgabe anzeigen zu lassen. Um eine formatierte Ausgabe des Moduls zu erstellen, ist der Punkt *Produce PDF Version* in dem Menü *File* auszuwählen. Die Ausgabe wird in einem separaten Fenster innerhalb der Toolbox angezeigt. Es ist möglich zwischen den Ansichten zu wechseln, indem entweder das TLA$^+$ Modul- oder das PDF Viewer- Tab in der linken oberen Ecke des Modul- Fensters aufgerufen wird. Das Bearbeiten des ASCII Version ändert nicht automatisch die formatierte Ausgabe. Diese muss manuell aktualisiert werden, indem der Befehl *Produce PDF Version* erneut ausgeführt wird. Eine PDF- Datei der Ausgabe wird automatisch erzeugt und in dem gleichen Verzeichnis abgelegt, wie die Modul- Datei mit der Endung .tla.

OneBitClock.tla | OneBitClock.pdf

MODULE *OneBitClock*

VARIABLE b

$$
\begin{aligned}
init1 &\triangleq (b = 0) \lor (b = 1) \\
next1 &\triangleq \lor \land b = 0 \\
&\qquad\;\; \land b' = 1 \\
&\qquad \lor \land b = 1 \\
&\qquad\;\; \land b' = 0
\end{aligned}
$$

* Modification History
* Last modified *Mon Sep* 25 16:02:54 *CEST* 2017 by *Jenni*
* Created *Mon Sep* 25 15:38:02 *CEST* 2017 by *Jenni*

Abbildung 11: Die formatierte Ausgabe am Beispiel der Ein-Bit-Uhr

6.5 Die Überprüfung der Spezifikation

Symbol	Bedeutung	TLA+
=	Gleichheitsbeziehung	`=`
∧	Konjunktion bzw. Und	`/\`
∨	Disjunktion bzw. Oder	`\/`
≜	Definition	`==`
∈	Element von	`\in`

Tabelle 4: Im Folgenden verwendete Symbole

Um die Spezifikation zu überprüfen, kommt nun der *TLC Model Checker* zum Einsatz. Hierfür wird unter dem Menüpunkt *TLC Model Checker* der Punkt *New Model* ausgewählt. Nachdem das Modell benannt wurde, öffnet sich ein neues Fenster mit einem Editor in der TLA$^+$ Toolbox. Dieser Editor hat drei Seiten. Auf der ersten Seite *Model Overview* wird in dem Unterabschnitt *inital predicate and next-state relation* unter *Init* der Name der Anfangsaussage und unter *Next* der Name der Nächster-Zustand-Aktion eingegeben. Mit dem Klick auf den grünen Startbutton im oberen linken Bereich der Seite wird der *TLC Model Checker* ausgeführt. Ein fehlerfreier Durchlauf bedeutet, dass das Modell semantisch richtig ist. Präziser ausgedrückt, bedeutet es, dass die Spezifikation für eine Sammlung von Verhalten vollständig ausführbar ist. Ist sie das

nicht, wird eine Fehlermeldung ausgegeben. Ein Doppelklick auf diese Fehlermeldung öffnet das TLA$^+$ Modul mit einer Markierung an der entsprechenden Stelle. Auf der Seite *Model Checking Results* findet sich der Unterabschnitt *Statistics*. Am Beispiel der Ein-Bit-Uhr kann hier folgendes abgelesen werden: Der *TLC Model Checker* hat zwei eindeutige Zustände gefunden, abzulesen in der Spalte *Distinct States*. In der Spalte *Diameter* steht die Anzahl an Schritten, die bei der Ausführung der Uhr möglich sind, bevor sich Zustände wiederholen.

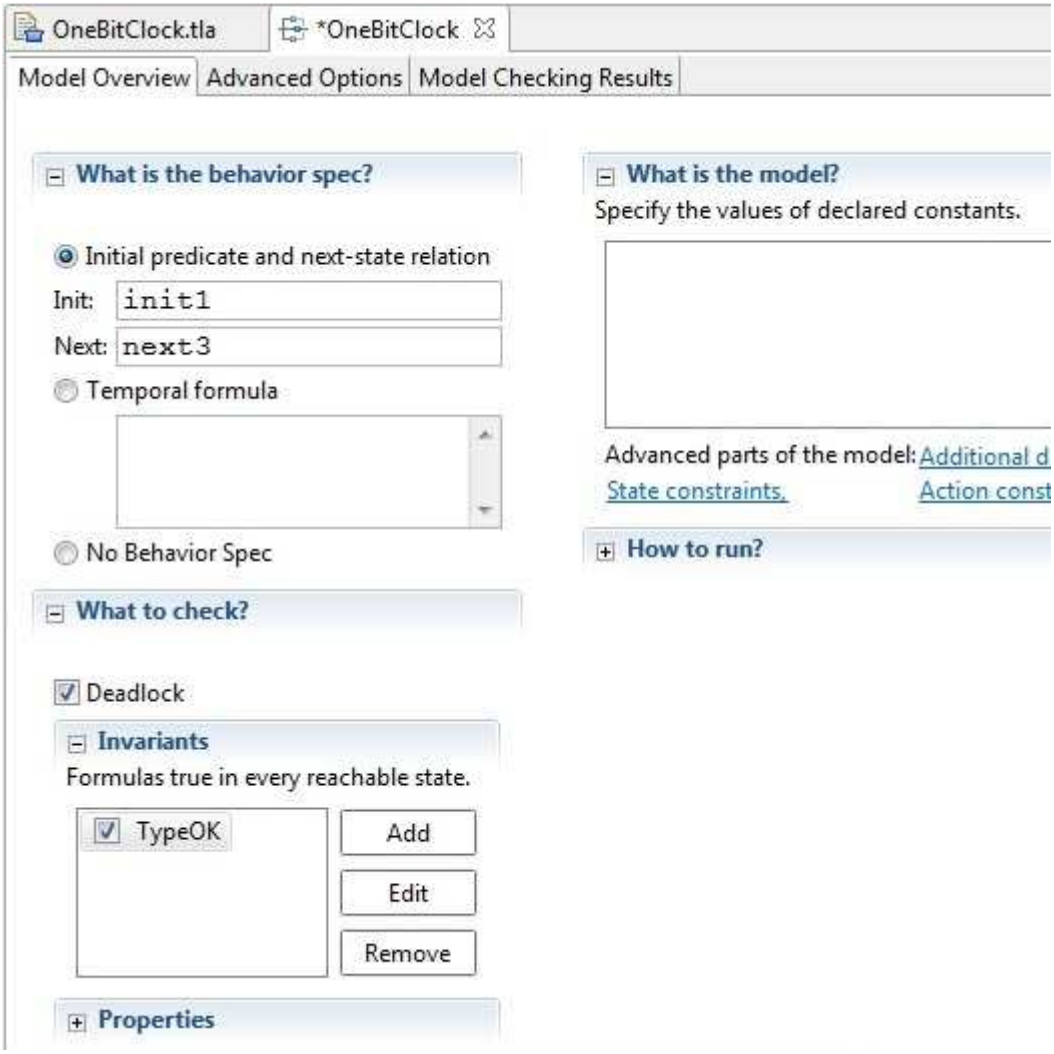


Abbildung 12: Die Model Overview- Seite inklusive Eingaben in TLC

Eine Eigenschaft sollte zudem bei jeder Spezifikation geprüft werden, die „type correct"-Eigenschaft. Typen- Korrektheit bedeutet in dem Fall einer TLA$^+$ Spezifikation, dass der Wert jeder Variable in jedem Zustand eines jeden Verhaltens ein Element in der Menge der Werte

ist, die für die Variablen vorgegeben wurde. Sprich der Wert im vorgegebenen Wertebereich liegt, ähnlich wie bei einer Domäne bei relationalen Datenbanken oder Datentypen in Programmiersprachen. Weil es in TLA$^+$, anders wie in beispielweise Java oder C, keine Datentypen gibt, gibt es auch keine Typ- Deklarationen. Eine „type correct“- Deklaration ist im Beispiel der Ein-Bit-Uhr eigentlich überflüssig, weil der Wertebereich von der Spezifikation und den Definitionen der Anfangsaussage und der Nächster-Zustand-Aktion impliziert wird. Da diese Definitionen bei vielen realen Spezifikationen allerdings schwer zu verstehen sind, ohne dass die Wertemenge der Variablen bekannt ist, ist es ratsam dem Leser der Spezifikation diese Information gleich zu Beginn mitzugeben. Dafür wird die TypeOK- Definition direkt unterhalb der Variablen- Definition in der Spezifikation ergänzt, sowie auf der Seite *Model Overview* im *TLC model checker* unter dem Punkt *Invariants* eingegeben.

TLA Module

```
1  ---------------------------- MODULE OneBitClock ----------------------------
2
3  VARIABLE b
4
5  TypeOK == b \in {0,1}
6
7  init1 == (b = 0) \/ (b = 1)
8  next1 == \/ /\ b = 0
9             /\ b' = 1
10         \/ /\ b = 1
11            /\ b' = 0
12
13 ==============================================================================
```

Abbildung 13: Die Spezifikation der Ein-Bit-Uhr in der TLA+ Toolbox inkl. TypeOK- Definition

Eine Formel, die für alle Zustände und Verhalten, die von der Spezifikation erlaubt sind, wahr ist, heißt invariant.

6.6 Die Berechnung des Verhaltens der Spezifikation

Im vorherigen Kapitel hat der *TLC Model Checker*, kurz *TLC*, überprüft, dass $TypeOK$ ein Invariant der Spezifikation einer Ein-Bit-Uhr ist, d.h. dass es in allen Zuständen in allen Verhalten der Spezifikation wahr ist. Bei der Überprüfung berechnet *TLC* alle möglichen Verhalten, die die Anfangsaussage $init1$ und die Nächster-Zustand-Aktion $next1$ erfüllen. Am Uhrenbeispiel sehen diese Berechnungen wie folgt aus. Zunächst wird ein mögliches Verhalten

berechnet. Um die Spezifikation zu erfüllen, muss der erste Zustand dieses Verhaltens die Anfangsaussage $init1$ erfüllen. Es muss also eine entsprechende Wertzuweisung zu der Variable b gefunden werden. Sowohl b = 1, als auch b = 0 erfüllen $init1$; in willkürlicher Weise wird als erstes b = 1 ausgewählt. Als nächstes muss ein möglicher zweiter Zustand gefunden werden. Damit ein Verhalten die Spezifikation erfüllt, muss jedes Paar aufeinanderfolgender Zustände die Nächster-Zustand-Aktion $next1$ erfüllen. Konkret bedeutet das: Um den zweiten Zustand zu bestimmen, muss ein Wert für b' gefunden werden, der $next1$ erfüllt, wenn b den Wert 1 hat. Dieser Wert wird gefunden, indem b in $next1$ durch den Wert 1 ersetzt und die Formel folgendermaßen vereinfacht wird.

1. Für b = 1 einsetzen:

$$\begin{array}{l} \vee \wedge\ 1 = 0 \\ \quad \wedge\ b' = 1 \\ \vee \wedge\ 1 = 1 \\ \quad \wedge\ b' = 0 \end{array}$$

2. Formel vereinfachen:

$$\begin{array}{l} \vee \wedge\ FALSE \\ \quad \wedge\ b' = 1 \\ \vee \wedge\ TRUE \\ \quad \wedge\ b' = 0 \end{array}$$

$$\begin{array}{l} = \vee\ FALSE \\ \quad \vee\ b' = 0 \end{array}$$

$$= b' = 0$$

Diese Berechnungen zeigen, dass, wenn b = 1 ist, der einzige Wert für b' = 0 ist, damit $next1$ wahr ist. Folglich ist der zweite Zustand des Verhaltens b = 0. Um den dritten Zustand des Verhaltens herauszufinden, wird b nun durch 0 ersetzt und die obigen Berechnungen erneut durchgeführt. Auf diese Weise wird auch hier ein Wert für b' gefunden, so dass $next1$ wahr ist. Die ersten drei Zustände des Verhaltens sind demzufolge

$$b = 1 \rightarrow b = 0 \rightarrow b = 1.$$

Anstatt den vierten Zustand zu berechnen, kann an dieser Stelle bereits angenommen werden, dass ein unendliches Verhalten bestimmt wurde. Um wirklich alle möglichen Verhalten zu

bestimmen, müssen die Berechnungen auch für den zweiten möglichen Startzustand mit $b = 0$ durchgeführt werden.

In *TLC* wird für diese Berechnungen ein gerichteter Graph G erstellt, der auch Zustandsgraph genannt wird, d.h. die Knoten stellen Zustände dar. (1) Zunächst werden alle möglichen Anfangszustände von Verhalten berechnet, für die die Anfangsaussage wahr ist, und diese zu G hinzugefügt. (2) Danach werden für jeden Zustand s in G alle möglichen Folgezustände t berechnet, so dass $s \rightarrow t$ ein Schritt in einem Verhalten sein kann. (3) Jeder Zustand t wird nun zu G hinzugefügt, sofern dieser noch nicht in G vorhanden ist, und eine Kante von s zu t gezeichnet. (4) Diese Vorgehensweise wird so lange wiederholt bis keine neuen Knoten und Kanten zu G hinzugefügt werden. Falls und wenn der Berechnungsprozess beendet wurde, bestehen die Knoten in G aus allen erreichbaren Zuständen der Spezifikation. Jedes Verhalten, das die Spezifikation zulässt, kann in dem gerichteten Graphen gefunden werden, indem man bei einem Anfangszustand bzw. -knoten startet und einem Weg in G folgt.

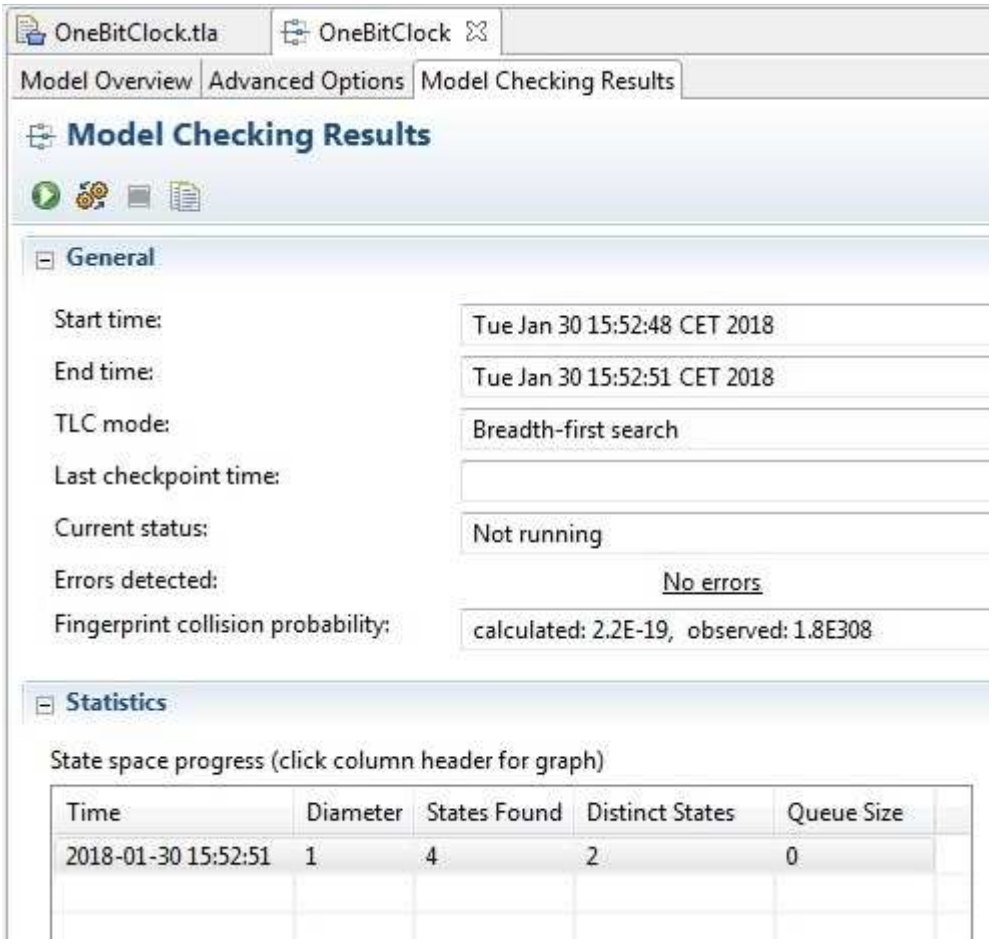


Abbildung 14: Die Model Checking Results- Seite inklusive State Space Progress- Tabelle in TLC

Die Tabelle *State Space Progress* in dem Unterabschnitt Statistics der Model Checking Results-Seite gibt folgende Informationen über den Graphen G wieder.

- *Diameter* beschreibt die Anzahl der Zustände in dem längsten Weg in G, in dem kein Zustand zweimal vorkommt.
- Bei *States Found* handelt es sich um die Gesamtanzahl der Zustände, die unter (1) oder unter (2) als Nachfolger- Zustand t geprüft werden.
- *Distinct States* gibt die Anzahl der Zustände an, die die Menge der Knoten in G bilden.
- *Queue Size* beschreibt die Anzahl der Zustände s in G, für die (2) noch nicht ausgeführt wurde.

6.7 Andere Möglichkeiten das Verhalten einer Spezifikation zu beschreiben

Symbol	**Bedeutung**	**TLA+**
$=$	Gleichheitsbeziehung	`=`
$\wedge$	Konjunktion bzw. Und	`/\`
$\vee$	Disjunktion bzw. Oder	`\/`
$\triangleq$	Definition	`==`
$\in$	Element von	`\in`
$\equiv$	Äquivalenz	`<=>`
mod	Divisionsrest	`%`

Tabelle 5: Im Folgenden verwendete Symbole

Sowohl die Formel $init1$ mit $(b = 0) \vee (b = 1)$, als auch die Formel $TypeOK$ mit $(b \in \{0, 1\})$ sagen aus, dass die Variable b entweder den Wert 1 oder den Wert 0 annimmt. Sie sind äquivalent, d.h. beide Formeln sind untereinander austauschbar.

$$(b = 0) \vee (b = 1) \equiv (b \in \{0, 1\})$$

Es gibt eine Vielzahl an verschiedenen Möglichkeiten die Nächster-Zustand-Aktion darzustellen. Zur Erinnerung: In dem Beispiel der Ein-Bit-Uhr besagt $next1$, dass b' gleich 1

ist, wenn b gleich 0 ist und b' gleich 0 ist, wenn b gleich 1 ist. Da in jedem Zustand des Verhaltens b entweder 1 oder 0 ist, ist eine äquivalente Schreibweise: b' ist gleich 1, wenn b gleich 0 ist, sonst ist b' gleich 0. Dies wird mit der Formel $next2$ ausgedrückt.

TLA Module

```
1  ---------------------------- MODULE OneBitClock ----------------------------
2
3  VARIABLE b
4
5  TypeOK == b \in {0,1}
6
7  init1 == (b = 0) \/ (b = 1)
8  next1 == \/ /\ b = 0
9             /\ b' = 1
10         \/ /\ b = 1
11            /\ b' = 0
12 next2 == b' = IF b = 0 THEN 1 ELSE 0
13
14 ==============================================================================
```

Abbildung 15: Die Spezifikation der Ein-Bit-Uhr in der TLA+ Toolbox mit alternativer next2- Definition

Anders als $init1$ und $TypeOK$, sind die zwei Formeln $next1$ und $next2$ nicht äquivalent. Dennoch ergeben sich daraus zwei äquivalente Spezifikationen, d.h. zwei Spezifikationen, die die gleiche Menge an erlaubten Verhalten haben.

Wie dieser konkrete Fall zeigt, können also auch zwei nicht äquivalente Nächster-Zustand-Aktionen äquivalente Spezifikationen ergeben. Umgekehrt können zwei nicht äquivalente Anfangsaussagen keine äquivalente Spezifikation hervorbringen, weil es hier immer einen Zustand geben würde, der gemäß der einen Anfangsaussage zulässig ist, aber gemäß der anderen nicht. Beginnt man nun von diesem Zustand aus ein Verhalten zu konstruieren, ist dieses zwar von einer Spezifikation zugelassen und von der anderen aber nicht.

Eine andere Methode, um die Nächster-Zustand-Aktion zu beschreiben, ist die Verwendung des Modulo Operators. Da der Divisionsrest von 1 mod 2 gleich 1 und der Divisionsrest von 2 mod 2 gleich 0 ist, ist es einfach zu überprüfen, dass $next1$ und $next2$ äquivalent zu der folgenden Formel $next3$ sind, wenn b gleich 0 oder 1 ist.

```
next3 == b' = (b + 1) % 2
```

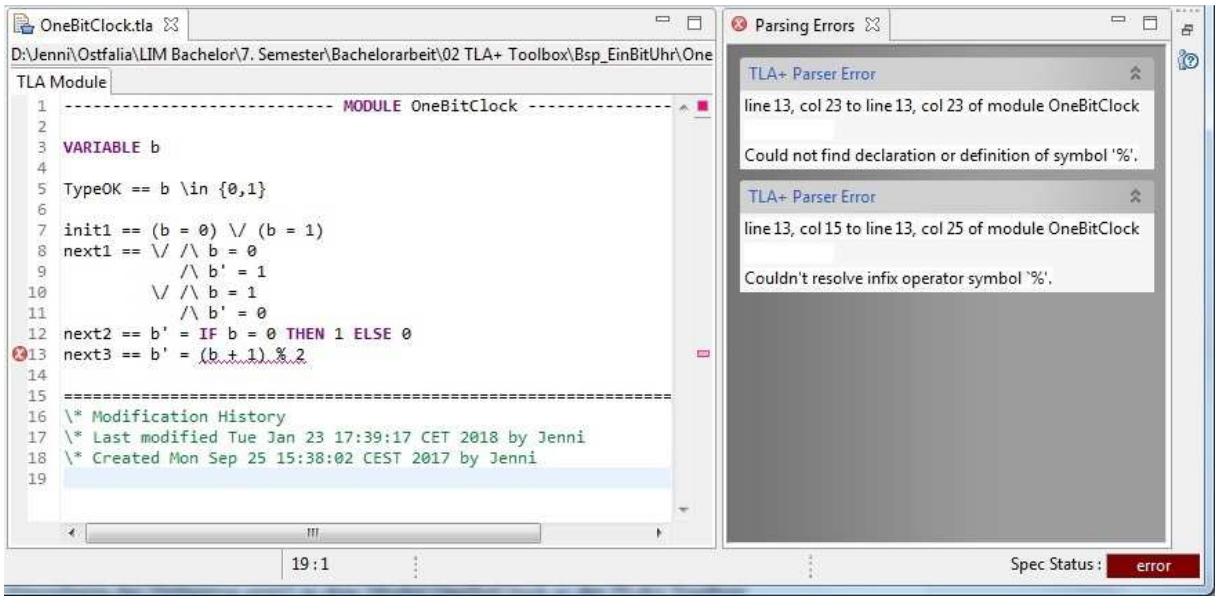


Abbildung 16: Die Spezifikation der Ein-Bit-Uhr in der TLA+ Toolbox mit alternativer next3- Definition und Fehlermeldung

Das Hinzufügen der Definition $next3$ in dem Modul *OneBitClock.tla* in der TLA$^+$ Toolbox erzeugt ein Fehler beim Parsen, nämlich dass der Modulo- Operator % nicht definiert sei. Das liegt daran, dass die gängigen arithmetischen Operatoren, darunter beispielsweise auch die Operatoren + und -, keine in TLA$^+$ integrierten Standardoperatoren sind. Diese müssen aus dem Integers- Standardmodul mit Hilfe des EXTENDS- Befehls importiert werden, der zu Beginn des Moduls, d.h. noch vor der Deklaration der Variablen, eingefügt werden muss.

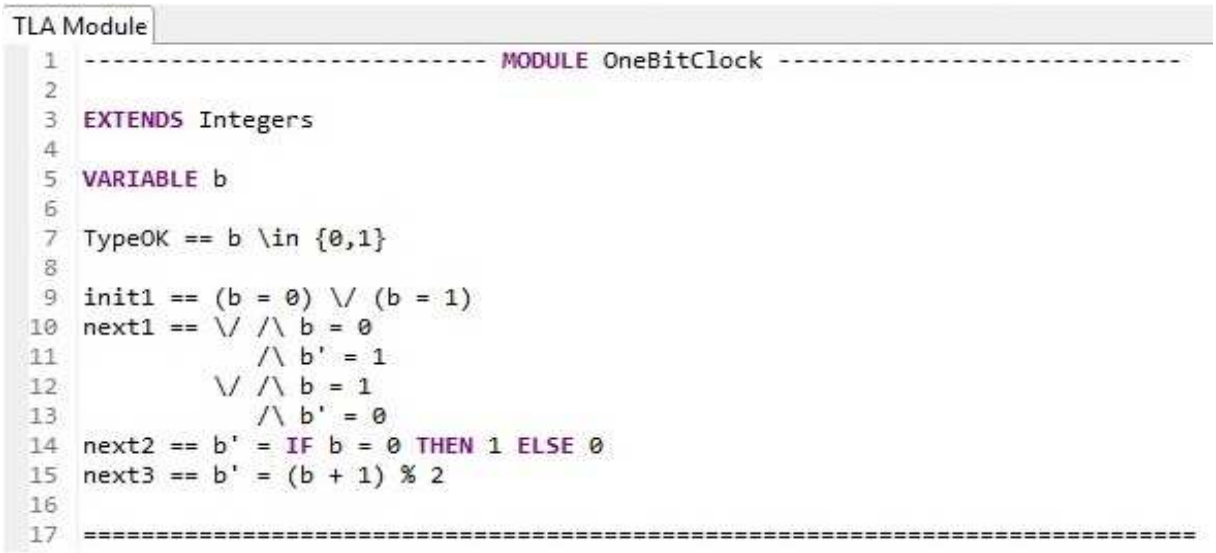


Abbildung 17: Die Spezifikation der Ein-Bit-Uhr in der TLA+ Toolbox mit alternativer next3- Definition ohne Fehlermeldung

6.8 Die Spezifizierung der Ein-Bit-Uhr in PlusCal

Symbol	Bedeutung
=	Gleichheitsbeziehung
:=	Wertzuweisung

Tabelle 6: Im Folgenden verwendete Symbole

Im nächsten Schritt wird die Ein-Bit-Uhr als PlusCal- Algorithmus näher beschrieben. Hierfür wird in der TLA$^+$ Toolbox über die Menüschritte *File – Open Spec – Add New Spec* eine neue Spezifikation inklusive Root- Modul mit dem Namen *PCalOneBitClock* angelegt. Es öffnet sich der Editor. Der Algorithmus wird in ein mehrzeiliges Kommentarfeld geschrieben, das mit der Symbolkombination (* beginnt und mit der Symbolkombination *) endet. Um das Kommentarfeld einzufügen, muss zunächst der Cursor innerhalb des Moduls platziert werden. Mit einem Rechtsklick wird nun ein Menü geöffnet und der Punkt *Start Boxed Comment* ausgewählt. Die Datei sieht danach folgendermaßen aus.

```
TLA Module
1  ------------------------- MODULE PCalOneBitClock -------------------------
2
3  (***************************************************************************
4
5
6
7   ***************************************************************************)
8
9  ===========================================================================
```

Abbildung 18: Eine leere Spezifikation in PlusCal

Für den PlusCal- Algorithmus wird ein beliebiger Name ausgewählt und dieser mit dem vorangestellten Zusatz --algorithm versehen, der keinerlei Bedeutung hat. Es handelt sich dabei lediglich um vorgeschriebene Syntax. Der Körper des Algorithmus steht zwischen zwei geschweiften Klammern und beginnt mit der Deklaration der Variablen, in dem Beispiel der Ein-Bit-Uhr der Variable *b*, und dem Festlegen des Wertebereiches aller möglichen Anfangswerte. Danach folgt, ebenfalls in geschweiften Klammern, der ausführbare Code. Dieser ähnelt bei Verwendung der C- Syntax C oder einer C- basierten Sprache, wie z.B. Java. Der wesentliche Unterschied besteht darin, dass in PlusCal eine Gleichheitsbeziehung mit

einem Gleichheitszeichen beschrieben wird; eine Wertzuweisung erfolgt mit einem Doppelpunkt gefolgt von einem Gleichheitszeichen. Kurz: In PlusCal werden die Operatoren wie folgt verwendet.

- Für eine Gleichheitsbeziehung ein =, anstatt ein ==
- Für eine Wertzuweisung ein :=, anstatt ein =

Nach dem Speichern des Moduls wird der *PlusCal translator* über das Menü *File – Translate PlusCal Algorithm option* aufgerufen. Der Übersetzer fügt die TLA$^+$ Übersetzung des PlusCal-Algorithmus am Ende zwischen den Kommentarzeilen * BEGIN TRANSLATION und * END TRANSLATION ein. Wenn die Datei diese bereits enthält, wird die bestehende Übersetzung dazwischen vollständig durch die neue ersetzt. Die wichtigsten Teile der Übersetzung sind die Deklaration der Variablen und die Definitionen der Anfangsaussage und Nächster-Zustand-Aktion. Die Übersetzung erstellt weiterhin Definitionen für die Symbole *var* und *Spec*.

Am Beispiel der Ein-Bit-Uhr sieht der PlusCal- Algorithmus vor der Übersetzung durch den *PlusCal translator* folgendermaßen aus.

TLA Module

```
1  ------------------------- MODULE PCalOneBitClock -------------------------
2
3  (***************************************************************************
4
5  --algorithm Clock {
6     variable b \in {0, 1};
7     { while (TRUE) { if (b = 0) b := 1 else b := 0
8                    }
9     }
10  }
11
12  ***************************************************************************)
13 ===========================================================================
```

Abbildung 19: PlusCal- Algorithmus vor der Übersetzung durch den PlusCal translator

Nach der Übersetzung durch den *PlusCal translator* sieht das Ganze wie folgt aus.

```
TLA Module
1  ---------------------------- MODULE PCalOneBitClock ----------------------------
2
3  (***************************************************************************
4
5  --algorithm Clock {
6     variable b \in {0, 1};
7     { while (TRUE) { if (b = 0) b := 1 else b := 0
8                    }
9     }
10  }
11
12  ***************************************************************************)
13 \* BEGIN TRANSLATION
14 VARIABLE b
15
16 vars == << b >>
17
18 Init == (* Global variables *)
19         /\ b \in {0, 1}
20
21 Next == IF b = 0
22            THEN /\ b' = 1
23            ELSE /\ b' = 0
24
25 Spec == Init /\ [][Next]_vars
26
27 \* END TRANSLATION
28
29 =============================================================================
```

Abbildung 20: PlusCal- Algorithmus nach der Übersetzung durch den PlusCal translator

6.9 Die TLA+ Übersetzung

Die TLA+ Übersetzung beschreibt präzise die Bedeutung des PlusCal- Algorithmus. Sie beginnt mit der Deklaration der Variablen. Im Fall der Ein-Bit-Uhr erfolgt die Deklaration der Variable b in Zeile 14 der Abbildung 20.

```
VARIABLE b
```

Im Anschluss an die Deklaration wird in Zeile 16 der Bezeichner $vars$ definiert. Die Definition geschieht mit Hilfe eines Tupels, das in eckigen Klammern geschrieben wird.

```
vars == << b >>
```

Als nächstes werden die Anfangsaussage und die Nächster-Zustand-Aktion in der TLA+ Übersetzung definiert, im Beispiel erfolgt die Definition in Zeile 18 und 21 der Abbildung 20.

```
Init == (* Global variables *)
/\ b \in {0, 1}

Next == IF b = 0
THEN /\ b' = 1
ELSE /\ b' = 0
```

Am Ende der Übersetzung, hier in Zeile 25, wird für den Bezeichner $Spec$ eine temporale Formel definiert. Diese ist eine Aussage über die Verhalten, denn die Formel ist für ein Verhalten entweder wahr oder falsch. Sie ist wahr, wenn das Verhalten eine mögliche Ausführung des Algorithmus ist, d.h. wenn $Init$ für den ersten Zustand des Verhaltens und $[Next]\_vars$ für jeden Schritt in dem Verhalten wahr ist. Mit anderen Worten bedeutet das: Die Formel $Spec$ ist die TLA$^+$ Verhaltensspezifikation des Algorithmus. [10]

```
Spec == Init /\ [][Next]_vars
```

7 Ein weiteres Beispiel: Fakultät einer natürlichen Zahl

Das obige Beispiel der Ein-Bit-Uhr stammt aus dem TLA$^+$ Hyperbook. Um das Verständnis noch tiefer zu festigen und das Gelernte direkt anzuwenden, dient im Folgenden das zweite, etwas anspruchsvollere Beispiel der Fakultät einer natürlichen Zahl. Die Fakultät einer natürlichen Zahl n aus der Menge $\mathbb{N}$ der natürlichen Zahlen lässt sich auf zwei Arten berechnen:

1. Als Produkt der natürlichen Zahlen von 1 bis n mit

$$n! = \prod_{i=1}^{n} i$$

Wenn die Menge der natürlichen Zahlen als $\mathbb{N}_o$ definiert ist und n gleich Null ist, gilt $0! = 1$, da das leere Produkt immer 1 ergibt.

2. Rekursiv mit

$$n! = \begin{cases} 1 & \text{für } n = 0 \\ n * (n-1)! & \text{für } n > 0 \end{cases}$$

In einem C- Programm kann die Fakultät in Fall 1 mit Hilfe einer for- Schleife berechnet werden. Wenn beispielsweise die Fakultät von fünf berechnet werden soll, ist diese initialisiert mit $i = 1$ und läuft bis $i \leq 5$, was die Abbruchbedingung darstellt. Der Befehl i++ erhöht die

Laufvariable i nach jedem Schleifendurchgang um eins. In jeder Iteration der for- Schleife wird nun der Wert der int- Variable $fakultaet$, die bereits vor dem Beginn der for- Schleife mit eins initialisiert wurde, mit i multipliziert und das Ergebnis in der Variable $fakultaet$ gespeichert. Konkret bedeutet das: Im ersten Durchgang wird $fakultaet$ = 1 mit i = 1 multipliziert, was wiederum $fakultaet$ = 1 ergibt. Im zweiten Durchgang wird $fakultaet$ = 1 mit i = 2 multipliziert, was wiederum $fakultaet$ = 2 ergibt. Im dritten Durchgang wird $fakultaet$ = 2 mit i = 3 multipliziert, was wiederum $fakultaet$ = 6 ergibt usw. Der Quellcode in der Programmiersprache C ist in der folgenden Abbildung dargestellt.

```
#include <stdio.h>

int main ()
{
    int f, fakultaet = 1;

    printf("\nZur Berechnung der Fakultaet bitte eine natuerliche Zahl eingeben: ");
    scanf("%d", &f);

    for (int i = 1; i <= f; i++) {
        fakultaet = fakultaet * i;
    }

    printf("%d! = %d\n", f, fakultaet);
    return 1;
}
```

Abbildung 21: Quellcode in C zur nicht rekursiven Berechnung der Fakultät

In Fall 2 erfolgt die Berechnung der Fakultät einer natürlichen Zahl in einem C- Programm durch den rekursiven Aufruf einer Funktion, in der die Fakultät berechnet wird. Rekursiver Aufruf bedeutet, dass die Funktion in der Funktion, ggf. sogar wiederholt, aufgerufen wird. Am Beispiel: Es soll die Fakultät von fünf berechnet werden. Hierfür wird in dem Programm *Fakultaet02* die int-Funktion $Fakultaet$ aufgerufen und die int- Variable f = 5 übergeben. Diese wird mit dem Ergebnis des Aufrufes der Funktion $Fakultaet$ mit f = 4 multipliziert, die wiederum mit dem Ergebnis des Aufrufes der Funktion $Fakultaet$ mit f = 3 multipliziert wird usw. In der folgenden Abbildung ist der Quellcode in der Programmiersprache C dargestellt.

```c
#include <stdio.h>

int Fakultaet(int f);

int main ()
{
    int f, fakultaet;

    printf("\nZur Berechnung der Fakultaet bitte eine natuerliche Zahl eingeben: ");
    scanf("%d", &f);

    fakultaet = Fakultaet(f);

    printf("%d! = %d\n", f, fakultaet);

    return 1;
}

int Fakultaet(int f)
{
    if (f == 1) return 1;
    else if (f > 1) return f*Fakultaet(f-1);
}
```

Abbildung 22: Quellcode in C zur rekursiven Berechnung der Fakultät

Im iterativen ersten Fall hat die Anzahl der Schleifendurchläufe Einfluss auf die Rechenzeit, aber nicht auf den benötigten Speicherplatz. Unabhängig von der natürlichen Zahl, die im Programmverlauf von dem Benutzer abgefragt wird, wird immer Speicherplatz für die zwei int- Variablen f und $fakultaet$ reserviert. Das unterscheidet diese Methode von der rekursiven Berechnung. Hier wird Speicher für die lokalen Variablen jedes Methodenaufrufs reserviert, und zwar so lange, bis die Methode endet. Am Beispiel von der Fakultät von fünf bedeutet das folgendes: Im Fall 1 wird während des gesamten Programmablaufs Speicher für zwei Variablen reserviert. Im Fall 2 wird Speicher für bis zu sieben Variablen gleichzeitig reserviert.

Darüber hinaus werden sowohl in der iterativen, als auch in der rekursiven Implementierung die folgenden Problemfälle nicht berücksichtigt.

1. Der Benutzer kann eine negative Zahl oder eine Kommazahl eingeben.
2. Aufgrund der Größe der zu berechnenden Zahl kann es zu einem Overflow des Speichers kommen.

Die folgende Abbildung zeigt, dass die Berechnung der Fakultät nur in dem ersten Fall korrekt erfolgt, weil der Benutzer, wie aufgefordert, mit der Zahl 5 eine natürliche Zahl eingibt. Im zweiten und im dritten Fall wurde eine Kommazahl bzw. eine negative Zahl eingegeben. Ein Ergebnis wird dem Benutzer dennoch ausgegeben, welches leider falsch ist. Im vierten und letzten Fall kommt es zu einem Speicher- Overflow, weil die zu berechnende Zahl zu groß ist und der Speicher hierfür nicht ausreicht.

```
Zur Berechnung der Fakultaet bitte eine natuerliche Zahl eingeben: 5
5! = 120

Zur Berechnung der Fakultaet bitte eine natuerliche Zahl eingeben: 5.5
5! = 120

Zur Berechnung der Fakultaet bitte eine natuerliche Zahl eingeben: -5
-5! = 1

Zur Berechnung der Fakultaet bitte eine natuerliche Zahl eingeben: 100
100! = 0
```

Abbildung 23: Problemfälle bei der Berechnung der Fakultät einer natürlichen Zahl

Für die Fakultät wurde gemäß der in den vorherigen Kapiteln beschriebenen Vorgehensweise ebenfalls eine Spezifikation in der TLA$^+$ Toolbox erstellt und das erstellte Modell in TLA$^+$ Schreibweise bzw. ASCII übertragen. Wenn die Menge der natürlichen Zahlen als $\mathbb{N}_o$ definiert ist, kann der Anfangszustand dabei entweder gleich Null oder größer als Null, d.h. eine beliebige natürliche Zahl, sein.

1. $f = 0$
2. $f > 0$

Für die Variable *f* in der Spezifikation bedeutet das, dass der Anfangszustand wie folgt zu definieren ist.

$$(f = 0) \vee (f > 0)$$

Für die Zustandsübergänge gelten die zwei folgenden Annahmen. Wenn *f* im ersten Zustand gleich Null ist, ist *f*‘ im nächsten Zustand gleich Eins.

$$(f = 0) \wedge (f' = 1)$$

Wenn *f* im ersten Zustand dagegen größer als Null ist, ist *f*‘ im nächsten Zustand die Fakultät der natürlichen Zahl *f*.

$$(f > 0) \wedge (f' = f!)$$

Daraus ergibt sich das folgende Modell:

$$((f = 0) \wedge (f' = 1)) \vee ((f > 0) \wedge (f' = f!))$$

Stellt man dieses nun in der Schreibweise von TLA$^+$ als Aufzählung dar, sehen die zwei Nächster-Zustand-Aktionen folgendermaßen aus.

$$\vee\ (f = 0) \wedge (f' = 1)$$

$$\vee\ (f > 0) \wedge (f' = f!)$$

Oder, noch weiter heruntergebrochen auf eine Schreibweise ohne Klammern, so:

$$\begin{aligned} \vee \wedge\ & f = 0 \\ \wedge\ & f' = 1 \\ \vee \wedge\ & f > 0 \\ \wedge\ & f' = f! \end{aligned}$$

In der TLA$^+$ Toolbox wird nun eine neue Spezifikation angelegt und das vollständige Modul mit $init$ als Anfangszustand und $next$ als Nächster-Zustand-Aktion in den Editor übertragen.

```
VARIABLE f

init == (f = 0) \/ (f > 0)
next == \/ /\ f = 0
           /\ f' = 1
        \/ /\ f > 0
           /\ f' = f!
```

Wenn das Modul gespeichert wird, wird es automatisch geparst und eventuelle Fehlermeldungen ausgegeben. Ein Klick auf die Meldung markiert die Stelle im Text, an der der Fehler auftritt. Im Falle des selbsterstellten Fakultät- Beispiels gibt es tatsächlich eine Fehlermeldung.

```
1  ---------------------------- MODULE Fakultaet ----------------------------
2
3  VARIABLE f
4
5  init == (f = 0) \/ (f > 0)
6  next == \/ /\ f = 0
7             /\ f' = 1
8          \/ /\ f > 0
9             /\ f' = f!
10
11 ==========================================================================
12 \* Modification History
13 \* Last modified Tue Oct 03 13:23:56 CEST 2017 by Jenni
14 \* Created Tue Oct 03 12:39:26 CEST 2017 by Jenni
```

Parsing Errors

TLA+ Parser Error

Parse Error
Encountered "==========

Residual stack trace follows:
Bang Extension starting at line 9, column 21.
Primitive expression starting at line 9, column 20.
ExtendableExpr starting at line 9, column 20.
ExtendableExpr starting at line 9, column 15.
Expression starting at line 9, column 15.

Abbildung 24: TLA⁺ Fehlermeldung Nr. 1 bei dem Fakultätsbeispiel

Die Vermutung liegt nahe, dass TLA$^+$ mit der in Zeile 9 verwendeten Fakultät nichts anfangen kann und hier eine andere Schreibweise erforderlich. Ein temporäres Abändern von $f!$ auf einen beliebigen Wert, z.B. 5, um die Vermutung zu bestätigen, führte allerdings zu einer bzw. zwei weiteren Fehlermeldungen beim Parsen, nämlich in Zeile 5 und Zeile 8. Der Grund für die Meldung ist in beiden Zeilen das Größer als- Zeichen, das gemäß TLA$^+$ nicht deklariert oder definiert ist.

```
1  ---------------------------- MODULE Fakultaet ----------------------------
2
3  VARIABLE f
4
5  init == (f = 0) \/ (f > 0)
6  next == \/ /\ f = 0
7             /\ f' = 1
8          \/ /\ f > 0
9             /\ f' = 5
10
11 ==========================================================================
12 \* Modification History
13 \* Last modified Tue Oct 03 13:20:17 CEST 2017 by Jenni
14 \* Created Tue Oct 03 12:39:26 CEST 2017 by Jenni
```

Parsing Errors

TLA+ Parser Error
line 5, col 21 to line 5, col 25 of module Fakultaet
Couldn't resolve infix operator symbol `>'.

TLA+ Parser Error
line 5, col 23 to line 5, col 23 of module Fakultaet
Could not find declaration or definition of symbol '>'.

TLA+ Parser Error
line 8, col 15 to line 8, col 19 of module Fakultaet
Couldn't resolve infix operator symbol `>'.

Abbildung 25: TLA⁺ Fehlermeldung Nr. 2 bei dem Fakultätsbeispiel

Die Fehlermeldung Nr. 2 informierte den Anwender darüber, dass das Größer als- Zeichen definiert sei. Das Einbinden des Integers- Standardmoduls gleich zu Beginn des Moduls, d.h. noch vor der Deklaration der Variable f, eliminiert die Fehlermeldung.

```
EXTENDS Integers
```

Die Fehlermeldung Nr. 1 bezüglich des Fakultätsoperators ! besteht weiterhin. Um diesen zu beheben, wird im folgenden Kapitel die Rekursion in TLA$^+$ am Beispiel der Fakultät einer natürlichen Zahl erklärt.

8 Rekursion in TLA$^+$ am Beispiel der Fakultät

Die Versuche das Fakultätsbeispiel auf nicht rekursive Art in TLA$^+$ umzusetzen, sind bisher sowohl bei der Definition der Anfangsaussage $init$ und der Nächster-Zustand-Aktion $next$ unter Einsatz des Fakultätsoperators !, als auch mit Hilfe einer for- Schleife in der TLA$^+$ Toolbox gescheitert, da diese nicht definiert sind.

8.1 Definition rekursiver Operatoren

Bei herkömmlichen Definitionen muss jedes Symbol deklariert oder definiert sein, bevor es verwendet werden kann. Rekursion erlaubt jedoch, dass der Operator oder die Funktion, die definiert wird, innerhalb des jeweiligen Ausdrucks auftaucht, in dem sie definiert wird. Der größte Unterschied zwischen einem Operator und einer Funktion besteht drin, dass eine Funktion an sich ein vollständiger Ausdruck ist, während ein Operator meist auf zwei Argumente, auch Operanden genannt, angewendet wird. [5]

Zur Definition rekursiver Operatoren kann die RECURSIVE- Deklaration verwendet werden. Am Beispiel der Fakultät sieht so eine Definition folgendermaßen aus.

TLA Module

```
1  ---------------------------- MODULE Fakultaet_rek ----------------------------
2
3  EXTENDS Integers
4
5  RECURSIVE FactorialOp(_)
6
7  FactorialOp(n) == IF n = 0 THEN 1 ELSE n * FactorialOp(n-1)
8
9
10 ==============================================================================
11
```

Abbildung 26: Definition rekursiver Operatoren am Beispiel der Fakultät einer natürlichen Zahl

Darüber hinaus ist es möglich gegenseitig rekursive Funktionen zu definieren. Ein Beispiel dazu findet sich im Folgenden.

RECURSIVE F(_,_), G(_)

$H \triangleq$

$F(x, y) \triangleq$

$G(z) \triangleq$

Dadurch dass F und G als rekursiv deklariert sind, können diese auf der rechten Seite von allen drei Definitionen auftauchen. H hingegen kann nur auf der rechten Seite von F und G stehen, da es bereits vorher definiert wurde.

Solange keine Funktion definiert werden soll, ist eine induktive Definition mit rekursiven Operatoren für gewöhnlich komfortabler, als die Definition einer rekursiven Funktion. Sollte jedoch das TLA$^+$ proof system, kurz TLAPS, verwendet werden, muss die Definition als rekursive Funktion erfolgen, da dieses rekursive Operatoren nicht verarbeiten kann. Um eine Definition mit rekursiven Operatoren in eine gewöhnliche Definition umzuwandeln, kann das LET/ IN- Konstrukt zusammen mit rekursiven Funktionen verwendet werden. [10]

8.2 Das LET / IN- Konstrukt

Eine Spezifikation kann vom Grundsatz her als eine einzige, lange Formel niedergeschrieben werden, ohne dabei jedwede Form von Definitionen zu verwenden. Jedoch ist eine Spezifikation, die aus einer einzigen, langen Formel besteht, schwer lesbar und nicht leicht zu verstehen. Um die Formel leichter lesbar und besser verständlich zu machen, kann diese mit Hilfe von Definitionen in ihre Einzelteile zerlegt werden. Es folgt ein einfaches Beispiel, um das Vorgehen zu verdeutlichen: Die Definition eines Operators H wird zerlegt, indem zwei weitere Operatoren F und G eingeführt werden. Wenn F und G ausschließlich in der Definition von H verwendet werden, ist die vollständige Spezifikation leichter zu verstehen, indem F und G lokal in der Definition von H definiert werden. Für diese lokalen Definitionen wird in TLA$^+$ das LET / IN- Konstrukt verwendet.

Das LET / IN- Konstrukt ist ein Ausdruck, der als Unterausdruck in jedem beliebigen Ausdruck vorkommen kann. Hauptsächlich wird dieser auf der rechten Seite einer Definition verwendet und hat die Form

$$H(a,b) \triangleq \exists\, i \in Nat : LET \ldots IN \ldots$$

Die LET- Klausel ist eine Liste von Definitionen, in der beispielsweise die Operatoren F und G definiert werden. Es können an dieser Stelle aber auch rekursiv definierte Funktionen oder Operatoren auftauchen. Die IN- Klausel ist ein Ausdruck, in dem die definierten Symbole, in dem folgenden Beispiel a, b und i, verwendet werden können. Eine vollständige Spezifikation unter Verwendung des LET / IN- Konstrukt könnte dementsprechend wie folgt aussehen.

$$\begin{aligned} H(a,b) \triangleq \exists\, i \in Nat :& \\ LET\ F(u) &\triangleq u + i \\ G &\triangleq a * F(2) \\ IN\ G &> F(b) \end{aligned}$$

Dieser Ausdruck definiert $H(x,y)$ gleich $x * (2 + i) > y + i$. [10]

8.3 Definition rekursiver Funktionen

TLA$^+$ stellt eine spezielle Syntax für die Definition von Funktionen bereit. Die zwei folgenden Definitionen können äquivalent verwendet werden. [10]

$$Successor[i \in Nat] \triangleq i + 1$$

$$Successor \triangleq [i \in Nat \mapsto i + 1]$$

Bezogen auf die untere der beiden Schreibweisen, ist für die rekursive Berechnung der Fakultät die folgende Definition gemäß TLA$^+$ Notation naheliegend.

$$fact \triangleq [n \in Nat \longmapsto if\ n = 0\ then\ 1\ else\ n * fact[n-1]]$$

Diese Definition ist allerdings unzulässig, weil *fact* auf der rechten Seite des Entspricht-Zeichens steht, hier jedoch noch nicht definiert ist. Die Definition erfolgt erst nach dem

Ausdruck. Im Allgemeinen erlaubt TLA$^+$ jedoch die Definition einer rekursiven Funktion f mit dem Wertebereich S in der folgenden Form.

$$f\ [x\ \in\ S] \triangleq e$$

Übertragen auf das Beispiel der Fakultät sieht die rekursive Definition in TLA$^+$ folgendermaßen aus. [5]

$$fact[n \in Nat] \triangleq if\ n\ =\ 0\ then\ 1\ else\ n * fact[n-1]$$

Die Darstellung in der erforderlichen ASCII- Schreibweise in der TLA$^+$ Toolbox erfolgt entsprechend in Abbildung 27.

TLA Module

```
1 ---------------------------- MODULE Fakultaet02 ----------------------------
2
3 EXTENDS Integers
4
5 fact[n \in {Nat}] == IF n = 0 THEN 1 ELSE n * fact[n-1]
6
7 ==============================================================================
8
9
```

Abbildung 27: Definition rekursiver Funktionen am Beispiel der Fakultät einer natürlichen Zahl

8.4 Der CHOOSE- Operator

Rekursiv definierte Funktionen sollten lediglich dann verwendet werden, wenn es eine eindeutige Funktion gibt, für die die CHOOSE- Formel erfüllt ist, wie es bei der Fakultät der Fall ist. Die obige Definition für $fact$ ist nämlich äquivalent zu der folgenden Definition mit Hilfe des CHOOSE- Operators.

$$fact \triangleq CHOOSE\ f: f = [n \in Nat \longmapsto IF\ n = 0\ THEN\ 1\ ELSE\ n * f[n-1]]$$

Der CHOOSE- Operator in TLA$^+$ ist eng verwandt mit dem Existenzquantor $\exists$. Der Existenzquantor ist neben dem Allquantor $\forall$ einer der gebräuchlichsten Operatoren der Prädikatenlogik. Der Allquantor $\forall$ wird gelesen als „für alle“. Die Formel $\forall\ n \in \mathbb{N}: P(n)$ sagt

entsprechend folgendes aus: Für alle n aus der Menge der natürlichen Zahlen $\mathbb{N}$ ist die Aussage $P(n)$ wahr. In diesem Fall ist auch die o.g. Formel wahr.

Der Existenzquantor $\exists$ wird gelesen als „es existiert“. Die Formel $\exists\, n \in \mathbb{N}: P(n)$ behauptet, dass mindestens ein n in der Menge der natürlichen Zahlen $\mathbb{N}$ existiert, für das die Aussage $P(n)$ wahr ist. Wenn diese Behauptung wahr ist, liefert die Formel $CHOOSE\ n \in \mathbb{N}: P(n)$ in TLA$^+$ solch einen Wert. Die häufigste Verwendung findet der CHOOSE- Operator also, um einen Wert auszuwählen, der spezifiziert ist durch die Eigenschaft, die er erfüllt. Gibt es kein Element n, wie beispielsweise für $CHOOSE\ n \in Int: n^2 = 2$, kann der Wert beliebig sein. In diesem Fall liefert der *TLC Model Checker*, kurz *TLC*, eine Fehlermeldung zurück. [10]

9 Implementierung des Fakultätsbeispiels

Gemäß Lamport [5] bietet TLA$^+$ zwei verschiedene Möglichkeiten die Fakultät einer natürlichen Zahl rekursiv zu spezifizieren:

1. Definition als rekursive Funktion
2. Definition als rekursiver Operator

Da aus einer TLA$^+$ Spezifikation jedoch nicht automatisch Code generiert werden kann, wird im Folgenden eine dritte Variante des Quellcodes der Fakultät einer natürliche Zahl in der Programmiersprache C umgesetzt, basierend auf den Erkenntnissen der zuvor erstellten TLA$^+$ Spezifikation mit einer rekursiven Funktion.

An der eigentlichen Berechnung der Fakultät einer natürlichen Zahl durch den rekursiven Aufruf einer Funktion hat sich im Grunde nichts geändert. Innerhalb der Funktion $Fakultaet$ wurde lediglich die if- Bedingung dahingehend angepasst, dass auch der Fall f = 0 berücksichtigt wird. Die wesentliche Änderung besteht darin, dass vor dem Funktionsaufruf, in dem die Berechnung stattfindet, die vom Benutzer eingegebene Zahl überprüft wird. Diese wird zunächst über die scanf- Funktion als String eingelesen. Die Zeichenkette wird daraufhin in einer for- Schleife auf ein Minuszeichen, ein Punkt oder Komma geprüft. Ist eines oder mehrere der besagten Zeichen enthalten, wird die int- Variable $natuerlich$ gleich Null gesetzt, die zuvor mit Eins initialisiert war. Nachdem der Durchlauf der for- Schleife beendet wurde, wird der Wert der Variablen überprüft. Ist $natuerlich$ = 0, handelt es sich bei der Zahl nicht um eine

natürliche Zahl. Es wird eine Fehlermeldung ausgegeben und das Programm beendet. Ist hingegen $natuerlich$ = 1, wird die Zeichenkette $zahl$ mit Hilfe des atoi- Befehls in eine Integer- Zahl umgewandelt und in die int- Variable f geschrieben. In Anschluss daran erfolgt noch eine Prüfung, ob f größer als 12 ist. Ist dies der Fall, wird ebenfalls eine Fehlermeldung ausgegeben und das Programm beendet, da die eingegebene Zahl zu groß für die Berechnung der Fakultät ist. Erst nach diesen zwei Prüfungen erfolgt der Aufruf der Funktion $Fakultät$. Der Quellcode in der Programmiersprache C ist in der folgenden Abbildung dargestellt.

```c
int main ()
{
    int i, f, fakultaet, natuerlich = 1;
    char zahl[10];
    printf("\nZur Berechnung der Fakultaet bitte eine natuerliche Zahl bis 12 eingeben: ");
    scanf("%s", zahl);

    for (i = 0; i < 10; i++) {
        if (zahl[i] == '-' || zahl[i] == '.' || zahl[i] == ',') natuerlich = 0;
    }

    if (natuerlich == 0) { // Es handelt sich nicht um eine natuerliche Zahl
        printf("\nEs handelt sich nicht um eine natuerliche Zahl, das Programm wird beendet.\n");
        return 1;
    }
    else {
        f = atoi(zahl);
    }

    if (f > 12) { // Die Zahl ist zu groß, bei der Berechnung der Fakultaet kommt es zu einem Overflow
        printf("\nDie eingegebene Zahl ist zu gross, das Programm wird beendet.\n");
        return 1;
    }

    fakultaet = Fakultaet(f);
    printf("%d! = %d\n", f, fakultaet);
    return 1;
}
int Fakultaet(int f)
{
    if (f == 0) return 1;
    else if (f > 0) return f*Fakultaet(f-1);
}
```

Abbildung 28: Quellcode in C zur rekursiven Berechnung der Fakultät gemäß der TLA⁺ Spezifikation

Die Ausführung der aus dem Quellcode kompilierten Anwendung *Fakultaet03.exe* mit den folgenden Zahlen führte dieses Mal zu dem gewünschten Ergebnis, wie die Abbildung 29 zeigt.

1. f = 5
2. f = 5.5
3. f = -5

4. f = 100

Die Berechnung der Fakultät erfolgt in dem ersten Fall korrekt, weil der Benutzer eine natürliche Zahl eingegeben hat. Im zweiten und im dritten Fall wurde hingegen eine Kommazahl bzw. eine negative Zahl eingegeben. Im vierten und letzten Fall war die eingegebene Zahl mit 100 deutlich größer, als die geforderte natürliche Zahl bis 12. In allen drei Fällen erfolgt keine Berechnung der Fakultät. Anstatt dessen wird das Programm beendet.

```
Zur Berechnung der Fakultaet bitte eine natuerliche Zahl bis 12 eingeben: 5
5! = 120

Zur Berechnung der Fakultaet bitte eine natuerliche Zahl bis 12 eingeben: 5.5
Es handelt sich nicht um eine natuerliche Zahl, das Programm wird beendet.

Zur Berechnung der Fakultaet bitte eine natuerliche Zahl bis 12 eingeben: -5
Es handelt sich nicht um eine natuerliche Zahl, das Programm wird beendet.

Zur Berechnung der Fakultaet bitte eine natuerliche Zahl bis 12 eingeben: 100
Die eingegebene Zahl ist zu gross, das Programm wird beendet.
```

Abbildung 29: Behobene Problemfälle bei der Berechnung der Fakultät einer natürlichen Zahl

Die erforderliche Überprüfung, ob es sich bei der eingegebenen Zahl um eine natürliche Zahl handelt, konnte direkt aus der TLA$^+$ Spezifikation abgelesen werden, denn hier wird mit `fact[n \in {Nat}]` gefordert, dass diese aus der Menge der natürlichen Zahl stammt. Was leider nicht mit TLA$^+$ herausgefunden werden konnte, war der Wert, bei dem es rechnerabhängig zu einer Überschreitung der Speicherkapazität, sprich zu dem besagten Overflow, kommt. Dieser wurde in diesem Beispiel durch Ausprobieren ermittelt. Ob TLA$^+$ eine entsprechende Funktion bietet, ist derzeit leider nicht bekannt.

10 Fazit

10.1 Vor- und Nachteile von TLA+

Nach Lamport [3] hat die Spezifikationssprache TLA$^+$ gegenüber Pseudocodes zwei essentielle Vorteile. Eine TLA$^+$ Spezifikation ist erstens präzise und kann mit den entsprechenden Tools, wie beispielsweise dem in der TLA$^+$ Toolbox zur Verfügung gestellten *TLC model checker* überprüft werden. Des Weiteren können mit TLA$^+$ formelle, maschinell überprüfbare Beweise für die Richtigkeit einer Spezifikation erstellt werden. Allerdings bringt TLA$^+$ auch Nachteile mit sich. Zwei geringfüge Nachteile sind, dass sowohl der Kontrollzustand, als auch die Variablen, die über den kompletten Programmablauf nicht verändert werden, explizit angegeben werden müssen. Beides erfolgt in Programmiersprachen implizit. Der größte und bedeutendste Nachteil ist jedoch, dass TLA$^+$ auf Mathematik basiert, was die meisten Entwickler und Programmierer von vornherein abschreckt.

10.2 Verwendung in der Industrie am Beispiel der Weltraummission Rosetta

Laut Lamport [3] wird die Spezifikationssprache TLA$^+$ in der Industrie verwendet, um komplexe Systementwürfe zu überprüfen. Als Beispiel sei hier der Einsatz von TLA$^+$ im Rahmen der Weltraummission Rosetta der Europäischen Weltraumorganisation, kurz ESA, zu nennen. Die gleichnamige Weltraumsonde startete am 02. März 2004 ihren Weg ins Weltall, um den Kometen Tschurjumow- Gerassimenko, kurz Komet 67P, zu erforschen. Innerhalb des Rosetta-Systems befindet sich ein Echtzeit- Betriebssystem mit dem Namen Virtuoso, das mehrere Instrumente kontrolliert. Dieses Echtzeit- Betriebssystem wurde mit Hilfe von TLA$^+$ entworfen. Eric Verhulst, der Leiter der Virtuoso- Entwicklung, sagte über TLA$^+$: „The TLA$^+$ abstraction helped a lot in coming to a much cleaner architecture. We witnessed first hand the brain washing done by years of C programming. One of the results was that the code size is about 10x less than the previous version.“ Um die Größe des Codes um den Faktor Zehn zu reduzieren, reicht es in so einem komplexen Fall nicht aus den Code selbst zu optimieren oder besser zu programmieren. Zu solch einer Reduktion gelangt man nur durch eine schlankere Architektur, die ebenfalls durch eine high-level Spezifikation vor der Code- Ebene beschrieben werden

kann. Die Architektur wurde entsprechend mit TLA$^+$ spezifiziert und mit Hilfe der TLA$^+$ Tools von Fehlern befreit.

10.3 Verwendung in der Industrie am Beispiel von Amazon Web Services

Stand Ende 2013 ist Amazon der führende Anbieter im Bereich Cloud- Dienstleistungen. Amazon Web Services erstellt und pflegt die komplette Cloud- Infrastruktur, d.h. sie erstellen große und komplexe Systeme im Auftrag von Amazon. Diese müssen richtig sein, da sich die Kunden darauf verlassen können müssen. Auf Lamports Vorschlag hin, veröffentlichten Chris Newcombe u.a. einen Artikel über ihre Arbeit mit TLA$^+$ bei Amazon Web Service: „Amazon has used TLA$^+$ on 10 large complex systems. In each, TLA$^+$ has added significant value, either finding subtle bugs we would not have found by other means, or giving us enough confidence to make aggressive optimizations without sacrificing correctness.“ Laut Lamport [4] stieg die Anzahl der Großprojekte bei Amazon Web Services, in denen TLA$^+$ verwendet wurde, von 10 in Ende 2013 auf 14 nach kurzer Zeit in 2014. Leider gibt es keine aktuellen Zahlen, aber dieser rasante Anstieg lässt vermuten, dass die Zahl inzwischen weiter gestiegen ist, zumal die Geschäftsleitung und die technische Leitung den Einsatz von TLA$^+$ unterstützen.

10.4 TLA$^+$ anstatt Pseudocode: Ja oder nein?

Leslie Lamport liefert mit seiner Spezifikationssprache TLA$^+$ keine Lösung für bekannte Probleme, sondern ein mächtiges Werkzeug zum Erkennen von unbekannten Problemen. TLA$^+$ wurde entwickelt, um nebenläufige und verteilte Systeme zu modellieren. Es kann dabei helfen Fehler im Entwurf zu finden, und zwar noch bevor die erste Zeile Code geschrieben wird, die durch das Testen alleine schwer bis unmöglich zu finden sind.

Wer solche Systeme entwirft und darum bemüht ist, dass diese sauber und richtig arbeiten, sollte laut Lamport [4] TLA$^+$ verwenden, um die Entwürfe präzise zu spezifizieren und zu überprüfen. Wenn man dagegen nur wenig programmiert, kann die Verwendung von TLA$^+$ zwar auch hilfreich sein, mitunter hat es aber wenig bis keinen Nutzen. Für die meisten Programmierer und Entwickler bietet TLA$^+$ eine neue Art des Denkens darüber, was sie tun.

Diese neue Art des Denkens kann sie zu besseren Programmierern und Entwicklern machen, selbst dann, wenn TLA$^+$ und seine Tools keinen direkten Nutzen haben.

Die Verwendung der Spezifikationssprache TLA$^+$ anstatt Pseudocode für komplexe Sachverhalte ist also durchaus empfehlenswert und vorteilhaft. Nach Lamport [3] verhindern Programmiersprachen, wie beispielsweise C, dass auf einem angemessenen Abstraktionsniveau gedacht wird, indem in Begriffen und Notationen der jeweiligen Sprache gedacht wird. Die Verwendung von TLA$^+$ hilft Schritt für Schritt dabei, sich von diesen einschränkenden Gedankenmustern zu lösen und komplexe Probleme auf ein starkes Abstraktionsniveau herunter zu brechen.

Digitale Systeme können nur dann richtig entwickelt werden, wenn sie richtig verstanden werden. Abstraktion ist der wahrscheinlich wichtigste Aspekt beim Programmieren und Entwickeln, weil komplexe Systeme erst durch den Prozess der Vereinfachung durch das Entfernen irrelevanter Details vollständig durchblickt und verstanden werden können. Brannon Batson, ein ehemaliger Entwickler bei Intel, sagte dazu: „The hard part of learning to write TLA$^+$ specs is learning to think abstractly about the system. With experience, engineers learn how to do it. Being able to think abstractly improves their design process.“

Aufgrund der vielen Nachteile, die Pseudocodes mit sich bringen – keine einheitliche Norm, unpräzise und damit anfällig für Fehler, sowie mangelnde Überprüfbarkeit –, sollte möglichst eine andere Art zur Beschreibung von Algorithmen gewählt werden. Die Mathematik zur Beschreibung von Algorithmen eignet sich nur bedingt. Sie ist mit ihren Notationen zwar eindeutig, lässt aber bei der späteren Implementierung zu viel Spielraum. Die Fakultät einer natürlichen Zahl n, um ein Beispiel zu nennen, sagt zwar aus, was getan werden muss, nämlich das Produkt der Zahlen 1 bis n zu berechnen. Sie sagt aber nichts darüber aus, auf welchem Wege das getan werden soll, wie auch die ersten zwei Codebeispiele in der Programmiersprache C gezeigt haben. In dem einen Fall wurde die Berechnung der Fakultät iterativ implementiert, in dem anderen Fall rekursiv. Erst ein Algorithmus sagt etwas darüber aus, wie etwas umgesetzt werden soll. Aus den bereits genannten Gründen macht es Sinn, diesen mit TLA$^+$ zu beschreiben. Leider gibt es noch keine Alternativen zu TLA$^+$ und somit keine Vergleichsmöglichkeiten, ob die Spezifikationssprache gut oder schlecht ist, um Algorithmen zu beschreiben. Ausgehend von den Status quo ist TLA$^+$ aber aktuell die beste Möglichkeit, um Algorithmen einheitlich, präzise und überprüfbar zu beschreiben.

11 Literatur- und Quellenverzeichnis

- [1] : Heinrich Müller und Frank Weichert: Vorkurs Informatik - Der Einstieg ins Informatikstudium, Springer Vieweg, Wiesbaden, 5. Auflage, 2017

- [2] : Hartmut Ernst, Jochen Schmidt und Gerd Beneken: Grundkurs Informatik, Springer Vieweg, Wiesbaden, 6. Auflage, 2016

- [3] : Leslie Lamport: The PlusCal Algorithm Language, Vortrag vom 22. September 2016 beim Heidelberg Laureate Forum. Link zum Vortrag: http://www.heidelberg-laureate-forum.org/blog/video/lecture-thursday-september-22-2016-leslie-lamport/

- [4] : Leslie Lamport: The TLA+ Video Course, Videokurs von Leslie Lamport, Stand 11. Dezember 2017. Link zum Kurs: http://lamport.azurewebsites.net/video/videos.html

- [5] : Leslie Lamport: Specifying Systems - The TLA+ Language and Tools for Hardware and Software Engineers, Addison-Wesley, Boston, 1. Auflage, 2002

- [6] : A.M. Turing Award: https://amturing.acm.org/

- [7] : Martin Dietzfelbinger, Kurt Mehlhorn, Peter Sanders: Algorithmen und Datenstrukturen. Die Grundwerkzeuge, Springer Vieweg, Berlin, 2014

- [8] : Leslie Lamport: The TLA Home Page. Link zur Homepage: https://lamport.azurewebsites.net/tla/tla.html

- [9] : Thomas Ottmann und Peter Widmayer: Algorithmen und Datenstrukturen, Springer Vieweg, Berlin, 7. Auflage, 2017

- [10] : Leslie Lamport: The TLA$^+$ Hyperbook, pdf- Ausgabe, Version vom 20. August 2015

- [11] : Günther Bengel, Christian Baun, Marcel Kunze und Karl-Uwe Stucky: Masterkurs Parallele und Verteilte Systeme, Springer Vieweg, Wiesbaden, 2., erweiterte und aktualisierte Auflage, 2015

- [12] : Wikipedia: Dijkstra- Algorithmus in Pseudocode. Link zum Artikel: https://de.wikipedia.org/wiki/Dijkstra-Algorithmus#Algorithmus_in_Pseudocode

12 Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne unerlaubte Hilfe angefertigt, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Ort, Datum

_____

(Unterschrift)

13 Anhang

- Quellcode „Fakultaet01“
- Quellcode „Fakultaet02“
- Quellcode „Fakultaet03“

Fakultaet01.c

```c
#include <stdio.h>

int main ()
{
    int f, fakultaet = 1;

    printf("\nZur Berechnung der Fakultaet bitte eine natuerliche Zahl eingeben: ");
    scanf("%d", &f);

    int i;
    for (i = 1; i <= f; i++) {
        fakultaet = fakultaet * i;
    }

    printf("%d! = %d\n", f, fakultaet);
    return 1;
}
```

Fakultaet02.c

```c
#include <stdio.h>

int Fakultaet(int f);

int main ()
{
    int f, fakultaet;

    printf("\nZur Berechnung der Fakultaet bitte eine natuerliche Zahl eingeben: ");
    scanf("%d", &f);

    fakultaet = Fakultaet(f);

    printf("%d! = %d\n", f, fakultaet);

    return 1;
}

int Fakultaet(int f)
{
    if (f == 1) return 1;
    else if (f > 1) return f*Fakultaet(f-1);
}
```

Fakultaet03.c

```c
#include <stdio.h>

int Fakultaet(int f);

int main ()
{
    int i, f, fakultaet, natuerlich = 1;
    char zahl[10];
    printf("\nZur Berechnung der Fakultaet bitte eine natuerliche Zahl bis 12 eingeben: ");
    scanf("%s", zahl);

    for (i = 0; i < 10; i++) {
        if (zahl[i] == '-' || zahl[i] == '.' || zahl[i] == ',') natuerlich = 0;
    }

    if (natuerlich == 0) {
        printf("\nEs handelt sich nicht um eine natuerliche Zahl, \n");
        printf("das Programm wird beendet.\n");
        return 1;
    }
    else {
        f = atoi(zahl);
    }

    if (f > 12) {
        printf("\nDie eingegebene Zahl ist zu gross, das Programm wird beendet.\n");
        return 1;
    }

    fakultaet = Fakultaet(f);

    printf("%d! = %d\n", f, fakultaet);

    return 1;
}

int Fakultaet(int f)
{
    if (f == 0) return 1;
    else if (f > 0) return f*Fakultaet(f-1);
}
```