FACHHOCHSCHULE **TRIER**
Hochschule für Technik, Wirtschaft und Gestaltung
University of Applied Sciences
Informatik

Formale Spezifikation mit TLA$^+$

formal spezification with TLA$^+$

Alexander Weigl

Fachseminar: *Software Engineering*

Betreuer: Prof. Dr. Georg Rock

Trier, 2012-01-02

Kurzfassung

TLA$^+$ (Temporal Logic for Action) ist eine Sprache zum Beschreiben von Systemen unter Verwendung von der Prädikatenlogik und Linearer Temporalen Logik (*LTL*). *LTL* erweitert die Prädikatenlogik (*propositional logic*) um Aspekte einer diskreten zeitlichen Abfolge von aufeinander folgenden Zuständen. Mit der LTL werden Anfordungen über Liveness und Fairness an Systemen gestellt. Diese Arbeit erklärt die Syntax und Semantik von TLA$^+$ anhand von Beispielen, sowie das Spezifizieren von Fairness-Anforderungen in TLA$^+$-Spezifikationen. Der Abschluss der Arbeit stellt die Vorstellung des Model-Checkers dar.

TLA$^+$ (Temporal Logic for Action) is a language for specifying systems with usage of propositional (PL) and linear temporal logic (LTL). LTL extends the propositional logic by quantors for statements about a linear discrete sequences about states. I explain liveness and fairness requirements in LTL. This paper shows the syntax and semantic of TLA$^+$ with examples and point out the strong and week fairness properties in TLA$^+$. I conclude with an introduction of the modelchecker TLC within the TLA$^+$-Toolbox.

Inhaltsverzeichnis

1 Einführung 1

2 Grundlagen 3
2.1 Prädikatenlogik 3
2.2 Lineare Temporale Logik 4
2.3 Invarianten, Liveness und Fairness 8

3 Die Sprache TLA$^+$ 10
3.1 Semantik und Syntax 11
3.2 Komplexe Datentypen 14
3.3 Liveness und Fairness 17

4 TLC – TLA$^+$-Modelchecker 19
4.1 Grenzen 19
4.2 Erzeugung der Verhalten 21
4.3 Optionen 22

5 Fazit 25

Literatur 27

1

Einführung

Softwaresysteme ersetzen zunehmend immer größere Teile von kritischen Systemen. Dazu gehören Kraftwerk-, Eisenbahn-, Fluttorsteuerung oder Flugzeugsysteme. Bei diesen Systemen ist ein hohes Maß an Zuverlässigkeit, Korrektheit und Robustheit erforderlich. Dabei sollten diese Eigenschaften unter jeder Eingabe garantiert werden. Durch das Testen von Software können wir lediglich bestätigen, dass das System die Spezifikation für eine Eingabe einhält. Zeigen wir dies für jede Eingabe, ist damit die Korrektheit der Software bewiesen.

Die andere Möglichkeit besteht in der formalen Beschreibung des System[1]. Dafür wird meist auf Logiken zurückgegriffen wie Lineare Temporalelogik (LTL) oder Computional Tree Logic (CTL), die wiederum auf der Prädikatienlogik aufbauen (Abschnitt 2.1). In dieser Ausarbeitung stelle ich die *Temporal Logic for Actions* (TLA$^+$) vor. TLA$^+$ ist eine formale Sprache zur Spezifikation von Systemen und nicht nur von Software. Die Spezifikationen können mittels eines Modelcheckers (*TLC*, Kapitel 4) interpretiert werden. Dabei können Eigenschaften, die in Form von LTL-Ausdrücke vorliegen, überprüft werden. TLA$^+$ wurde von Leslie Lamport als Erweiterung zur Prädikatenlogik (Abschnitt 2.1) und Lineare Temporale Logik (Abschnitt 2.2) entwickelt.

Nun stelle ich innerhalb dieses Dokumentes die formale Spezifikation mit TLA$^+$ vor. Im Unterschied zur Spezifikation mit natürlicher Sprache oder UML greifen wir bei der Spezifikation auf eine Sprache mit festgelegter Syntax und Semantik zurück. Als Vorteil erhalten wir eine einheitliche deterministische Interpretation und Beweisbarkeit von Eigenschaften der Spezifikation. Außerdem werden Fehler innerhalb der Spezifikation in einer frühen Phase des Projektes sichtbar und können kostengünstig korrigiert werden.

Im Nachteil steht der Aufwand für die Einarbeitung der formalen Sprache und Werkzeuge sowie das Schreiben der Spezifikation. Die Spezifikation unterliegt einer Komplexität, die zusätzlich separat dokumentiert werden muss. Diese Zeit könnte in das Schreiben von Prototypen oder Testfällen (im Rahmen von *Test-Driven-Development*) gesteckt werden. Im Unterschied zur Spezifikation wären Teile der Software früher umgesetzt und getestet. Wir werden sehen, dass Spezifikationen auf

[1] Im folgenden bezeichne ich die formale Systembeschreibung als Spezifikation, die Informellen als (System-)Anfordungen

einem hohen abstrakten Niveau geschrieben werden und ein Testen des endgültigen Produktes nicht abgenommen wird.

Zusammenfassend haben wir einem erheblichen Aufwand bei der formalen Spezifikation und gewinnen eine Nachweisbarkeit von Eigenschaften. Im Gegenzug könnten wir informell Anforderungen festhalten, die wir mit Softwaretests= versuchen partiell für gewisse Eingaben nachzuweisen.

Diese Ausarbeitung befasst sich mit den Grundlagen von TLA$^+$. Dafür repetiere ich im nächsten Kapitel Grundlagen der Prädikatenlogik und LTL und erläutere *Liveness* und *Fairness* in LTL. Danach stelle ich die Sprachelemente von TLA$^+$ und *Liveness* und *Fairness*-Eigenschaften vor. Ich werde nicht auf die Spezifikation von Echtzeitsystemen mit TLA$^+$ oder komplexere Konstrukte der Sprache eingehen. Der Abschluss bietet die Vorstellung des TLA$^+$ Toolsets.

Der Ziel der Ausarbeitung ist die Vermittlung eines Verständnisses für die Spezfikationssprache und Umsetzung und Anwendung des Modelcheckers.

2

Grundlagen

TLA$^+$ baut auf die Prädikatenlogik und Linearen Temporalen Logik auf, indem es die Syntax und Semantik in seinen Ausdrücken zulässt. In diesem Kapitel stelle ich die Prädikatenlogik und LTL kurz vor.

2.1 Prädikatenlogik

Die Basis der Prädikatenlogik bildet die Aussagenlogik. Diese Logik umfasst einfache Aussagen, die entweder wahr w oder falsch f sind. Sie werden wie folgt gebildet und ausgewertet:

Definition 1 (*Aussagenlogik*)

Wir definieren die Aussagenlogik als einen Ausdruck über eine Menge von Variablen und den Operatoren $\wedge, \vee, \neg$. Hinzu kommen noch komplexere Operatoren: Impliziert $\Rightarrow$ und Äquivalenz $\Leftrightarrow$.

Syntax

Induktionsanfang *Jede Variable x ist eine aussagenlogische Formel ϕ.*

Induktionsschritt *Wenn wir ϕ, ψ syntaktisch korrekte Formeln nach Induktionsvoraussetzung sind, dann sind folgende Formeln auch syntaktisch korrekt:*

$$\neg\phi \tag{2.1}$$

$$\phi \wedge \psi \tag{2.2}$$

$$\phi \vee \psi \tag{2.3}$$

Semantik *Wir können eine aussagenlogische Formel anhand folgender Wertetabelle für eine Variablenbelegung festlegen:*

ϕ	ϕ	$\neg\phi$	$\phi \wedge \psi$	$\phi \vee \psi$	$\phi \Rightarrow \psi$	$\phi \Leftrightarrow \psi$
f	f	w	f	f	w	w
f	w	f	f	w	w	f
w	f	w	f	w	f	f
w	w	f	w	w	w	w

Dabei gilt folgende Bindungsreihenfolge: $\neg$, $\wedge$, $\vee$, $\Rightarrow$.
Impliziert und Äquivalenz können wie folgt abgebildet werden:

$$\phi \Rightarrow \psi \equiv \neg\phi \vee \psi \tag{2.4}$$
$$\phi \Leftrightarrow \psi \equiv \phi \Rightarrow \psi \wedge \psi \Rightarrow \phi \tag{2.5}$$

Die Operatoren $\wedge, \vee, \neg$ *bilden eine Basis.*

Wir können damit beliebige Ausdrücke der Aussagenlogik unter einer bestimmen Instanz (Variablenbelegung) auswerten. Wir erweitern die Aussagenlogik zur Prädikatenlogik (1. Stufe) indem wir Prädikate und Quantoren einführen.

Definition 2 (*Prädikat*)

Ein Prädikat P *ist eine* n*-stellige Abbildung von* $P\colon A^n \to \{w, f\}$ *aus einer beliebigen Menge zu einem Wahrheitswert.*

Ein Prädikat kann anstelle einer Variablen stehen. Dabei können wir Relationen wie $<, >, =$ als 2-stellige Prädikaten auffassen. Außerdem können wir Ausdrücke quantifizieren und damit Aussagen über Mengen formulieren.

Definition 3 (*Quantoren*)

Mit dem All-Quantor drücken wir aus, dass eine Aussage $P(x)$ *für jedes Element gilt.*

$$\forall x\colon P(x) \equiv P(x_1) \wedge P(x_2) \wedge \ldots \equiv \bigwedge_x P(x) \tag{2.6}$$

Der Existenz-Quantor beschreibt, dass mindestens ein Element existiert, das die Aussage $P(x)$ *erfüllt:*

$$\exists x\colon P(x) \equiv P(x_1) \vee P(x_2) \vee \ldots \equiv \bigvee_x P(x) \tag{2.7}$$

Der All-Quantor ist dabei eine unendliche Verkettung mit dem land-Operator. Analog dazu Existenz-Quantor mit dem $\vee$*-Operator.*

Wir haben bei den Quantoren keine Mengen angegeben. Wir können daher für x jedes Element (meist eines vordefinierten Universums) einsetzen. Wenn wir nun eine Aussage über eine Menge A machen wollen, können wir folgenden Zusammenhang ausnutzen:

$$\forall x \in A\colon P(x) \equiv \forall x\colon x \in A \Rightarrow P(x) \tag{2.8}$$
$$\exists x \in A\colon P(x) \equiv \exists x\colon x \in A \wedge P(x) \tag{2.9}$$

2.2 Lineare Temporale Logik

Mittels der Prädikaten können Aussagen über Mengen beschrieben werden, allerdings keine zeitlichen Abläufe innerhalb eines Programmablaufes. Diese Lücke

schließt die Lineare Temporale Logik (LTL). Dafür werden neue Quantoren eingeführt, mit denen wir relativ zeitliche Aussagen treffen können.

Im Unterschied zu der *Computation Tree Logik* CTL ([BK08] Kapitel 6) mit der Aussagen über Bäume getroffen werden können, formulieren wir mit LTL Aussagen über eine Abfolge von Zuständen.

Zunächst definieren wir die zeitliche Abfolge eines Programms. Dabei fassen wir ein Programm als Menge von linearen Abfolgen (Verhalten) von Zuständen auf.

Definition 4 (*Zustände, Schritt, Stotternde Schritte*)

Zustand *Ein Zustand ist eine konkrete Belegung aller Variablen innerhalb des Systems. Wir schreiben für einen Zustand:*

$$\begin{vmatrix} var_1 = value_1 \\ var_2 = value_2 \\ \vdots \quad = \quad \vdots \\ var_n = value_n \end{vmatrix} \tag{2.10}$$

Schritt *Einen Übergang zwischen zwei Zuständen nennen wir Schritt. In TLA$^+$ entsteht ein Schritt durch die Ausführung einer Aktion. Wir notieren einen Schritt wie folgt:*

$$\begin{vmatrix} var_1 = value_1 \\ var_2 = value_2 \\ \vdots \quad = \quad \vdots \\ var_n = value_n \end{vmatrix} \longrightarrow \begin{vmatrix} var_1 \quad = \quad value'_1 \\ var_2 = value'_2 \\ \vdots \quad = \quad \vdots \\ var_n \quad = \quad value'_n \end{vmatrix} \tag{2.11}$$

Ändert die Variablenbelegung in einem Schritt nicht $\sigma_i = \sigma_{i+1}$, so bezeichnen wir diesen Schritt als ***stotternder Schritt*** *(*suttering steps*).*

Eine konkrete Abfolge von möglichen Schritten und den dazwischen liegenden Zuständen bezeichnen wir als Verhalten (*behaviour*) ([Lam02] S. 88). Wobei wir ein Verhalten als σ und als eine Folge von Zuständen ansehen:

$$\sigma \stackrel{\text{def}}{=} \sigma_0 \to \sigma_1 \to \sigma_2 \cdots \tag{2.12}$$

Dabei ist die Sequenz von Zuständen unendlich. Wir nehmen an, dass das Programm divergiert. Für die Modellierung einer Termination können wir uns einem Fangzustand (*trap state*) bedienen.

Die Abbildung 2.1 visualisiert die Auswertung von Aussagen über ein Verhalten. Wir definieren

$$\sigma^{+n} \stackrel{\text{def}}{=} \sigma_n \to \sigma_{n+1} \to \sigma_{n+2} \cdots \tag{2.13}$$

um Sequenzfolgen in der *Zukunft* anzusprechen. Mit den obigen Definitionen können wir die Operatoren: $\Box, \Diamond$ und $\bigcirc$ definieren:

Definition 5 ($\sigma \models F$, *Temporale Operatoren*)

$\sigma \models F$ bedeutet, dass die temporale Aussage F im Verhalten σ erfüllt ist. Falls F eine prädikatenlogische Aussage ist, ist diese erfüllt wenn gilt:

$$\sigma \models F \Leftrightarrow \sigma_1 \models F \tag{2.14}$$

d. h. F ist erfüllt, wenn F im ersten Zustand erfüllt ist.

Abbildung 2.1 zeigt einen Überblick über Semantik von temporalen Operatoren und ihre Bedeutung.

Der Operator $\bigcirc$ (gesprochen next*) referenziert den nächsten Zustand. Damit gilt $\bigcirc F$ genau dann, wenn im Folgezustand F gilt.*

$$\sigma_i \models \bigcirc F \Leftrightarrow \sigma_{i+1} \models F \tag{2.15}$$

Mit $\Box$ (gesprochen box*) drücken wir aus, dass eine Formel für alle folgenden Zustände eines Verhaltens σ erfüllt ist - inklusive des Aktuellen.*

$$\sigma \models \Box F \Leftrightarrow \forall n \in \mathbb{N}\colon \sigma^{+n} \models F \tag{2.16}$$

Um zu spezifizieren, dass F irgendwann in der Zustandsfolge gilt, verwenden wir $\Diamond F$ (gesprochen diamond*):*

$$\sigma \models \Diamond F \Leftrightarrow \exists n \in \mathbb{N}\colon \sigma^{+n} \models F \tag{2.17}$$

Es gelten die in Abbildung 2.2 gezeigten Regeln für die temporale Logik. Oft in der Literatur verwendet wird noch der *Until*-Operator $\omega \mathsf{U} \psi$, der einen Übergang von ω nach ψ im Verlauf des Verhaltens festlegt. Auf diesen gehe ich näher ein, da dieser in TLA$^+$ nicht definiert ist. Wir können $\Box$ und $\Diamond$ alternativ definieren:

$$\Box F \stackrel{\text{def}}{=} \neg \Diamond \neg F \tag{2.18}$$

$$\Diamond F \stackrel{\text{def}}{=} \mathit{true} \mathsf{U} F \tag{2.19}$$

Die Abbildung 2.2 fasst die Regeln mit dem Umformen von Formeln der LTL zusammen. Die Beweise und Erläuterungen sind aus [BK08] zu entnehmen.

Wir erhalten durch die Kombinationen von $\Diamond$ und $\Box$ neue Modellierungsmöglichkeiten: $\Diamond\Box F$ steht dafür, dass F ab einen späteren Zeitpunkt F für immer gilt. Während $\Box\Diamond F$ besagt, dass für jeden zukünftigen Zustand i ein späterer Zustand $j \geq i$ existiert, wobei das für $\sigma_j \models F$ gilt. Aufgrund der divergierenden Verhalten gilt F unendlich oft.

$\Box\Diamond F$ unendlich oft *infinitely often F*
$\Diamond\Box F$ schlussendlich für immer *eventually forever F*

Ein kleines Beispiel aus [BK08] (*Example 5.4 Properties for a Traffic Light*, S. 253) soll die Formulierung mit LTL verdeutlichen:

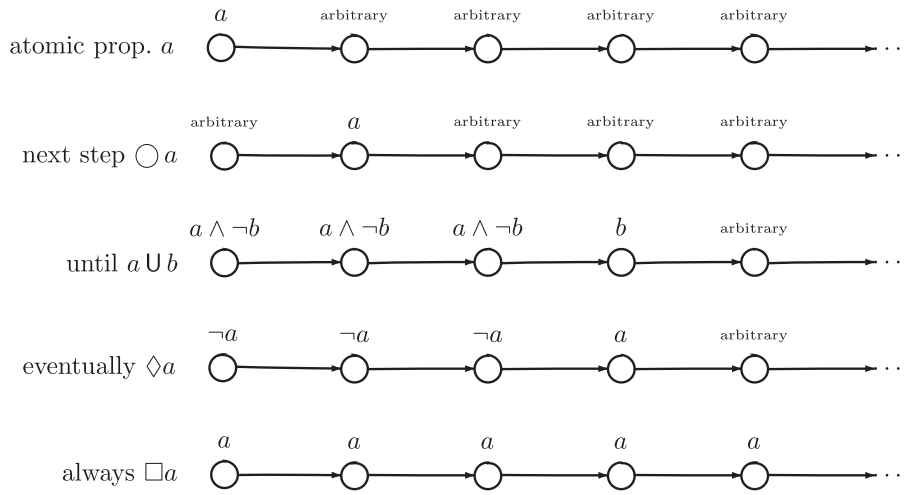

Abbildung 2.1: Darstellung der Temporalen Operatoren aus [BK08] S. 233.

duality law $\neg \bigcirc \varphi \equiv \bigcirc \neg\varphi$ $\neg\Diamond\varphi \equiv \Box\neg\varphi$ $\neg\Box\varphi \equiv \Diamond\neg\varphi$	*idempotency law* $\Diamond\Diamond\varphi \equiv \Diamond\varphi$ $\Box\Box\varphi \equiv \Box\varphi$ $\varphi \mathsf{U} (\varphi \mathsf{U} \psi) \equiv \varphi \mathsf{U} \psi$ $(\varphi \mathsf{U} \psi) \mathsf{U} \psi \equiv \varphi \mathsf{U} \psi$
absorption law $\Diamond\Box\Diamond\varphi \equiv \Box\Diamond\varphi$ $\Box\Diamond\Box\varphi \equiv \Diamond\Box\varphi$	*expansion law* $\varphi \mathsf{U} \psi \equiv \psi \vee (\varphi \wedge \bigcirc(\varphi \mathsf{U} \psi))$ $\Diamond\psi \equiv \psi \vee \bigcirc\Diamond\psi$ $\Box\psi \equiv \psi \wedge \bigcirc\Box\psi$
distributive law $\bigcirc(\varphi \mathsf{U} \psi) \equiv (\bigcirc\varphi) \mathsf{U} (\bigcirc\psi)$ $\Diamond(\varphi \vee \psi) \equiv \Diamond\varphi \vee \Diamond\psi$ $\Box(\varphi \wedge \psi) \equiv \Box\varphi \wedge \Box\psi$	

Abbildung 2.2: Regeln der temporalen Logik aus [BK08]

Eine Verkehrsampel hat die Phasen *grün*, *rot* und *gelb*. Dabei soll die Ampel folgende Zustandsfolge einhalten:

$$rot \longrightarrow gelb \longrightarrow grün \longrightarrow gelb \longrightarrow rot \longrightarrow \ldots \tag{2.20}$$

Zusätzlich spezifizieren wir eine Liveness-Anfordung, d. h. die Ampel soll ihren Zustand wechseln müssen. Sobald sie einmal *rot* ist, wird sie auch wieder *grün*.

Wir können z. B.festhalten, dass *grün* unendlich oft gilt:

$$\Box\Diamond grün \tag{2.21}$$

Dabei haben wir noch keine Abfolge von Zuständen festgelegt. Wir können ausschließen, dass nach *rot grün* folgt:

$$\Box(rot \Rightarrow \neg\bigcirc grün) \tag{2.22}$$

Dies können wir für jeden entsprechenden Schritt festlegen. Für die Anforderung, dass sobald die Ampel *rot* ist, sie wieder nach einer *gelb*-Phase *grün* wird, können wir folgendes festhalten:

$$\Box(rot \Rightarrow (rot \mathsf{U} (gelb \wedge \bigcirc(gelb \mathsf{U} grün)))). \tag{2.23}$$

Wir können die Aussage wie folgt interpretieren:

□ Für jeden Zustand,
rot ⇒ wenn die Ampel *rot* ist, folgt daraus
U dass sie einen Übergang von *rot* nach *gelb* hat.
gelb ∧ Dabei ist die Ampel *gelb* und
U kann nach dem nächsten Zustand ○ von *gelb* nach *grün* wechseln.

Die Aussage, dass *rot* immer zu *grün* führt (*leads-to*), kann man durch $rot \rightsquigarrow grün$ ausdrücken. Dabei steht $\omega \rightsquigarrow \psi$ für: *leads-to*

$$\Box(\omega \Rightarrow \Diamond\psi)$$

. Wir können mit der oben angegeben Definition die Bedeutung herleiten (vgl. [Lam02]):

$$\sigma \models (F \rightsquigarrow G) \Leftrightarrow \tag{2.24}$$
$$\forall n \in \mathbb{N}\colon (\sigma^{+n} \models F) \Rightarrow (\exists m \in \mathbb{N}\colon (\sigma^{+(n+m)} \models G) \tag{2.25}$$

Oft werden noch weitere Operatoren wie *weak-until* oder *release* eingeführt, wenn die Aufgabenstellung dies erfordert (vgl. [BK08] Kapitel 5).

2.3 Invarianten, Liveness und Fairness

Wir spezifizieren Systeme in TLA$^+$ durch das Erlauben von Aktionen abhängig der aktuellen Variablenbelegung. Dadurch verbieten wir Aktionen, die eine Gefahr darstellen können. Solche Spezifikationen genügen in erster Hinsicht der Safety[1]. Einprägsam in [BK08] (S. 107) beschrieben als »nothing bad should happen«. Safety-Eigenschaften

Solche Eigenschaften stellen Invarianten sicher. Invarianten I sind Aussagen, die über jeden Zustand $\Box I$ in jedem Pfad des Ablaufes gelten müssen. Zum Beispiel kann mithilfe einer Invarianten der Variablentyp festgelegt werden. Als Invariante für unsere Ampel aus dem letzten Abschnitt könnten wir mit: Invarianten

[1] Sicherheit im Bezug auf Leib und Leben

$$rot \vee gelb \vee grün$$

die möglichen Zustände begrenzen.

Mit Safety-Anforderungen verbieten oder erlauben wir bestimmte Aktionen. Wir erzwingen aber ein Ausführen von Aktionen nicht. Wir können grundsätzlich davon ausgehen, dass in einem System *stuttering steps* zulässig sind. Dies ist erforderlich damit die Wiederverwendbarkeit bei der Kombination von mehreren Modulen gewährleistet wird. Als Beispiel könnte die Ampel aus dem obigen Beispiel immer in der *rot*-Phase verharren. Aussagen, die einen Wechsel des Zustandes eines Systems verlangen, schreiben eine Fairness vor. Für unsere Ampel könnten wir vorschreiben: Liveness

$$\Box\Diamond rot \wedge \Box\Diamond grün \tag{2.26}$$

.

Wir schreiben vor, dass *rot* und *grün* unendlich oft gilt. Wenn wir nur ein Teil der Fairness-Anforderung stellen würden. Zum Beispiel Aussage $\Box\Diamond grün$, die auch gelten würde, wenn die Ampel für immer in *grün* verharren würde.

Wir halten fest, dass Safety-Eigenschaften in endlicher Zeit verletzt werden, während Fairness-Eigenschaften nur in unendlicher Zeit verletzt werden ([BK08] S. 120). Für ein System spezifizieren wir Fairness-Anforderungen, die gelten müssen, um damit die Liveness zu beweisen. Wir erlangen bei nachgewiesener Liveness zur Folgerung, dass das System nicht verharrt bzw. blockiert.

Kombinieren wir Ampeln zu einer Ampelanlage, um den Verkehrsfluss einer ganzen Kreuzung zu regeln, würden wir vorschreiben, dass die Ampeln sich ab und zu umschalten. Die obige Liveness-Anforderung für jede Ampel würde dies sicherstellen, dass eine Ampeln keine Grün-Phase bekäme (*Starvation*). Das Entstehen von Unfällen wäre ein Aspekt der Safety und würde über die Invarianten abgefangen. Fairness

In der Praxis werden feinere Liveness-Aussagen getroffen. Dabei wird an die Ausführung von Aktionen Bedingungen gestellt. Wir bezeichnen dies als Fairness-Anforderungen. Allgemein können wir Fairness wie folgt unterscheiden:

Unconditional Fairness Jeder Teilnehmer bekommt seinen Zug unendlich oft.

Strong Fairness Jeder Prozess, der unendlich oft aktiviert wird, bekommt seinen Zug unendlich oft.

Weak Fairness Jeder Prozess, der für immer aktiviert ist, bekommt seinen Zug unendlich oft.

Die *Unconditional Fairness* haben wir indirekt über unsere Liveness-Eigenschaft (2.26) definiert, in der jede Ampel unendlich oft umschalten muss. In der Praxis würde die Ampel auf *grün* umschalten, obwohl niemand auf der Spur wartet. Hier wäre die Verwendung von *Strong Fairness* oder *Weak Fairness* angebracht, um ein Umschalten nur zu fordern, wenn ein Auto die Ampel aktiviert.

Im folgenden Kapitel gehe ich auf die Umsetzung von *Strong Fairness* und *Weak Fairness* in TLA^+ ein.

3

Die Sprache TLA$^+$

Die *Temporal Logic for Actions* TLA$^+$ ist eine Erweiterung der Prädikaten und Linearen Temporalen Logik. Dabei werden die Logiken um Aspekte für die Systemmodellierung ergänzt. Dieses Kapitel soll einen Überblick über die Semantik und Anwendung von TLA$^+$ geben. Eine vollständige Beschreibung ist in [Lam02].

TLA$^+$ beschreibt eine Temporale Logik im Sinne der Linearen Temporalen Logik. Dabei greift es auf Zustände, Schritte und Operatoren zurück. Hinzu kommt allerdings die Beschreibung von Aktionen, die angewendet werden können.

Eine Aktion überführt einen Zustand in einen (oder mehrere Folgezustände). In der Spezifikation drückt sich eine Aktion als eine Konjunktion aus.

Innerhalb der Konjunktion befindet sich die Zuweisung an die *zukünftigen* Variablen. Diese Variablen des nächsten Zustandes kennzeichnen wir mit $\ll'\gg$ (*prime*) z. B.$x' = 2$. Jede Variable muss einen Wert zugewiesen bekommen. Sollen eine Variable nicht geändert werden, dann schreiben wir $x' = x$ oder kürzer UNCHANGED x.

prime ′

UNCHANGED

Um die Anwendung von Aktionen zu unterbinden, können wir neben den Zuweisungen noch Bedingungen in die Konjunktion aufnehmen. Diese Aussagen werden als *enabling condition* bezeichnet und beziehen sich auf den aktuellen Zustand. Eine Aktion ist zusammenfassend eine Konjunktion, bestehend aus der:

enabling condition

- *Enabling Condition*
- Wertzuweisungen
- UNCHANGED Aussagen.

Ein weiterer Operator meist *Init* beschreibt die Initialisierung des System. Bei den Wertzuweisungen sind keine *prime*-Variablen erlaubt und meistens werden keine konkreten Werte zugewiesen (*HCini* in Abbildung 3.1, *IInit* in Abbildung 3.3). Weitere Einblicke in die Syntax und Semantik folgen im nächsten Abschnitt.

TLA$^+$ ist eine ungetypte Sprache. Die Variablen haben keine Typ, damit sind Ausdrücke wie "*abc*" + 2 syntaktisch und semantisch gültig. Diese werden dann vom Modelchecker TLC beim Auswerten bemängelt. Wir müssen die Typen unserer Variablen durch Invarianten festlegen (*HCini* in Abbildung 3.1, *TypeInvariant* in Abbildung 3.3)

Spezifikationen werden in einer ASCII-Notation verfasst. Ich werde mich im Verlauf des nächsten Abschnitts auf die LaTeX-Notation konzentrieren. Die ASCII-Notation in Tabelle 3.1 aufgeführt. Abbildung 3.2 zeigt die ASCII-Notation für

Module HourClock (Abbildung 3.1). Eine Übersicht über die ASCII-Notation ist in [Lam02] Seite 273.

MODULE *HourClock*

EXTENDS *Naturals*
VARIABLE *hr*
$HCini \triangleq hr \in (1 \,..\, 12)$
$HCnxt \triangleq hr' = \text{IF } hr \neq 12 \text{ THEN } hr + 1 \text{ ELSE } 1$
$HC \triangleq HCini \wedge \Box[HCnxt]_{hr}$

THEOREM $HC \Rightarrow \Box HCini$

Abbildung 3.1: Module HourClock

```
---------------------- MODULE HourClock ----------------------
EXTENDS Naturals
VARIABLE hr
HCini  ==  hr \in (1 .. 12)
HCnxt  ==  hr' = IF hr # 12 THEN hr + 1 ELSE 1
HC  ==  HCini /\ [][HCnxt]_hr
-------------------------------------------------------------
THEOREM  HC => []HCini
=============================================================
```

Abbildung 3.2: Module HourClock (Abbildung 3.1) in ASCII-Notation

3.1 Semantik und Syntax

Ich möchte hier anhand von zwei Beispielspezifikationen (Abbdildung 3.1 und 3.3) die Syntax und Semantik erläutern.

Wir definieren ein System als eine Menge von Modulen. Für den Modelchecker wird jedes Modul separat in einer gleichnamigen Datei abgelegt. Jedes Modul hat eine Kopfzeile mit ihrem Namen und eine Fußzeile. Innerhalb dieser stehen die einzelnen Anweisungen. Dabei bildet ein Modul einen Namensraum.

Mit EXTENDS können wir andere Module in unseren Namensraum importieren. Dabei können wir direkt auf die dort definierten Variablen, Funktionen und Operatoren zugreifen. Eine andere Möglichkeit besteht in der Instanzierung von Modulen (INSTANCE). Dabei wird der Namensraum des instanzierten Moduls an die Variable gebunden. EXTENDS INSTANCE

Über VARIABLE (bzw. VARIABLES) können wir die Variablen definieren. Wir können Variable von ihrer Bedeutung wie aus gängigen Programmiersprachen

MODULE *InternalMemory*

EXTENDS *MemoryInterface*
VARIABLES *mem*, *ctl*, *buf*

$$
\begin{array}{l}
IInit \triangleq \\
\quad \land\, mem \in [Adr \to Val] \\
\quad \land\, ctl \;\; = [p \in Proc \mapsto \text{“rdy”}] \\
\quad \land\, buf \;\; = [p \in Proc \mapsto NoVal] \\
\quad \land\, memInt \in InitMemInt
\end{array}
$$

$$
\begin{array}{l}
TypeInvariant \triangleq \\
\quad \land\, mem \in [Adr \to Val] \\
\quad \land\, ctl \;\; \in [Proc \to \{\text{“rdy”}, \text{“busy”}, \text{“done”}\}] \\
\quad \land\, buf \;\; \in [Proc \to MReq \cup Val \cup \{NoVal\}]
\end{array}
$$

$$
\begin{array}{l}
Req(p) \triangleq \land\, ctl[p] = \text{“rdy”} \\
\qquad\qquad \land\, \exists\, req \in MReq : \\
\qquad\qquad\quad \land\, Send(p,\, req,\, memInt,\, memInt') \\
\qquad\qquad\quad \land\, buf' = [buf \text{ EXCEPT } ![p] = req] \\
\qquad\qquad\quad \land\, ctl' \;\; = [ctl \text{ EXCEPT } ![p] \;\; = \text{“busy”}] \\
\qquad\qquad \land\, \text{UNCHANGED } mem
\end{array}
$$

$$
\begin{array}{l}
Do(p) \triangleq \\
\quad \land\, ctl[p] = \text{“busy”} \\
\quad \land\, mem' = \text{IF } buf[p].op = \text{“Wr”} \\
\qquad\qquad\quad \text{THEN } [mem \text{ EXCEPT } ![buf[p].adr] = buf[p].val] \\
\qquad\qquad\quad \text{ELSE } mem \\
\quad \land\, buf' \;\; = [buf \text{ EXCEPT } ![p] = \text{IF } buf[p].op = \text{“Wr”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{THEN } NoVal \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE } mem[buf[p].adr]] \\
\quad \land\, ctl' \;\; = [ctl \text{ EXCEPT } ![p] \;\; = \text{“done”}] \\
\quad \land\, \text{UNCHANGED } memInt
\end{array}
$$

$$
\begin{array}{l}
Rsp(p) \triangleq \land\, ctl[p] = \text{“done”} \\
\qquad\qquad \land\, Reply(p,\, buf[p],\, memInt,\, memInt') \\
\qquad\qquad \land\, ctl' = [ctl \text{ EXCEPT } ![p] = \text{“rdy”}] \\
\qquad\qquad \land\, \text{UNCHANGED } \langle mem,\, buf \rangle
\end{array}
$$

$$INext \triangleq \exists\, p \in Proc : Req(p) \lor Do(p) \lor Rsp(p)$$

$$ISpec \triangleq IInit \;\land\; \Box[INext]_{\langle memInt,\, mem,\, ctl,\, buf \rangle}$$

THEOREM $ISpec \Rightarrow \Box\, TypeInvariant$

Abbildung 3.3: Module InternalMemory

auffassen. Über CONSTANTS können wir Konstanten definieren, die von außerhalb des Moduls gesetzt werden, z. B.vom Modelchecker aus. Möchten wir, dass der Modelchecker bestimmte Aussagen über die Konstanten prüft, so können wir ASSUME gefolgt der Aussage verwenden.

Über $\triangleq$ deklarieren wir Operatoren wie *HCini* oder *HCnxt* (Abbildung 3.1). Wir können diese Operatoren als Stellvertreter der Aussagen betrachten. Dabei ist es auch möglich, Operatoren zu parametrisieren wie $Rsp(x)$ oder $Send(x)$ in Abbildung 3.3. Es besteht auch die Möglichkeiten binäre Infix-Operatoren zu definieren:

$$f@@g == [x \in (\text{DOMAIN } f) \cup (\text{DOMAIN } g) \mapsto \tag{3.1}$$
$$\text{IF } x \in \text{DOMAIN } f \text{ THEN } f[x] \text{ ELSE } g[x]] \tag{3.2}$$

.

Mit DOMAIN erhalten wir den Urbildbereich einer Funktion. Dabei steht $f@@g$ für eine Funktion h, sodass

$$h(x) \stackrel{\text{def}}{=} \begin{cases} f(x) & : x \in \text{DOMAIN } f \\ g(x) & \text{sonst} \end{cases} \tag{3.3}$$

Eine detaillierte Diskussion zu Datentypen wie Funktionen oder Tupel ist in Abschnitt 3.2.

Mit THEOREM deklararieren wir eine Aussage, die vom Modelchecker zu beweisen oder widerlegen ist. Im Beispiel von *HourClock* ist das die Einhaltung der Typinvarianten.

Das Modul *HourClock* ist wie folgt zu verstehen. Sobald wir *HC* „ausführen“ soll das THEOREM $HC \Rightarrow \Box HCini$ gelten. *HCinit* sagt aus, dass $hr \in \{x \in \mathbb{N} \| 1 \leq x \leq 12\}$ liegt. Der Operator *HC* ist der Einstiegspunkt in die Spezifikation. Dieser legt fest, dass das System mit *HCini* initialisiert wird und durch Ausführung der Aktion *HCnxt* in den nächsten Zustand überführt. Dabei steht $[A]_v$ für

$$[A]_v \stackrel{\text{def}}{=} A \vee (v' = v)$$

und erweitert die Aktion A um *stuttering steps*. Wenn v ein Tupel ist, dann gilt

$$[A]_{\langle v_1,\ldots,v_n\rangle} \stackrel{\text{def}}{=} A \vee (v_1' = v_1) \vee \ldots \vee (v_n' = v_n)$$

Der Operator .. in *HCini* $\triangleq$ $hr \in (1..12)$ wird über das Modul *Naturals* bereitgestellt. *HCnxt* inkrementiert hr, wenn $hr \neq 12$ sonst gilt $hr' = 1$.

Die Fallunterscheidung kann zum Einen über IF abgewickelt werden IF

IF *condition* THEN e_1 ELSE e_2

.

Dabei ist die ELSE -Fall obligatorisch. Eine weitere Möglichkeit besteht mit CASE dem CASE -Ausdruck:

$$v' = \text{CASE} \quad \begin{array}{ll} p_1 & \to e_1 \\ \vdots & \to \vdots \\ p_n & \to e_n \\ [\text{OTHER} & \to e] \end{array} \tag{3.4}$$

Dabei wählt CASE den Wert e_i für v, sodass p_i wahr ist. Die Bedingungen p_i sollten sich gegenseitig ausschließen und es sollte immer ein Fall p_i zutreffen. Sonst ist der Wert v undefiniert. Der OTHER -Fall ist obligatorisch

IF und CASE lassen sich auf CHOOSE zurückführen ([Lam02] S. 298). Mit CHOOSE kann ein Wert gewählt werden, sodass die angegeben Bestimmungen eingehalten werden:

$$\text{CHOOSE } v : \textit{constraints}$$

wobei die *constraints* beliebige Einschränkungen an v darstellen, wie z. B.$v \in Nat$ oder $v \leq 10$. CHOOSE ist deterministisch und hilft die Anzahl von Folgezuständen zu minimieren ([Mer] S. 90).

rekursive Funktionen

Es gibt neben den Operatoren auch die Möglichkeit Funktionen in TLA$^+$ zu definieren. Eine Funktion kann dabei wie folgt definiert werden:

$$fact[x \in Nat] \triangleq \text{IF } x = 0 \text{ THEN } 1 \text{ ELSE } n \cdot fact[n-1]$$

.

Dabei ist $fact[x] \equiv x!$ eine rekursive Funktionsdefinition. Rekursive Operatoren werden seit TLA$^+$ in Version 2 unterstützt (vgl. [Lam10]). Im Unterschied zu Operatoren besitzen Funktionen eine Urbild- und Bildmenge, liefern immer einen Wert zurück. Es sind keine Funktionen in der Infixnotation erlaubt. Außerdem können Operatoren auch Operatoren als Argument übergeben. Funktionen sind nicht höherer Ordnung (vgl. [Lam02] S. 69 ff.).

3.2 Komplexe Datentypen

Innerhalb von TLA$^+$ gibt es diverse Datentypen. Durch die Sprache unterstützt TLA$^+$ Prädikatenlogik, Mengen und Funktionen.[1] Andere Datentypen werden durch Module unterstützt, z. B.natürliche und reelle Zahlen, Sequenzen, Zeichenketten, Multisets (*Bag*) aber Bauckus-Naur-Form (*BNF*) und Graphen.

Im Verlauf dieses Abschnittes gehe ich auf Mengen, Funktionen, Records und Sequenzen bzw. Strings ein. Die natürlichen, ganzen bzw. reellen Zahlen sind in den Modulen *Naturals*, *Integer* bzw. *Reals* mit den üblichen arithmetischen und logischen Operatoren $(+, -, \cdot, <, \geq, \ldots)$ definiert. Wichtig dabei sind die Zahlenmengen $\mathbb{N}$ *Nat*, $\mathbb{Z}$ *Int* und $\mathbb{R}$ *Real*. Die Datenstrukturen in TLA$^+$ sind in den Kapiteln 11 und

[1] TLA$^+$ baut Zermelo-Fraenkel-Mengenlehre auf

18 in [Lam02] beschrieben. Eine Zusammenfassung der Syntax und Semantik ist in[Lam02] S. 268 ff.

Mengen können deklariert werden durch Aufzählung der einzelnen Elemente e_i, durch Angabe einer Menge S und Aussage p oder durch Angabe eines Ausdruckes e und einer Menge S:

$$\{e_1, \ldots, e_n\} \qquad \{x \in S : p\} \qquad \{e : p\} \tag{3.5}$$

Beispielweise definiert das Modul *Naturals* den Infixoperator _.._:

$$a..b \triangleq \{i \in Nat : (a \leq i) / (i \leq b)\}$$

Mengen

TLA$^+$ hält für die Mengen die üblichen Operatoren bereit: Vereinigung ($\cup$), Durschnitt ($\cap$) Teilmenge ($\subset$), Differenz ($\setminus$) sowie Enthaltenprädikat ($\in, \notin$) bereit. Die Potenzmenge $\mathcal{P}(S)$ kann durch SUBSET erreicht werden. Mächtigkeit $|S|$ befindet sich im Modul *FiniteSets*.

Funktionen

Funktionen sind Abbildungen von der Urbild- zur Bildmenge. Bei Funktionen unterscheiden wir zwischen der Definition der Funktion sowie der Funktionsmenge. Wir definieren eine Funktion wie folgt:

$$f = [x \in S \mapsto e]$$

.

bzw.

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

Dabei gilt DOMAIN $f = S$ ist. Wir können auch die Funktionsmenge defnieren. Diesen Ausdruck brauchen wir für Typinvarianten, um festzulegen, dass die Funktion von der Menge S nach T abbildet:

$$f \in [S \to T]$$

Eine Funktion rufen wir mit eckigen Klammern auf $f[x]$. Das Ändern von Funktion wird über EXCEPT durchgeführt.

$$f' = [f \text{ EXCEPT } ![x_1] = e_1, \ldots ![x_n] = e_n]$$

Dabei wird f' zur Funktion f, wobei für die Funktionsargumente x_i der Wert abgeändert wurde. Wir können auf den alten Funktionswert mit $f[x_i]$ bzw. @ zugreifen.

Records sind strukturierte Datentyp mit Attributen, wie Records in Pascal oder Structs in C. TLA$^+$ behandelt diese wie Funktionen mit der Urbildmenge *STRING* und einigen syntaktischen Abkürzungen. Ein Record definieren wir mit

$$r = [h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$$

Der Zugriff auf die Felder können wir mit $r.h$. Wir können auch Mengen aller Records mit bestimmten Felder und Mengen bilden:

$$r \in [h_1 : S_1, \ldots, h_n : S_n]$$

Records werden auch mit EXCEPT abgeändert

$$\hat{r} = [r \text{ EXCEPT } !.h_1 \mapsto S_1, \ldots, !.h_n \mapsto S_n]$$

Funktionen und Records können beliebig verschachtelt werden.

Sequenz & Tupel

Auch Tupels bzw. Sequenzen gehen von ihrer Semantik auf Funktionen zurück. Dabei bildet ein Tupel von $Nat \setminus \{0\}$ zu einer beliebigen Menge ab. Wir definieren ein Tupel in TLA$^+$ mit spitzigen Klammern

$$t = \langle e_1, \ldots, e_n \rangle$$

und können mit $e[i]$ auf das i-te Elemente zugreifen. Daneben können wir eine Menge von Tupeln wie üblich definieren:

$$t \in S_1 \times \ldots \times S_n$$

Wir können Tupel genau wie Funktionen behandeln - also mit EXCEPT abändern. Für den praktischen Einsatz stellt das Modul *Sequences* Operatoren bereit. Darunter Konkatenation $\circ$, Zugriff auf das Kopfelemente *Head* und Restliste *Tail* sowie Länge *Len* der Sequenz und das Einfügen eines Elementes e an das Ende eines Tupels s ($Append(s, e)$). Zusätzlich kann mit $Seq(S)$ eine Menge mit allen Tupel beliebiger Länge mit Elementen aus S erzeugen.

$$\forall n \in Nat \setminus \{0\} : \langle e_1 \in S, \ldots, e_n \in S \rangle \in Seq(S)$$

Es gibt noch $SubSeq(s, m, n)$ um eine Teil Sequenz $\langle e_m, \ldots, e_n \rangle$ aus s zu erhalten. Daneben kann man mit $SelectSeq(s, Test(\_))$ eine Sequenz aus s in der jedes Element e den Operator $Test(e)$ erfüllt.

$\wedge$	`/\ \land`	$\vee$	`\/ \lor`	$\Rightarrow$	`=>`
$\neg$	`\neg \lot`	$\equiv$	`<=> \equiv`	$\triangleq$	`==`
$\in$	`in`	$\notin$	`notin`	$\neq$	`# /=`
$\langle$	`<<`	$\rangle$	`>>`	$\Box$	`[]`
$<$	`<`	$>$	`>`	$\Diamond$	`<>`
$\leq$	`<= \leq`	$\geq$	`>= \geq`	$\rightsquigarrow$	`~>`
$\mapsto$	`|->`	$\rightarrow$	`->`	$\div$	`\div`
$\exists$	`\E`	$\forall$	`\A`	$'$	`'`
$]_v$	`]_v`	$\rangle_v$	`>_v`	$\cap$	`\intersect \cap`
WF_v	`WF_v`	SF_v	`SF_v`	$\cup$	`\cup \union`

Tabelle 3.1: Ausgewählte ASCII-Repräsentation in TLA$^+$ (vgl. [Lam02] S. 273)

3.3 Liveness und Fairness

Grundsätzlich gilt auch in TLA$^+$ die Eigenschaften zur Liveness und Fairness aus Abschnitt 2.3. Dort habe ich von aktivierten Teilnehmern geschrieben. Nun können wir dies genauer formulieren. In TLA$^+$ reden wir von Aktionen anstatt von Teilnehmern. Dabei ist eine Aktion A aktiviert, wenn diese durchgeführt werden kann genau dann, wenn es gilt ENABLED A.

Hinzu kommt nun, dass wir grundsätzlich *suttering steps* haben. Wir möchten aber durch Liveness vorschreiben, dass eine Aktion ausgeführt wird - nicht jedoch ein *stuttered step* also $v' = v$. In der Semantik von TLA$^+$ können wir für einen A Schritt folgendes schreiben:

$$\langle A \rangle_v \stackrel{\text{def}}{=} A \wedge (v' \neq v)$$

Damit können wir Fairness-Bedingungen kürzer formulieren. TLA$^+$ hält dafür Weak- und Strong-Fairness die Prädikate $WF_{vars}(A)$ bzw. $SF_{vars}(A)$ bereit. Die wie folgt definiert sind:

$$WF_{vars}(A) \stackrel{\text{def}}{=} \quad \Diamond\Box(\text{ENABLED } \langle A \rangle_{vars}) \Rightarrow \Box\Diamond\langle A \rangle_{vars} \tag{3.6}$$

$$SF_{vars}(A) \stackrel{\text{def}}{=} \quad \Box\Diamond(\text{ENABLED } \langle A \rangle_{vars)}) \Rightarrow \Box\Diamond\langle A \rangle_{vars} \tag{3.7}$$

Wir schreiben die Fairness-Anfordungen in die Konjunktion der Spezifikationen. Spezifikation sind immer daher nach dem gleichen Muster aufgebaut, der TLA$^+$-Normalform

TLA$^+$-Normalform

$$Init \wedge [Next]_{vars} \wedge Fairness \tag{3.8}$$

Dabei ist *Fairness* eine Junktion von *SF* und *WF*-Aussagen. *WF* bzw. *SF*-Anfordungen können über die **SF** bzw. **WF Conjunction Rule** zusammengefasst werden ([Lam02] S. 105 ff.).

Definition 6 (*WF/SF Conjunction Rule*)

Seien $A_1, \ldots, A_n$ Aktionen, die sich paarweise ausschließen, d. h. für je zwei A_i, A_j gilt: Sobald ENABLED $\langle A_i \rangle_v$ *gilt, kann A_j nicht ausgeführt solange dass A_i nicht ausgeführt wurde.*

Schließen sich $A_1, \ldots, A_n$ paarweise aus, dann können wir die Fairness-Anforderungen zusammenfassen:

$$WF_v(A_1) \wedge \ldots \wedge WF_v(A_n) \equiv WF_v(A_1 \vee \ldots \vee A_n) \tag{3.9}$$

$$SF_v(A_1) \wedge \ldots \wedge SF_v(A_n) \equiv SF_v(A_1 \vee \ldots \vee A_n) \tag{3.10}$$

*Außerdem lassen sich in Weak-Fairness-Anforderungen reinziehen (*WF Quantifier Rule*):*

$$\forall i \in S : \; WF_v(A_i) \equiv WF_v(\exists i \in S : \; A_i)$$

Die Beweise seien [Lam02] (S. 105 ff.) überlassen.

In manchen Fällen ist selbst die Weak-Fairness-Anfordung noch zu stark. Zum Beispiel sollen Speicherbereiche erst verdrängt werden, sobald keine freien Speicherplätze zur Verfügung stehen. Obwohl es zu jeder Zeit möglich ist, Speicherbereiche auf die Festplatte auszulagern und damit die entsprechende Aktion immer aktiviert ist, fordern wir eine Ausführung unter einer weiteren Bedingung ρ. Solche zusätzlichen Bedingungen für das Ausführen einer Aktionen setzen wir konjungiert in die Fairness-Anforderung:

$$WF_{vars}((\rho \wedge A) \vee \ldots)$$

Die obigen Angaben sind aus [Lam02] Kapitel 8.

4

TLC – TLA$^+$-Modelchecker

Diesen Abschnitt widme ich der Erläuterung des Modelcheckers TLC. Dieser kann eine Untermenge von TLA$^+$-Spezifikation verarbeiten. Wir betrachten die Grenzen in Abschnitt 4.1.

Mithilfe von TLC können Eigenschaften nachgewiesen werden. Dazu wird jeder mögliche Zustand, der nach Spezifikation aus erreichbar wäre, erzeugt. Anhand der erzeugten Zustände können die Invarianten und Fairness-Anforderungen überprüft werden (Abschnitt 4.2). Mit diesem Wissen können wir dann die Optionen des TLC betrachten (Abschnitt 4.3).

In TLC unterscheiden wir zwischen der Spezifikation, die in TLA$^+$ zugrundeliegt, und dem Model, das die entsprechenden Optionen für TLC bereithält.

TLC gibt als alleinstehendes Programm, das anhand einer Konfigurationsdatei und einer Spezifikation arbeitet. Ich betrachte hier nur die Verwendung von TLC innerhalb der TLA$^+$-Toolbox[1], die noch TLATEX, PlusCal-Translator sowie den TLAPS (TLA$^+$-Proof System) beinhaltet.

Die hier gemachten Aussagen gehen zurück auf [Lam02] Kapitel 14. Dort wird die direkte Verwendung von TLC behandelt sowie weitere Hilfestellung gegeben.

Grundsätzlich gilt, dass Safety-Properties (Invarianten) mit einer endlichen Folge von Zuständen widerlegt werden können. Während dagegen Liveness und Fairness eine unendliches Verhalten zur Widerlegung brauchen.

4.1 Grenzen

TLC Werte

Die Grenzen von TLC liegen in den verwendbaren Datentypen und deren Vergleichbarkeit ([Lam02] Kapitel 14.2.1) und in der Auswertung von Ausdrücken ([Lam02] Kapitel 14.2.2).

TLC Werte sind induktiv definiert. *(Induktionsanfang)* Als primitive Datentypen stehen

Booleans Wahrheitswert (TRUE, FALSE)
Integers Ganze Zahlen
Strings Zeichenketten wie “abc”

[1] `http://www.tlaplus.net/tools/tla-toolbox/`

Model Values Werte, die durch die Definition von CONSTANTS im Model gesetzt werden (z. B.d1, d2). Bei unterschiedlichen Namen geht TLC davon aus, dass die Werte sich unterscheiden.

(Induktionsschritt) Wenn wir TLC Werte vorliegen haben, können wir daraus weitere TLC Werte bauen:

- Eine Menge von unterscheidbaren TLC Werten. TLC Werte sind unterscheidbar, wenn sie des gleichen Typs sind. Vergleichbar ist also “abc” mit “def” aber nicht 123 und “abc”.
- Eine Funktion f, wobei DOMAIN f ein TLC Wert sowie $f[x]$ für jedes $x \in$ DOMAIN f ein TLC Wert ist.

TLC führt Anweisungen von links nach rechts aus. Im Einzelnen heißt dies für $p \wedge q$, $p \vee q$, dass p vor q ausgewertet werden. Bei IF p THEN e_1 ELSE e_2 wird zuerst p ausgewertet und dann abhängig von p e_1 oder e_2. Die Ausführung von Junktionen können vorzeitig abgebrochen werden, wenn das Ergebnis fest steht (*short-curcuit, lazy-evaluation*). Sollten falsche Ausdrücke (z. B.“123”+1) nicht ausgeführt werden, liefert TLC auch keine Fehlermeldung.

Mengen müssen in TLC endlich sein. TLC zählt bei Anweisungen wie $\exists e \in ScolonP(x)$ alle Elemente von S auf. Sollte S nicht aufzählbar bzw. unendlich sein, bricht TLC mit einer Fehlermeldung. Diese Regel gilt für folgende Konstrukte[2]:

$\exists x \in S : p$	$\forall x \in S : p$	CHOOSE $x \in S : p$
$\{x \in S : p\}$	$\{e : x \in S\}$	$[x \in S \mapsto e]$
SUBSET S	UNION S	

Die Verwendung von ungebundenen Variablen in $\exists x : p$ bzw. beim Allquantor und CHOOSE -Befehl ist nicht möglich.

Im Weiteren sollte man beachten, dass die Auswertung von rekursiven Funktionen nicht divergieren.

TLC kann nicht alle Temporalen Formeln auswerten. Wir können folgende Klassen von Formeln auswerten (vgl. [Lam02] S. 236).

State Predicate Formeln der Prädikatenlogik, die keine temporalen Operatoren oder *primed*-Variablen (x') beinhalten.

Invariance Formula Formeln der Form $\Box P$ mit P als *state predicate*.

Box-Action Formel Eine Formeln der Form $\Box[A]_v$ wobei A eine Aktion und v eine *state function* ist.

Simple Temporale Formel Besteht aus der Kombination von:

- Einfache Bool’schen Operatoren wie $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$, TRUE, FALSE sowie Quantoren der Prädikatenlogik über endliche Menge.
- *state functions* kombiniert mit $\Box, \Diamond, \rightsquigarrow$ und den Einfachen Bool’schen Operatoren.

[2] Einige Ausnahmen gibt es: z. B.$[n \in Nat \mapsto n \cdot (n+1)][3]$. Aber $[n \in Nat \mapsto n]$ ist selber nicht auswertbar.

- Sowie Formeln mit Aktionen A:

$$WF_v(A) \qquad SF_v(A) \qquad \Box\Diamond\langle A\rangle_v \qquad \Diamond\Box[A]_v \tag{4.1}$$

  sowie ENABLED $\langle A\rangle_v$.

TLC kann nicht $3+3$ berechnen aufgrund der Definition im Modul *Naturals*, aufgrund der Definition über unendliche Menge. Die Einbindung von *Naturals* wird durch eine Java-Klasse ersetzt, die die Operationen übernimmt. Möchte man nun eine eigene Definition + in TLC haben, so ist eine Änderungen von TLC erforderlich. Dies gilt auch für weitere Standardspezifikationen: *Naturals*, *Integers*, *Sequences*, *FiniteSets* und *Bags*.

4.2 Erzeugung der Verhalten

Erzeugen Folgenzustände

Verhalten werden durch Auswertung des *next-state* Operator (oft *Next*) erzeugt. Dabei ist das Vorgehen in zwei Punkten abweichend vom vorherigen Kapitel.

Alle Variablen erhalten den Wert im aktuellen Zustand. *Next* ist eine Disjunktion von Aktionen. Abweichend von Abschnitt 4.1 wird jeder Term parallel ausgewertet. Der zweite Unterschied ist die Auswertung von *prime*-Variablen. Sobald TLC auf einen Ausdruck $x' = e$ wird erhält x den Wert durch die Auswertung von v im nächsten Zustand und der Ausdruck ist TRUE. Stößt TLC auf einen Ausdruck $x' \in S$ bzw. $\exists v \in S : x' = v$ erstellt TLC mehrere Folgezustände. Sollte die bei der Auswertung eines Terms FALSEergeben, so können wir diese Aktion nicht ausführen.

TLC unterscheidet zwei Moden – Model-Checking Modus und Simulationsmodus. Ich gehe zuerst auf den Model-Checking Modus ein ([Lam02] S. 240, Kapitel 14.3).

Dem Model-Checking Modus liegt die Breitensuche zugrunde. Dabei pflegt TLC einen gerichteten $\mathcal{G}$ sowie eine Queue $\mathcal{U}$ von Zuständen. Jeder Knoten $s \in \mathcal{G}$ ist ein Zustand und ist erreichbar, d. h.e r konnte durch Ausführen der Spezifikation erreicht werden. $\mathcal{U}$ beinhaltet alle Knoten des Graphen die noch nicht expandiert wurden, also die möglichen Folgezustände noch nicht ermittelt wurden. Der Algorithmus hat folgende Invarianten für $\mathcal{G}$ und $\mathcal{U}$:

- Jeder Zustand in $\mathcal{G}$ erfüllt die Invarianten (*state predicates*) und *state constraint*. *State constraint* ist ein *state predicate*, dass TLC anweist, bestimmte Zustände nicht in den Graphen aufzunehmen.
- Jeder Zustand hat eine Kante zu sich selbst: $\forall s \in \mathcal{G} : (s, s) \in \mathcal{G}$. Dies repräsentiert die *suttering steps*.
- Für jede Transition $(s, t) \in \mathcal{G}$ gilt, dass sie *Next* $\wedge$ *ActionConstraint* einhält. *ActionConstraint* ist ein Prädikat aus der TLC-Configuration, das abhängig vom Zustand, Aktionen deaktivieren kann.
- Jeder Zustand $s \in \mathcal{G}$ ist erreichbar vom Startzustand aus (*Init*).
- $\mathcal{U}$ beinhaltet nur unterschiedliche Zustände aus $\mathcal{G}$.
- Jeder Zustand $s, t \in \mathcal{G} \wedge s \notin \mathcal{U}$ und t erfüllt die Invarianten, existiert eine $(s, t) \in \mathcal{G}$ Kante genau dann, wenn *Next* $\wedge$ *ActionConstraint* dies zulässt.

TLC starten den folgenden Algorithmus mit leeren $\mathcal{G}$ und $\mathcal{U}$:

1. Alle Annahmen (ASSUME) werden anhand der zugewiesenen CONSTANT - Parameters geprüft.
2. Die initialen Zustände werden anhand von *Init* berechnet. Dabei wird für jeden Zustand s überprüft:
   a) ob die Invarianten eingehalten werden; sonst Abbruch.
   b) ob das *state constraint* eingehalten wird.

   alle Zustände s werden in $\mathcal{U}$ übernommen.
3. Solange $\mathcal{U}$ ist nicht leer
   a) Entnehme den ersten Zustand s aus $\mathcal{U}$.
   b) Berechne die Menge T der Nachfolgezustände (s.o.)
   c) Wenn $T = \emptyset$, dann haben wir einen Deadlock. Abhängig von der TLC-Model wird abgebrochen oder fortgefahren.
   d) Für jeden Zustand $t \in T$:
      i. Wenn die Invarianten für t unerfüllt sind, breche ab.
      ii. Wenn *state constraint* erfüllt ist in t und der Schritt $s \rightarrow t$ *ActionConstraint* erfüllt dann,
         A. nehme t und (t, t) in $\mathcal{G}$ und t in $\mathcal{U}$ auf, falls es nicht vorhanden ist.
         B. nehme (s, t) in $\mathcal{G}$ auf.

Periodisch und zur Fertigstellung der Graphegenerierung werden die Temporalen Formeln (Liveness, Fairness, usw.) überprüft. Dafür werden aus dem Graphen $\mathcal{G}$ eine Menge $\mathcal{T}$ von konsistenten Verhalten gebildet. Jeder dieser Verhalten ist ein unendlicher Pfad vom einem Startknoten aus. Anhand dieser Menge verifiziert TLC, ob die Temporalen Aussagen eingehalten werden (vgl. [Lam02] S 242).

Der obige Algorithmus endet nur, wenn der $\mathcal{G}$ endlich ist.

Der Model-Checking Modus geht per Breitensuche vor, während der Simulationsmodus inviduelle Verhalten per Tiefensuche erzeugt und überprüft. Die Suchen werden abgebrochen sobald das Verhalten eine gewisse Länge (Tiefe) errreicht hat (Standardwert: 100).

Dafür wählt der Algorithmus zufällig aus der Menge der Folgezustände $\mathcal{T}$ der Startzustände einen Zustand und expandiert diesen wie im Model-Checking Modus. Sollten Invarianten oder sonstige Fehler bei der Ausführung auftauchen, liefert TLC den Seed und *Ariel* zurück, mit der die Simulation an der Stelle aufsetzen kann.

4.3 Optionen

In Abschnitt 4.2 wurden weitere Prädikate und Einstellungen für TLC eingeführt. In diesem Abschnitt erläutere ich die wichtigsten Optionen von TLC in der TLA$^+$-Toolbox. Die Toolbox legt die entsprechen TLC-Model Dateien innerhalb eines Verzeichnisses unterhalb der Spezifikation ab.

Model Overview

Die wichtigste Option befindent sich auf der *Model Overview* (Abbildung 4.1). Die *Behavior Spec* beschreibt den Programmablauf und kann als Operator in der

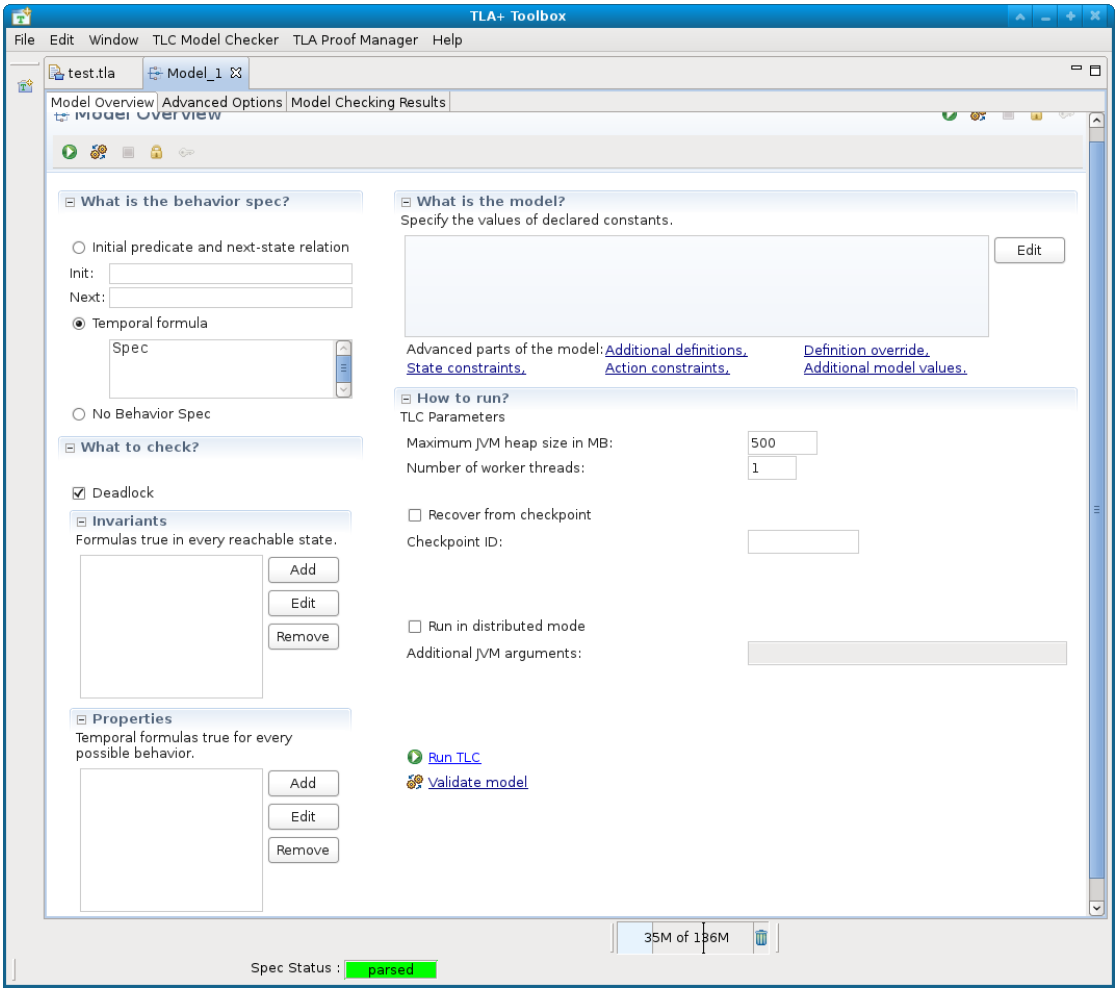

Abbildung 4.1: Übersicht über TLC Konfiguration

TLA$^+$-Normalform erfolgen (3.8) oder als einzelne Angabe des *Init* und *Next*-Operatoren aus der Spezifikationen.

In der nächsten Kategorie *What to check?* können wir Aussagen über Zustände und Verhalten festlegen. Unter anderen ist dort die Deadlock-Erkennung abzuschalten. Ist die Option *Deadlock* aktiviert, so wirft TLC beim Detektieren von Deadlocks im obigen Algorithmus an der Stelle 3c einen Fehler.[3] *Invariants* sind *state predicats*, die für jeden Zustand gelten müssen. Findet TLC einen Zustand, der die Invarianten nicht erfüllt, terminiert TLC mit einem Fehler - selbiges bei den *Properties*. Nur können wir hier LTL-Aussagen auf alle Verhalten vorschreiben. Unter *What is the model?* werden die konstanten Parameter für das Model festlegt.

Weitere Einstellungen zu diversen Prädikaten sind unter *Advanced Options* (Abbildung 4.2) zu sehen. Advanced Options

Über *Additional Definitions* können weitere Operationen und Funktionen in die TLA$^+$-Spezifikation hinzu definiert werden. Mit *Definition Override* können Operatoren aus der Spezifikation überschrieben werden.

[3] Deadlock ist eine *Property* mit ENABLED $\langle Next \rangle_v$

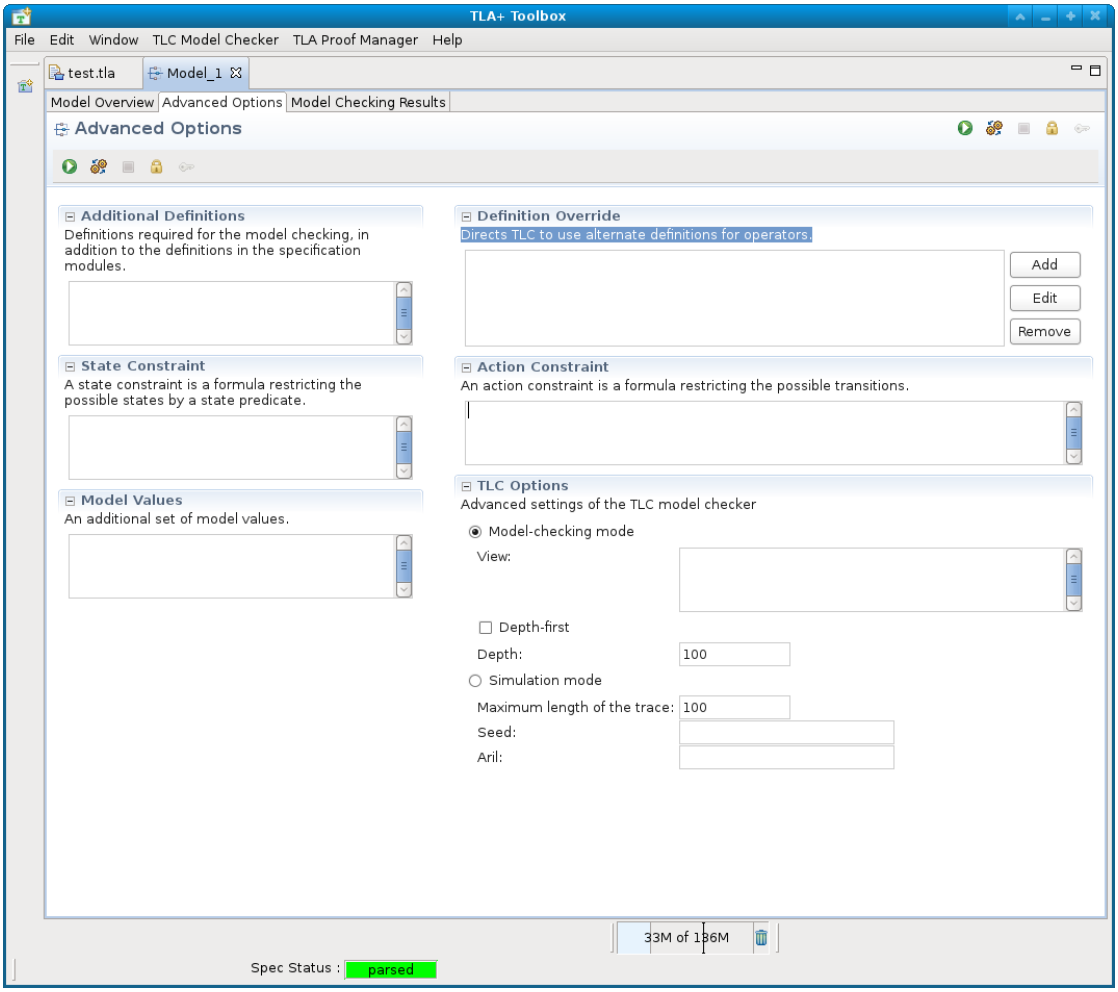

Abbildung 4.2: Erweiterte Optionen von TLC

State Constraint sind *state predicates*, die ein Expandieren von Knoten verbieten. Ähnlich dazu ist *Action Constraint*, damit wird die Ausführung von Aktionen verboten.

Unter *TLC Options* kann der TLC-Modus ausgewählt werden (siehe Abschnitt 4.2). Dabei kann jeweils die maximale Tiefe des Expandierens eingestellt werden. Der Simulation Mode erlaubt die Angabe von *Seed* und *Aril*, um Ausführungen zu wiederholen.

5

Fazit

Was fehlt?

Dieser Ausarbeitung sollte einen kleinen Einblick in die formale Spezifikation unter Verwendung von TLA$^+$ und TLC geben. Viele Themen wurden dabei nicht behandelt:

Echtzeit Mit LTL und TLA$^+$ lassen sich Echtzeit-Anforderungen umsetzen. Dabei schlägt [BK08] die Beschreibung von Echtzeitanfordungen in Abhängigkeiten der Schritte unter Einführung von Operatoren wie $\bigcirc^k$ oder $\Diamond^{\leq k}$. TLA$^+$ führt eine Variable *now* für jeden Zustand mit, der die Zeitpunkt des Zustandes beschreibt. Abhängig von *now* können dann Aussagen getroffen werden, wann Aktionen ausgeführt werden dürfen.

PlusCal Auf TLA$^+$ aufbauend gibt es eine Sprache für Algorithmen. Algorithmen die in PlusCal geschrieben sind, können direkt in eine TLA$^+$-Spezifikation transformiert werden und mittels TLC ausgeführt werden. Die Sprache baut dabei auf der TLA$^+$-Sprache auf und erweitert diese um imperative Anweisungen. Vollständige Vorstellung von PlusCal befindet sich in [Lam11].

TLAPS Neu in Verstion 2 von TLA$^+$ ist ein System zum maschinellen Beweisen von Aussagen über eine Spezifikation.

TLATeX TLATeX übersetzt die in ASCII geschriebene Spezifikation nach LaTeX. So sind die Module 3.1 und 3.3 entstanden.

Module Meine Vorstellung der Zusammensetzung von Modulen ist sehr kurz und bezog sich auf EXTENDS und INSTANCE . Weitere Informationen dazu gibt in Kapitel 10 und 17 aus [Lam02].

Semantik Die Sprache TLA$^+$ ist mit ihrer Syntax und Semantik wohl defniert im Teil IV in [Lam02].

Operatoren In TLA$^+$ gibt es noch weitere Operatoren z. B.$Fxrightarrow[]+G$ und $\exists$. Erstere wird dafür verwendet, um Garantien auszudrücken. $\exists$ wird verwendet, um Variablen nach außenhin zu verbergen und wird nicht von TLC unterstützt. Die Definition der Beiden befindet sich auf S. 316 [Lam02].

Rückblick

Innerhalb dieser Arbeit erläuterte ich Nutzen und Nachteile von formalen Spezifikationen. Im Anschluss stellte ich Grundlagen in Prädikatenlogik und Linearer Temporale Logik vor. Die LTL grenzt sich durch andere Logiken wie der Computation Tree Logik[1] durch eine lineare Abfolge von Zuständen ab. LTL macht aber keine

[1] Für nähere Informationen [BK08] Kapitel 6

Aussagen über die Erstellen von Zuständen. Hier kommt TLA$^+$ zum Einsatz, das eine Spezifizierung von Schritten (Aktionen) zwischen Zuständen erlaubt. Hinzu erlaubt TLA$^+$ noch Modularisierung und Wiederverwendung unser Spezifikationen. Im letzten Abschnitt habe ich die Ausführung von TLA$^+$-Spezifikationen zu einem Graphen von Zuständen erläutert. Aus diesen Graphen können Aussagen der Temporalen Logik bewiesen werden.

Abschluss

TLA$^+$ bietet eine kompakte Sprache zur Spezifikation von Systemen. Dies ersetzt aber nicht die nötige Dokumenation des Systems und Spezifikation. Mathematische Spezifikationen führen zu einer schnellen Konkretisierung von Systemen und lassen Freiheiten für die spätere Implementierung zu. Ob nun eine frühe Konkretisierung des Systems zur Verbesserung der Softwarequalität und Verkürzung der Projektzeiten beiträgt, hängt eher vom verwendeten Entwicklungsprozessmodel und den Erfahrungen des eingesetzten Teams ab.

Während meiner Arbeit TLA$^+$ stellte sich nicht das Spezifizieren eines Systems als Problem heraus, eher die Formulierung von Eigenschaften, die das System einhalten soll.

Literatur

BK08. Baier, Christel und Joost-Pieter Katoen: *Principles of Model Checking*. MIT Press, 2008.

Lam02. Lamport, Leslie: *Specifying Systems*. 1 Auflage, June 2002.

Lam10. Lamport, Leslie: *TLA^{+2}*. October 2010.

Lam11. Lamport, Leslie: *A PlusCal User's Manual*. 1.5 Auflage, April 2011.

Mer. Merz, Stephan: *Modeling and Developing Systems Using TLA+*.