AG Rechnernetze und verteilte Systeme

Kleine Einführung in die Spezifikationssprache TLA+

Version 3.2 by Michael Blume

Stand 20.1.97

Inhaltsverzeichnis

Eine kurze Anmerkung zu der Version dieses Dokuments:
Dieses Dokument trägt den Veränderungen der Syntax und Semantik von TLA+ durch ihren Entwickler Lesslie Lamport Rechnung. Die Beispiele und Erläuterungen sind hinsichtlich der neuen Sepzifikation zur Modulstruktur und der Operatoren in TLA+ überarbeitet worden. So wie auch zuvor, werden nicht alle Möglichkeiten von TLA+ beschrieben da sie nur den einleitenden Charakter unnötig verwischen würden. Es werden z.B. sog. "Assumptions" und Theoreme nur am Rande in Bezug auf das Einbinden von Modulen behandelt. Im Anhang sind Quelle aufgeführt, unter denen der interresierte Leser weitere und vorallem tiefergehende Einblicke in die TLA-Logik und die Sprache TLA+ erhält.

1 Einführung

Die Spezifikation von Algorithmen spielt in der Informatik eine große Rolle. Zum einen liegt das daran, daß eine Spezifikation die gewünschten Eigenschaften des Algorithmus exakt beschreibt, zum anderen kann man auf Grund der Spezifikation den Algorithmus verifizieren d.h. seine Korrektheit beweisen. TLA+ ist eine Spezifikationssprache, die auf der **T**-emporalen **L**-ogik der **A**-ktionen basiert, die von Leslie Lamport entwickelt wurde. Die **TLA**-Logik ist die mathematische Grundlage für TLA+. TLA+ ist genauso mächtig wie TLA und jeder Satz in TLA+ ist in TLA übersetzbar. D.h. eine Verifikation in TLA+ ist auch ein Beweis in der formalen Logik TLA. Für diese Einleitung in TLA+ benötigt man aber keine Kenntnisse über TLA. Der Grund, warum man TLA+ anstelle der TLA benutzt liegt in den Vorteilen, die TLA+ bietet: die TLA+-Schreibweise ist unter anderem wesentlich besser zu lesen und zu schreiben. TLA+ hat auch den Vorteil, daß es eine Sprache ist und damit Syntax und Semantik besitzt. Die Kontrolle der Semantik obliegt wie immer dem "Programmierer" selbst; es ist aber möglich, die syntaktische Korrektheit z.B. mit Hilfe eines Parsers festzustellen. TLA+ bietet die Möglichkeit, die Spezifikation programmiersprachennah zu gestalten, da es ähnlich aufgebaut ist wie **Modula3**. Hierbei darf man aber nicht vergessen, daß es sich immernoch um eine Logik handelt und somit nur Zustände abgefragt werden können! Die größte Stärke von TLA+ liegt aber auf dem Gebiet der Verifikation, denn zum einen basiert sie auf einer Logik, die temporale Abläufe (siehe Abschnitt **4 Temporal- oder das Problem der "Fairness"** Seite 18) berücksichtigt, und zum anderen folgen die Beweise einem sog. "Proofscheme" – was eine Automatisierung ermöglicht. Im Anhang befinden sich Quellenhinweise zum Thema TLA+, in denen man evtl. Antworten auf weitere Fragen finden kann.
Anmerkung:
Die

2 TLA+

TLA+ bietet eine modulare Schreibweise der Spezifikation - ähnlich wie Modula3. Die Modulschreibweise ist der Übersichtlichkeit sehr dienlich. Um die Übersichtlickeit noch zu unterstreichen wurden von **uns** sog. Labels vergeben: **parameters, predicate, action, temporal**. Diese Labels sind nicht Teil von TLA+ und haben in dieser hinsicht auch keine semantische Bedeutung! Gleiches gilt auch für die ein gezeichneten horizontalen Linien in den Modulen. Im Folgenden wird immer von Modulen die Rede sein, diese sind aber als äquivalent zu einer Spezifikation zu verstehen. Um TLA+ zu beschreiben, liegt es nahe, in zwei Schritten vorzugehen. Zuerst wird nun die Struktur innerhalb und im Anschluß die Struktur außerhalb

des Moduls beschrieben. Diese Struktur ist aber nur ein Muster, wie Spezifikationen in TLA+ aussehen können.
Die TLA+ -Spezifikationen wurden in LaTeXgeschrieben. Hinweise, wie man die TLA-Umgebung in LaTeXbenutzt, findet man im Anhang.

2.1 Struktur des Moduls

2.1.1 Beispielmodul

Das Modul *SemaphoreBin* spezifiziert den binären Semaphore. Der Semaphore ist eine Variable, die -weil binär- den Wert 1 oder 0 haben kann. Der Semaphore dient dazu, sog. kritische Bereiche z.B. in einem System zu schützen.
Ein Beispiel:
Ein Drucker und zwei Druckprozesse (A und B). Wenn der erste Prozeß A seine Arbeit beginnt und von dem zweiten Prozeß B unterbrochen wird, so ist das Ergebnis wahrscheinlich wenig brauchbar. Darum wird der kritische Bereich des Druckens mit der Semaphore geschützt. Jeder Prozeß der drucken möchte, **muß** zuvor den Semaphore erniedrigen - was man mit dem Wegnehmen eines Schlüssels vergleichen könnte. Wenn der Prozeß den Druckauftrag erledigt hat, erhöht dieser den Semaphore wieder und der nächste Prozeß könnte nun den "Schlüsssel" nehmen.
Das Prinzip der Semaphore wurde von dem Niederländer E.W. Dijkstra erfunden.
Fassen wir kurz das Semaphorekonzept zusammen:

1. Der Semaphore wirkt wie ein Schlüssel, der das Betreten eines kritischen Systembereiches ermöglicht.

2. Jeder Prozeß muß zuerst den Schlüssel nehmen, d.h. der Semaphore um eins erniedriegen, **bevor** er seine Aktion(en) in dem kritischen Bereich ausführen kann.

3. Nachdem der Prozeß in dem kritischen Bereich seine Aktion(en) beendet hat, verläßt er diesen Bereich **und** legt den Schlüssel wieder zurück, d.h. der Prozeß erhöht der Semaphore wieder um eins.

4. Die Veränderungen der Semaphorevariablen sind immer atomar, d.h. bei Systemen, die nebenläufige Prozesse ermöglichen, wird der Prozeß, der die Semphore verändert, hierbei nicht durch denn Scheduler unterbrochen. Das ist sehr wichtig, denn sonst könnte es vorkommen, daß ein Prozeß den kritischen Bereich verläßt und bevor dieser der Semaphore wieder erhöhen kann vom Scheduler unterbrochen wird, weil z.B. seine Rechenzeit ausgeschöpft ist. Dann würde der kritische Bereich für alle anderen Prozesse nicht mehr betretbar sein. Das wiederum könnte zu einem **DEADLOCK** führen!

Diese Eigenschaften der Semaphore-Spezifikation werden unter **" 2.1.3 temporal"** beschrieben. Das Erhöhen oder Erniedrigen des Semaphore geschieht durch die unter **"2.1.2 actions"** beschriebenen Operationen auf der Variablen *semVar*. Diese Variable kann entweder durch "P" erhöht, oder mit "V" erniedrigt werden. Weil die Variable *semVar* aber niemals negativ sein darf - sonst würde dieses Module kein Semaphore spezifizieren - wird bei "P" abgefragt, ob $semVar > 0$ ist.

module *SemaphoreBin*

parameters

Hier stehen alle VARIABLEN und KONSTANTEN
die in diesem Modul vorkommen :

$semVar$: VARIABLE

Hier steht alles, was fuer die Verknuepfung der Module untereinander von Bedeutung ist. Ein Beispiel folgt spaeter unter dem Punkt: **2.3 Einbinden von Modulen***(S.10) .*

predicate

Hier stehen die Werte, mit welchen die Parameter
initialisiert werden :

$$Init \triangleq semVar = 1$$

actions

Hier stehen die Definitionen :
In diesem Fall sind es Operationen, die auf den deklarierten Variablen ablaufen

$$P \triangleq (semVar > 0) \wedge (semVar' = semVar - 1)$$
$$V \triangleq (semVar = 0) \wedge (semVar' = semVar + 1)$$

temporal

Hier steht die eigentliche Spezifikation des Semaphore − Moduls :

$$spec \triangleq \wedge\ Init \\ \wedge\ \Box[P \vee V]_{semVar}$$

2.1.2 actions

Wie oben zu sehen ist, definieren wir die Operation "P" so, daß sie nur ausgeführt, wird wenn $SemVar > 0$ ist. Die Aktion: ($P \triangleq semVar' = semVar - 1$) bedeutet, daß nach der Ausführung der Operation "P" die Variable $semVar$ dekrementiertiert ist. Also beschreibt $semVar'$ den **nächsten** Zustand der Variable!
Definitionen haben immer die Form: [Name] $\triangleq$ [Bedeutung der Definition].
Die Namen dieser Definitionen sind geschichtlich bedingt, da der Erfinder des Semaphorekonzepts ein Niederländer war:
P : Passeren (übergeben)
V : Vrijgeven (freigeben)
(siehe auch Kap. 2.2.5, Tanenbaum, Literaturangabe Seite 19) Bei aller Nähe zu Modula3 ist immer zu beachten, daß TLA+ eine Logiksprache ist.

2.1.3 temporal

In diesem Abschnitt steht die Spezifikation des Moduls.

$$spec \triangleq \begin{array}{l} \land\ init \\ \land\ \Box[P \lor V]_{semVar} \end{array}$$

Der Ausdruck:

$$\triangleq \Box[Actions]_{Variablen}$$

besagt, daß entweder die definierten Actionen die Variablen verändern oder die Variablen unverändert bleiben. In diesem Fall bedeutet die Spezifikation, daß "init" (die Initialisierung) erfüllt sein muß, und entweder die Operation "P" oder "V" die Variable verändern, oder nichts passiert $\rightarrow$ *semVar* bleibt unverändert. Diese Box definiert die sog. "Safetyconditions", die im Gegensatz zu der Liveliness "leicht" zu beweisen ist. Deshalb bemüht man sich, auch die meisten Elemente der Spezifikation immer durch die "Safetyconditions" zu beschreiben. In dem Abschnitt **temporal** stehen auch die temporalen Bedingungen – die sog. "Liveliness" -Eigenschaften. Diese Bedingungen sind aber für eine Einleitung zu komplex und wurden deshalb weggelassen. Eine kurze Abhandlung über die Grundzüge der temporalen Bedingungen steht auf Seite 18.

2.1.4 Ein weiteres Beispiel

Das Modul "ProcessTemplate" spezifiziert einen Prozeß, bei welchem die Variable "d" inkrementiert wird. Der Name dieses Moduls deutet schon auf eine mehrfache Einbindung in ein weiteres Modul hin. Das Modul hat die Aktion *increment*. Aus dem Ausdruck: d' = d+1 entnimmt man, daß **nach** dem Abschluß von *increment* die Variable in einem neuen Zustand und um eins erhöht ist.

module *ProcessTemplate*

parameters

d : VARIABLE

predicate

$$init \triangleq d = 0$$

actions

$$increment \triangleq d' = d + 1$$

temporal

$$spec \triangleq init \land \Box[increment]_d$$

2.2 Einbinden der Module

In den vorherigen Beispielen wurde gezeigt, wie man ein Modul spezifiziert. Nun folgt ein Beispiel, wie man Module in andere einbinden kann. Hinweise, wie man eine Spezifikation bzw. eine Verifikation in LaTex schreibt, findet man im Anhang.

2.2.1 Beispiel

module *ProcessInc*

parameters

$x, y, sem, pc_1, pc_2 \; : \; \text{VARIABLE}$

$P1 \triangleq$ **instance** *ProcessTemplate* **with** $d \longleftarrow x$
$P2 \triangleq$ **instance** *ProcessTemplate* **with** $d \longleftarrow x$
Semaphore $\triangleq$ **instance** *SemaphoreBin* **with** $SemVar \longleftarrow sem$
extends *GreekeValues*

---

predicates

$$
\begin{array}{lll}
Init & \triangleq & \land\ pc_1 = \alpha \\
 & & \land\ pc_2 = \beta \\
 & & \land\ sem = 1
\end{array}
$$

---

actions

$$
\begin{array}{lll}
increment(x) & \triangleq & \lor \land\ pc_1 = \alpha \\
 & & \quad \land\ Semaphore.P \\
 & & \quad \land\ pc_1{}' = \beta \\
 & & \quad \land\ \text{UNCHANGED } x, y, pc_2 \\
 & & \lor \land\ pc_1 = \beta \\
 & & \quad \land\ P1.increment \\
 & & \quad \land\ pc_1{}' = \gamma \\
 & & \quad \land\ \text{UNCHANGED } y, Sem, pc_2 \\
 & & \lor \land\ pc_1 = \gamma \\
 & & \quad \land\ Semaphore.V \\
 & & \quad \land\ pc' = \alpha \\
 & & \quad \land\ \text{UNCHANGED } x, y, pc_2 \\
increment(y) & \triangleq & \lor \land\ pc_2 = \alpha \\
 & & \quad \land\ Semaphore.P \\
 & & \quad \land\ pc_2{}' = \beta \\
 & & \quad \land\ \text{UNCHANGED } x, y, pc_1 \\
 & & \lor \land\ pc_2 = \beta \\
 & & \quad \land\ P2.increment \\
 & & \quad \land\ pc_2{}' = \gamma \\
 & & \quad \land\ \text{UNCHANGED } x, Sem, pc_1 \\
 & & \lor \land\ pc_2 = \gamma \\
 & & \quad \land\ Semaphore.V \\
 & & \quad \land\ pc_2' = \alpha \\
 & & \quad \land\ \text{UNCHANGED } x, y, pc_1
\end{array}
$$

temporal

$$spec \triangleq init \wedge \Box[increment(x) \vee increment(y)]_{sem, pc_1, pc_2, x, y}$$

2.2.2 Eigenschaft des Moduls: ProcessInc

Wechselseitiger Ausschluß

In dem Modul sind zwei Prozesse möglich – nämlich *increment(x)* und *increment(y)*. Welcher der beiden Prozesse gerade aktiv ist spielt keine Rolle, aber es darf immer nur einer aktiv sein. Diese Bedingung nennt man auch wechselseitiger Auschluß oder "mutual exclusion" und ist ein wichtiger Bestandteil der Betriebssystemtheorien. Diese "mutual exclusion" wird in diesem Beispiel über der Semaphore implementiert: Bevor ein Prozeß seine eigentliche Aufgabe, hier das Inkrementieren einer Variablen, ausführen kann, muß der Semaphore *Sem* erniedrigt werden (siehe auch Abschnit 2.2.1). Nun wird die Variable z.B."x" durch den Ausdruck *P1.increment* erhöht (siehe Abschnitt **Actions**). Danach wird der Semaphore wieder erhöht und ein neuer Prozeß kann beginnen.

Einbinden anderer Module

In diesem Fall werden die Module **SemaphoreBin** und **ProcessInc** eingebunden. Durch diese Einbindung ist es z.B. möglich, daß mit *P1.increment* die Variable x und mit *Semaphore.P* die Variable *Sem* verändert wird. Diese Möglichkeit sollte man übrigens so oft es sinnvoll ist nutzen, weil dadurch die Spezifikation übersichtlicher bleibt. Eine Beschreibung der Methoden zum Einbinden folgt unter: **2.3 Das Einbinden von Modulen** (S. 10).

2.2.3 Parameter

Wir haben hier fünf Variablen. x, y sind die Variablen, die erhöht werden sollen. *Sem* ist ein Aliasname für die Variable *SemVar*. pc_1, pc_2 sind zwei Programmzähler (engl. programcounter), die eingeführt wurden, da TLA+ nur Zustände unterscheiden kann.

2.2.4 Actions

In diesem Modul gibt es zwei Prozesse :

increment(x)

In dieser Operation, wird die Variable x um den Wert 1 erhöht. Besonders zu beachten ist hier die Schreibweise mit dem Programmzähler (pc_1). Dieser wurde eingeführt, weil wir die Operation in drei Zustände unterteilen müssen. Hier zeigt sich der Unterschied zu einer Programmiersprache, denn z.B. in C könnte man einfach eine Kette von Anweisungen schreiben und würde natürlich nach jeder Anweisung einen neuen Zustand des Prozesses erreichen, aber ohne diesen genau zu formulieren. In TLA+ hingegen müssen wir jeden dieser Zustände in Bezug zu den Variablen explizit formulieren bzw. definieren. Wie schon bereits erwähnt, kann eine Logik ja nur Zustände beschreiben und somit kann man in ihr auch nur Zustände verifizieren.

$pc_1 = \alpha$ In diesem Schritt wird durch : *Semaphore.P* die Variable *Sem* die in **SemaphoreBin** spezifiziert ist erniedrigt und der Programmzähler wird um einen Schritt weitergesetzt. Durch das Erniedrigen der globalen Variable *Sem* ist es dem anderen Prozeß increment(y) nicht mehr möglich aktiv zu sein, d.h *der wechselseitige Ausschluß (MUTUAL EXCLUSION) ist erfüllt.* Der Programmzähler wird weitergesetzt.
Wichtig : Alle Variablen, die nicht verändert werden dürfen, müssen als "**UNCHANGED**" deklariert werden!

$pc_1 = \beta$ In diesem Schritt wird die eigentliche Aufgabe der Operation geleistet, nämlich die Erhöhung von x um eins. Dieses geschieht durch:
$increment(x) \triangleq P1.increment.$
Die Operation *increment* ist in dem Modul **ProcessTemplate** spezifiziert. Bei INSTANCE kann eine Angabe über die Herkunft der Operation (eine Referenz) gemacht werden, wie später noch genauer beschrieben wird (siehe S. 11). *P1.increment* ist demnach folgendermaßen zu lesen:

P1.(   ): wir sehen in dem Abschnitt mit INSTANCE, daß dieses eine Aliasdefinition für **ProcessTemplate** ist. - Der andere Name wäre P2. Aus dem Modul sind also zwei " neue" Module entstanden, die auch **getrennt** voneinander zu behandeln sind.

(   ).increment: Die Operation, die unter **P1** $\rightarrow$ **ProcessTemplate** definiert ist, verändert in dem Modul **ProcessInc** demnach die Variable x in der Weise, wie es im Module **ProcessTemplate** für d beschrieben ist. Der Grund, warum hier x und nicht d verändert wird, liegt in der Umbenennung: $x \rightarrow d$.

Also verändert *P1.increment* die Variable in der gewünschten Weise! Am Ende wird der Programmzähler wieder weitergesetzt.

$pc_1 = \gamma$ In diesem Schritt wird der Zyklus beendet. Der Semaphore *Sem* wird wieder erhöht: *Semaphore.V* bedeutet nach Beendigung dieses Zustands hat *Sem* den Wert "1" und ein neuer Zyklus kann beginnen.

increment(y) Dieser Zyklus verhält sich äquivalent zu dem von *increment(x)*, bis auf die Tatsache, daß einige Namen verschieden sind.

2.2.5 Temporal

Die Spezifikation

temporal

$$spec \triangleq init \land \Box[increment(x) \lor increment(y)]_{Sem,pc_1,pc_2,x,y}$$

dieses Moduls besagt, daß :

1. Die Bedingungen der Initialisierung erfüllt sein müssen.
2. Entweder der Prozeß *increment(x)* oder *increment(y)* aktiv ist, oder alle Variablen unverändert bleiben.

2.3 Das Einbinden von Modulen

Wie schon zu Beginn geschrieben, sind die Beispiele in dieser Einleitung bewußt sehr einfach gehalten. Sie enthalten dadurch auch nicht alle Features der TLA+-Sprache. So werden sowohl die Vorbedingungen, bzw. Annahmen in TLA+, die sog. "Assumptions" (Schlsselbegriff:*ASSUME*) und die Theoreme (*THEOREM*) hier nur hinsichtlich der Eigenschaften von *EXTENDS* und *INSTANCE* erwähnt. Der interressierte Leser findet aber im Anhang Quellenhinweise wo weiterführende Literatur zu finden ist.

Die Syntax und Semantik von EXTENDS entspricht im wesentlichen dem frühre geräuchlichen IMPORT. Im Gegensatz dazu ist die jetzt aktuelle INSTANCE Syntax und Semantik sehr von der alten IMPORT Definition abgewichen.

2.3.1 EXTENDS

Syntax : EXTENDS [Name des einzubindenen Moduls, Name des einzub. Moduls,...]

Erläuterung E1: Durch diese Form der Einbindung sind alle Details der Spezifikation, wie z.B. die Variablen und Definitionen in dem "aufrufenden" Modul sichtbar. Im Gegensatz zu INSTANCE sind **immer** alle Elemente des importierten Moduls unter dem gleichen Namen bekannt. Also kann ich ein Modul nur jeweils einmal (sinnvoll !) in ein anderes importieren, da z.B. die Variablen des eingebundenen Moduls nun global bekannt sind.
Auch *Assumptions* und *Theoreme* werden importiert!

Beispiel: Im folgenden Fall binden wir das Modul SemaphoreBin ein.

module *ImpBeispiel*

(...)

extends *SemaphoreBin*

(...)

Ich kann nun die Action *P* bzw. *V* in **ImpBeispiel** benutzen und verändere dabei immer *SemVar*. Würde ich das Module **SemaphoreBin** noch einmal importieren, so würde sich nichts ändern, weil *SemVar* ja schon eine globale Variable im Modul **ImpBeispiel** ist.
Sei z.B. a : VARIABLE im Modul M und a: VARIABLE im Modul L so bedeutet dies für ein Modul B welches EXTENDS M,L beinhaltet, daß hier nun das Symbol "a" zweimal deklariert ist. TLA+ erlaubt eine mehrfache Deklaration oder Definition von Symbolen, sofern diese identisch sind.

Erläuterung E2: Einfach gesprochen ist die Bedeutung des EXTENDS mit einem COPY-Befehl zu vergleichen. In Beiden Fällen werden Daten an eine andere Stelle Kopiert. Bei TLA+ sind dieses in unserem Beispiel nur Spezifikations-, Definitions-, und Deklarationsdaten. Wegen der Gefahr der **Mehrdeutigkeit** sog. it Nameclashes von Symbolen aufgrund der Einbindung von Modulen mit EXTENDS sollte dieses Feature von TLA+ nur soweit eingesetzt werden, wie es der Übersichtlickeit dient. Im Zweifelsfall ist der Gebrauch von INSTANCE vorzuziehen.

Um die Gefahr von Mehrdeutigkeiten zu vermeiden, bzw. um gezielt bestimmte Definitionen oder Deklarationen nur **lokal** in einem Modul zu belassen, kann man das "LOCAL" Keyword benutzen (siehe Lokale Definitionen, S.12).

2.3.2 INSTANCE

Syntax (anhand von Beispielen):

module *Module-B*

instance *ModNameA* **with** $SymbModA_1 \longleftarrow SymbAliasInModB_1$,
$SymbModA_2 \longleftarrow SymbAliasInModB_2$,
...

Erläuterung I1: Durch die oben gezeigte Verknüpfung, werden die Definitionen aus einem anderen Modul in das Modul B eingebunden. Hierbei werden allerdings alle hinter WITH aufgeführten Symbole mit **eindeutigen** Aliasnamen versehen. Symbole, welche keine Aliasbezeichnung erhalten werden unter ihrem alten Namen weitergeführt. Unter dem Begriff *Symbole* werden in TLA+ deklarierte Constanten und Variablen zusammengefaßt. Soll kein Aliasmatching benutzt werden, so kann der WITH-Bereich weggelassen werden! Im Gegensatz zu EXTENDS **müssen** alle Symbole aus den über INSTANCE eingebundenen Modulen (z.B. ModNameA) nochmals deklariert werden. Entweder unter ihrem alten, oder unter ihrem zugewiesenen Aliasnamen!
Die sog. *Assumptions* und *Theoreme* werden nicht weitervererbt!

module *Module-B*

FOO $\triangleq$ **instance** *ModNameA* **with** $SymbModA_1 \longleftarrow SymbAliasInModB_1$,
$SymbModA_2 \longleftarrow SymbAliasInModB_2$,
...

Erläuterung I2: Sollen die Symboldeklarationen und die Definitionen eines Moduls **mehrfach** in ein anderes Modul eingebunden werden, so muß der INSTANCE Verknüpfung eine eindeutige Definition zugeordnet werden! Wichtig ist hierbei, daß es weiterhin keine uneindeutige Symbolmappings geben darf!
Die jeweiligen Definitionen können nun mit dem Definitionslabel zusammen genutzt werden, z.B. FOO.Action-aus-Mod-A .

Beispiel: Im folgenden Fall binden wir das Modul ProcessTemplate zweimal ein.

module *Ausschnitt von ProcessInc*

*P*1 $\triangleq$ **instance** *ProcessTemplate* **with** $d \longleftarrow x$
*P*2 $\triangleq$ **instance** *ProcessTemplate* **with** $d \longleftarrow x$
Semaphore $\triangleq$ **instance** *SemaphoreBin* **with** $SemVar \longleftarrow sem$
extends *GreekeValues*

Zum einen als Definition *P1* und zum anderen als *P2*. Damit ist gewährleistet, daß die beiden Module *P1 und P2* in **ProcessInc** völlig unabhängig voneinander sind! Völlig unabhängig heißt hier, daß man *P1* und *P2* jeweils

als eigenständige Module versteht. Dieses gilt sowohl für Variablen als auch für die Definitionen, z.B. **actions**. Der Ausdruck: d $\longleftarrow$ x bedeutet, daß die Variable *d* aus dem **ProcessTemplate** unter *ProcessInc* nun *x* genannt wird und zu *P1* gehört!
Alle **Variablen** oder **Konstanten**, die **keinen Alias** zugewiesen bekommen behalten ihren alten Namen auch unter ProcessInc ! Im Gegensatz zu EXTENDS **müssen** alle in den Modulen *P1* und *P2* vorkommenden Variablen und Konstanten nochmals in dem Modul *ProcessInc* deklariert werden!

2.3.3 Lokale Definitionen

Es kann durchaus sinnvoll sein, bestimmte Definitionen eines Moduls nicht weiter zu vererben, d.h. lokal zu halten. Die Definitionen in einem Modul werden dafür mit dem Wort LOCAL bezeichnet.
Deklarationen können nicht mit LOCAL geschützt werden. Es wäre sondt möglich Symbole lokal zu halten, während die zugehörige Definition global wäre.

module *Beispiel*

$P1 \;\triangleq\; \textbf{instance}\;\; \mathit{ProcessTemplate}\;\; \textbf{with}\;\; d \longleftarrow x$

$\mathit{LOCAL}\; P2 \;\triangleq\; \textbf{instance}\;\; \mathit{ProcessTemplate}\;\; \textbf{with}\;\; d \longleftarrow x$

$\mathit{LOCAL}\; \mathit{Semaphore} \;\triangleq\; \textbf{instance}\;\; \mathit{SemaphoreBin}\;\; \textbf{with}\;\; \mathit{SemVar} \longleftarrow \mathit{sem}$

$\mathit{LOCAL}\; \mathit{INC}(x) \;\triangleq\; x' = x + 1$

Die erste Definition (P1) ist nicht geschützt und kann somit sowohl durch EXTENDS als auch mit INSTANCE von einem anderen Modul eingebunden werden. Die anderen mit LOCAL bezeichneten Definitionen hingegen **nicht**.
Symboldeklarationen sind nicht durch LOCAL zu schützen! Also z.B. sind keine "LOCAL VarA :VARIABLE" gestattet.
Die Definitionen und Deklarationen welche mittles EXTENDS copiert werden, sind von der LOCAL-Eigenschaft **ausgeschloßen**. Es darf also nicht heißen:

module *!!!!!Fehler Beispiel!!!!*

LOCAL **extends** *SemaphoreBin*

Weitere Beispiele sind unter dem Begriff 2.3.4 Transitivität aufgeführt.

Anmerkung
Unter Berücksichtigung des Einleitungscharakters, gehe hier nicht weiter auf die Syntax und Semantik des sog. "Moduls innerhalb eines Moduls" ein. Es sei hier nur angemerkt, daß es in TLA+ erlaubt ist, ein oder mehrere Module real ineinander zu verschachteln (s.u.). Weitergehende Erläuterungen findet man unter "The Module Structure of TLA+" von Leslie Lamport (siehe Literaturangabe Seite 19).

```
---------------------------- module OuterMod ----------------------------
  VarA : VARIABLE
  ...
----------------------------- module InnMod -----------------------------
VarB : VARIABLE
...
=========================================================================

=========================================================================
```

2.3.4 Transitivität

Unter Transitivität wird hier die Eigenschaft der Weitervererbung von Definitionen und Deklarationen bei der Verknüpfung der TLA+ - Modulen beschrieben. Die Weitervererbung von "Assumptions" und "Theoremen" wird hier nicht behandelt. Es sei dazu auf die Beschreibung "The Module Structure of TLA+" von L. Lamport hingewiesen (siehe Literaturangabe Seite 19) .

MODULE A
varA: VARIABLE
DefA $\triangleq \ldots$

MODULE B
varB: VARIABLE
EXTENDS A
DefB $\triangleq \ldots$

Kennt varA und DefA

MODULE C
varC, varA: VARIABLE
INSTANCE A
DefC $\triangleq \ldots$

Kennt varA und A.DefA

MODULE D
varD: VARIABLE
EXTENDS B
DefD $\triangleq \ldots$

Kennt varA, varB,
DefA und DefB

MODULE E
varE, varA,
varB: VARIABLE
INSTANCE B
DefE $\triangleq \ldots$

Kennt varA,varB
DefA und DefB

MODULE F
varF: VARIABLE
EXTENDS C
DefF $\triangleq \ldots$

Kennt varA,varC,
A.DefC und DefC

MODULE G
varG, varA,
varC: VARIABLE
INSTANCE C
DefG $\triangleq \ldots$

Kennt varA,varC
A.DefC und DefC

Erläuterung T1:
Die Skizze zeigt deutlich den wesentlichen Unterschied zwischen EXTEND und INSTANCE hinsichtlich der Deklaration der Symbole. Es zeigt sich auch, das ein Verschachtelungstiefe von mehr als "2" sehr unübersichtlich wird (besonders wenn die einzelnen Module nicht mehr auf einer Seite stehen). Dies gillt besonders für die Verwendung von EXTENDS. INSTANCE bietet hier den Vorteil, daß die Symbole nochmals deklariert werden müssen. Ist eine größere Verschachtelungstiefe unausweichlig, so empfhielt es sich das INSTANCE wie im nächsten Beispiel einzusetzen.

MODULE A
varA: VARIABLE
DefA $\triangleq \dots$

MODULE B
varB, varA: VARIABLE
LOCAL DeflocalB
LA $\triangleq$ **INSTANCE A**
DefB $\triangleq \dots$

Kennt varA,varB und LA.DefA

MODULE C
varC, varA,
varB: VARIABLE
LC $\triangleq$**INSTANCE B**
DefC $\triangleq \dots$

Kennt varA,varB,varC
LC.LA.DefA und LC.DefB
NICHT DeflocalB

Erläuterung T2:

Durch die Vergabe von Definitionen an die INSTANCE-Module, ist es nicht nur möglich ein Modul mehrmals (sinnvoll!) einzubinden, sondern es kann auch zur Steigerung der Übersichtlichkeit einer Spezifikation dienen!
Die LOCALE Definition "DeflocalB" ist für alle anderen Module nicht sichtbar, d.h. sie kann weder von EXTENDS noch von INSTANCE eingebunden werden (siehe dazu auch 2.3.3 Lokale Definitionen, S. 12)!

3 Résumé

In den vorhergehenden Abschnitten wurden viele Eigenschaften von TLA+ mehr oder weniger ausführlich vorgestellt. Fassen wir nun ein paar wichtige Punkte noch einmal gekürzt zusammen:

3.1 Form eines Moduls

Die Struktur eines TLA+ - Modules kann man zur Übersicht in die wesentlichen fünf Abschnitte aufspalten. Es ist aber hierbei wichtig zu wissen, daß dies nicht speziell von TLA+ vorgegeben ist!:

1. Parameter (Variablen und Konstanten)
2. evtl. Einbindungen von weiteren Modulen
3. Predicate (Initialisierung)
4. Definitionen: z.B. Actions (Operationen)
5. Temporal (eigentl. Spezifikation)

module *Die Struktur eines Moduls*

parameters

Deklaration der Variablen und Konstanten

---

extends
instance
Einbindung anderer Module

---

predicates

Initialisierung der Variablen,
ggf. Belegung von Konstanten

---

definitions

Definition von Mengen, Funktionen etc.

---

actions

Definition der Operationen

---

temporal

die eigentliche Spezifikation

Wie überall gibt es natürlich auch von dieser Form Abweichungen, die durch die jeweilige Aufgabe der Spezifikation verursacht werden.

3.2 Eigenschaften von EXTENDS und INSTANCE

3.2.1 EXTENDS

- Bei EXTENDS sind alle Details der Spezifikation des eingebundenen Moduls in dem nächst einbindenen Modul sichtbar. Die so vererbten Variablen und Konstanten müssen nicht zusätzlich im Modul deklariert werden.
- Bei Benutzung der importierten Spezifikation (z.B. einer *action*) müssen keine weiteren Angaben (Referenzen) über die Herkunft (z.B. der *action*) gemacht werden.
- EXTENDS kann nicht in Verbindung mit LOCAL genutzt werden. Also bleiben alle die so "copierten" Definitionen und Deklarationen ohne Einschränkung weitervererbbar.
- Die "Assumptions" und Theoreme werden weitervererbt.

3.2.2 INSTANCE

- Durch diese Form des Einbindens kann man ein Modul mehrmals sinnvoll mit einem anderen Modul verknüpfen.
- Das so eingebundene Modul kann mit einem Aliasnamen definiert werden, unter dem dann allle Definitionen aus diesem angesprochen werden können.
- Alle Deklarierten Symbole die keine Aliasbezeichnung erhalten werden unter ihren Originalnamen weitervererbt. Alle Variablen und Konstanten, auch die vererbten, müssen im Modul deklariert werden.
- Bei Benutzung einer Aliasdefinition in Bezug auf ein Modul wird bei der Benutzung einer Definition aus diesem immer der Aliasname vorgehängt.
- Durch LOCAL können die eingebundenen Definitionen aus einem anderen Modul vor weiteren Verërbungen geschützt werden, d.h. sie ist lokal.
- Deklariert ein Modul keine Symbole, so kann die Verknüpfung durch EXTENDS mit Hilfe von INSTANCE ersetzt werden - (ermöglicht wieder eine evtl. gewünschte "lokale" Definition!).
- "Assumptions" und Theoreme werden nicht weitervererbt.

Mit dieser Zusammenfassung endet die Einführung in TLA+. Wer noch weitere Fragen zu diesem Thema hat, findet im Anhang Seite 19 Hinweise auf Quellen über TLA, bzw. TLA+.

4 Temporal - oder das Problem der "Fairness"

Den schon in der Einleitung erwähnte Bereich der "zeitabhängigen" Logik beschreibt man unter TLA+ mit dem Begriff *FAIRNESS*. Dieser Begriff steht für das Prinzip des Möglichen. Da wir uns in einer Logik befinden bedeutet das in diesem Fall etwas einfach ausgedrückt:

> **Alle Zustände, die erreichbar sind, werden auch irgendwann eintreten.**

Diese Bedingung ist natürlich sehr schwer zu beweisen, und deshalb versucht man auch so viele Teile des Problems wie möglich durch die statischen Logikbereiche von TLA+ zu spezifizieren und zu verifizieren. Diese Bereiche haben wir unter "**2.1.3 Temporal"** kennengelernt. Dieses ist auch der Grund, warum wir uns nur am Rande mit dem Problem der Fairness beschäftigen - es ist für eine Einleitung zu schwer. Um zumindest einen Eindruck von der komplexen Materie zu bekommen, folgen nun zwei Beispiele.
Es gibt viele Abstufungen von "Fairness" und wir stellen nur exemplarisch zwei Arten vor:

WEAK FAIRNESS Gegeben sei eine Ampel mit den zwei möglichen Schaltzuständen "Rot" und "Grün". Die Abfolge der Zustände sei beliebig. Dann bedeutet eine Weak Fairness - Aussage, über daß Auto, Das es fahren wird, wenn die Ampel den Zustand Grün annimmt und behält. Diese Aussage betrifft **nur** das Auto und hat keine Konsequenzen für die Ampel!

STRONG FAIRNESS In diesem Beispiel bedeutet die "Strong Fairness" folgendes: Das Auto wird fahren, wenn die Ampel unendlich oft den Zustand "Grün" annimmt. Die Strong Fairness macht keine Aussage darüber was mit dem Auto passiert, wenn die Ampel immer Rot anzeigt. (Hier muß man wieder besonders beachten, daß es sich um eine abstrakte Logik handelt)

Es ist ersichtlich,**STRONG FAIRNESS** $\Rightarrow$ **WEAK FAIRNESS**. An diesem Beispiel wird erkennbar, wie schwer der Beweis solcher Bedingungen ist. Dabei spielt die "Fairness" bei der Deadlockerkennung eine große Rolle.

5 Anhang

Zunächst möchte ich noch all denen danken, die mich bei der Verfassung dieser Einleitung unterstützt haben. Ohne sie wäre dieses Dokument weder grammatisch noch inhaltlich so korrekt wie es jetzt ist. Besonderen Dank gilt Dirk Henkel und Lutz Sommerfeld, mit deren Hilfe wesentliche Verbesserungen dieses Skriptes erfolgten.

5.1 Quellenhinweise

- Auf der Homepage von Prof. Peter Ladkin.
  http://www.rvs.uni-bielefeld.de/~ladkin/
  finden sich diverse Unterpunkte:
  - "Formal but Lively Buffers in TLA+"
  - "Analysis of a Technical Description of the Airbus A320 Braking System"
- WWW-Page zum Thema TLA+
  http://www.research.digital.com/SRC/tla/
  Diese Seite ist auch über die Homepage von P. Ladkin und Michael Blume erreichbar.
  - "Introduction to TLA" by Leslie Lamport
  - "The Module Structure of TLA+" by Leslie Lamport
  - "The Operators of TLA+" by Leslie Lamport
- Meine eigene Homepage:
  http://www.rvs.uni-bielefeld.de/~mblume/
- Die Tutoren-Homepage zur Vorlesung: Technische Informatik I
  http://www.rvs.uni-bielefeld.de/lecture/Courses/
- Die Homepage von Dirk Henkel
  http://www.rvs.uni-bielefeld.de/~dhenkel
- Tanebaum: Moderne Betriebssysteme
  http://www.cs.vu.nl/ ~ast/

Eine Frage habe wurde bis jetzt offen gelassen, nämlich wie man unter LaTex die Spezifikationen bzw. Verifikationen auf "Papier" bringt. Zu diesem Zweck hat Leslie Lamport einige Styles geschrieben. Auf der Webseite von Lesslie Lamport und auch auf der vom Michael Blume befinden sich die folgenden Files: *tla.sty*, *abbrev.sty*, *pf.sty* und *l*atexsym.sty.
Dieses Stylefiles, können frei kopiert und in das eigene Tex-Verzeichniß gestellt werden, wodurch dem Benutzer eine von Lesly Lamport entwickelte "TLA+ -Umgebung" unter Latex zur Verfügung steht. Es folgen noch zwei Beispiele für die LaTex-Preamble (auch nachzulesen auf der Seite von Dirk Henkel.):

LaTex-Version 2 (alt)	**LaTex-Version 2e (neu)**
`\documentstyle[tla,a4]{article}`	`\documentclass[a4paper]{article}` `\usepackage{latexsym,tla,pf}`

Allgemeine Hinweise zu oft gestellten Fragen in Bezug zu Anwendungsprogrammen enthält folgende Seite: http://www.rvs.uni-bielefeld.de/ mblume/faq.html