Goethe-Universität Frankfurt am Main
Fachbereich 12 - Informatik und Mathematik
Institut für Informatik

Automatische Überprüfung von Übersetzungen von synchroner in asynchrone Kommunikation bei blockierenden Speicheroperationen

Erstgutachter: Prof. Dr. Manfred Schmidt-Schauß

Modul M-MSC: Masterarbeit
Studiengang: M.Sc. Informatik
Wintersemester 2021
Semesterzahl: 6

Abgabetermin: 08.11.2021

Von

Simon Rudel
Matrikelnummer: 5799950
Email: s8486068@stud.uni-frankfurt.de

Frankfurt am Main, den 31.10.2021

Erklärung zur Abschlussarbeit

gemäß § 34, Abs. 16 der Ordnung für den Masterstudiengang Informatik vom 17. Juni 2019.

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Ebenso bestätige ich, dass diese Arbeit nicht, auch nicht auszugsweise, für eine andere Prüfung oder Studienleistung verwendet wurde.

Frankfurt am Main, den 31.10.2021

Ort, Datum

Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Zusammenfassung

In dieser Arbeit geht es darum, die relative Ausdrucksstärke zweier Ansätze zur Umsetzung von Nebenläufigkeit in Programmiersprachen zu vergleichen. Dazu werden Rechenprozesse betrachtet, welche die Kommunikation zwischen parallelen Prozessen modellieren.
Der erste Ansatz beruht auf einer synchronen Kommunikation zwischen den parallel laufenden Prozessen, während der zweite Ansatz eine asynchrone Kommunikation verwendet, welche durch einen gemeinsam genutzten Speicher ermöglicht wird.
Es werden Übersetzungen von Rechenprozessen mit synchroner Kommunikation in Rechenprozesse mit asynchroner Kommunikation analysiert. Dazu werden die zwei Sprachen *SYNCSIMPLE* und *LOCKSIMPLE* verwendet, bei denen es sich jeweils um ein minimalistisches Modell für synchrone Kommunikation (*SYNCSIMPLE*) bzw. asynchrone Kommunikation (*LOCKSIMPLE*) handelt.
Das Kriterium für die Korrektheit von Übersetzungen von *SYNCSIMPLE* in *LOCKSIMPLE* ist, dass übersetzte Rechenprozesse in *LOCKSIMPLE* das gleiche Konvergenzverhalten wie die ursprünglichen Rechenprozesse in *SYNCSIMPLE* besitzen.
Zur Analyse wird ein Programm entwickelt, welches Übersetzungen automatisch auf Rechenprozesse anwenden und die Übersetzungen in der Folge überprüfen kann. Mit Hilfe dieses Programms werden anschließend Übersetzungen mit bestimmten Strukturen in einigen Testläufen widerlegt.

1. Einleitung

Nebenläufigkeit spielt eine große Rolle in der Informatik, da durch sie ermöglicht wird, dass Rechenprozesse effizient durchgeführt werden können.
Schmidt-Schauß und Sabel haben die relative Ausdrucksstärke zweier Ansätze zur Umsetzung von Nebenläufigkeit in Programmiersprachen in einer gemeinsamen Arbeit [1] miteinander verglichen.
Der erste dieser Ansätze besteht in dem Austausch von Nachrichten zwischen parallel laufenden Prozessen. Die Prozesse senden und empfangen dabei Nachrichten von anderen Prozessen, um sich miteinander zu synchronisieren. Die an der Kommunikation beteiligten Prozesse müssen dabei immer warten, bis die Kommunikation abgeschlossen ist. Es handelt sich hierbei also um eine synchrone Kommunikation.
Der andere Ansatz besteht darin, dass ein gemeinsamer Speicher verwendet wird, mit dessen Hilfe parallele Prozesse auf asynchrone Art miteinander kommunizieren können. Das grundlegende Prinzip besteht hierbei darin, dass die Prozesse den Zustand einzelner Speicherzellen durch bestimmte Operationen ändern können. Darüber hinaus sind bestimmte Operationen eines Prozesses abhängig von dem Zustand einzelner Speicherzellen, d.h. je nach Zustand der Speicherzellen können bestimmte Operationen nicht ausgeführt werden. Somit kann ein Prozess blockiert werden, bis ein bestimmtes Ereignis in einem anderen Prozess den Zustand der entsprechenden Speicherzelle ändert.
In der angesprochenen Arbeit wurde die relative Ausdrucksstärke dieser beiden Ansätze miteinander verglichen, indem Rechenprozesse betrachtet wurden, welche die Kommunikation zwischen parallel laufenden Prozessen modellieren. Es wurden zwei vereinfachte abstrakte Modelle eingeführt, um Rechenprozesse unter Verwendung der beiden Kommunikationsmethoden zu beschreiben. Zur Beschreibung von Rechenprozessen, welche den Ansatz der synchronen Kommunikation verwenden, wurde die Sprache *SYNCSIMPLE* definiert. Für Rechenprozesse, die mittels eines gemeinsamen Speichers kommunizieren, wurde die Sprache LOCKSIMPLE eingeführt. Diese verwendet Speicherzellen, welche entweder leer oder voll sein können. Mit Hilfe dieser Speicherzellen kann schließlich die Ausführung der nebenläufigen Prozesse kontrolliert werden. Innerhalb der Prozesse gibt es dazu die zwei Speicheroperationen *Put* und *Take*, welche eine Speicherzelle füllen bzw. leeren können. Je nach Zustand der entsprechenden Speicherzelle können diese Operationen innerhalb eines Prozesses blockieren. Im Rahmen ihrer Arbeit haben Schmidt-Schauß und Sabel die Fälle betrachtet, in denen nur eine der Operationen blockiert. Das heißt, entweder hat *Put* blockiert, wenn die entsprechende Speicherzelle bereits voll war, oder *Take* hat blockiert, wenn die zugehörige Speicherzelle leer war.
Um die Ausdrucksstärke der beiden Ansätze zu vergleichen, wurde analysiert, ob es möglich ist, korrekte Übersetzungen von Rechenprozessen in *SYNCSIMPLE* in Rechenprozesse in *LOCKSIMPLE* zu konstruieren. Als Kriterium für korrekte Übersetzungen galt dabei, dass das Konvergenzverhalten von jedem Rechenprozesses in *SYNCSIMPLE* auch in dem zugehörigen übersetzten Rechenprozess in *LOCKSIMPLE* erhalten bleibt. Das bedeutet z.B., dass ein *SYNCSIMPLE*-Prozess, der in jedem Fall erfolgreich terminiert, auch nach der Übersetzung in einen *LOCKSIMPLE*-Prozess in jedem Fall erfolgreich terminieren muss. Zur Beschreibung des Konvergenzverhaltens wurden unter anderem die Eigenschaften may-convergent und must-convergent verwendet. May-convergent gibt dabei an, dass ein Prozess auf mindestens eine Art erfolgreich terminieren kann, während ein Prozess, der must-convergent ist, in jedem Fall erfolgreich terminieren muss.
Es wurde gezeigt, dass man in in *LOCKSIMPLE* mindestens drei der genannten Spei-

cherzellen benötigt, um eine korrekte Übersetzung zu erhalten, falls eine der genannten Speicheroperationen pro Speicherzelle blockierend ist. Offen blieb allerdings die Frage, ob möglicherweise zwei Speicherzellen ausreichen, wenn jeweils beide Speicheroperationen blockierend sind.
Das Ziel dieser Arbeit ist es, ein Programm zu entwickeln, welches Übersetzungen von der Sprache *SYNCSIMPLE* in die Sprache *LOCKSIMPLE* automatisch überprüft, wobei beide Speicheroperationen blockierend sind. Damit soll ermöglicht werden, Übersetzungen gezielt zu widerlegen und nach möglicherweise korrekten Übersetzungen zu suchen.
Dazu werden in Kapitel 2 zunächst die beiden Sprachen *SYNCSIMPLE* und *LOCKSIMPLE* erläutert. Außerdem wird beschrieben, wie eine Übersetzung definiert ist und welche Anforderungen an eine Übersetzung gestellt werden, damit sie als korrekt bezeichnet werden kann.
In Kapitel 3 wird dieses Programm, welches in der funktionalen Programmiersprache Haskell geschrieben wurde, dargestellt und erläutert.
Anschließend wird es in Kapitel 4 verwendet, um mit Hilfe von Tests einige grundlegende Bedingungen zu ermitteln, welche korrekte Übersetzungen erfüllen müssen.

2. Die Sprachen für nebenläufige Prozesse

Im folgenden Kapitel werden die vereinfachten Modelle für nebenläufige Prozesse mit synchroner und asynchroner Kommunikation beschrieben. Die Sprache *SYNCSIMPLE* dient zur Beschreibung von nebenläufigen Prozessen mit synchroner Kommunikation. Die Sprache *LOCKSIMPLE* modelliert nebenläufige Prozesse mit asynchroner Kommunikation. In diesem Fall läuft die Kommunikation zwischen den Subprozessen eines Rechenprozesses mit Hilfe eines gemeinsamen Speichers ab. Die in diesem Kapitel beschriebenen Definitionen stammen aus der Arbeit von Schmidt-Schauß und Sabel [1] und wurden zum größten Teil daraus übernommen. Nur die Sprache *LOCKSIMPLE* wurde geringfügig angepasst, sodass beide Speicheroperationen blockierend sind und es genau zwei Speicherzellen gibt. Außerdem wurde in Kapitel 2.2 und Kapitel 2.3 jeweils ein Beispiel hinzugefügt.

2.1. Die Sprache SYNCSIMPLE

Die Sprache *SYNCSIMPLE* dient zur vereinfachten Darstellung von nebenläufigen Prozessen mit synchroner Kommunikation. Zur Kommunikation findet ein Nachrichtenaustausch zwischen den parallel laufenden Subprozessen statt, sodass immer zwei Subprozesse an der Kommunikation beteiligt sind. Einer der Subprozesse sendet dabei eine Nachricht, während der andere Subprozess die Nachricht empfangen muss. Erst wenn die Kommunikation abgeschlossen ist, können beide Subprozesse weiterlaufen. In *SYNCSIMPLE* werden die zwei Symbole ? und ! verwendet, um das Senden (!) und das Empfangen (?) von Nachrichten darzustellen. In einem Kommunikationsschritt muss dementsprechend jeweils ein ! eines Subprozesses zusammen mit einem ? eines anderen Subprozess ausgeführt werden.

Definition 2.1 *Die Syntax von Prozessen und Subprozessen in SYNCSIMPLE wird durch folgende Grammatik beschrieben:*

$$\textit{Subprozess } U = 1 \mid 0 \mid !U \mid ?U$$
$$\textit{Prozess } P = U \mid U \parallel P$$

Die einzelnen Symbole haben folgende Bedeutungen:

- 0 bedeutet, der Subprozess wird im weiteren Verlauf nicht mehr kommunizieren. Der Prozess ist allerdings noch nicht erfolgreich abgeschlossen.
- 1 bedeutet, der Prozess ist erfolgreich.
- ! steht für einen Output (das Senden einer Nachricht) eines Subprozesses
- ? steht für einen Input (das Empfangen einer Nachricht) innerhalb eines Subprozesses
- $\parallel$ beschreibt die parallele Verkettung mehrerer Subprozesse. Sie ist kommutativ und assoziativ. Außerdem fungiert das Symbol 0 als Einheitselement von $\parallel$, d.h. $0 \parallel P = P$ für alle P.

Ein Prozess ist dementsprechend ein Multiset aus Subprozessen.

Definition 2.2 *Die operationale Semantik der Prozesse besteht aus nicht-deterministischen Kommunikationsschritten $\xrightarrow{SYS}$, welche folgendermaßen definiert sind:*

$$!U_1 \parallel ?U_2 \parallel P \xrightarrow{SYS} U_1 \parallel U_2 \parallel P$$

U_1 und U_2 sind dabei zwei beliebige Subprozesse und P ist ein beliebiger Prozess.
Die reflexiv-transitive Hülle von $\xrightarrow{SYS}$ wird im weiteren Verlauf mit $\xrightarrow{SYS,*}$ dargestellt.
Eine Sequenz aus Kommunikationsschritten $\xrightarrow{SYS}$ wird Ausführung von P genannt. Es kann mehrere Ausführungen zu einem Prozess geben. Alle Ausführungen terminieren, allerdings können sie zu unterschiedlichen Ergebnissen führen.
Ein Prozess ist erfolgreich, falls es in P einen Subprozess U gibt mit U = 1, also wenn gilt $P = P' \parallel 1$ für ein beliebiges P'.

Beispiel 2.1 *Der Prozess $P = \ ?!0 \parallel !!1 \parallel ?0$ kann auf zwei Arten ausgeführt werden:*

$$\begin{aligned}
1.\ ?!0 \parallel\ !!1 \parallel\ ?0 &\xrightarrow{SYS}\ !0 \parallel\ !1 \parallel\ ?0 \\
&\xrightarrow{SYS}\ !0 \parallel\ 1 \parallel\ 0 \\
2.\ ?!0 \parallel\ !!1 \parallel\ ?0 &\xrightarrow{SYS}\ !0 \parallel\ !1 \parallel\ ?0 \\
&\xrightarrow{SYS}\ 0 \parallel\ !1 \parallel\ 0
\end{aligned}$$

Im ersten Fall ist der Prozess erfolgreich beendet. Im zweiten Fall ist kein weiterer Kommunikationsschritt mehr möglich und der Prozess konnte nicht erfolgreich beendet werden.

Definition 2.3 *Ein Prozess P gilt als*

- *may-convergent, falls es einen erfolgreichen Prozess P' gibt, sodass $P \xrightarrow{SYS,*} P'$.*
  D.h. es gibt mindestens eine Ausführung, die zu einem erfolgreichen Prozess führt
- *must-convergent, falls alle Prozesse P', für die gilt $P \xrightarrow{SYS,*} P'$, may-convergent sind.*
  D.h. alle möglichen Ausführungen von P führen zu einem erfolgreichen Prozess
- *must-divergent, falls es keine mögliche Ausführung gibt, die zu einem erfolgreichen Prozess führt*
- *may-divergent, falls es mindestens eine mögliche Ausführung $P \xrightarrow{SYS,*} P'$ gibt, sodass P' must-divergent ist.*

Dementsprechend ist jeder Prozess, der must-convergent ist, auch may-convergent und jeder must-divergent Prozess ist auch may-divergent. Außerdem kann ein Prozess nicht gleichzeitig must-convergent und may-divergent oder may-convergent und must-divergent sein.

2.2. Die Sprache LOCKSIMPLE

Die Sprache *LOCKSIMPLE* dient zur vereinfachten Darstellung von nebenläufigen Prozessen mit asynchroner Kommunikation. Sie ist ähnlich aufgebaut wie die Sprache *SYNCSIMPLE*. Allerdings wird der Austausch von Nachrichten hier entfernt und durch die Speicheroperationen *Put* und *Take* ersetzt. Diese Operationen wirken auf einen gemeinsam genutzten Speicher mit zwei Speicherzellen c_1, c_2. Eine Speicherzelle c_i kann leer $(c_i = 0)$ oder voll $(c_i = 1)$ sein. Der Befehl *Put* wird zum Füllen einer Speicherzelle verwendet, während ein *Take* eine Speicherzelle leert. Der Startzustand des Speichers wird durch ein Tupel (c_1, c_2) beschrieben, wobei $c_1, c_2 \in \{0, 1\}$. Durch die Schreibweise *LOCKSIMPLE*$_{IS}$ wird die Sprache mit initialem Speicher IS beschrieben.

Definition 2.4 *Die Syntax von Prozessen und Subprozessen in* $LOCKSIMPLE_{IS}$ *wird durch folgende Grammatik beschrieben:*

$$\begin{aligned} \textit{Subprozess } \mathcal{U} &= 1 \mid 0 \mid P_1\mathcal{U} \mid P_2\mathcal{U} \mid T_1\mathcal{U} \mid T_2\mathcal{U} \\ \textit{Prozess } \mathcal{P} &= \mathcal{U} \mid \mathcal{U} \parallel \mathcal{P} \end{aligned}$$

Die einzelnen Symbole haben folgende Bedeutung:

- 0 bedeutet, der Subprozess wird im weiteren Verlauf nicht mehr kommunizieren. Der Prozess ist allerdings noch nicht erfolgreich abgeschlossen.
- 1 bedeutet, der Prozess ist erfolgreich.
- P_i steht für *Put* und gibt an, dass der Subprozess Speicherzelle i füllt, für $i \in \{1, 2\}$
- T_i steht für *Take* und gibt an, dass der Subprozess Speicherzelle i leert, für $i \in \{1, 2\}$
- $\parallel$ steht für die parallele Verkettung von Subprozessen. Diese ist kommutativ und assoziativ.

Ein Prozess in *LOCKSIMPLE*$_{IS}$ ist ein Multiset aus Subprozessen.

Definition 2.5 *Die Ausführung von Prozessen in LOCKSIMPLE*$_{IS}$ *verändert den Zustand der Speicherzellen. Dementsprechend benötigt man bei der Betrachtung der Ausführung neben dem Prozess in LOCKSIMPLE*$_{IS}$ *den aktuellen Zustand des Speichers. Dieser wird als* $\mathcal{C}$ *bezeichnet und in Form eines Tupel beschrieben:*

$$\mathcal{C} = (c_1, c_2) \textit{ , wobei } c_1, c_2 \in \{0, 1\}$$

Mit der Schreibweise $\mathcal{C}[c_i = 0]$ *bzw.* $\mathcal{C}[c_i = 1]$ *wird angegeben, dass Speicherzelle i in dem Speicherzustand* $\mathcal{C}$ *leer bzw. voll ist.*

Definition 2.6 *Im Folgenden wird ein Prozess $\mathcal{P}$ in LOCKSIMPLE$_{IS}$ zusammen mit dem aktuellen Speicherzustand $\mathcal{C}$ in Form eines Tupels $(\mathcal{P}, \mathcal{C})$ geschrieben, um den aktuellen Zustand der Ausführung eines Prozesses zu beschreiben. Die operationale Semantik der Prozesse besteht aus nicht-deterministischen Reduktionsschritten $\xrightarrow{LS}$, welche immer auf solch einen Zustand wirken. Ein Reduktionsschritt $\xrightarrow{LS}$ wird durch die beiden folgenden Regeln definiert:*

$$(P_i\mathcal{U} \parallel \mathcal{P}\text{ , }\mathcal{C}[c_i = 0]) \xrightarrow{LS} (\mathcal{U} \parallel \mathcal{P}\text{ , }\mathcal{C}[c_i = 1])$$
$$(T_i\mathcal{U} \parallel \mathcal{P}\text{ , }\mathcal{C}[c_i = 1]) \xrightarrow{LS} (\mathcal{U} \parallel \mathcal{P}\text{ , }\mathcal{C}[c_i = 0])$$

Dabei ist $\mathcal{U}$ ein beliebiger Subprozess und $\mathcal{P}$ ein beliebiger Prozess. Eine Sequenz aus den oben beschriebenen Reduktionsschritten ergibt eine Ausführung. Die Ausführung eines Prozesses ist an den Zustand des Speichers gebunden und kann gegebenenfalls blockieren. Dies geschieht genau dann, wenn ein Prozess Speicherzelle i füllen möchte und diese aber bereits voll ist oder wenn ein Prozess Speicherzelle i leeren möchte, aber die Speicherzelle leer ist. Die Reduktion eines Prozesses $\mathcal{P}$ in Locksimple$_{IS}$ beginnt in dem Zustand ($\mathcal{P}$,IS). Die reflexiv-transitive Hülle von $\xrightarrow{LS}$ wird mit $\xrightarrow{LS,*}$ dargestellt.

Definition 2.7 *Ein Prozess $\mathcal{P}$ in LOCKSIMPLE$_{IS}$ gilt als erfolgreich, falls es in $\mathcal{P}$ einen Subprozess $\mathcal{U}$ gibt mit $\mathcal{U} = 1$. Ein Zustand $(\mathcal{P}, \mathcal{C})$ gilt als*

- *erfolgreich, falls $\mathcal{P}$ erfolgreich ist.*
- *may-convergent, falls es einen erfolgreichen Zustand $(\mathcal{P}', \mathcal{C}')$ gibt, mit $(\mathcal{P}, \mathcal{C}) \xrightarrow{LS,*} (\mathcal{P}', \mathcal{C}')$*
- *must-convergent, falls alle Zustände $(\mathcal{P}', \mathcal{C}')$, für die gilt $(\mathcal{P}, \mathcal{C}) \xrightarrow{LS,*} (\mathcal{P}', \mathcal{C}')$, may-convergent sind*
- *must-divergent, falls es keine mögliche Ausführung gibt, die zu einem erfolgreichen Zustand führt*
- *may-divergent, falls es mindestens eine mögliche Ausführung $(\mathcal{P}, \mathcal{C}) \xrightarrow{LS,*} (\mathcal{P}', \mathcal{C}')$ gibt, sodass $(\mathcal{P}', \mathcal{C}')$ must-divergent ist.*

Ein Prozess $\mathcal{P}$ gilt als may-convergent bzw. must-convergent, falls der Zustand $(\mathcal{P}, IS)$ may-convergent bzw. must-convergent ist.

Wie bereits zuvor bei der Sprache *SYNCSIMPLE* gibt es auch in *LOCKSIMPLE*$_{IS}$ oft verschiedene Möglichkeiten zur Ausführung eines Prozesses. Auch hier terminieren alle Prozesse. Allerdings können unterschiedliche Ausführungen wieder zu unterschiedlichen Resultaten führen. Wenn ein Subprozess blockiert ist, dann muss er solange warten, bis sich der Speicherzustand durch einen anderen Subprozess ändert und seine nächste Operation ausgeführt werden kann.

Beispiel 2.2 *Die Reduktion des Prozesses* $\mathcal{P} = P_1T_11 \parallel T_10$ *mit initialem Speicher IS = (0,0) kann auf verschiedene Weisen ablaufen:*

$$\begin{aligned}
1.\ (P_1T_11 \parallel T_10\,,\,(0,0)) &\xrightarrow{LS} (T_11 \parallel T_10\,,\,(1,0)) \\
&\xrightarrow{LS} (1 \parallel T_10\,,\,(0,0)) \\
2.\ (P_1T_11 \parallel T_10\,,\,(0,0)) &\xrightarrow{LS} (T_11 \parallel T_10\,,\,(1,0)) \\
&\xrightarrow{LS} (T_11 \parallel 0\,,\,(0,0))
\end{aligned}$$

Im ersten Fall ist der Prozess erfolgreich. Im zweiten Fall blockiert das verbleibende T_1 *des ersten Subprozesses, sodass der Prozess nicht erfolgreich beendet werden kann. Dementsprechend ist der Prozess may-convergent.*

2.3. Übersetzungen

Das Ziel ist es, korrekte Übersetzungen von synchroner in asynchrone Kommunikation, also von *SYNCSIMPLE* in *LOCKSIMPLE*, zu finden bzw. aufzuzeigen, dass es keine korrekte Übersetzung gibt. Als Kriterium für die Korrektheit einer Übersetzung gilt, dass jeder Prozess in *SYNCSIMPLE* das gleiche Konvergenzverhalten besitzen muss wie der entsprechende übersetzte Prozess in *LOCKSIMPLE*. Das heißt z.B., wenn ein Prozess in *SYNCSIMPLE* bei jeder möglichen Ausführung erfolgreich ist, dann muss auch der übersetzte Prozess in *LOCKSIMPLE* bei jeder möglichen Ausführung erfolgreich sein.

Definition 2.8 *Eine Zuordnung τ von SYNCSIMPLE-Ausdrücken zu LOCKSIMPLE-Ausdrücken bildet eine Übersetzung, falls damit alle Prozesse in SYNCSIMPLE zu einem Prozess in LOCKSIMPLE zugeordnet werden.*

τ wird als kompositionell bezeichnet, falls folgende Bedingungen erfüllt werden:

- $\tau(0) = 0$ *und* $\tau(1) = 1$
- $\tau(P_1 \parallel P_2) = \tau(P_1) \parallel \tau(P_2)$
- *Für jeden Subprozess U darf $\tau(U)$ nicht den Operator $\parallel$ enthalten*
- $\tau(!U) = \tau(!)\tau(U)$
- $\tau(?U) = \tau(?)\tau(U)$

Eine Zuordnung τ ist korrekt, falls für alle SYNCSIMPLE-Prozesse P gilt:

- *ist P may-convergent, dann ist auch $\tau(P)$ may-convergent*
- *ist P must-convergent, dann ist auch $\tau(P)$ must-convergent*

Kompositionelle Übersetzungen können mit Hilfe eines Tupels $(\tau(?), \tau(!))$ bestehend aus zwei Strings beschrieben werden. Eine Übersetzung hat Länge n, falls $|\tau(?)| + |\tau(!)| = n$.

Beispiel 2.3 *Die Übersetzung $(\tau(?), \tau(!)) = (P_1P_2, T_2)$ hat Länge 3. Der SYNCSIMPLE-Prozess $?!0 \parallel !?1$ wird damit in den LOCKSIMPLE-Prozess $P_1P_2T_20 \parallel T_2P_1P_21$ übersetzt.*

3. Das Programm

In dem folgenden Kapitel wird gezeigt, wie die zuvor beschriebenen Inhalte mit Hilfe der funktionalen Programmiersprache Haskell implementiert wurden, um Übersetzungen von *SYNCSIMPLE* in *LOCKSIMPLE* automatisch zu überprüfen. Dazu werden jeweils einzelne Codeblöcke gezeigt und anschließend beschrieben, welche Funktionen dort enthalten sind und was diese im Programm bewirken. Einige der verwendeten Datenstrukturen und Funktionen wurden dabei aus dem Tool „Refute Regex Translations Tool“ übernommen bzw. orientieren sich daran. Dieses Tool wurde von Schmidt-Schauß und Sabel entwickelt und im Rahmen einer ihrer Arbeiten [2] verwendet.
In dem Programm werden sowohl Listen als auch Sets verwendet, sodass die entsprechenden Libraries zu Beginn importiert werden müssen. Außerdem wird die „regexpr-symbolic“-Library [3] verwendet. Um diese zu nutzen, muss sich der Ordner „RegExpr“ mit der Datei „RegExprOperations.lhs“ in dem Verzeichnis befinden, in dem das Programm gestartet wird. Insgesamt werden zu Beginn also drei Libraries importiert:

Vewendete Libraries

```
1  import RegExpr.RegExprOperations
2  import Data.List
3  import qualified Data.Set as Set
```

3.1. Implementierung der Sprache SYNCSIMPLE

Die Sprache *SYNCSIMPLE* wurde folgendermaßen implementiert:

SYNCSIMPLE

```
1  data SYNCSIMPLE = Zero | One | Out SYNCSIMPLE | In SYNCSIMPLE
2   deriving(Eq,Ord)
3
4  instance Show SYNCSIMPLE where
5      show (Zero) = "0"
6      show (One) = "1"
7      show (Out x) = "!"++(show x)
8      show (In x) = "?" ++(show x)
9
10 newtype SYNCSIMPLEPROCESS = SYNCSIMPLEPROCESS [SYNCSIMPLE]
11  deriving(Eq,Ord)
12
13 instance Show SYNCSIMPLEPROCESS where
14     show (SYNCSIMPLEPROCESS xs) = intercalate "||" (map show xs)
```

Ein Subprozess wird mit Hilfe eines neuen Datentyps gemäß der in Kapitel 2.1 beschriebenen Grammatik modelliert. *Zero* und *One* stehen dabei für 0 bzw. 1. *Out* und *In* stehen für ! bzw. ?. Ein Subprozess muss immer mit *Zero* oder *One* enden.

In der Ausgabe werden die Komponenten eines Subprozesses mit den in Kapitel 2.1 beschriebenen Symbolen dargestellt.
Anschließend können mehrere Subprozesse mit dem Typ *SYNCSIMPLEPROCESS* zu einem Prozess zusammengefasst werden. Dazu müssen die Subprozesse in Form einer Liste an den Konstruktor übergeben werden.
Die operationale Semantik von *SYNCSIMPLE*-Prozessen wurde folgendermaßen umgesetzt:

SYNCSIMPLE-Reduktion

```
1  maycon :: SYNCSIMPLEPROCESS -> Bool
2  maycon (SYNCSIMPLEPROCESS []) = False
3  maycon xs = case oneStepSync xs of
4                [] -> successful xs
5                ys -> any maycon ys
6
7  mustcon :: SYNCSIMPLEPROCESS -> Bool
8  mustcon xs = maycon xs && all mustcon (oneStepSync xs)
9
10 successful :: SYNCSIMPLEPROCESS -> Bool
11 successful (SYNCSIMPLEPROCESS proc) =
12     One `elem` proc
```

Die Funktionen *maycon* und *mustcon* überprüfen, ob ein gegebener *SYNCSIMPLE*-Prozess die jeweilige Eigenschaft besitzt. Dazu müssen die Reduktionsschritte gemäß den in Definition 2.2 beschriebenen Regeln ausgeführt werden. Diese Schritte sind in der Funktion *oneStepSync* modelliert, welche im nächsten Block gezeigt wird.
Nach jedem Schritt können je nach Ausführung mehrere unterschiedliche Prozesse resultieren, wie in Beispiel 2.1 gezeigt. In der Funktion *maycon* werden zwei Fälle unterschieden, die nach Ausführung eines Schritts auftreten können. Im ersten Fall war es nicht möglich, einen weiteren Kommunikationsschritt auszuführen. In diesem Fall ist die Ergebnismenge leer und es bleibt nur noch zu prüfen, ob der Prozess erfolgreich ist. Dies geschieht mit Hilfe der Funtion *successful*, welche überprüft, ob einer der Subprozesse mit einer 1 beginnt. Ist der Prozess erfolgreich, dann ist er may-convergent.
Im zweiten Fall konnte mindestens ein weiterer Kommunikationsschritt ausgeführt werden, sodass man eine Menge mit neuen Prozessen erhält. In diesem Fall wird dann mit *any* überprüft, ob mindestens einer dieser Prozesse may-convergent ist.
Die Funktion *mustcon* verwendet die Funktion *maycon* um zu testen, ob alle möglichen Ausführungen eines Prozesses erfolgreich sind. Ist dies der Fall, ist der Prozess must-convergent.

SYNCSIMPLE-Reduktion

```
1  oneStepSync :: SYNCSIMPLEPROCESS -> [SYNCSIMPLEPROCESS]
2  oneStepSync (SYNCSIMPLEPROCESS p) =
3      let ones = filter isOne p
4          outs = filter isOut p
5          ins = filter isIn p
6          new = [ones ++ [removeHead reducedOut] ++ [removeHead reducedIn] ++ (removeSync
               reducedOut outs) ++ (removeSync reducedIn ins) | reducedOut <- outs, reducedIn
               <- ins]
7      in map SYNCSIMPLEPROCESS $ Set.elems (Set.fromList [(sort a) | a <- new])
8
9  isOne One = True
10 isOne _ = False
11 isZero Zero = True
12 isZero _ = False
13 isOut (Out _) = True
14 isOut _ = False
15 isIn (In _) = True
16 isIn _ = False
17
18 removeHead :: SYNCSIMPLE -> SYNCSIMPLE
19 removeHead (In rest) = rest
20 removeHead (Out rest) = rest
21 removeHead p = p
22
23 removeSync:: SYNCSIMPLE -> [SYNCSIMPLE] -> [SYNCSIMPLE]
24 removeSync p [] = []
25 removeSync p (x:xs)
26             | x == p = xs
27             | otherwise = [x] ++ removeSync p xs
```

Die Funktion *oneStepSync* erzeugt alle möglichen Prozesse, welche durch Ausführung eines Kommunikationsschritts bei einem Prozess entstehen können. Dabei werden alle Subprozesse nach dem jeweils vordersten Ausdruck der Subprozesse sortiert. Subprozesse, bei denen eine 0 zu Beginn steht, können entfernt werden, da sie im Bezug auf die Ausführung keine Rolle mehr spielen und auch nicht dazu beitragen, dass ein Prozess erfolgreich ist.
Anschließend werden alle möglichen Paare aus Subprozessen gebildet, wobei je ein Subprozess mit einem ! und der andere mit einem ? beginnen muss. Dann wird eine Menge aus neuen Prozessen berechnet, welche dadurch entsteht, dass jeweils eines der zuvor gebildeten Paare verwendet wird, um einen Kommunikationsschritt in dem ursprünglichen Prozess auszuführen, d.h. das ! und ? der entsprechenden Subprozesse wird entfernt.
Dies geschieht mit Hilfe der Funktion *removeHead*, welche lediglich dafür sorgt, dass ein *In* (?) oder ein *Out* (!) zu Beginn eines Subprozesses entfernt wird. Mit der Funktion *removeSync* werden die beiden verwendeten Subprozesse aus den Listen *outs* und *ins* entfernt.
Die neuen Prozesse werden als Set gespeichert. So werden identische Prozesse, die im Laufe der Ausführung entstehen können, direkt herausgefiltert, um die Komplexität in der Folge zu reduzieren.

3.2. Implementierung der Sprache LOCKSIMPLE

Die Sprache *LOCKSIMPLE* wurde wie folgt implementiert:

LOCKSIMPLE

```
1  data LOCKSIMPLE = Success | Stop | P1 | T1 | P2 | T2  | P1T1Star | T1P1Star | P2T2Star |
       T2P2Star | P1T1Plus | P2T2Plus | T1P1Plus | T2P2Plus | More | Alt [[LOCKSIMPLE]]
2   deriving(Eq,Ord)
3
4  instance Show LOCKSIMPLE where
5      show (Success) = "1"
6      show (Stop) = "0"
7      show (P1) = "P1"
8      show (T1) = "T1"
9      show (P2) = "P2"
10     show (T2) = "T2"
11     show (P1T1Star) = "(P1T1)*"
12     show (T1P1Star) = "(T1P1)*"
13     show (P2T2Star) = "(P2T2)*"
14     show (T2P2Star) = "(T2P2)*"
15     show (P1T1Plus) = "(P1T1)+"
16     show (T1P1Plus) = "(T1P1)+"
17     show (P2T2Plus) = "(P2T2)+"
18     show (T2P2Plus) = "(T2P2)+"
19     show (More) = "More"
20     show (Alt xs) = "Alt" ++ show xs
21
22 newtype LOCKSIMPLEPROCESS = LOCKSIMPLEPROCESS [[LOCKSIMPLE]]
23  deriving(Eq,Ord)
24
25 instance Show LOCKSIMPLEPROCESS where
26     show (LOCKSIMPLEPROCESS x) = intercalate "||" (map show x)
```

Ein *LOCKSIMPLE*-Subprozess besteht aus einer Sequenz von *LOCKSIMPLE*-Ausdrücken. *Success* steht hierbei für 1 und *Stop* für 0. Mit den vier Ausdrücken P_1, P_2, T_1, T_2 werden die Speicheroperationen beschrieben. Außerdem ist es möglich, eine Folge von *LOCKSIMPLE*-Operationen durch reguläre Ausdrücke zu beschreiben. Dafür gibt es einige zusätzliche Ausdrücke. P_iT_iStar steht hierbei für den Ausdruck $(P_iT_i)^*$ und P_iT_iPlus steht für $(P_iT_i)^+$. Der Ausdruck *More* beschreibt eine beliebige Sequenz von *LOCKSIMPLE*-Ausdrücken und kann an Stellen eingefügt werden, wo der weitere Verlauf nicht bekannt ist. *Alt* ermöglicht es, eine Menge an alternativen *LOCKSIMPLE*-Sequenzen anzugeben, welche an einer Stelle im Subprozess vorkommen können.
Mehrere *LOCKSIMPLE*-Subprozesse, welche in Form einer Liste definiert werden, können schließlich einen *LOCKSIMPLEPROCESS* bilden, ähnlich wie zuvor bei der Sprache *SYNCSIMPLE*.

Zusätzlich zu dem Prozess benötigt man bei der Sprache *LOCKSIMPLE* noch den Zustand des Speichers.

LOCKSIMPLE

```
1  data LOCKVAR = Full | Emp
2   deriving(Eq,Ord)
3
4  instance Show LOCKVAR where
5      show (Full) = "1"
6      show (Emp) = "0"
7
8  newtype STORAGE = STORAGE [LOCKVAR]
9   deriving(Eq,Ord)
10
11 instance Show STORAGE where
12     show (STORAGE xs) = "(" ++ intercalate "," (map show xs) ++ ")"
13
14 newtype LOCKSIMPLESTATE = LOCKSIMPLESTATE (LOCKSIMPLEPROCESS,STORAGE)
15  deriving(Eq,Ord)
16
17 instance Show LOCKSIMPLESTATE where
18     show (LOCKSIMPLESTATE (a,b)) = (show a) ++ "␣with␣storage:␣" ++ (show b)
```

Um den Speicher zu modellieren wird zunächst ein neuer Datentyp *LOCKVAR* verwendet, welcher entweder *Full* oder *Emp* (Empty) sein kann. Damit wird eine Speicherzelle und ihr Zustand beschrieben, also ob sie voll oder leer ist. In der Ausgabe wird der Zustand einer Speicherzelle mit 0 (= *Emp*) oder 1 (= *Full*) angezeigt. Mehrere *LOCKVAR*s bilden schließlich den Speicher, welcher ebenfalls durch einen neuen Datentyp *STORAGE* beschrieben wird. Dieser wird mit Hilfe einer Liste von *LOCKVAR*s erzeugt.
Der *LOCKSIMPLE*-Prozess und der *STORAGE* bilden gemeinsam den aktuellen Zustand einer Ausführung. Deshalb gibt es den Datentyp *LOCKSIMPLESTATE*, welcher diese beiden Komponenten in einem Tupel verbindet. Somit ist es nun möglich, auch die Sprache *LOCKSIMPLE* gemäß den Definitionen aus Kapitel 2.2 darzustellen.

Bei der Ausführung von *LOCKSIMPLE*-Prozessen werden zwei Fälle unterschieden. Der erste Fall umfasst alle Prozesse, in denen nur die Ausdrücke P_1, P_2, T_1, T_2 sowie *Success* und *Stop* vorkommen. Für diese Prozesse können alle möglichen Ausführungen berechnet werden, da alle Operation innerhalb der Prozesse konkret angegeben sind.
Im zweiten Fall kommen alle Prozesse vor, in denen auch die restlichen Ausdrücke wie zum Beispiel $(P_1T_1)^*$ und $(P_1T_1)^+$ vorkommen. Bei diesen Prozessen handelt es sich im Grunde genommen um mehrere verschiedene Prozesse, da diese Ausdrücke auf viele unterschiedliche Arten erweitert werden können, wodurch jedes Mal ein anderer Prozess entsteht. Da es aber nicht möglich ist, jede dieser Möglichkeiten zu betrachten, muss man hier bei der Reduktion etwas anders vorgehen als bei den konkreten Prozessen.
Zunächst wird die Reduktion für den ersten Fall implementiert und beschrieben. Die Reduktion von Prozessen, die dem zweiten Fall entsprechen, folgt später.

LOCKSIMPLE-Reduktion von konkreten Prozessen

```
1  mayconLock :: LOCKSIMPLESTATE -> Bool
2  mayconLock (LOCKSIMPLESTATE (LOCKSIMPLEPROCESS [],b)) = False
3  mayconLock xs = case oneStepLock xs of
4                  [] -> successfulLock xs
5                  ys -> any mayconLock ys
6
7  mustconLock :: LOCKSIMPLESTATE -> Bool
8  mustconLock xs = mayconLock xs && all mustconLock (oneStepLock xs)
9
10 successfulLock:: LOCKSIMPLESTATE -> Bool
11 successfulLock (LOCKSIMPLESTATE(LOCKSIMPLEPROCESS p, s)) = any (isHeadSuccess) p
12
13 isHeadSuccess x = isSuccess $ head x
14
15 isSuccess Success = True
16 isSuccess _ = False
```

Die Funktionen *mayconLock* und *mustconLock* dienen der Überprüfung von Prozessen im Bezug auf die Eigenschaften *maycon* bzw. *mustcon*. Diesmal beziehen sich die Funktionen allerdings auf *LOCKSIMPLE*-Prozesse. Dabei funktionieren sie jedoch ähnlich, wie zuvor die Funktionen *maycon* und *mustcon* für SYNCSIMPLE-Prozesse.
Bei einem Aufruf von *mayconLock* wird die Funktion *oneStepLock* auf den übergebenen Prozess angewendet. Damit werden alle möglichen Reduktionsschritte eines Prozesses, wie in Kapitel 2.2 beschrieben, ausgeführt und eine Menge mit allen resultierenden Zuständen bestehend aus einem Prozess und dem zugehörigen Speicherzustand erstellt. Ist diese Menge leer, wird mit der Funktion *successfulLock* überprüft, ob mindestens einer der Subprozesse des Prozesses mit *Success* beginnt, also ob der Prozess erfolgreich war.
Falls die Menge nicht leer ist, wird überprüft, ob mindestens einer der resultieren Zustände maycon-convergent ist, d.h die Funktion *mayconLock* wird auf alle resultierenden Zustände angewendet.

LOCKSIMPLE-Reduktion von konkreten Prozessen

```
1  oneStepLock :: LOCKSIMPLESTATE -> [LOCKSIMPLESTATE]
2  oneStepLock (LOCKSIMPLESTATE (LOCKSIMPLEPROCESS p, s)) = map LOCKSIMPLESTATE $ Set.elems (
       Set.fromList $ [(LOCKSIMPLEPROCESS $ sort a, b) | (LOCKSIMPLEPROCESS a,b) <-
       removeNothing $ map (oneStep s p) p])
3
4  removeNothing :: [Maybe (LOCKSIMPLEPROCESS,STORAGE)] -> [(LOCKSIMPLEPROCESS,STORAGE)]
5  removeNothing (Just x:xs) = [x] ++ (removeNothing xs)
6  removeNothing (Nothing:xs) = removeNothing xs
7  removeNothing [] = []
8
9  oneStep:: STORAGE -> [[LOCKSIMPLE]] -> [LOCKSIMPLE] -> Maybe (LOCKSIMPLEPROCESS,STORAGE)
10 oneStep (STORAGE [s1,s2]) procs p
11         | isSuccess (head p) = Nothing
12         | isStop (head p) = Nothing
13         | isPut1 (head p) = if s1 == Emp then Just (LOCKSIMPLEPROCESS([tail p] ++ remove p
               procs),STORAGE [Full,s2]) else Nothing
14         | isPut2 (head p) = if s2 == Emp then Just (LOCKSIMPLEPROCESS([tail p] ++ remove p
               procs),STORAGE [s1,Full]) else Nothing
15         | isTake1 (head p) = if s1 == Full then Just (LOCKSIMPLEPROCESS([tail p] ++ remove p
                procs),STORAGE [Emp,s2]) else Nothing
16         | isTake2 (head p) = if s2 == Full then Just (LOCKSIMPLEPROCESS([tail p] ++ remove p
                procs),STORAGE [s1,Emp]) else Nothing
17         | otherwise = error "Patterns␣are␣not␣supported␣in␣this␣function"
18
19
20 remove:: [LOCKSIMPLE] -> [[LOCKSIMPLE]] -> [[LOCKSIMPLE]]
21 remove p [] = [[]]
22 remove p (x:xs)
23             | x == p = xs
24             | otherwise = [x] ++ remove p xs
25
26 isStop Stop = True
27 isStop _ = False
28 isPut1 (P1) = True
29 isPut1 _ = False
30 isPut2 (P2) = True
31 isPut2 _ = False
32 isTake1 (T1) = True
33 isTake1 _ = False
34 isTake2 (T2) = True
35 isTake2 _ = False
```

Bei der Ausführung der Funktion *oneStepLock* wird die Funktion *oneStep* auf alle Subprozesse angewendet. Diese Funktion prüft, ob der aktuelle Speicherzustand es ermöglicht, einen Schritt im betrachteten Subprozess auszuführen. Ist dies der Fall, wird der Subprozess mittels der Funktion *remove* aus dem Prozess entfernt. Dann wird dem Prozess ein neuer Subprozess hinzugefügt, welcher dem zu reduzierenden Subprozess ohne den Ausdruck an vorderster Stelle entspricht. Außerdem wird der Zustand der modifizierten Speicherzelle entsprechend angepasst. So erhält man ein Paar bestehend aus dem neuen

Prozess und dem neuen Speicher, welches der Ergebnismenge hinzugefügt wird. Lässt sich der Subprozess nicht ausführen, weil der Zustand der Speichers nicht passend ist und der Subprozess somit blockiert ist, gibt die Funktion ein *Nothing* zurück. Entsprechend ist die Ausgabe im Fall, dass es möglich ist, vom Typ *Maybe*. Die Funktion *removeNothing*, welche in der Ausführung von *oneStepLock* aufgerufen wird, entfernt alle *Nothings* aus der Ergebnismenge und entfernt bei allen anderen Ergebnissen den Typ *Maybe*. Die Ergebnismenge wird anschließend sortiert und als Set dargestellt, sodass es keine doppelten Ergebnisse gibt. Dann werden alle übrigen Ergebnisse wieder mit *LOCKSIMPLESTATE* in einen Zustand umgeformt.
Die Ausführung von Prozessen, welche durch reguläre Ausdrücke beschrieben werden, wird wie bereits erwähnt etwas anders behandelt, da man hier keine konkreten Prozesse vorfindet. Im Folgenden wird die Reduktion solcher Prozesse beschrieben:

Reduktion von LOCKSIMPLE-Prozessen mit regulären Ausdrücken

```
1  mayconLockPattern :: LOCKSIMPLESTATE -> Bool
2  mayconLockPattern (LOCKSIMPLESTATE (LOCKSIMPLEPROCESS [],b)) = False
3  mayconLockPattern xs = case oneStepLockPattern xs of
4              [] -> successfulLock xs
5              ys -> any mayconLockPattern ys
6
7  maydivLockPattern :: LOCKSIMPLESTATE -> Bool
8  maydivLockPattern xs = mustdivLockPattern xs || (any maydivLockPattern (oneStepLockPattern
       xs))
9
10 mustdivLockPattern:: LOCKSIMPLESTATE -> Bool
11 mustdivLockPattern xs = case oneStepLockPattern xs of
12                            [] -> deadlock xs
13                            _ -> False
14
15 deadlock:: LOCKSIMPLESTATE -> Bool
16 deadlock (LOCKSIMPLESTATE (LOCKSIMPLEPROCESS p,s)) = all lock p
17
18 lock:: [LOCKSIMPLE] -> Bool
19 lock x = not (isMore $ head x) && not (isSuccess $ head x)
20
21 isMore (More) = True
22 isMore _ = False
```

Die Reduktion von *LOCKSIMPLE*-Prozessen, welche mit regulären Ausdrücken beschrieben werden, ist ähnlich aufgebaut, wie die Reduktion von konkreten *LOCKSIMPLE*-Prozessen. In diesem Fall werden zusätzlich die Ausdrücke $(P_iT_i)^+$,$(T_iP_i)^+$ und $More$ berücksichtigt. Was mit den Ausdrücken $(P_iT_i)^*$, $(T_iP_i)^*$ und *Alt* geschieht, wird später beschrieben. Das Problem bei Prozessen, die Ausdrücke enthalten, welche erweitert werden können, ist wie bereits erwähnt, dass es sich hierbei eigentlich um viele verschiedene Prozesse handelt, die mit Hilfe eines Musters zusammengefasst werden. Dementsprechend ist es nicht möglich, solche Prozesse im Detail auszuführen. Allerdings haben alle Prozesse, die durch Erweiterung dieser Ausdrücke entstehen können, einige Ausführungen gemeinsam.

Und das ist der Fall, wenn die Ausdrücke $(P_iT_i)^+$ und $(T_iP_i)^+$ in einem Zug ausgeführt werden. Dann ist es nicht von Bedeutung, wie oft sich z.B. die Folge P_iT_i wiederholt. In der Funktion *oneStepPattern*, welche in *oneStepLockPattern* verwendet wird, werden die Ausdrücke $(P_iT_i)^+$ und $(T_iP_i)^+$ genau auf diese Weise betrachtet. Der Ausdruck *More* wird so behandelt, dass die Reduktion an dieser Stelle nicht weiter ausgeführt werden kann. Wenn man nun zeigen kann, dass eine dieser Ausführungen erfolgreich ist, weiß man zumindest, dass alle Prozesse, die durch Erweiterung der Ausdrücke entstehen können, may-convergent sind. Und falls es eine Ausführung gibt, die nicht erfolgreich ist, sind alle diese Prozesse mit Sicherheit may-divergent. Diese Erkenntnis kann schon ausreichend sein, um Übersetzungen zu widerlegen. Wenn der ursprüngliche Prozess in *SYNCSIMPLE* zum Beispiel must-convergent ist und man zeigen kann, dass der übersetzte *LOCKSIMPLE*-Prozess mit erweiterbaren Ausdrücken may-divergent ist, spielt es keine Rolle, wie man die Ausdrücke erweitert. Dementsprechend werden bei Übersetzungen mit erweiterbaren Ausdrücken keine *SYNCSIMPLE*-Prozesse zum Widerlegen verwendet, die may-convergent sind.
Die Eigenschaft may-convergent wird genau wie bei Prozessen ohne reguläre Ausdrücke überprüft. Die Frage, ob es eine Ausführung gibt, die zu einem nicht erfolgreichen Zustand führt, wird mit der Funktion *maydivLockPattern* untersucht. Dabei wird zum einen getestet, ob der Prozess must-divergent ist. Das könnte dann der Fall sein, wenn kein Ausführungsschritt möglich ist. In diesem Fall würde beim Aufruf von *oneStepLockPattern* in der Funktion *mustdivLockPattern* eine leere Ergebnismenge erzeugt werden. Eine leere Ergebnismenge würde allerdings nicht zwangsläufig bedeuten, dass der Prozess nicht erfolgreich ist. Deswegen muss mit der Funktion *deadlock* bestätigt werden, dass alle Subprozesse blockiert sind, ohne dass einer der Subprozesse ein *Success* oder ein *More* an vorderster Stelle beinhaltet. Dies würde nämlich bedeuten, dass der Prozess erfolgreich ist bzw. an einem *More* nicht weiter ausgeführt wird, obwohl es theoretisch möglich wäre. Zum anderen wird überprüft, ob mindestens einer der Zustände, welche durch Ausführung eines Schritts entstehen, may-divergent ist. Falls eine dieser beiden Bedingungen eintritt, ist der Prozess may-divergent. Die Funktion *oneStepLockPattern* sieht wie folgt aus:

Reduktion von LOCKSIMPLE-Prozessen mit regulären Ausdrücken

```
1  oneStepLockPattern :: LOCKSIMPLESTATE -> [LOCKSIMPLESTATE]
2  oneStepLockPattern (LOCKSIMPLESTATE (LOCKSIMPLEPROCESS p, s)) = map LOCKSIMPLESTATE $ Set.
       elems (Set.fromList $ [(LOCKSIMPLEPROCESS $ sort a, b) | (LOCKSIMPLEPROCESS a,b) <-
       removeNothing $ map (oneStepPattern s p) p])
3
4  oneStepPattern:: STORAGE -> [[LOCKSIMPLE]] -> [LOCKSIMPLE] -> Maybe (LOCKSIMPLEPROCESS,
       STORAGE)
5  oneStepPattern (STORAGE [s1,s2]) procs p
6          | isSuccess (head p) = Nothing
7          | isStop (head p) = Nothing
8          | isPut1 (head p) = if s1 == Emp then Just (LOCKSIMPLEPROCESS([tail p] ++ remove p
               procs),STORAGE [Full,s2]) else Nothing
9          | isPut2 (head p) = if s2 == Emp then Just (LOCKSIMPLEPROCESS([tail p] ++ remove p
               procs),STORAGE [s1,Full]) else Nothing
10         | isTake1 (head p) = if s1 == Full then Just (LOCKSIMPLEPROCESS([tail p] ++ remove p
               procs),STORAGE [Emp,s2]) else Nothing
11         | isTake2 (head p) = if s2 == Full then Just (LOCKSIMPLEPROCESS([tail p] ++ remove p
               procs),STORAGE [s1,Emp]) else Nothing
12         | isP1T1Plus (head p) = if s1 == Emp then Just (LOCKSIMPLEPROCESS([tail p] ++ remove
               p procs),STORAGE [s1,s2]) else Nothing
13         | isP2T2Plus (head p) = if s2 == Emp then Just (LOCKSIMPLEPROCESS([tail p] ++ remove
               p procs),STORAGE [s1,s2]) else Nothing
14         | isT1P1Plus (head p) = if s1 == Full then Just (LOCKSIMPLEPROCESS([tail p] ++
               remove p procs),STORAGE [s1,s2]) else Nothing
15         | isT2P2Plus (head p) = if s2 == Full then Just (LOCKSIMPLEPROCESS([tail p] ++
               remove p procs),STORAGE [s1,s2]) else Nothing
16         | isMore (head p) = Nothing
17         | otherwise = error "Some␣patterns␣must␣be␣expanded␣before␣using␣this␣function"
18
19 isP1T1Plus (P1T1Plus) = True
20 isP1T1Plus _ = False
21 isP2T2Plus (P2T2Plus) = True
22 isP2T2Plus _ = False
23 isT1P1Plus (T1P1Plus) = True
24 isT1P1Plus _ = False
25 isT2P2Plus (T2P2Plus) = True
26 isT2P2Plus _ = False
```

Damit können nun auch Ausführungen von Prozessen, die mit regulären Ausdrücken dargestellt werden, berechnet werden. Allerdings fehlt noch, wie mit den Ausdrücken $(P_iT_i)^*$) und *Alt* umgegangen wird. Diese beiden Ausdrücke müssen vor der Ausführung mit der Funktion *expand* folgendermaßen erweitert werden:

Expandieren der Ausdrücke

```
1  expand :: [LOCKSIMPLE] -> [[LOCKSIMPLE]]
2  expand [] = [[]]
3  expand ((P1T1Star):ys)  = concat [[p,P1T1Plus:p] |  p <- expand ys]
4  expand ((T1P1Star):ys)  = concat [[p,T1P1Plus:p] |  p <- expand ys]
5  expand ((P2T2Star):ys)  = concat [[p,P2T2Plus:p] |  p <- expand ys]
6  expand ((T2P2Star):ys)  = concat [[p,T2P2Plus:p] |  p <- expand ys]
7  expand ((Alt y):ys) = [x ++ p | a <- y, x <- expand a, p <- expand ys]
8  expand (y:ys) = [y:p |  p <- expand ys]
```

Der Ausdruck $(P_iT_i)^*$ wird erweitert, indem der ursprüngliche Prozess zwei mal erzeugt wird, wobei die Folge P_iT_i in einem der Prozesse gar nicht vorkommt und in dem anderen Prozess kommt sie mindestens einmal vor und kann somit als $(P_iT_i)^+$ dargestellt werden. Der Ausdruck *Alt* wird so erweitert, dass der ursprüngliche Prozess mehrmals erzeugt wird und dann jeweils eine der angegeben Alternativen an der entsprechenden Stelle in einem der Prozesse eingefügt wird.
Dadurch erhält man bei Prozessen, in denen diese beiden Ausdrücke enthalten sind, mehrere Prozesse, bei denen man mit den zuvor beschriebenen Funktionen Reduktionsschritte durchführen kann. Wichtig ist, dass die Funktion *expand* vorher mit solchen Prozessen aufgerufen wird, da es sonst zu einem Fehler innerhalb der Funktionen *mayconLockPattern* und *maydivLockPattern* kommt.

3.3. Übersetzen eines Prozesses

Zum Übersetzen von Prozessen benötigt man eine Übersetzung gemäß den Beschreibungen in Kapitel 2.3. Eine solche Übersetzung wird mit dem folgenden Datentyp definiert:

Eine Übersetzung erzeugen

```
1  newtype TRANSLATION = TRANSLATION ([LOCKSIMPLE],[LOCKSIMPLE], STORAGE)
2   deriving(Eq,Ord)
3
4  instance Show TRANSLATION where
5      show (TRANSLATION (a,b,c)) = "(" ++ show a ++ "␣,␣" ++ show b ++ ")," ++ "␣with␣initial␣
           storage:␣" ++ show c
6
7  createTranslation:: ([LOCKSIMPLE],[LOCKSIMPLE]) -> TRANSLATION
8  createTranslation (inp,outp) = TRANSLATION (inp,outp,(STORAGE [Emp,Emp]))
```

Eine Übersetzung wird durch einen neuen Datentyp namens *TRANSLATION* dargestellt. Eine *TRANSLATION* enthält zwei Sequenzen von *LOCKSIMPLE*-Ausdrücken, welche die Übersetzung von einem Input bzw. Output in *SYNCSIMPLE* darstellen. Dazu enthält eine *TRANSLATION* den Zustand des initialen Speichers, um anzugeben, in welchem Zustand eine Ausführung des übersetzten Prozesses beginnen soll. Mit diesen drei Angaben, lassen sich *SYNCSIMPLE*-Prozesse komplett in LOCKSIMPLE übersetzen.
Mit Hilfe der Funktion *createTranslation* lässt sich außerdem eine *TRANSLATION* erzeugen, welche den initialen Speicherzustand (0,0) besitzt. Dies ist sinnvoll beim Testen (siehe Kapitel 4.2), da man hier meist diese Konstellation verwendet und man somit nicht immer den Speicher mitangeben muss, sondern nur zwei Listen mit *LOCKSIMPLE*-Ausdrücken.
Die Übersetzung eines Prozesses funktioniert dann folgendermaßen:

Übersetzen eines Prozesses

```
1  translate:: [LOCKSIMPLE] -> [LOCKSIMPLE] -> SYNCSIMPLE -> [LOCKSIMPLE]
2  translate input output (In p) = input ++ translate input output p
3  translate input output (Out p) = output ++ translate input output p
4  translate input output (Zero) = [Stop]
5  translate input output (One) = [Success]
6
7  translateProcess:: [LOCKSIMPLE] -> [LOCKSIMPLE] -> SYNCSIMPLEPROCESS -> LOCKSIMPLEPROCESS
8  translateProcess input output (SYNCSIMPLEPROCESS p) = LOCKSIMPLEPROCESS $ map (translate
       input output) p
9
10 applyTranslation:: TRANSLATION -> SYNCSIMPLEPROCESS -> LOCKSIMPLESTATE
11 applyTranslation (TRANSLATION(input,output,s)) p = LOCKSIMPLESTATE (translateProcess input
       output p, s)
```

Die Funktion *translate* nimmt die Übersetzung für den Input und Output in einem *SYNCSIMPLE*-Prozess und übersetzt damit schrittweise einen *SYNCSIMPLE*-Subprozess, indem

die Ausdrücke *In* und *Out* des Subprozesses durch die Ausdrücke der *TRANSLATION* ersetzt werden. Die Ausdrücke *Zero* und *One* werden durch *Success* bzw. *Stop* ersetzt.
Die Funktion *translateProcess* nimmt schließlich einen kompletten *SYNCSIMPLE*-Prozess und übersetzt alle Subprozesse, indem die Funktion *translate* auf alle diese Subprozesse angewendet wird. Als Ergebnis erhält man eine Liste mit den übersetzten Subprozessen, mit deren Hilfe dann ein *LOCKSIMPLE*-Prozess erzeugt wird.
Die Funktion *applyTranslation* verwendet die zuvor beschriebenen Funktionen und fügt dem damit erzeugten *LOCKSIMPLE*-Prozess den initialen Speicher hinzu, den die *TRANSLATION* beinhaltet, um somit den Zustand zu erstellen, in dem die Ausführung des Prozesses startet.

Beispiel 3.1 *In Abbildung 1 ist zu sehen, wie ein SYNCSIMPLE-Prozess innerhalb des Programms mit Hilfe einer Übersetzung in einen LOCKSIMPLE-Zustand übersetzt werden kann. Der Prozess* $!0 \parallel ?1$ *wird in diesem Beispiel mit der Übersetzung* $(\tau(?), \tau(!)) = (P_1P_2, T_2)$ *in den Prozess* $T_20 \parallel \ P_1P_21$ *übersetzt. Zusammen mit dem initialen Speicher (0,0) ergibt das den Zustand* $(T_20 \parallel P_1P_21\ ,\ (0,0))$.

```
*Main> let sync = SYNCSIMPLEPROCESS [Out Zero, In One]
*Main> let trans = TRANSLATION ([P1,P2],[T2],STORAGE [Emp,Emp])
*Main> applyTranslation trans sync
[T2,0] || [P1,P2,1] with storage: (0,0)
```

Abbildung 1: Übersetzen eines Prozesses (Quelle: Eigene Darstellung)

3.4. Überprüfen einer Übersetzung

Beim Überprüfen von Übersetzungen wird wieder zwischen konkreten Übersetzungen und Übersetzungen, die durch reguläre Ausdrücke angegeben werden, unterschieden. Zuerst wird die Überprüfung von konkreten Übersetzungen beschrieben:

Überprüfung einer konkreten Übersetzung

```
1  checkTranslation :: SYNCSIMPLEPROCESS -> LOCKSIMPLESTATE -> Bool
2  checkTranslation s l = maycon s == mayconLock l && mustcon s == mustconLock l
3
4  newtype RESULT = RESULT (Bool,TRANSLATION,SYNCSIMPLEPROCESS,LOCKSIMPLESTATE)
5
6  instance Show RESULT where
7      show (RESULT (a,b,c,d)) = show a ++ " --- " ++ show b ++ " --- " ++ show c ++ " --- " ++
           show d ++ "\n"
8
9  checkMultiple :: [SYNCSIMPLEPROCESS] -> [LOCKSIMPLESTATE] -> TRANSLATION -> RESULT
10 checkMultiple (p:ps) (l:ls) t = if checkTranslation p l == False then RESULT (False,t,p,l)
       else checkMultiple ps ls t
11 checkMultiple [] [] t = RESULT (True, t, SYNCSIMPLEPROCESS [], LOCKSIMPLESTATE (
       LOCKSIMPLEPROCESS [], STORAGE []))
12
13 propString :: SYNCSIMPLEPROCESS -> [Char]
14 propString p = if mustcon p then "must-convergent" else if maycon p then "may-convergent, "
       but not must-convergent" else "must-divergent"
15
16 propStringLock :: LOCKSIMPLESTATE -> [Char]
17 propStringLock l = if mustconLock l then "must-convergent" else if mayconLock l then "may-
       convergent, but not must-convergent" else "must-divergent"
```

Die Funktion *checkTranslation* nimmt einen *SYNCSIMPLEPROCESS* und einen *LOCKSIMPLE*-Zustand und überprüft, ob beide Prozesse dieselben Eigenschaften bezüglich ihres Konvergenzverhaltens besitzen. Damit kann man überprüfen, ob eine Übersetzung eines *SYNCSIMPLE*-Prozesses in *LOCKSIMPLE* korrekt ist, indem man der Funktion den *SYNCSIMPLE*-Prozess und den übersetzten *LOCKSIMPLE*-Zustand übergibt.
Um eine korrekte Übersetzung darzustellen, muss eine *TRANSLATION* jeden *SYNCSIMPLE*-Prozess korrekt übersetzen. Deshalb ist es beim Testen notwendig, mehrere *SYNCSIMPLE*-Prozesse und deren übersetzte *LOCKSIMPLE*-Prozesse auf Gleichheit bei diesen Eigenschaften zu überprüfen, um die Wahrscheinlichkeit zu erhöhen, dass inkorrekte Übersetzungen auch dementsprechend erkannt werden. Dafür gibt es die Funktion *checkMultiple*, welche eine Liste von *SYNCSIMPLE*-Prozessen und eine Liste mit den zugehörigen übersetzten *LOCKSIMPLE*-Zuständen nimmt und dann jeweils die zusammengehörigen Paare überprüft. Sobald ein Paar gefunden wurde, welches in den Eigenschaften nicht übereinstimmt, wurde nachgewiesen, dass die *TRANSLATION* nicht korrekt sein kann.

Die Funktion gibt ein *RESULT* zurück. Dabei handelt es sich um einen neuen Datentyp, welcher sich aus vier Komponenten zusammensetzt:

- ein Bool-Wert, der *False* ist, falls die *TRANSLATION* falsch ist, oder *True*, falls die *TRANSLATION* nicht widerlegt werden kann.
- die *TRANSLATION*, welche zum Übersetzen verwendet wurde
- der *SYNCSIMPLE*-Prozess, welcher als Gegenbeispiel dient, falls die Übersetzung nicht korrekt ist. Ist der Bool-Wert *True* wird hier ein leerer Prozess ausgegeben.
- der *LOCKSIMPLE*-Zustand, der durch Übersetzen des *SYNCSIMPLE*-Prozesses, welcher als Gegenbeispiel dient, entstanden ist. Falls es einen solchen Prozess nicht gibt, wird hier ein leerer Zustand ausgegeben.

Falls kein Paar gefunden wurde, welches bei den Eigenschaften nicht übereinstimmt, war es mit den betrachteten Prozessen nicht möglich die *TRANSLATION* zu widerlegen.
Die Funktionen *propString* und *propStringLock* geben einen String aus, der beschreibt, welche Eigenschaft ein bestimmter *SYNCSIMPLE*-Prozess bzw. ein *LOCKSIMPLE*-Zustand besitzt. Diese Funktionen werden später in der Ausgabe verwendet, um zu zeigen, warum ein SYNCSIMPLE-Prozess und der übersetzte LOCKSIMPLE-Zustand in ihrem Konvergenzverhalten nicht übereinstimmen.
Alle diese Funktionen werden in der Funktion *refute* zusammengeführt, um es zu ermöglichen, eine Übersetzung mit einem Aufruf vollständig mit Textausgabe zu testen. Diese Funktion sieht wie folgt aus:

Überprüfung einer konkreten Übersetzung

```
1  refute :: TRANSLATION -> IO RESULT
2  refute t = do
3                              let l = map (applyTranslation t) testProcs
4                                  r = checkMultiple testProcs l t
5                                  RESULT(b,_,sync,lock) = r
6                                  syncString = propString sync
7                                  lockString = propStringLock lock
8                              putStrLn $ "Checking translation " ++ show t
9                              if not b then
10                                 do
11                                     putStrLn $ "Translation refuted"
12                                     putStrLn $ "Counter example: sync = " ++ show sync
13                                     putStrLn $ "Translated: lock = " ++ show lock
14                                     putStrLn $ "sync is " ++ syncString
15                                     putStrLn $ "lock is " ++ lockString ++ "\n"
16                                     return r
17                             else
18                                 do
19                                     putStrLn $ "Translation could not be refuted\n"
20                                     return r
```

Die Funktion *refute* nimmt eine *TRANSLATION* und wendet diese mit Hilfe der zuvor beschriebenen Funktion *applyTranslation* auf alle verwendeten *SYNCSIMPLE*-Testprozesse

an, welche unter *testProcs* definiert sind (mehr dazu später). Anschließend überprüft sie alle diese Prozesse und die jeweiligen übersetzten *LOCKSIMPLE*-Zustände mit Hilfe der zuvor beschriebenen Funktionen. Mit Hilfe des Datentyps *RESULT* ist es hierbei möglich, auf alle wichtigen Informationen zuzugreifen. Falls eine Übersetzung falsch ist, wird in der Ausgabe angegeben, dass die *TRANSALTION* widerlegt wurde. Außerdem wird der *SYNCSIMPLE*-Prozess und der dazugehörige übersetzte *LOCKSIMPLE*-Zustand angegeben, welche in dem *RESULT* als Gegenbeispiel gespeichert wurden. Zusätzlich werden noch die Eigenschaften dieser beiden Prozesse ausgegeben, um zu zeigen, inwiefern sie nicht übereinstimmen.
Falls eine *TRANSLATION* nicht widerlegt werden konnte, wird dies entsprechend in der Ausgabe angegeben.
Zum Schluss gibt die Funktion das *RESULT* zurück, da dieses in der als nächstes beschriebenen Funktion *refuteMultiple* benötigt wird:

Überprüfung von mehreren konkreten Übersetzungen

```
1  refuteMultiple :: [TRANSLATION] -> [RESULT] -> IO ()
2  refuteMultiple [] r = do putStrLn $ "Translations␣that␣could␣not␣be␣refuted:\n"
3                           showTransList r
4                           putStrLn ""
5
6  refuteMultiple (t:ts) r = do
7                              RESULT(b,tr,sync,lock) <- refute t
8                              if b then
9                                  refuteMultiple ts (r++[RESULT(b,tr,sync,lock)])
10                             else
11                                 refuteMultiple ts r
12
13 showTransList [] = return()
14 showTransList (x:xs) = do
15                        putStrLn $ show x
16                        showTransList xs
```

Mit Hilfe der Funktion *refuteMultiple* lassen sich mehrere *TRANSLATION*s überprüfen, indem schrittweise jede einzelne *TRANSLATION* an die Funktion *refute* übergeben wird. Alle *TRANSLATION*s, die dabei nicht widerlegt werden konnten, werden in einer Liste gesammelt, um diese am Ende der Ausführung in der Ausgabe anzuzeigen. Deshalb braucht man in der Funktion *refute* das jeweilige *RESULT* als Ausgabewert. Die Liste mit den *RESULT*s wird als zweites Argument in der Funktion *refuteMultiple* mitgeführt und in jedem Schritt erweitert, falls die getestete *TRANSLATION* nicht widerlegt werden konnte. Dementsprechend muss der erste Aufruf der Funktion *refuteMultiple* mit einer leeren Liste als zweites Argument durchgeführt werden.
Um am Ende die Liste mit den *RESULT*s aller nicht widerlegten *TRANSLATION*s anzuzeigen, wird die Funktion *showTranslist* verwendet, welche lediglich alle *RESULT*s der übergebenen Liste schrittweise in der Ausgabe anzeigt.
Damit lassen sich nun mehrere konkrete Übersetzungen auf einmal testen.

Für das Überprüfen von Übersetzungen mit regulären Ausdrücken gibt es eigene Funktionen, welche allerdings ähnlich wie die zuvor beschriebenen Funktionen für konkrete Übersetzungen aufgebaut sind. Diese Funktion werden im Folgenden gezeigt:

Überprüfung einer Übersetzung mit regulären Ausdrücken

```
1  checkTranslationPattern :: SYNCSIMPLEPROCESS -> LOCKSIMPLESTATE -> Bool
2  checkTranslationPattern s l
3                              | mustcon s = not (maydivLockPattern l)
4                              | not (maycon s) = not (mayconLockPattern l)
5                              | maycon s = True
6
7  checkMultiplePattern :: [SYNCSIMPLEPROCESS] -> [LOCKSIMPLESTATE] -> TRANSLATION -> RESULT
8  checkMultiplePattern (p:ps) (l:ls) t = if checkTranslationPattern p l == False then RESULT (
       False,t,p,l) else checkMultiplePattern ps ls t
9  checkMultiplePattern [] [] t = RESULT (True, t, SYNCSIMPLEPROCESS [], LOCKSIMPLESTATE (
       LOCKSIMPLEPROCESS [], STORAGE []))
10
11 propStringLockPat :: LOCKSIMPLESTATE -> [Char]
12 propStringLockPat l = if (maydivLockPattern l) && (mayconLockPattern l) then "may-convergent
       ␣and␣may-divergent" else if mayconLockPattern l then "may-convergent" else if
       maydivLockPattern l then "may-divergent" else ""
```

Die Überprüfung von Übersetzungen mit regulären Ausdrücken läuft ähnlich ab wie bei konkreten Übersetzungen. Der einzige Unterschied ist, wie bereits zuvor erwähnt, dass *SYNCSIMPLE*-Prozesse, die may-convergent sind, hier nicht als Gegenbeispiel dienen. Dementsprechend sind die Funktionen *checkTranslationPattern* und *checkMultiplePattern* etwas anders, als die entsprechenden Funktionen im Fall mit konkreten Übersetzungen. Hier wird nun überprüft, ob der übersetzte *LOCKSIMPLE*-Prozess eines *SYNCSIMPLE*-Prozesses, welcher must-convergent ist, may-divergent ist oder ob der *LOCKSIMPLE*-Prozess eines *SYNCSIMPLE*-Prozesses mit der Eigenschaft must-divergent may-convergent ist. In diesen beiden Fällen könnte man mit Sicherheit sagen, dass die Übersetzung nicht korrekt ist. In allen anderen Fällen kann sie nicht widerlegt werden.
Die Funktion *propStringLockPat* dient dazu, einen String auszugeben, der das Konvergenzverhalten eines *LOCKSIMPLE*-Zustands beschreibt, dessen Prozess mit regulären Ausdrücken dargestellt wird. Dieser String wird später in der Ausgabe verwendet, um zu beschreiben, warum ein bestimmter *SYNCSIMPLE*-Prozess zum Widerlegen einer Übersetzung verwendet wird.

Diese Funktionen werden schließlich wieder in einer Funktion zusammengeführt, mit der man eine Übersetzung auf alle Testprozesse anwenden und anschließend überprüfen kann:

Überprüfung einer Übersetzung mit regulären Ausdrücken

```
1  refutePattern :: TRANSLATION -> IO RESULT
2  refutePattern t = do
3                              let l = map (applyTranslation t) testProcs
4                                  r = checkMultiplePattern testProcs l t
5                                  RESULT(b,_,sync,lock) = r
6                                  syncString = propString sync
7                                  lockString = propStringLockPat lock
8                              putStrLn $ "Checking translation " ++ show t
9                              if not b then
10                                 do
11                                     putStrLn $ "Translation refuted"
12                                     putStrLn $ "Counter example: sync = " ++ show sync
13                                     putStrLn $ "Translated: lock = " ++ show lock
14                                     putStrLn $ "sync is " ++ syncString
15                                     putStrLn $ "lock is " ++ lockString ++ "\n"
16                                     return r
17                             else
18                                 do
19                                     putStrLn $ "Translation could not be refuted\n"
20                                     return r
```

Die Funktion *refutePattern* funktioniert genau wie die Funktion *refute*, nur das hier die zuvor beschriebenen Funktionen für Übersetzungen mit regulären Ausdrücken eingesetzt werden. In der Ausgabe wird dann wieder gezeigt, ob die Übersetzung widerlegt werden konnte und welcher *SYNCSIMPLE*-Prozess in diesem Fall als Gegenbeispiel dient. Als Rückgabewert besitzt die Funktion auch ein *RESULT*, welches in der folgenden Funktion benötigt wird:

Überprüfung mehrerer Übersetzungen mit regulären Ausdrücken

```
1  refuteMultiplePattern :: [TRANSLATION] -> [TRANSLATION] -> IO ()
2  refuteMultiplePattern [] r = do
3                               putStrLn $ "Translations␣that␣could␣not␣be␣refuted:\n"
4                               showTransList r
5                               putStrLn ""
6
7  refuteMultiplePattern (t:ts) r = do
8                               let TRANSLATION (inp,outp,s) = t
9                                   tlist = [TRANSLATION(a,b,s) | a <- expand inp, b <- expand
                                        outp]
10                              if length tlist == 1 then
11                                  do
12                                      RESULT(b,tr,sync,lock) <- refutePattern t
13                                      if b then
14                                          refuteMultiplePattern ts (r++[tr])
15                                      else
16                                          refuteMultiplePattern ts r
17                              else
18                                  do
19                                      putStrLn $ "Unfolding␣translation␣" ++ show t ++ ":\n"
20                                      showTransList tlist
21                                      putStrLn ""
22                                      refuteMultiplePattern (tlist ++ ts) r
```

Die Funktion *refuteMultiplePattern* läuft nach demselben Prinzip ab wie die Funktion *refuteMultiple*. Allerdings werden hier bei Übersetzungen mit regulären Ausdrücken diese Ausdrücke erweitert, sodass man eventuell mehrere neue Übersetzungen erhält. Diese Übersetzungen werden in einer Liste dargestellt, welche vorne an die Liste mit den zu überprüfenden Übersetzungen gestellt wird. Diese neuen Übersetzungen werden in der Ausgabe unter dem Hinweis „Unfolding translation“ aufgelistet. Damit kann man nachverfolgen, welche Übersetzungen durch Erweiterung der Ausdrücke entstanden sind. Anschließend werden alle Übersetzungen wie bei *refuteMultiple* nacheinander getestet und am Ende werden alle nicht widerlegten Übersetzungen gesammelt ausgegeben. Hat man eine Menge, die aus konkreten Übersetzungen und Übersetzungen mit regulären Ausdrücken besteht, kann man auch diese Funktion verwenden. Allerdings werden die konkreten Übersetzungen hierbei nicht so ausführlich getestet, da *SYNCSIMPLE*-Prozesse, die may-convergent sind, hier nicht betrachtet werden. Eventuell kann eine Übersetzung dann nicht widerlegt werden, die sonst widerlegt werden würde.

Zum Überprüfen von Übersetzungen wird eine Menge an *SYNCSIMPLE*-Prozessen verwendet. Diese werden unter der Bezeichnung *testProcs* definiert und in den Funktionen *refute* und *refutePattern* verwendet. Die folgenden Prozesse wurden im Laufe dieser Arbeit verwendet:

Verwendete SYNCSIMPLE-Prozesse zum Überprüfen von Übersetzungen

```
1  testProcs = map SYNCSIMPLEPROCESS
2          [[Out One, In Zero]                                  -- !1 || ?0              must-con
3          ,[Out Zero, In One]                                  -- ?1 || !0              must-con
4          ,[Out $ In Zero, In One]                             -- !?0 || ?1             must-conv
5          ,[In $ Out Zero, Out One]                            -- ?!0 || !1             must-conv
6          ,[In One, In Zero]                                   -- ?1 || ?0              must-div
7          ,[Out One, Out Zero]                                 -- !1 || !0              must-div
8          ,[Out $ Out One, In Zero]                            -- !!1 || ?0             must-div
9          ,[In $ In One, Out Zero]                             -- ??1 || !0             must-div
10         ,[Out $ In One, Out Zero]                            -- !?1 || ?0             must-div
11         ,[In $ Out One, In Zero]                             -- ?!1 || !0             must-div
12         ,[Out $ In Zero, In $ In One]                        -- !?0 || ??1            must-div
13         ,[In $ Out Zero, Out $ Out One]                      -- ?!0 || !!1            must-div
14         ,[Out One, Out One, In Zero]                         -- !1 || !1 || ?0        must-con
15         ,[In One, In One, Out Zero]                          -- ?1 || ?1 || !0        must-conv
16         ,[Out Zero, Out Zero, In One]                        -- !0 || !0 || ?1        must-con
17         ,[In Zero, In Zero, Out One]                         -- ?0 || ?0 || !1        must-con
18         ,[Out One, Out One, In Zero]                         -- !1 || !1 || ?0        must-con
19         ,[In One, In One, Out Zero]                          -- ?1 || ?1 || !0        must-con
20         ,[Out One, Out $ In $ Out Zero, In Zero]             -- !1 || !?!0 || ?0      must-con
21         ,[In One, In $ Out $ In Zero, Out Zero]              -- ?1 || ?!?0 || !0      must-con
22         ,[In Zero, Out Zero, Out One]                        -- ?0 || !0 || !1        may-con
23         ,[Out Zero, In Zero, In One]                         -- !0 || ?0 || ?1        may-con
24         ,[Out Zero, In One, In Zero]                         -- !0 || ?1 || ?0        may-con
25         ,[In Zero, Out One, Out Zero]                        -- ?0 || !1 || !0        may-con
26         ,[In One, In Zero, In Zero]                          -- ?1 || ?0 || ?0        must-div
27         ,[Out One, Out Zero, Out Zero]                       -- !1 || !0 || !0        must-div
28         ,[Out One, Out One, Out One, In Zero]                -- !1 || !1 || !1 || ?0  must-con
29         ,[In One, In One, In One, Out Zero]]                 -- ?1 || ?1 || ?1 || !0  must-con
```

Mit den Testprozessen soll ein möglichst großes Spektrum abgedeckt werden, um Übersetzungen möglichst aussagekräftig zu testen. Dementsprechend sind alle Arten von Prozessen enthalten. Diese Menge an Prozessen kann beliebig angepasst werden.

3.5. Generieren von Übersetzungen

Mit Hilfe der folgenden Funktionen lassen sich Übersetzungen mit einer bestimmten Länge automatisch generieren:

Generieren von konkreten Übersetzungen

```
1  genLocksimple :: (Num a, Eq a) => a -> [[LOCKSIMPLE]]
2  genLocksimple 1 = [[P1],[P2],[T1],[T2]]
3  genLocksimple len = [x ++ [y]| x <- genLocksimple (len-1), y <- [P1,P2,T1,T2]]
4
5  genTranslation:: (Num a1, Num a, Eq a1, Eq a) => a -> a1 -> [TRANSLATION]
6  genTranslation inputLen outputLen = map createTranslation [(x,y) | x <- genLocksimple
       inputLen, y <- genLocksimple outputLen]
7
8  genTranslationOfLength:: (Num a, Eq a, Enum a) => a -> [TRANSLATION]
9  genTranslationOfLength len = [ b | a <- [1..len-1], b <- genTranslation a (len-a)]
```

Die Funktion *genLockSimple* erzeugt alle möglichen Sequenzen aus *LOCKSIMPLE*-Ausdrücken einer bestimmten Länge, welche als Argument angegeben werden muss.
In der Funktion *genTranslation* wird diese Funktion verwendet, um alle möglichen *TRANSLATION*s zu generieren, bei denen die Übersetzungen für *In* und *Out* jeweils eine bestimmte Länge haben. Diese beiden Längen können unterschiedlich sein. Dazu werden alle möglichen Sequenzen zur Übersetzung von *In* und *Out* mit *genLocksimple* erzeugt und dann alle möglichen Paare gebildet. Als initialer Speicher wird dabei immer (0,0) verwendet.
Die Funktion *genTranslationOfLength* erzeugt mit Hilfe der beiden zuvor beschriebenen Funktionen schließlich alle *TRANSLATION*s einer bestimmten Gesamtlänge. Dazu wird die Gesamtlänge auf alle möglichen Arten auf die beiden Übersetzungen für *In* und *Out* aufgeteilt.

3.6. Funktionen zum Suchen nach korrekten Übersetzungen

Die folgenden drei Funktionen sind die Hauptfunktionen zum Testen von Übersetzungen:

Hauptfunktionen

```
1  test :: [TRANSLATION] -> IO ()
2  test t = do putStrLn "\n--------- Test starts here ----------\n"
3              refuteMultiple t []
4              putStrLn "\n--------- Test finished ----------\n"
5
6  testPattern :: [TRANSLATION] -> IO ()
7  testPattern t = do putStrLn "\n--------- Test starts here ----------\n"
8                     refuteMultiplePattern t []
9                     putStrLn "\n--------- Test finished ----------\n"
10
11 testLength n = do putStrLn "\n--------- Test starts here ----------\n"
12                   refuteMultiple (genTranslationOfLength n) []
13                   putStrLn "\n--------- Test finished ----------\n"
```

Die Funktion *test* erwartet eine Liste mit Übersetzungen, welche getestet werden sollen. Diese Funktion ist allerdings nur zum Testen von konkreten Übersetzungen geeignet. Für Übersetzungen mit regulären Ausdrücken wird die Funktion *testPattern* verwendet. Die Funktion *testLength* testet alle konkreten Übersetzungen mit einer bestimmten Länge n. Dazu werden alle Übersetzungen der Länge n mit Hilfe der zuvor beschriebenen Funktion *genTranslationOfLength* erzeugt und anschließend mit *refuteMultiple* getestet.

Beispiel 3.2 *Die Überprüfung der Übersetzung* $(\tau(?), \tau(!)) = (P_1T_2, P_2P_2)$ *mit der Funktion test läuft beispielsweise wie in Abbildung 2 zu sehen ab.*

```
*Main> test $ [createTranslation([P1,T2],[P2,P2])]

-------- Test starts here ---------

Checking translation ([P1,T2] , [P2,P2]), with initial storage: (0,0)
Translation refuted
Counter example: sync = ?!0 || !1
Translated: lock = [P1,T2,P2,P2,0] || [P2,P2,1] with storage: (0,0)
sync is must-convergent
lock is may-convergent, but not must-convergent

Translations that could not be refuted:



-------- Test finished ---------
```

Abbildung 2: Beispielaufruf der Funktion *test* (Quelle: Eigene Darstellung)

Beispiel 3.3 *Enthält die zu überprüfende Übersetzung Ausdrücke, welche erweitert werden können, dann muss die Funktion testPattern verwendet werden. In Abbildung 3 sieht man ein Beispiel, wie Ausdrücke der Übersetzung* $(\tau(?), \tau(!)) = (P_1 More, P_2(T_1 P_1)^*)$ *vor dem Überprüfen erweitert werden und sich dadurch mehrere Übersetzungen ergeben. Eine dieser Übersetzungen kann nicht widerlegt werden und wird dementsprechend am Ende des Testlaufs unter den nicht widerlegten Übersetzungen aufgeführt.*

```
*Main> testPattern $ [createTranslation([P1,More],[P2,T1P1Star])]

-------- Test starts here ---------

Unfolding translation ([P1,More] , [P2,(T1P1)*]), with initial storage: (0,0):

([P1,More] , [P2]), with initial storage: (0,0)
([P1,More] , [P2,(T1P1)+]), with initial storage: (0,0)

Checking translation ([P1,More] , [P2]), with initial storage: (0,0)
Translation refuted
Counter example: sync = !1 || !0
Translated: lock = [P2,1] || [P2,0] with storage: (0,0)
sync is must-divergent
lock is may-convergent and may-divergent

Checking translation ([P1,More] , [P2,(T1P1)+]), with initial storage: (0,0)
Translation could not be refuted

Translations that could not be refuted:

([P1,More] , [P2,(T1P1)+]), with initial storage: (0,0)


-------- Test finished ---------
```

Abbildung 3: Beispielaufruf der Funktion *testPattern* (Quelle: Eigene Darstellung)

Beim Suchen nach korrekten Übersetzungen bzw. beim Widerlegen von Übersetzungen kann es hilfreich sein, bestimmte Ausdrücke zu erweitern, um den Suchbereich einzuschränken. Beim Erweitern von Ausdrücken kann es allerdings zu Fehlern kommen. Deshalb kann es nützlich sein, wenn sich die Erweiterung von Ausdrücken auf Korrektheit überprüfen lässt. Dafür können die folgenden Funktionen verwendet werden, welche die „regexpr-symbolic“-Library verwenden. Darin sind Funktionen zum Darstellen und Prüfen von regulären Ausdrücken enthalten.

Überprüfung von regulären Ausdrücken

```
1  lockSimpleToRegex :: LOCKSIMPLE -> RE Char
2  lockSimpleToRegex P1 = L 'P'
3  lockSimpleToRegex P2 = L 'X'
4  lockSimpleToRegex T1 = L 'T'
5  lockSimpleToRegex T2 = L 'Y'
6  lockSimpleToRegex P1T1Plus = let p1t1 = Seq (L 'P') (L 'T') in Seq p1t1 (Star p1t1)
7  lockSimpleToRegex T1P1Plus = let t1p1 = Seq (L 'T') (L 'P') in Seq t1p1 (Star t1p1)
8  lockSimpleToRegex P1T1Star = let p1t1 = Seq (L 'P') (L 'T') in (Star p1t1)
9  lockSimpleToRegex T1P1Star = let t1p1 = Seq (L 'T') (L 'P') in (Star t1p1)
10 lockSimpleToRegex P2T2Plus = let p2t2 = Seq (L 'X') (L 'Y') in Seq p2t2 (Star p2t2)
11 lockSimpleToRegex T2P2Plus = let t2p2 = Seq (L 'Y') (L 'X') in Seq t2p2 (Star t2p2)
12 lockSimpleToRegex P2T2Star = let p2t2 = Seq (L 'X') (L 'Y') in (Star p2t2)
13 lockSimpleToRegex T2P2Star = let t2p2 = Seq (L 'Y') (L 'X') in (Star t2p2)
14 lockSimpleToRegex (Alt xs) = resToRE (map lockSimpleSequenceToRegex xs)
15 lockSimpleToRegex More = Star (Choice (Choice (L 'P') (L 'T')) (Choice (L 'X') (L 'Y')))
16
17 lockSimpleSequenceToRegex :: [LOCKSIMPLE] -> RE Char
18 lockSimpleSequenceToRegex [] = Empty
19 lockSimpleSequenceToRegex  (x:xs) = Seq (lockSimpleToRegex x) (lockSimpleSequenceToRegex xs)
20
21 checkEquality :: LOCKSIMPLE -> LOCKSIMPLE -> Bool
22 checkEquality e1 e2 = equality (lockSimpleToRegex e1) (lockSimpleToRegex e2)
23
24 isContainedIn :: LOCKSIMPLE -> LOCKSIMPLE -> Bool
25 isContainedIn e1 e2 = contains (lockSimpleToRegex e1) (lockSimpleToRegex e2)
```

Die Funktion *lockSimpleToRegex* stellt die Schnittstelle zwischen den *LOCKSIMPLE*-Ausdrücken und der importierten Library dar. Damit werden alle *LOCKSIMPLE*-Ausdrücke in reguläre Ausdrücke umgeformt, so wie sie in der Library dargestellt werden. Die Funktion *lockSimpleSequenceToRegex* überführt eine Sequenz aus *LOCKSIMPLE*-Ausdrücken in einen solchen regulären Ausdruck.

Überprüfung von regulären Ausdrücken

```
1  isSubsetOfTranslations :: [TRANSLATION] -> [TRANSLATION] -> Bool
2  isSubsetOfTranslations l1 l2 =
3    (foldl1 Choice [  (Seq (Seq (lockSimpleSequenceToRegex a) (L '#')) (
         lockSimpleSequenceToRegex b)) | TRANSLATION (a,b,s) <- l1])
4    `contains`
5    (foldl1 Choice [  (Seq (Seq (lockSimpleSequenceToRegex a) (L '#')) (
         lockSimpleSequenceToRegex b)) | TRANSLATION (a,b,s) <- l2])
6
7  equalTranslations :: [TRANSLATION] -> [TRANSLATION] -> Bool
8  equalTranslations t1 t2 = isSubsetOfTranslations t1 t2 && isSubsetOfTranslations t2 t1
9
10 isExpContainedIn :: [LOCKSIMPLE] -> [LOCKSIMPLE] -> Bool
11 isExpContainedIn e1 e2 = contains (lockSimpleSequenceToRegex e1) (lockSimpleSequenceToRegex
       e2)
12
13 isExpEqual :: [LOCKSIMPLE] -> [LOCKSIMPLE] -> Bool
14 isExpEqual e1 e2 = isExpContainedIn e1 e2 && isExpContainedIn e2 e1
```

Mit der Funktion *isSubesetOfTranslations* kann geprüft werden, ob eine Menge von Übersetzungen in einer anderen Menge aus Übersetzungen enthalten ist. Die Funktion *equalTranslations* prüft, ob zwei Mengen aus Übersetzungen gleich sind, d.h. ob die beiden regulären Ausdrücke, welche die Mengen beschreiben, die gleiche Sprache darstellen. Die Funktion *isExpContainedIn* überprüft, ob eine Sequenz aus *LOCKSIMPLE*-Ausdrücken in einer anderen Sequenz aus *LOCKSIMPLE*-Ausdrücken vorkommt. Dies wird in der Funktion *isExpEqual* verwendet, um zu prüfen, ob zwei Sequenzen aus *LOCKSIMPLE*-Ausdrücken gleich sind. Dazu werden die beiden Sequenzen in reguläre Ausdrücke umgeformt und anschließend überprüft, ob sie die gleiche Sprache bilden.

4. Suche nach korrekten Übersetzungen

Mit Hilfe des zuvor beschrieben Programms ist es nun möglich, Übersetzungen anzugeben und automatisch zu überprüfen. Somit kann man unter Verwendung des Programms nach möglicherweise korrekten Übersetzungen suchen. Dabei ist allerdings wichtig, zu beachten, dass eine Übersetzung, die nicht von dem Programm widerlegt wurde, nicht zwangsläufig korrekt ist. Da nur eine begrenzte Auswahl an SYNCSIMPLE-Prozessen zur Überprüfung verwendet wird, kann es sein, dass es außerhalb dieser Auswahl einen Prozess gibt, welcher mit der geprüften Übersetzung nicht korrekt übersetzt werden kann. Mit dem Programm lassen sich Übersetzungen also nur mit Sicherheit ausschließen. Falls eine Übersetzung nicht widerlegt werden kann, benötigt es weitere Überprüfungen, bevor man annehmen kann, dass diese Übersetzung tatsächlich korrekt ist.
Für die Suche nach eventuell korrekten Übersetzungen gibt es verschiedene Herangehensweisen. Mit Hilfe der Funktion *testLength* ist es beispielsweise möglich unter allen Übersetzungen einer bestimmten Länge nach einer nicht widerlegbaren Übersetzung zu suchen. Ein anderer Ansatz besteht darin, einen Teil der Übersetzung, also z.B. $\tau(?)$, festzulegen und $\tau(!)$ zunächst mit *More* darzustellen. Den Ausdruck *More* kann man dann erweitern und testen, ob einer der resultieren Übersetzungen nicht widerlegt werden kann. Nicht widerlegte Übersetzungen, bei denen $\tau(!)$ wieder auf *More* endet, können anschließend wieder erweitert werden. So kann man möglicherweise eine korrekte Übersetzung finden oder zeigen, dass es keine korrekte Übersetzung mit dem festgelegten $\tau(?)$ gibt, falls sich der Ausdruck *More* irgendwann nicht mehr erweitern lässt, ohne in eine endlose Schleife von *LOCKSIMPLE*-Operationen zu geraten.
Mit diesen Methoden kann man den Suchbereich einschränken, indem man zeigt, dass bestimmte Strukturen innerhalb einer Übersetzung nicht zulässig sind, wenn man eine korrekte Übersetzung erhalten möchte.
In diesem Kapitel werden mit Hilfe von Tests einige Bedingungen an korrekte Übersetzungen dargelegt, welche sich daraus ergeben, dass alle Übersetzungen, die diese Bedingungen nicht erfüllen, von dem Algorithmus widerlegt wurden. Dazu wurden die in Kapitel 3.4 gezeigten Beispielprozesse in *SYNCSIMPLE* verwendet.

4.1. Konfiguration des initialen Speichers

Bei der Suche nach korrekten Übersetzungen spielt der anfängliche Zustand des Speichers eine Rolle, da die Ausführung der Prozesse davon abhängt. Bei der Verwendung von zwei Speicherzellen gibt es vier verschiedene Konfigurationen des initialen Speichers. Während der Suche genügt es allerdings, nur eine der möglichen Startkonfigurationen des Speichers zu betrachten. Das liegt daran, dass es möglich ist jeden Prozess in *LOCKSIMPLE* mit einem beliebigen initialen Speicherzustand in einen isomorphen *LOCKSIMPLE*-Prozess mit jedem anderen möglichen initialen Speicherzustand umzuwandeln. Dies ist möglich, indem man innerhalb eines Prozesses jedes P_i in ein T_i und jedes T_i in ein P_i umwandelt und gleichzeitig den Startzustand von Speicherzelle i umkehrt. Dadurch kann man bei der Suche beispielsweise immer den Startzustand (0,0) beim Speicher betrachten, da alle *LOCKSIMPLE*-Prozesse mit einem anderen initialen Speicher isomorph zu einem der *LOCKSIMPLE*-Prozesse mit initialem Speicher (0,0) sind. Im Rahmen dieser Arbeit wird bei der Suche, falls nicht anders angegeben, immer der initiale Speicher (0,0) verwendet.

Beispiel 4.1 *Der Prozess $\mathcal{P} = P_1P_21 \parallel T_10$ in der Sprache LOCKSIMPLE$_{(0,0)}$ ist isomorph zu dem Prozess $\mathcal{P}' = T_1T_21 \parallel P_10$ in LOCKSIMPLE$_{(1,1)}$.*

4.2. Übersetzungen mit Länge < 10

Mit Hilfe der Funktion *testLength* können alle Übersetzungen einer bestimmten Länge überprüft werden. Allerdings steigt der Rechenaufwand bei steigender Länge der Übersetzungen sehr schnell, sodass die Verwendung dieser Funktion nur bis zu einer gewissen Länge sinnvoll ist. Bis einschließlich Länge 9 wurden schrittweise alle Übersetzungen beginnend bei Länge 2 getestet. Dabei wurden alle Übersetzungen widerlegt. Das bedeutet, dass eine korrekte Übersetzung also mindestens eine Länge von 10 besitzen muss.

4.3. Suche durch Erweiterung von *More*

Eine weitere Möglichkeit nach korrekten Übersetzungen zu suchen, ist die zuvor beschriebene Methode, bei der man den Ausdruck *More* erweitert, um Übersetzungen zu konkretisieren. Dazu lassen sich verschiedene Ausdrücke definieren, mit denen man *More* erweitern kann. Je nach Struktur der bisherigen Sequenz von *LOCKSIMPLE*-Ausdrücken lassen sich unterschiedliche Ausdrücke verwenden. Im Folgenden werden die im Rahmen dieser Arbeit verwendeten Ausdrücke zur Erweiterung von *More* beschrieben.

Erweiterung von More

```
1  step = Alt[[P1],[P2],[T1],[T2]]
2
3  someSteps = Alt[[],[step],[step,step],[step,step,More]]
```

Der Ausdruck *step* beschreibt einen einzelnen Schritt in einem *LOCKSIMPLE*-Prozess. Für einen einzelnen Schritt gibt es vier Möglichkeiten: P_1, P_2, T_1, T_2.
Der Ausdruck *someSteps* ermöglicht es, *More* so zu erweitern, dass bis zu zwei weitere Schritte konkretisiert werden. Es werden alle Fälle abgedeckt, wie *More* erweitert werden kann. Ein *More* kann entsprechend entweder leer sein, genau eine Operation enthalten, genau zwei Operationen enthalten oder wenn es mindestens drei Operationen enthält, dann werden die ersten beiden konkret dargestellt und alle restlichen Operationen werden wieder mit einem *More* zusammengefasst.
Mit der Funktion *isExpEqual* wurde zusätzlich sichergestellt, dass diese Erweiterung von *More* korrekt ist.

Erweiterung von More

```
1  expP1More = [Alt[[P1T1Plus],[P1T1Star,P1],[P1T1Plus,Alt[[T1],[T2],[P2]]],[P1T1Plus,Alt[[P2
       ],[T1],[T2]],More],[P1T1Star,P1,Alt[[P1],[P2],[T2]]],[P1T1Star,P1,Alt[[P1],[P2],[T2]],
       More]]]
2
3  expP1MoreAlt = [Alt[[P1,T1,P1T1Star],[Alt[[],[P1,T1,P1T1Star]],P1],[P1,T1,P1T1Star,Alt[[T1
       ],[T2],[P2]]],[P1,T1,P1T1Star,Alt[[P2],[T1],[T2]],More],[Alt[[],[P1,T1,P1T1Star]],P1,
       Alt[[P1],[P2],[T2]]],[Alt[[],[P1,T1,P1T1Star]],P1,Alt[[P1],[P2],[T2]],More]]]
```

Falls vor dem *More* ein P_1 steht, dann lässt sich die Sequenz $P_1 More$ mit Hilfe des Ausdrucks *expP1More* erweitern. Auch hier werden alle möglichen Fälle, wie die ursprüngliche Sequenz $P_1 More$ erweitert werden kann, abgedeckt. Der Vorteil bei diesem Ausdruck ist, dass hierbei auch die Ausdrücke $(P_1 T_1)^*$ und $(P_1 T_1)^+$ verwendet werden und somit endlose Schleifen abgefangen werden können. Dadurch taucht am Ende der *LOCKSIMPLE*-Sequenz eventuell kein *More* mehr auf, welches wieder erweitert werden müsste, falls die Übersetzung nicht widerlegt werden konnte.
Eine Alternative zu *expP1More* ist der Ausdruck *expP1MoreAlt*, in welchem alle Instanzen von $(P_1 T_1)^+$ so erweitert werden, dass das erste P_1 und T_1 einzeln von dem Algorithmus betrachtet werden. In manchen Fällen lassen sich damit Übersetzungen widerlegen, die sonst nicht widerlegt worden wären, da der Algorithmus die Ausführung von $(P_1 T_1)^+$ immer in einem Schritt durchführt und nicht einzeln betrachtet. Falls sich dadurch entstehende Übersetzungen allerdings nicht widerlegen lassen, werden die Übersetzungen länger als bei der Verwendung von *expP1More*. Deswegen muss man immer abwägen, welcher Ausdruck in einer bestimmten Situation besser geeignet ist. Außerdem wird $(P_1 T_1)^*$ so erweitert, dass die Folge $P_1 T_1$ entweder gar nicht vorkommt oder dass das erste P_1 und T_1 wieder jeweils einzeln betrachtet werden.
Die Korrektheit dieser beiden Erweiterungen von P_1*More* wurde auch wieder mit der Funktion *isExpEqual* überpüft.

Erweiterung von More

```
1  expP2More = [Alt[[P2T2Plus],[P2T2Star,P2],[P2T2Plus,Alt[[T1],[T2],[P1]]],[P2T2Plus,Alt[[P1
       ],[T1],[T2]],More],[P2T2Star,P2,Alt[[P1],[P2],[T1]]],[P2T2Star,P2,Alt[[P1],[P2],[T1]],
       More]]]
2
3  expP2MoreAlt = [Alt[[P2,T2,P2T2Star],[Alt[[],[P2,T2,P2T2Star]],P2],[P2,T2,P2T2Star,Alt[[T1
       ],[T2],[P1]]],[P2,T2,P2T2Star,Alt[[P1],[T1],[T2]],More],[Alt[[],[P2,T2,P2T2Star]],P2,
       Alt[[P1],[P2],[T1]]],[Alt[[],[P2,T2,P2T2Star]],P2,Alt[[P1],[P2],[T1]],More]]]
4
5  expT1More = [Alt[[T1P1Plus],[T1P1Star,T1],[T1P1Plus,Alt[[P1],[P2],[T2]]],[T1P1Plus,Alt[[P1
       ],[P2],[T2]],More],[T1P1Star,T1,Alt[[T1],[T2],[P2]]],[T1P1Star,T1,Alt[[T1],[T2],[P2]],
       More]]]
6
7  expT1MoreAlt = [Alt[[T1,P1,T1P1Star],[Alt[[],[T1,P1,T1P1Star]],T1],[T1,P1,T1P1Star,Alt[[P1
       ],[P2],[T2]]],[T1,P1,T1P1Star,Alt[[P1],[P2],[T2]],More],[Alt[[],[T1,P1,T1P1Star]],T1,
       Alt[[T1],[T2],[P2]]],[Alt[[],[T1,P1,T1P1Star]],T1,Alt[[T1],[T2],[P2]],More]]]
8
9  expT2More = [Alt[[T2P2Plus],[T2P2Star,T2],[T2P2Plus,Alt[[P1],[P2],[T1]]],[T2P2Plus,Alt[[T1
       ],[P1],[P2]],More],[T2P2Star,T2,Alt[[T1],[T2],[P1]]],[T2P2Star,T2,Alt[[T1],[T2],[P1]],
       More]]]
10
11 expT2MoreAlt = [Alt[[T2,P2,T2P2Star],[Alt[[],[T2,P2,T2P2Star]],T2],[T2,P2,T2P2Star,Alt[[P1
       ],[P2],[T1]]],[T2,P2,T2P2Star,Alt[[T1],[P1],[P2]],More],[Alt[[],[T2,P2,T2P2Star]],T2,
       Alt[[T1],[T2],[P1]]],[Alt[[],[T2,P2,T2P2Star]],T2,Alt[[T1],[T2],[P1]],More]]]
```

Nach dem gleichen Prinzip lassen sich auch Ausdrücke definieren, mit denen sich die Folgen P_2More, T_1More bzw. T_2More erweitert darstellen lassen. Damit hat man nun einige Ausdrücke definiert, mit denen man Übersetzungen schrittweise erweitern kann und so die Suche nach korrekten Übersetzungen gegebenenfalls weiter einschränken kann. Je nach Situation muss man dabei abwägen, welcher der beschriebenen Ausdrücke am besten für die Erweiterung geeignet ist bzw. gegebenenfalls ausprobieren, womit man eine gute Ausgangslage für die weitere Suche erhält.
Im nächsten Abschnitt werden Fälle gezeigt, in denen $\tau(?)$ festgelegt wurde, und anschließend alle damit möglichen Übersetzungen mit Hilfe dieser Methode und unter Verwendung der hier definierten Ausdrücke widerlegt wurden.

4.4. Übersetzungen mit $|\tau(?)| = 1$ bzw. $|\tau(!)| = 1$

Im Folgenden werden alle Übersetzungen widerlegt, bei denen eine der Komponenten der Übersetzung auf Länge 1 beschränkt ist, d.h. $|\tau(?)| = 1$ oder $|\tau(!)| = 1$. Dabei genügt es, nur einen der beiden Fälle zu betrachten, da die Argumente aufgrund von Symmetrie auch für den anderen Fall gelten. Im Folgenden wird der Fall $|\tau(?)| = 1$ betrachtet.
Für $\tau(?)$ gibt es bei einer Länge von 1 nur vier Möglichkeiten.

$\tau(?) = P_1$ und $\tau(?) = P_2$

Eine Übersetzung mit $\tau(?) = P_1$ bzw. $\tau(?) = P_2$ kann nicht korrekt sein, da sonst der Prozess $P = ?1 \parallel ?1$ in *LOCKSIMPLE* must-convergent wäre. In *SYNCSIMPLE* ist er hingegen must-divergent. Entsprechend bleiben nur die zwei Möglichkeiten, in denen gilt $\tau(?) = T_1$ bzw. $\tau(?) = T_2$.

$\tau(?) = T_1$ und $\tau(?) = T_2$

Angenommen $\tau(?) = T_1$. Mit Hilfe der zuvor beschriebenen Methode lässt sich eine schrittweise Suche starten, indem man $\tau(!) =$ *More* festlegt. Da $\tau(!)$ mindestens eine Operation enthalten muss, kann man *More* erweitern, sodass $\tau(!) =$ *step More*. Die Übersetzung $(\tau(?), \tau(!)) = (T_1, step\ More)$ kann man nun überprüfen.
Für den Fall $\tau(?) = T_1$ wird die gesamte Suche beschrieben, um die Vorgehensweise bei dem Suchverfahren zu zeigen. Gestartet wird mit dem Aufruf von *test1_1*, wobei gilt:

```
1   test1_1 = testPattern $ [createTranslation ([T1],[step,More])]
```

Das Ergebnis des Testlaufs ist in Abbildung 4 zu sehen.

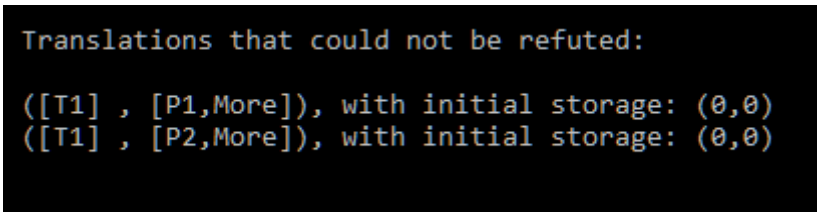

Abbildung 4: Ausgabe des Programms nach Aufruf von *test1_1* (Quelle: Eigene Darstellung)

$\tau(!)$ kann nur mit P_1 oder P_2 beginnen. Die nicht widerlegten Übersetzungen werden in Form einer Liste als *result1_1* definiert. Anschließend werden die nicht widerlegten Übersetzungen in einer Liste, welche mit *newCase1_1* definiert wird, erweitert. Mit Hilfe des Aufrufs von *check1_1* wird danach sichergestellt, dass beim Erweitern der Übersetzungen keine Fehler gemacht wurden, also dass die Übersetzungen in *result1_1* und *newCase1_1* inhaltlich übereinstimmen. Anschließend werden alle erweiterten Übersetzungen mit *test1_2* überprüft.

```
result1_1 = map createTranslation [([T1],[P1,More]),([T1],[P2,More])]

newCase1_1 = map createTranslation [([T1],expP1MoreAlt),([T1],expP2More)]
check1_1 = equalTranslations newCase1_1 $ result1_1
test1_2 = testPattern $ newCase1_1
```

Nach der Ausführung erhält man das in Abbildung 5 dargestellte Ergebnis.

```
Translations that could not be refuted:

([T1] , [P1,P2,More]), with initial storage: (0,0)
([T1] , [(P2T2)+,P1,More]), with initial storage: (0,0)
([T1] , [P2,P1,More]), with initial storage: (0,0)
([T1] , [(P2T2)+,P2,P1,More]), with initial storage: (0,0)
```

Abbildung 5: Ausgabe des Programms nach Aufruf von *test1_2* (Quelle: Eigene Darstellung)

Das Ergebnis wird wieder gespeichert, diesmal unter *result1_2*. Anschließend werden die übrig gebliebenen Übersetzungen wieder erweitert und unter *newCase1_2* gespeichert. Dann wird mit *check1_2* überprüft, ob dabei kein Fehler gemacht wurde und schließlich werden alle erweiterten Übersetzungen mit *test1_3* getestet.

```
result1_2 = map createTranslation [([T1] , [P1,P2,More]),
                                   ([T1] , [P2T2Plus,P1,More]),
                                   ([T1] , [P2,P1,More]),
                                   ([T1] , [P2T2Plus,P2,P1,More])]

newCase1_2 = map createTranslation [([T1] , [P1]++expP2More),
                                    ([T1] , [P2T2Plus]++expP1MoreAlt),
                                    ([T1] , [P2,P1,someSteps]),
                                    ([T1] , [P2T2Plus,P2,P1,someSteps])]
check1_2 = equalTranslations newCase1_2 result1_2
test1_3 = testPattern $ newCase1_2
```

Das Ergebnis nach dem dritten Testschritt ist in Abbildung 6 abgebildet.

```
Translations that could not be refuted:

([T1] , [(P2T2)+,P1,P2,More]), with initial storage: (0,0)
```

Abbildung 6: Ausgabe des Programms nach Aufruf von *test1_3* (Quelle: Eigene Darstellung)

Damit hat man den Suchraum schon stark eingeschränkt. Die zuvor beschriebenen Schritte wiederholt man nun solange, bis alle Übersetzungen widerlegt wurden.

```
1  result1_3 = createTranslation ([T1] , [P2T2Plus,P1,P2,More])
2
3  newCase1_3 = createTranslation ([T1] , [P2T2Plus,P1]++expP2More)
4  check1_3 = equalTranslations [newCase1_3] [result1_3]
5  test1_4 = testPattern $ [newCase1_3]
```

Wie man in Abbildung 7 sieht, gibt es nach dem vierten Testschritt keine Übersetzungen mehr, die nicht widerlegt werden können.

```
Translations that could not be refuted:


-------- Test finished ---------
```

Abbildung 7: Ausgabe des Programms nach Aufruf von *test1_4* (Quelle: Eigene Darstellung)

Damit wurden schrittweise alle Übersetzungen, bei den $\tau(?) = T_1$ gilt, widerlegt. Die Suche im Fall $\tau(?) = T_2$ würde bei initialem Speicher (0,0) symmetrisch zur Suche mit $\tau(?) = T_1$ ablaufen, sodass auch in diesem Fall keine korrekte Übersetzung möglich ist. Damit wurde gezeigt, dass bei einer korrekten Übersetzung sowohl $|\tau(?)| > 1$ als auch $|\tau(!)| > 1$ gelten muss.

4.5. Übersetzungen mit $|\tau(?)| = 2$ bzw. $|\tau(!)| = 2$

Auf die gleiche Art und Weise wie zuvor bei $|\tau(?)| = 1$ lässt sich zeigen, dass es auch unter den Übersetzungen mit $|\tau(?)| = 2$ keine korrekte Übersetzung gibt. Das Gleiche gilt aufgrund von Symmetrie für Übersetzungen mit $|\tau(!)| = 2$.
Angenommen $|\tau(?)| = 2$. Dann gibt es 16 Möglichkeiten, wie $\tau(?)$ aussehen könnte. Allerdings lassen sich vier davon direkt ausschließen, da bei den Kombinationen P_1T_1, P_1P_2, P_2T_2 und P_2P_1 der Prozess $P = ?1 \parallel ?1$, welcher must-divergent ist, übersetzt jeweils must-convergent ist.
Die restlichen zwölf Varianten lassen sich mit dem Suchverfahren alle widerlegen. Aufgrund von Symmetrie muss man hierbei aber nicht alle zwölf Varianten betrachten. Jeder Fall, in dem $\tau(?)$ mit P_2 beginnt, ist symmetrisch zu einem der Fälle, in denen $\tau(?)$ mit P_1 beginnt. Das Gleiche gilt für die Fälle, in denen $\tau(?)$ mit T_1 und T_2 beginnt. Es reicht also alle Fälle zu betrachten, in denen $\tau(?)$ mit P_1 oder T_1 beginnt. Da die Suche im Fall $|\tau(?)| = 2$ allerdings länger als im Fall $|\tau(?)| = 1$ ist, wird sie an dieser Stelle nicht ausführlich beschrieben. Alle durchgeführten Suchaufrufe sind in Anhang A zu finden. Festzuhalten bleibt an dieser Stelle als Ergebnis, dass es keine korrekte Übersetzung mit $|\tau(?)| = 2$ bzw. $|\tau(!)| = 2$ geben kann.

4.6. Übersetzungen mit $|\tau(?)| = 3$ bzw. $|\tau(!)| = 3$

Für den Fall, dass $|\tau(?)| = 3$ bzw. $|\tau(!)| = 3$ kann ebenfalls gezeigt werden, dass es keine korrekte Übersetzung gibt. Es genügt wieder, nur einen der beiden Fälle zu betrachten, da beide Fälle symmetrisch sind. Bei der Suche wird $|\tau(?)| = 3$ festgelegt. Insgesamt gibt es für $\tau(?)$ 64 Möglichkeiten. Allerdings können acht Möglichkeiten direkt ausgeschlossen werden, da der *SYNCSIMPLE*-Prozess $P = ?1 \parallel ?1$, welcher must-divergent ist, damit in einen *LOCKSIMPLE*-Prozess mit der Eigenschaft must-convergent übersetzt wird. Bei diesen acht Möglichkeiten handelt es sich um die folgenden Sequenzen:
$P_1T_1P_1$, $P_1T_1P_2$, $P_1P_2T_1$, $P_1P_2T_2$, $P_2P_1T_1$, $P_2P_1T_2$, $P_2T_2P_1$, $P_2T_2P_2$
Es bleiben also noch 56 Möglichkeiten übrig. Allerdings kann man sich wieder die Symmetrie zu Nutze machen, sodass diese nicht alle getestet werden müssen. Nur die Fälle, in denen $\tau(?)$ mit P_1 oder T_1 beginnt, müssen überprüft werden. Alle durchgeführten Suchaufrufe befinden sich im Anhang B. Die Suche ist in diesem Fall allerdings deutlich länger als bei den Fällen $|\tau(?)| = 1$ und $|\tau(?)| = 2$. Deswegen wurden hierbei neue Ausdrücke definiert, um *More* effizienter zu erweitern. Diese Ausdrücke werden im Folgenden gezeigt:

Erweiterung von More

```
1  expP1MoreLong = [Alt[[P1T1Plus],[P1T1Star,P1],[P1T1Plus,Alt[[T1],[T2],[P2]]],[P1T1Plus,Alt[
       expP2More,expT1More,expT2More]],[P1T1Star,P1,Alt[[P1],[P2],[T2]]],[P1T1Star,P1,Alt[
       expP1More,expP2More,expT2More]]]
2
3  expP1MoreLongAlt = [Alt[[P1,T1,P1T1Star],[P1T1Star,P1],[P1,T1,P1T1Star,Alt[[T1],[T2],[P2
       ]]],[P1,T1,P1T1Star,Alt[expP2MoreAlt,expT1MoreAlt,expT2MoreAlt]],[P1T1Star,P1,Alt[[P1
       ],[P2],[T2]]],[P1T1Star,P1,Alt[expP1MoreAlt,expP2MoreAlt,expT2MoreAlt]]]
4
5  expP2MoreLong = [Alt[[P2T2Plus],[P2T2Star,P2],[P2T2Plus,Alt[[T1],[T2],[P1]]],[P2T2Plus,Alt[
       expP1More,expT1More,expT2More]],[P2T2Star,P2,Alt[[P1],[P2],[T1]]],[P2T2Star,P2,Alt[
       expP1More,expP2More,expT1More]]]
6
7  expP2MoreLongAlt = [Alt[[P2,T2,P2T2Star],[P2T2Star,P2],[P2,T2,P2T2Star,Alt[[T1],[T2],[P1
       ]]],[P2,T2,P2T2Star,Alt[expP1MoreAlt,expT1MoreAlt,expT2MoreAlt]],[P2T2Star,P2,Alt[[P1
       ],[P2],[T1]]],[P2T2Star,P2,Alt[expP1MoreAlt,expP2MoreAlt,expT1MoreAlt]]]
8
9  expT1MoreLong = [Alt[[T1P1Plus],[T1P1Star,T1],[T1P1Plus,Alt[[P1],[P2],[T2]]],[T1P1Plus,Alt[
       expP1More,expP2More,expT2More]],[T1P1Star,T1,Alt[[T1],[T2],[P2]]],[T1P1Star,T1,Alt[
       expT1More,expT2More,expP2More]]]
10
11 expT1MoreLongAlt = [Alt[[T1,P1,T1P1Star],[T1P1Star,T1],[T1,P1,T1P1Star,Alt[[P1],[P2],[T2
       ]]],[T1,P1,T1P1Star,Alt[expP1MoreAlt,expP2MoreAlt,expT2MoreAlt]],[T1P1Star,T1,Alt[[T1
       ],[T2],[P2]]],[T1P1Star,T1,Alt[expT1MoreAlt,expT2MoreAlt,expP2MoreAlt]]]
12
13 expT2MoreLong = [Alt[[T2P2Plus],[T2P2Star,T2],[T2P2Plus,Alt[[P1],[P2],[T1]]],[T2P2Plus,Alt[
       expT1More,expP1More,expP2More]],[T2P2Star,T2,Alt[[T1],[T2],[P1]]],[T2P2Star,T2,Alt[
       expT1More,expT2More,expP1More]]]
14
15 expT2MoreLongAlt = [Alt[[T2,P2,T2P2Star],[T2P2Star,T2],[T2,P2,T2P2Star,Alt[[P1],[P2],[T1
       ]]],[T2,P2,T2P2Star,Alt[expT1MoreAlt,expP1MoreAlt,expP2MoreAlt]],[T2P2Star,T2,Alt[[T1
       ],[T2],[P1]]],[T2P2Star,T2,Alt[expT1MoreAlt,expT2MoreAlt,expP1MoreAlt]]]
```

Diese Ausdrücke erweitern P_1More, P_2More, T_1More und T_2More in gleich mehreren Schritten auf alle möglichen Arten. Dadurch kann die Suche um einige Schritte verkürzt werden. In den Definitionen werden unter anderem die vorher eingeführten Ausdrücke *expP1More*, *expP2More*, *expT1More* und *expT2More* verwendet. In den Ausdrücken mit der Endung Alt werden wieder alle Instanzen von $(P_iT_i)^+$ und $(T_iP_i)^+$ so dargestellt werden, dass das jeweils erste P_iT_i bzw. T_iP_i in zwei einzelnen Schritten dargestellt wird. Somit können auch hier Übersetzungen in einigen Fällen direkt widerlegt werden, welche sonst nicht widerlegt werden könnten. Auch hier wurde wieder mit der Funktion *isExpEqual* sichergestellt, dass keine Fehler bei dem Erweitern gemacht wurden.
Unter Verwendung dieser Ausdrücke konnten schließlich alle möglichen Übersetzungen mit $|\tau(?)| = 3$ widerlegt werden. Damit gilt, sowohl $\tau(?)$ als auch $\tau(!)$ müssen mindestens eine Länge von 4 besitzen.

5. Fazit

Das Ziel dieser Arbeit war es, eine Möglichkeit zur automatischen Überprüfung von Übersetzungen von nebenläufigen Prozessen mit synchroner Kommunikation in nebenläufige Prozesse mit asynchroner Kommunikation zu entwickeln, wobei bei der asynchronen Kommunikation blockierende Speicheroperationen verwendet werden, die auf zwei Speicherzellen wirken. Außerdem gilt dabei, dass alle der verwendeten Speicheroperationen blockierend sind. Damit sollte ermöglicht werden, dass man gezielt nach möglicherweise korrekten Übersetzungen suchen kann bzw. Übersetzungen auf effiziente Art und Weise widerlegen kann.
Dazu wurden zunächst die beiden Sprachen *SYNCSIMPLE* und *LOCKSIMPLE* beschrieben, welche jeweils ein vereinfachtes Modell zur Beschreibung von synchroner Kommunikation bzw. asynchroner Kommunikation unter den zuvor erwähnten Bedingungen in nebenläufigen Prozessen darstellen. Mit diesen Sprachen ist es möglich, das Konvergenzverhalten von nebenläufigen Prozessen zu untersuchen. Außerdem ist es damit möglich, Übersetzungen von einem Modell in das andere zu beschreiben.
Mit Hilfe der funktionalen Programmiersprache Haskell wurde schließlich ein Programm geschrieben, welches Übersetzungen von der Sprache *SYNCSIMPLE* in die Sprache *LOCKSIMPLE* automatisch überprüft und angibt, ob eine Übersetzung widerlegt werden kann. Außerdem gibt das Programm an, warum eine Übersetzung widerlegt wurde, sodass es nachvollziehbar ist. Damit ist es möglich, innerhalb kurzer Zeit viele verschiedene Übersetzungen zu überprüfen und Mengen an Übersetzungen nach möglichen korrekten Übersetzungen zu durchsuchen. Durch die Verwendung von regulären Ausdrücken ist es dabei möglich, Übersetzungen effizient zu beschreiben.
Unter Verwendung des Programms konnte in den durchgeführten Testläufen keine korrekte Übersetzung gefunden werden. Es konnte allerdings gezeigt werden, dass eine Übersetzung mindestens Länge 10 besitzen muss, um korrekt zu sein. Außerdem wurde gezeigt, dass es keine korrekten Übersetzungen geben kann, bei denen gilt $|\tau(?)| < 4$ oder $|\tau(!)| < 4$.
Ob es für den Fall, dass beide Speicheroperationen bei Verwendung von zwei Speicherzellen blockierend sind, eine korrekte Übersetzung gibt, ist weiterhin offen. Falls es keine korrekte Übersetzung gibt, müsste man dies zeigen, indem alle möglichen Übersetzungen widerlegt werden. Da es unendlich viele Übersetzungen gibt, müsste man zum Widerlegen aller Übersetzungen eine Möglichkeit finden, um die Übersetzungen anhand von Merkmalen in eine überschaubare Anzahl von Fällen einzuteilen. Diese Fälle lassen sich dann eventuell einzeln widerlegen.
Ein möglicher Ansatz zur Einteilung der Übersetzungen in verschiedene Fälle wurde bereits in der Arbeit [1] von Schmidt-Schauß und Sabel verwendet. Dazu wurden „Blocking Types“ beschrieben, die eine Fallunterscheidung unter den Übersetzungen ermöglichen. Das gleiche Prinzip lässt sich auch in dem Fall, dass beide Speicheroperationen blockierend sind, anwenden. Ob sich damit schließlich alle Übersetzungen widerlegen lassen, müsste in weiteren Arbeiten untersucht werden.

Literaturverzeichnis

[1] Manfred Schmidt-Schauß & David Sabel (2021): *Minimal Translations from Synchronous Communication to Synchronizing Locks (Extended Version)*, Computing Research Repository (CoRR) 2021.

[2] Manfred Schmidt-Schauß & David Sabel (2020): *On Impossibility of Simple Modular Translations of Concurrent Calculi*, präsentiert bei WPTE 2020.

[3] *regexpr-symbolic: Regular expressions via symbolic manipulation*, URL: https://hackage.haskell.org/package/regexpr-symbolic, Stand 21.10.2021.

Anhang

Anhangsverzeichnis

A. Widerlegung von Übersetzungen mit $|\tau(?)| = 2$

A.1. Übersetzungen mit $\tau(?) = P_1P_1$

```
1  test2_1_1 = testPattern $ map createTranslation [([P1,P1],[step,More])]
2  result2_1_1 = map createTranslation [([P1,P1] , [P2,More]),
3                                       ([P1,P1] , [T1,More])]
4  newCase2_1_1 = map createTranslation [([P1,P1] , expP2MoreAlt),
5                                        ([P1,P1] , expT1MoreAlt)]
6  check2_1_1 = equalTranslations newCase2_1_1 result2_1_1
7
8  test2_1_2 = testPattern newCase2_1_1
9  result2_1_2 = map createTranslation [([P1,P1] , [P2,T2,T1,More]),
10                                      ([P1,P1] , [P2,T2,P2T2Plus,T1,More]),
11                                      ([P1,P1] , [P2,T1,More]),
12                                      ([P1,P1] , [P2,T2,P2,T1,More]),
13                                      ([P1,P1] , [P2,T2,P2T2Plus,P2,T1,More]),
14                                      ([P1,P1] , [T1,P2,More])]
15 newCase2_1_2 = map createTranslation [([P1,P1] , [P2,T2]++expT1MoreAlt),
16                                       ([P1,P1] , [P2,T2,P2T2Plus]++expT1MoreAlt),
17                                       ([P1,P1] , [P2]++expT1MoreAlt),
18                                       ([P1,P1] , [P2,T2,P2]++expT1MoreAlt),
19                                       ([P1,P1] , [P2,T2,P2T2Plus,P2]++expT1MoreAlt),
20                                       ([P1,P1] , [T1]++expP2MoreAlt)]
21 check2_1_2 = equalTranslations newCase2_1_2 result2_1_2
22
23 test2_1_3 = testPattern newCase2_1_2
24 result2_1_3 = map createTranslation [([P1,P1] , [P2,T2,T1,P2,More]),
25                                      ([P1,P1] , [P2,T2,P2T2Plus,T1,P2,More]),
26                                      ([P1,P1] , [P2,T1,T1,More]),
27                                      ([P1,P1] , [P2,T1,T2,More]),
28                                      ([P1,P1] , [P2,T2,P2,T1,T1,More]),
29                                      ([P1,P1] , [P2,T2,P2,T1,T2,More]),
30                                      ([P1,P1] , [P2,T2,P2T2Plus,P2,T1,T1,More]),
31                                      ([P1,P1] , [P2,T2,P2T2Plus,P2,T1,T2,More])]
32 newCase2_1_3 = map createTranslation [([P1,P1] , [P2,T2,T1]++expP2MoreAlt),
33                                       ([P1,P1] , [P2,T2,P2T2Plus,T1]++expP2MoreAlt),
34                                       ([P1,P1] , [P2,T1,T1,someSteps]),
35                                       ([P1,P1] , [P2,T1,T2,someSteps]),
36                                       ([P1,P1] , [P2,T2,P2,T1,T1,someSteps]),
37                                       ([P1,P1] , [P2,T2,P2,T1,T2,someSteps]),
38                                       ([P1,P1] , [P2,T2,P2T2Plus,P2,T1,T1,someSteps]),
39                                       ([P1,P1] , [P2,T2,P2T2Plus,P2,T1,T2,someSteps])]
40 check2_1_3 = equalTranslations newCase2_1_3 result2_1_3
41
42 test2_1_4 = testPattern newCase2_1_3
```

A.2. Übersetzungen mit $\tau(?) = P_1T_2$

```
test2_2_1 = testPattern $ map createTranslation [([P1,T2],[step,More])]
result2_2_1 = map createTranslation [([P1,T2] , [P2,More]),
                                     ([P1,T2] , [T1,More])]
newCase2_2_1 = map createTranslation [([P1,T2] , expP2MoreAlt),
                                      ([P1,T2] , expT1MoreAlt)]
check2_2_1 = equalTranslations newCase2_2_1 result2_2_1

test2_2_2 = testPattern newCase2_2_1
result2_2_2 = map createTranslation [([P1,T2] , [P2,T1,More]),
                                     ([P1,T2] , [T1,P2,More])]
newCase2_2_2 = map createTranslation [([P1,T2] , [P2]++expT1More),
                                      ([P1,T2] , [T1]++expP2More)]
check2_2_2 = equalTranslations newCase2_2_2 result2_2_2

test2_2_3 = testPattern newCase2_2_2
```

A.3. Übersetzungen mit $\tau(?) = T_1P_1$

```
test2_3_1 = testPattern $ map createTranslation [([T1,P1],[step,More])]
result2_3_1 = map createTranslation [([T1,P1] , [P2,More])]
newCase2_3_1 = map createTranslation [([T1,P1] , expP2More)]
check2_3_1 = equalTranslations newCase2_3_1 result2_3_1

test2_3_2 = testPattern newCase2_3_1
```

A.4. Übersetzungen mit $\tau(?) = T_1P_2$

```
1  test2_4_1 = testPattern $ map createTranslation [([T1,P2],[step,More])]
2  result2_4_1 = map createTranslation [([T1,P2] , [P1,More]),
3                                       ([T1,P2] , [P2,More])]
4  newCase2_4_1 = map createTranslation [([T1,P2] , expP1MoreAlt),
5                                        ([T1,P2] , expP2More)]
6  check2_4_1 = equalTranslations newCase2_4_1 result2_4_1
7
8  test2_4_2 = testPattern newCase2_4_1
9  result2_4_2 = map createTranslation [([T1,P2] , [P1,T2,More]),
10                                      ([T1,P2] , [P2T2Plus,P1,More]),
11                                      ([T1,P2] , [P2,P1,More]),
12                                      ([T1,P2] , [P2T2Plus,P2,P1,More])]
13 newCase2_4_2 = map createTranslation [([T1,P2] , [P1]++expT2MoreAlt),
14                                       ([T1,P2] , [P2T2Plus]++expP1MoreAlt),
15                                       ([T1,P2] , [P2]++expP1MoreAlt),
16                                       ([T1,P2] , [P2T2Plus,P2]++expP1MoreAlt)]
17 check2_4_2 = equalTranslations newCase2_4_2 result2_4_2
18
19 test2_4_3 = testPattern newCase2_4_2
20 result2_4_3 = map createTranslation [([T1,P2] , [P2T2Plus,P1,T2]),
21                                      ([T1,P2] , [P2T2Plus,P1,T2,More]),
22                                      ([T1,P2] , [P2,P1,P1,More]),
23                                      ([T1,P2] , [P2,P1,T2,More]),
24                                      ([T1,P2] , [P2T2Plus,P2,P1,P1,More]),
25                                      ([T1,P2] , [P2T2Plus,P2,P1,T2,More])]
26 newCase2_4_3 = map createTranslation [([T1,P2] , [P2,T2,P2T2Star,P1,T2]),
27                                       ([T1,P2] , [P2,T2,P2T2Star,P1]++expT2MoreAlt),
28                                       ([T1,P2] , [P2,P1,P1,someSteps]),
29                                       ([T1,P2] , [P2,P1]++expT2MoreAlt),
30                                       ([T1,P2] , [P2T2Plus,P2,P1,P1,someSteps]),
31                                       ([T1,P2] , [P2T2Plus,P2,P1]++expT2MoreAlt)]
32 check2_4_3 = equalTranslations newCase2_4_3 result2_4_3
33
34 test2_4_4 = testPattern newCase2_4_3
35 result2_4_4 = map createTranslation [([T1,P2] , [P2,P1,T2,T2,More]),
36                                      ([T1,P2] , [P2T2Plus,P2,P1,T2,T2,More])]
37 newCase2_4_4 = map createTranslation [([T1,P2] , [P2,P1,T2]++expT2MoreAlt),
38                                       ([T1,P2] , [P2T2Plus,P2,P1,T2]++expT2MoreAlt)]
39 check2_4_4 = equalTranslations newCase2_4_4 result2_4_4
40
41 test2_4_5 = testPattern newCase2_4_4
```

A.5. Übersetzungen mit $\tau(?) = T_1T_1$

```
1  test2_5_1 = testPattern $ map createTranslation [([T1,T1],[step,More])]
2  result2_5_1 = map createTranslation [([T1,T1] , [P1,More]),
3                                       ([T1,T1] , [P2,More])]
4  newCase2_5_1 = map createTranslation [([T1,T1] , expP1MoreAlt),
5                                        ([T1,T1] , expP2More)]
6  check2_5_1 = equalTranslations newCase2_5_1 result2_5_1
7
8  test2_5_2 = testPattern newCase2_5_1
9  result2_5_2 = map createTranslation [([T1,T1] , [P1,P2,More]),
10                                      ([T1,T1] , [P2T2Plus,P1,More]),
11                                      ([T1,T1] , [P2,P1,More]),
12                                      ([T1,T1] , [P2T2Plus,P2,P1,More])]
13 newCase2_5_2 = map createTranslation [([T1,T1] , [P1]++expP2MoreAlt),
14                                       ([T1,T1] , [P2T2Plus]++expP1MoreAlt),
15                                       ([T1,T1] , [P2]++expP1MoreAlt),
16                                       ([T1,T1] , [P2T2Plus,P2]++expP1MoreAlt)]
17 check2_5_2 = equalTranslations newCase2_5_2 result2_5_2
18
19 test2_5_3 = testPattern newCase2_5_2
20 result2_5_3 = map createTranslation [([T1,T1] , [P2T2Plus,P1,P2,More]),
21                                      ([T1,T1] , [P2,P1,P1,More]),
22                                      ([T1,T1] , [P2,P1,T2,More]),
23                                      ([T1,T1] , [P2T2Plus,P2,P1,P1,More]),
24                                      ([T1,T1] , [P2T2Plus,P2,P1,T2,More])]
25 newCase2_5_3 = map createTranslation [([T1,T1] , [P2T2Plus,P1]++expP2MoreAlt),
26                                      ([T1,T1] , [P2,P1]++expP1MoreAlt),
27                                      ([T1,T1] , [P2,P1]++expT2MoreAlt),
28                                      ([T1,T1] , [P2T2Plus,P2,P1]++expP1MoreAlt),
29                                      ([T1,T1] , [P2T2Plus,P2,P1]++expT2MoreAlt)]
30 check2_5_3 = equalTranslations newCase2_5_3 result2_5_3
31
32 test2_5_4 = testPattern newCase2_5_3
33 result2_5_4 = map createTranslation [([T1,T1] , [P2,P1,P1,P1,More]),
34                                      ([T1,T1] , [P2,P1,P1,T2,More]),
35                                      ([T1,T1] , [P2T2Plus,P2,P1,P1,P1,More]),
36                                      ([T1,T1] , [P2T2Plus,P2,P1,P1,T2,More])]
37 newCase2_5_4 = map createTranslation [([T1,T1] , [P2,P1,P1,P1,someSteps]),
38                                       ([T1,T1] , [P2,P1,P1,T2,someSteps]),
39                                       ([T1,T1] , [P2T2Plus,P2,P1,P1,P1,someSteps]),
40                                       ([T1,T1] , [P2T2Plus,P2,P1,P1,T2,someSteps])]
41 check2_5_4 = equalTranslations newCase2_5_4 result2_5_4
42
43 test2_5_5 = testPattern newCase2_5_4
```

A.6. Übersetzungen mit $\tau(?) = T_1 T_2$

```
1  test2_6_1 = testPattern $ map createTranslation [([T1,T2],[step,More])]
2  result2_6_1 = map createTranslation [([T1,T2] , [P1,More]),
3                                       ([T1,T2] , [P2,More])]
4  newCase2_6_1 = map createTranslation [([T1,T2] , [P1,someSteps]),
5                                        ([T1,T2] , expP2More)]
6  check2_6_1 = equalTranslations newCase2_6_1 result2_6_1
7
8  test2_6_2 = testPattern newCase2_6_1
9  result2_6_2 = map createTranslation [([T1,T2] , [P2T2Plus,P1,More]),
10                                      ([T1,T2] , [P2,P1,More]),
11                                      ([T1,T2] , [P2T2Plus,P2,P1,More])]
12 newCase2_6_2 = map createTranslation [([T1,T2] , [P2T2Plus,P1,someSteps]),
13                                       ([T1,T2] , [P2,P1,someSteps]),
14                                       ([T1,T2] , [P2T2Plus,P2,P1,someSteps])]
15 check2_6_2 = equalTranslations newCase2_6_2 result2_6_2
16
17 test2_6_3 = testPattern newCase2_6_2
```

B. Widerlegung von Übersetzungen mit $|\tau(?)| = 3$

B.1. Übersetzungen mit $\tau(?) = P_1P_1P_1$

```
1  test3_1_1 = testPattern $ map createTranslation [([P1,P1,P1],[step,More])]
2  result3_1_1 = map createTranslation [([P1,P1,P1] , [P2,More]),
3                                       ([P1,P1,P1] , [T1,More])]
4  newCase3_1_1 = map createTranslation [([P1,P1,P1] , expP2MoreLongAlt),
5                                        ([P1,P1,P1] , expT1MoreLongAlt)]
6  check3_1_1 = equalTranslations result3_1_1 newCase3_1_1
7
8  test3_1_2 = testPattern newCase3_1_1
9  result3_1_2 = map createTranslation [([P1,P1,P1] , [P2,T2,T1,P2,More]),
10                                      ([P1,P1,P1] , [P2,T2,P2T2Plus,T1,P2,More]),
11                                      ([P1,P1,P1] , [P2,T1,T1,More]),
12                                      ([P1,P1,P1] , [P2T2Plus,P2,T1,T1,More]),
13                                      ([P1,P1,P1] , [P2,T1,T2,More]),
14                                      ([P1,P1,P1] , [P2T2Plus,P2,T1,T2,More])]
15 newCase3_1_2 = map createTranslation [([P1,P1,P1] , [P2,T2,T1]++expP2MoreAlt),
16                                       ([P1,P1,P1] , [P2,T2,P2T2Plus,T1]++expP2MoreAlt),
17                                       ([P1,P1,P1] , [P2,T1]++expT1MoreAlt),
18                                       ([P1,P1,P1] , [P2T2Plus,P2,T1]++expT1MoreAlt),
19                                       ([P1,P1,P1] , [P2,T1]++expT2MoreAlt),
20                                       ([P1,P1,P1] , [P2T2Plus,P2,T1]++expT2MoreAlt)]
21 check3_1_2 = equalTranslations result3_1_2 newCase3_1_2
22
23 test3_1_3 = testPattern newCase3_1_2
24 result3_1_3 = map createTranslation [([P1,P1,P1] , [P2,T1,T1,T1,More]),
25                                      ([P1,P1,P1] , [P2,T1,T1,T2,More]),
26                                      ([P1,P1,P1] , [P2T2Plus,P2,T1,T1,T1,More]),
27                                      ([P1,P1,P1] , [P2T2Plus,P2,T1,T1,T2,More])]
28 newCase3_1_3 = map createTranslation [([P1,P1,P1] , [P2,T1,T1,T1,someSteps]),
29                                      ([P1,P1,P1] , [P2,T1,T1,T2,someSteps]),
30                                      ([P1,P1,P1] , [P2T2Plus,P2,T1,T1,T1,someSteps]),
31                                      ([P1,P1,P1] , [P2T2Plus,P2,T1,T1,T2,someSteps])]
32 check3_1_3 = equalTranslations result3_1_3 newCase3_1_3
33
34 test3_1_4 = testPattern newCase3_1_3
```

B.2. Übersetzungen mit $\tau(?) = P_1 P_1 T_1$

```
test3_2_1 = testPattern $ map createTranslation [([P1,P1,T1],[step,More])]
result3_2_1 = map createTranslation [([P1,P1,T1] , [P2,More]),
                                     ([P1,P1,T1] , [T1,More])]
newCase3_2_1 = map createTranslation [([P1,P1,T1] , expP2MoreLongAlt),
                                      ([P1,P1,T1] , expT1MoreLongAlt)]
check3_2_1 = equalTranslations result3_2_1 newCase3_2_1

test3_2_2 = testPattern newCase3_2_1
result3_2_2 = map createTranslation [([P1,P1,T1] , [P2,T2,T1,P2,More]),
                                     ([P1,P1,T1] , [P2,T2,P2T2Plus,T1,P2,More]),
                                     ([P1,P1,T1] , [P2,T1,T2,More]),
                                     ([P1,P1,T1] , [P2T2Plus,P2,T1,T2,More])]
newCase3_2_2 = map createTranslation [([P1,P1,T1] , [P2,T2,T1]++expP2MoreAlt),
                                      ([P1,P1,T1] , [P2,T2,P2T2Plus,T1]++expP2MoreAlt),
                                      ([P1,P1,T1] , [P2,T1,T2,someSteps]),
                                      ([P1,P1,T1] , [P2T2Plus,P2,T1,T2,someSteps])]
check3_2_2 = equalTranslations result3_2_2 newCase3_2_2

test3_2_3 = testPattern newCase3_2_2
```

B.3. Übersetzungen mit $\tau(?) = P_1 P_1 P_2$

```
test3_3_1 = testPattern $ map createTranslation [([P1,P1,P2],[step,More])]
result3_3_1 = map createTranslation [([P1,P1,P2] , [P2,More]),
                                     ([P1,P1,P2] , [T1,More])]
newCase3_3_1 = map createTranslation [([P1,P1,P2] , expP2MoreLongAlt),
                                      ([P1,P1,P2] , expT1MoreLongAlt)]
check3_3_1 = equalTranslations result3_3_1 newCase3_3_1

test3_3_2 = testPattern newCase3_3_1
result3_3_2 = map createTranslation [([P1,P1,P2] , [P2,T1,T1,More]),
                                     ([P1,P1,P2] , [P2T2Plus,P2,T1,T1,More]),
                                     ([P1,P1,P2] , [P2,T1,T2,More]),
                                     ([P1,P1,P2] , [P2T2Plus,P2,T1,T2,More])]
newCase3_3_2 = map createTranslation [([P1,P1,P2] , [P2,T1,T1,someSteps]),
                                      ([P1,P1,P2] , [P2T2Plus,P2,T1,T1,someSteps]),
                                      ([P1,P1,P2] , [P2,T1,T2,someSteps]),
                                      ([P1,P1,P2] , [P2T2Plus,P2,T1,T2,someSteps])]
check3_3_2 = equalTranslations result3_3_2 newCase3_3_2

test3_3_3 = testPattern newCase3_3_2
```

B.4. Übersetzungen mit $\tau(?) = P_1 P_1 T_2$

```
1  test3_4_1 = testPattern $ map createTranslation [([P1,P1,T2],[step,More])]
2  result3_4_1 = map createTranslation [([P1,P1,T2] , [P2,More]),
3                                       ([P1,P1,T2] , [T1,More])]
4  newCase3_4_1 = map createTranslation [([P1,P1,T2] , expP2MoreLongAlt),
5                                        ([P1,P1,T2] , expT1MoreLongAlt)]
6  check3_4_1 = equalTranslations result3_4_1 newCase3_4_1
7
8  test3_4_2 = testPattern newCase3_4_1
9  result3_4_2 = map createTranslation [([P1,P1,T2] , [P2,T2,T1,P2,More]),
10                                      ([P1,P1,T2] , [P2,T2,P2T2Plus,T1,P2,More])]
11 newCase3_4_2 = map createTranslation [([P1,P1,T2] , [P2,T2,T1,P2,someSteps]),
12                                       ([P1,P1,T2] , [P2,T2,P2T2Plus,T1,P2,someSteps])]
13 check3_4_2 = equalTranslations result3_4_2 newCase3_4_2
14
15 test3_4_3 = testPattern newCase3_4_2
```

B.5. Übersetzungen mit $\tau(?) = P_1 T_1 T_1$

```
1  test3_5_1 = testPattern $ map createTranslation [([P1,T1,T1],[step,More])]
```

B.6. Übersetzungen mit $\tau(?) = P_1T_1T_2$

```
1  test3_6_1 = testPattern $ map createTranslation [([P1,T1,T2],[step,More])]
2  result3_6_1 = map createTranslation [([P1,T1,T2] , [P1,More]),
3                                       ([P1,T1,T2] , [P2,More])]
4  newCase3_6_1 = map createTranslation [([P1,T1,T2] , expP1MoreLongAlt),
5                                        ([P1,T1,T2] , expP2MoreLongAlt)]
6  check3_6_1 = equalTranslations result3_6_1 newCase3_6_1
7
8  test3_6_2 = testPattern newCase3_6_1
9  result3_6_2 = map createTranslation [([P1,T1,T2] , [P1,T1,P2,P1,More]),
10                                      ([P1,T1,T2] , [P1,T1,P1T1Plus,P2,P1,More]),
11                                      ([P1,T1,T2] , [P1,P2,T1,More]),
12                                      ([P1,T1,T2] , [P1T1Plus,P1,P2,T1,More])]
13 newCase3_6_2 = map createTranslation [([P1,T1,T2] , [P1,T1,P2]++expP1MoreAlt),
14                                       ([P1,T1,T2] , [P1,T1,P1T1Plus,P2]++expP1MoreAlt),
15                                       ([P1,T1,T2] , [P1,P2,T1,someSteps]),
16                                       ([P1,T1,T2] , [P1T1Plus,P1,P2,T1,someSteps])]
17 check3_6_2 = equalTranslations result3_6_2 newCase3_6_2
18
19 test3_6_3 = testPattern newCase3_6_2
```

B.7. Übersetzungen mit $\tau(?) = P_1P_2P_1$

```
1  test3_7_1 = testPattern $ map createTranslation [([P1,P2,P1],[step,More])]
2  result3_7_1 = map createTranslation [([P1,P2,P1] , [T1,More]),
3                                       ([P1,P2,P1] , [T2,More])]
4  newCase3_7_1 = map createTranslation [([P1,P2,P1] , expT1MoreLongAlt),
5                                        ([P1,P2,P1] , expT2MoreLongAlt)]
6  check3_7_1 = equalTranslations result3_7_1 newCase3_7_1
7
8  test3_7_2 = testPattern newCase3_7_1
9  result3_7_2 = map createTranslation [([P1,P2,P1] , [T2,P2,T1,T2,More]),
10                                      ([P1,P2,P1] , [T2,P2,T2P2Plus,T1,T2,More]),
11                                      ([P1,P2,P1] , [T2,T1,T1,More]),
12                                      ([P1,P2,P1] , [T2P2Plus,T2,T1,T1,More])]
13 newCase3_7_2 = map createTranslation [([P1,P2,P1] , [T2,P2,T1,T2,someSteps]),
14                                       ([P1,P2,P1] , [T2,P2,T2P2Plus,T1,T2,someSteps]),
15                                       ([P1,P2,P1] , [T2,T1,T1,someSteps]),
16                                       ([P1,P2,P1] , [T2P2Plus,T2,T1,T1,someSteps])]
17 check3_7_2 = equalTranslations result3_7_2 newCase3_7_2
18
19 test3_7_3 = testPattern newCase3_7_2
```

B.8. Übersetzungen mit $\tau(?) = P_1P_2P_2$

```
test3_8_1 = testPattern $ map createTranslation [([P1,P2,P2],[step,More])]
result3_8_1 = map createTranslation [([P1,P2,P2] , [T1,More]),
                                     ([P1,P2,P2] , [T2,More])]
newCase3_8_1 = map createTranslation [([P1,P2,P2] , expT1MoreLongAlt),
                                      ([P1,P2,P2] , expT2MoreLongAlt)]
check3_8_1 = equalTranslations result3_8_1 newCase3_8_1

test3_8_2 = testPattern newCase3_8_1
result3_8_2 = map createTranslation [([P1,P2,P2] , [T1,T2,T2,More]),
                                     ([P1,P2,P2] , [T1P1Plus,T1,T2,T2,More])]
newCase3_8_2 = map createTranslation [([P1,P2,P2] , [T1,T2,T2,someSteps]),
                                      ([P1,P2,P2] , [T1P1Plus,T1,T2,T2,someSteps])]
check3_8_2 = equalTranslations result3_8_2 newCase3_8_2

test3_8_3 = testPattern newCase3_8_2
```

B.9. Übersetzungen mit $\tau(?) = P_1T_2P_1$

```
test3_9_1 = testPattern $ map createTranslation [([P1,T2,P1],[step,More])]
result3_9_1 = map createTranslation [([P1,T2,P1] , [P2,More]),
                                     ([P1,T2,P1] , [T1,More])]
newCase3_9_1 = map createTranslation [([P1,T2,P1] , expP2MoreLongAlt),
                                      ([P1,T2,P1] , expT1MoreLongAlt)]
check3_9_1 = equalTranslations result3_9_1 newCase3_9_1

test3_9_2 = testPattern newCase3_9_1
```

B.10. Übersetzungen mit $\tau(?) = P_1T_2T_1$

```
1  test3_10_1 = testPattern $ map createTranslation [([P1,T2,T1],[step,More])]
2  result3_10_1 = map createTranslation [([P1,T2,T1] , [P2,More]),
3                                        ([P1,T2,T1] , [T1,More])]
4  newCase3_10_1 = map createTranslation [([P1,T2,T1] , expP2MoreLongAlt),
5                                         ([P1,T2,T1] , expT1MoreLongAlt)]
6  check3_10_1 = equalTranslations result3_10_1 newCase3_10_1
7
8  test3_10_2 = testPattern newCase3_10_1
9  result3_10_2 = map createTranslation [([P1,T2,T1] , [T1,P2,P1,More]),
10                                       ([P1,T2,T1] , [T1P1Plus,T1,P2,P1,More]),
11                                       ([P1,T2,T1] , [T1,P2,P2,More]),
12                                       ([P1,T2,T1] , [T1P1Plus,T1,P2,P2,More])]
13 newCase3_10_2 = map createTranslation [([P1,T2,T1] , [T1,P2,P1,someSteps]),
14                                        ([P1,T2,T1] , [T1P1Plus,T1,P2,P1,someSteps]),
15                                        ([P1,T2,T1] , [T1,P2,P2,someSteps]),
16                                        ([P1,T2,T1] , [T1P1Plus,T1,P2,P2,someSteps])]
17 check3_10_2 = equalTranslations result3_10_2 newCase3_10_2
18
19 test3_10_3 = testPattern newCase3_10_2
```

B.11. Übersetzungen mit $\tau(?) = P_1T_2P_2$

```
1  test3_11_1 = testPattern $ map createTranslation [([P1,T2,P2],[step,More])]
2  result3_11_1 = map createTranslation [([P1,T2,P2] , [P2,More]),
3                                        ([P1,T2,P2] , [T1,More])]
4  newCase3_11_1 = map createTranslation [([P1,T2,P2] , expP2MoreLongAlt),
5                                         ([P1,T2,P2] , expT1MoreLongAlt)]
6  check3_11_1 = equalTranslations result3_11_1 newCase3_11_1
7
8  test3_11_2 = testPattern newCase3_11_1
```

B.12. Übersetzungen mit $\tau(?) = P_1T_2T_2$

```
1  test3_12_1 = testPattern $ map createTranslation [([P1,T2,T2],[step,More])]
2  result3_12_1 = map createTranslation [([P1,T2,T2] , [P2,More]),
3                                        ([P1,T2,T2] , [T1,More])]
4  newCase3_12_1 = map createTranslation [([P1,T2,T2] , expP2MoreLongAlt),
5                                         ([P1,T2,T2] , expT1MoreLongAlt)]
6  check3_12_1 = equalTranslations result3_12_1 newCase3_12_1
7
8  test3_12_2 = testPattern newCase3_12_1
9  result3_12_2 = map createTranslation [([P1,T2,T2] , [T1,P2,P2,More]),
10                                       ([P1,T2,T2] , [T1P1Plus,T1,P2,P2,More])]
11 newCase3_12_2 = map createTranslation [([P1,T2,T2] , [T1,P2,P2,someSteps]),
12                                        ([P1,T2,T2] , [T1P1Plus,T1,P2,P2,someSteps])]
13 check3_12_2 = equalTranslations result3_12_2 newCase3_12_2
14
15 test3_12_3 = testPattern newCase3_12_2
```

B.13. Übersetzungen mit $\tau(?) = T_1P_1P_1$

```
1  test3_13_1 = testPattern $ map createTranslation [([T1,P1,P1],[step,More])]
2  result3_13_1 = map createTranslation [([T1,P1,P1] , [P1,More]),
3                                        ([T1,P1,P1] , [P2,More])]
4  newCase3_13_1 = map createTranslation [([T1,P1,P1] , expP1MoreLongAlt),
5                                         ([T1,P1,P1] , expP2MoreLongAlt)]
6  check3_13_1 = equalTranslations result3_13_1 newCase3_13_1
7
8  test3_13_2 = testPattern newCase3_13_1
9  result3_13_2 = map createTranslation [([T1,P1,P1] , [P1,P2,T2,T1,More]),
10                                       ([T1,P1,P1] , [P1T1Plus,P1,P2,T2,T1,More]),
11                                       ([T1,P1,P1] , [P1,P2,T2,P2T2Plus,T1,More]),
12                                       ([T1,P1,P1] , [P1T1Plus,P1,P2,T2,P2T2Plus,T1,More]),
13                                       ([T1,P1,P1] , [P1,P2,T1,More]),
14                                       ([T1,P1,P1] , [P1T1Plus,P1,P2,T1,More]),
15                                       ([T1,P1,P1] , [P1,P2,T2,P2,T1,More]),
16                                       ([T1,P1,P1] , [P1T1Plus,P1,P2,T2,P2,T1,More]),
17                                       ([T1,P1,P1] , [P1,P2,T2,P2T2Plus,P2,T1,More]),
18                                       ([T1,P1,P1] , [P1T1Plus,P1,P2,T2,P2T2Plus,P2,T1,More])
                                              ,
19                                       ([T1,P1,P1] , [P2,T2,P1,T1,P2,More]),
20                                       ([T1,P1,P1] , [P2,T2,P2T2Plus,P1,T1,P2,More]),
21                                       ([T1,P1,P1] , [P2,T2,P1,P2,More]),
22                                       ([T1,P1,P1] , [P2,T2,P2T2Plus,P1,P2,More]),
23                                       ([T1,P1,P1] , [P2,P1,T1,T2,More]),
24                                       ([T1,P1,P1] , [P2T2Plus,P2,P1,T1,T2,More]),
25                                       ([T1,P1,P1] , [P2,P1,T2,More]),
26                                       ([T1,P1,P1] , [P2T2Plus,P2,P1,T2,More])]
```

```
27  newCase3_13_2 = map createTranslation [([T1,P1,P1] , [P1,P2,T2]++expT1MoreLongAlt),
28                                         ([T1,P1,P1] , [P1T1Plus,P1,P2,T2]++expT1MoreLongAlt),
29                                         ([T1,P1,P1] , [P1,P2,T2,P2T2Plus]++expT1MoreLongAlt),
30                                         ([T1,P1,P1] , [P1T1Plus,P1,P2,T2,P2T2Plus]++
                                               expT1MoreLongAlt),
31                                         ([T1,P1,P1] , [P1,P2]++expT1MoreLongAlt),
32                                         ([T1,P1,P1] , [P1T1Plus,P1,P2]++expT1MoreLongAlt),
33                                         ([T1,P1,P1] , [P1,P2,T2,P2]++expT1MoreLongAlt),
34                                         ([T1,P1,P1] , [P1T1Plus,P1,P2,T2,P2]++
                                               expT1MoreLongAlt),
35                                         ([T1,P1,P1] , [P1,P2,T2,P2T2Plus,P2]++
                                               expT1MoreLongAlt),
36                                         ([T1,P1,P1] , [P1T1Plus,P1,P2,T2,P2T2Plus,P2]++
                                               expT1MoreLongAlt),
37                                         ([T1,P1,P1] , [P2,T2,P1,T1]++expP2MoreLongAlt),
38                                         ([T1,P1,P1] , [P2,T2,P2T2Plus,P1,T1]++
                                               expP2MoreLongAlt),
39                                         ([T1,P1,P1] , [P2,T2,P1]++expP2More),
40                                         ([T1,P1,P1] , [P2,T2,P2T2Plus,P1]++expP2More),
41                                         ([T1,P1,P1] , [P2,P1,T1]++expT2MoreLongAlt),
42                                         ([T1,P1,P1] , [P2T2Plus,P2,P1,T1]++expT2MoreLongAlt),
43                                         ([T1,P1,P1] , [P2,P1]++expT2MoreLongAlt),
44                                         ([T1,P1,P1] , [P2T2Plus,P2,P1]++expT2MoreLongAlt)]
45  check3_13_2 = equalTranslations result3_13_2 newCase3_13_2
46
47  test3_13_3 = testPattern newCase3_13_2
48  result3_13_3 = map createTranslation [([T1,P1,P1] , [P2,T2,P1,P2T2Plus,T1,More]),
49                                        ([T1,P1,P1] , [P2,T2,P1,P2,T1,More]),
50                                        ([T1,P1,P1] , [P2,T2,P1,P2T2Plus,P2,T1,More]),
51                                        ([T1,P1,P1] , [P2,T2,P2T2Plus,P1,P2T2Plus,T1,More]),
52                                        ([T1,P1,P1] , [P2,T2,P2T2Plus,P1,P2,T1,More]),
53                                        ([T1,P1,P1] , [P2,T2,P2T2Plus,P1,P2T2Plus,P2,T1,More])
                                            ]
54  newCase3_13_3 = map createTranslation [([T1,P1,P1] , [P2,T2,P1,P2T2Plus]++expT1MoreLongAlt),
55                                         ([T1,P1,P1] , [P2,T2,P1,P2]++expT1MoreLongAlt),
56                                         ([T1,P1,P1] , [P2,T2,P1,P2T2Plus,P2]++
                                               expT1MoreLongAlt),
57                                         ([T1,P1,P1] , [P2,T2,P2T2Plus,P1,P2T2Plus]++
                                               expT1MoreLongAlt),
58                                         ([T1,P1,P1] , [P2,T2,P2T2Plus,P1,P2]++
                                               expT1MoreLongAlt),
59                                         ([T1,P1,P1] , [P2,T2,P2T2Plus,P1,P2T2Plus,P2]++
                                               expT1MoreLongAlt)]
60  check3_13_3 = equalTranslations result3_13_3 newCase3_13_3
61
62  test3_13_4 = testPattern newCase3_13_3
```

B.14. Übersetzungen mit $\tau(?) = T_1P_1T_1$

```
1  test3_14_1 = testPattern $ map createTranslation [([T1,P1,T1],[step,More])]
2  result3_14_1 = map createTranslation [([T1,P1,T1] , [P1,More]),
3                                        ([T1,P1,T1] , [P2,More])]
4  newCase3_14_1 = map createTranslation [([T1,P1,T1] , expP1MoreLongAlt),
5                                         ([T1,P1,T1] , expP2MoreLongAlt)]
6  check3_14_1 = equalTranslations result3_14_1 newCase3_14_1
7
8  test3_14_2 = testPattern newCase3_14_1
9  result3_14_2 = map createTranslation [([T1,P1,T1] , [P2,T2,P1,P2,More]),
10                                       ([T1,P1,T1] , [P2,T2,P2T2Plus,P1,P2,More]),
11                                       ([T1,P1,T1] , [P2,P1,T2,More]),
12                                       ([T1,P1,T1] , [P2T2Plus,P2,P1,T2,More])]
13 newCase3_14_2 = map createTranslation [([T1,P1,T1] , [P2,T2,P1]++expP2MoreAlt),
14                                        ([T1,P1,T1] , [P2,T2,P2T2Plus,P1]++expP2MoreAlt),
15                                        ([T1,P1,T1] , [P2,P1,T2,someSteps]),
16                                        ([T1,P1,T1] , [P2T2Plus,P2,P1,T2,someSteps])]
17 check3_14_2 = equalTranslations result3_14_2 newCase3_14_2
18
19 test3_14_3 = testPattern newCase3_14_2
```

B.15. Übersetzungen mit $\tau(?) = T_1P_1P_2$

```
1  test3_15_1 = testPattern $ map createTranslation [([T1,P1,P2],[step,More])]
2  result3_15_1 = map createTranslation [([T1,P1,P2] , [P1,More]),
3                                        ([T1,P1,P2] , [P2,More])]
4  newCase3_15_1 = map createTranslation [([T1,P1,P2] , expP1MoreLongAlt),
5                                         ([T1,P1,P2] , expP2MoreLongAlt)]
6  check3_15_1 = equalTranslations result3_15_1 newCase3_15_1
7
8  test3_15_2 = testPattern newCase3_15_1
9  result3_15_2 = map createTranslation [([T1,P1,P2] , [P2,P1,T1,T2,More]),
10                                       ([T1,P1,P2] , [P2T2Plus,P2,P1,T1,T2,More]),
11                                       ([T1,P1,P2] , [P2,P1,T1,P1T1Plus,T2,More]),
12                                       ([T1,P1,P2] , [P2T2Plus,P2,P1,T1,P1T1Plus,T2,More]),
13                                       ([T1,P1,P2] , [P2,P1,T2,More]),
14                                       ([T1,P1,P2] , [P2T2Plus,P2,P1,T2,More]),
15                                       ([T1,P1,P2] , [P2,P1,T1,P1,T2,More]),
16                                       ([T1,P1,P2] , [P2T2Plus,P2,P1,T1,P1,T2,More]),
17                                       ([T1,P1,P2] , [P2,P1,T1,P1T1Plus,P1,T2,More]),
18                                       ([T1,P1,P2] , [P2T2Plus,P2,P1,T1,P1T1Plus,P1,T2,More])
                                             ]
19 newCase3_15_2 = map createTranslation [([T1,P1,P2] , [P2,P1,T1]++expT2MoreLongAlt),
20                                        ([T1,P1,P2] , [P2T2Plus,P2,P1,T1]++expT2MoreLongAlt),
21                                        ([T1,P1,P2] , [P2,P1,T1,P1T1Plus]++expT2MoreLongAlt),
22                                        ([T1,P1,P2] , [P2T2Plus,P2,P1,T1,P1T1Plus]++
                                              expT2MoreLongAlt),
23                                        ([T1,P1,P2] , [P2,P1]++expT2MoreLongAlt),
24                                        ([T1,P1,P2] , [P2T2Plus,P2,P1]++expT2MoreLongAlt),
25                                        ([T1,P1,P2] , [P2,P1,T1,P1]++expT2MoreLongAlt),
26                                        ([T1,P1,P2] , [P2T2Plus,P2,P1,T1,P1]++
                                              expT2MoreLongAlt),
27                                        ([T1,P1,P2] , [P2,P1,T1,P1T1Plus,P1]++
                                              expT2MoreLongAlt),
28                                        ([T1,P1,P2] , [P2T2Plus,P2,P1,T1,P1T1Plus,P1]++
                                              expT2MoreLongAlt)]
29 check3_15_2 = equalTranslations result3_15_2 newCase3_15_2
30
31 test3_15_3 = testPattern newCase3_15_2
```

B.16. Übersetzungen mit $\tau(?) = T_1 P_1 T_2$

```
1  test3_16_1 = testPattern $ map createTranslation [([T1,P1,T2],[step,More])]
2  result3_16_1 = map createTranslation [([T1,P1,T2] , [P1,More]),
3                                        ([T1,P1,T2] , [P2,More])]
4  newCase3_16_1 = map createTranslation [([T1,P1,T2] , expP1MoreAlt),
5                                         ([T1,P1,T2] , expP2More)]
6  check3_16_1 = equalTranslations result3_16_1 newCase3_16_1
7
8  test3_16_2 = testPattern newCase3_16_1
9  result3_16_2 = map createTranslation [([T1,P1,T2] , [P1,T1,P2,More]),
10                                       ([T1,P1,T2] , [P1,T1,P1T1Plus,P2,More]),
11                                       ([T1,P1,T2] , [P1,P2,More]),
12                                       ([T1,P1,T2] , [P1,T1,P1,P2,More]),
13                                       ([T1,P1,T2] , [P1,T1,P1T1Plus,P1,P2,More]),
14                                       ([T1,P1,T2] , [P2T2Plus,P1,More]),
15                                       ([T1,P1,T2] , [P2,P1,More]),
16                                       ([T1,P1,T2] , [P2T2Plus,P2,P1,More])]
17 newCase3_16_2 = map createTranslation [([T1,P1,T2] , [P1,T1]++expP2MoreLongAlt),
18                                        ([T1,P1,T2] , [P1,T1,P1T1Plus]++expP2MoreLongAlt),
19                                        ([T1,P1,T2] , [P1,P2,someSteps]),
20                                        ([T1,P1,T2] , [P1,T1,P1,P2,someSteps]),
21                                        ([T1,P1,T2] , [P1,T1,P1T1Plus,P1,P2,someSteps]),
22                                        ([T1,P1,T2] , [P2T2Plus]++expP1MoreLongAlt),
23                                        ([T1,P1,T2] , [P2]++expP1MoreAlt),
24                                        ([T1,P1,T2] , [P2T2Plus,P2]++expP1MoreAlt)]
25 check3_16_2 = equalTranslations result3_16_2 newCase3_16_2
26
27 test3_16_3 = testPattern newCase3_16_2
28 result3_16_3 = map createTranslation [([T1,P1,T2] , [P2T2Plus,P1,T1,P2,P1,More]),
29                                       ([T1,P1,T2] , [P2T2Plus,P1,T1,P1T1Plus,P2,P1,More]),
30                                       ([T1,P1,T2] , [P2T2Plus,P1,P2,T1,More]),
31                                       ([T1,P1,T2] , [P2T2Plus,P1T1Plus,P1,P2,T1,More])]
32 newCase3_16_3 = map createTranslation [([T1,P1,T2] , [P2T2Plus,P1,T1,P2]++expP1MoreAlt),
33                                        ([T1,P1,T2] , [P2T2Plus,P1,T1,P1T1Plus,P2]++
                                              expP1MoreAlt),
34                                        ([T1,P1,T2] , [P2T2Plus,P1,P2,T1,someSteps]),
35                                        ([T1,P1,T2] , [P2T2Plus,P1T1Plus,P1,P2,T1,someSteps])
                                              ]
36 check3_16_3 = equalTranslations result3_16_3 newCase3_16_3
37
38 test3_16_4 = testPattern newCase3_16_3
```

B.17. Übersetzungen mit $\tau(?) = T_1T_1P_1$

```
1  test3_17_1 = testPattern $ map createTranslation [([T1,T1,P1],[step,More])]
2  result3_17_1 = map createTranslation [([T1,T1,P1] , [P1,More]),
3                                        ([T1,T1,P1] , [P2,More])]
4  newCase3_17_1 = map createTranslation [([T1,T1,P1] , expP1MoreLongAlt),
5                                         ([T1,T1,P1] , expP2MoreLongAlt)]
6  check3_17_1 = equalTranslations result3_17_1 newCase3_17_1
7
8  test3_17_2 = testPattern newCase3_17_1
9  result3_17_2 = map createTranslation [([T1,T1,P1] , [P2,T2,P1,P2,More]),
10                                       ([T1,T1,P1] , [P2,T2,P2T2Plus,P1,P2,More]),
11                                       ([T1,T1,P1] , [P2,P1,P1,More]),
12                                       ([T1,T1,P1] , [P2T2Plus,P2,P1,P1,More]),
13                                       ([T1,T1,P1] , [P2,P1,T2,More]),
14                                       ([T1,T1,P1] , [P2T2Plus,P2,P1,T2,More])]
15 newCase3_17_2 = map createTranslation [([T1,T1,P1] , [P2,T2,P1]++expP2MoreAlt),
16                                        ([T1,T1,P1] , [P2,T2,P2T2Plus,P1]++expP2MoreAlt),
17                                        ([T1,T1,P1] , [P2,P1,P1,someSteps]),
18                                        ([T1,T1,P1] , [P2T2Plus,P2,P1,P1,someSteps]),
19                                        ([T1,T1,P1] , [P2,P1,T2,someSteps]),
20                                        ([T1,T1,P1] , [P2T2Plus,P2,P1,T2,someSteps])]
21 check3_17_2 = equalTranslations result3_17_2 newCase3_17_2
22
23 test3_17_3 = testPattern newCase3_17_2
```

B.18. Übersetzungen mit $\tau(?) = T_1T_1T_1$

```
1  test3_18_1 = testPattern $ map createTranslation [([T1,T1,T1],[step,More])]
2  result3_18_1 = map createTranslation [([T1,T1,T1] , [P1,More]),
3                                        ([T1,T1,T1] , [P2,More])]
4  newCase3_18_1 = map createTranslation [([T1,T1,T1] , expP1MoreAlt),
5                                         ([T1,T1,T1] , expP2MoreAlt)]
6  check3_18_1 = equalTranslations result3_18_1 newCase3_18_1
7
8  test3_18_2 = testPattern newCase3_18_1
9  result3_18_2 = map createTranslation [([T1,T1,T1] , [P1,P1,More]),
10                                       ([T1,T1,T1] , [P1,P2,More]),
11                                       ([T1,T1,T1] , [P2,T2,P1,More]),
12                                       ([T1,T1,T1] , [P2,T2,P2T2Plus,P1,More]),
13                                       ([T1,T1,T1] , [P2,P1,More]),
14                                       ([T1,T1,T1] , [P2,T2,P2,P1,More]),
15                                       ([T1,T1,T1] , [P2,T2,P2T2Plus,P2,P1,More])]
16 newCase3_18_2 = map createTranslation [([T1,T1,T1] , [P1]++expP1MoreLongAlt),
17                                        ([T1,T1,T1] , [P1]++expP2More),
18                                        ([T1,T1,T1] , [P2,T2]++expP1MoreAlt),
19                                        ([T1,T1,T1] , [P2,T2,P2T2Plus]++expP1MoreAlt),
20                                        ([T1,T1,T1] , [P2]++expP1MoreAlt),
21                                        ([T1,T1,T1] , [P2,T2,P2]++expP1MoreAlt),
22                                        ([T1,T1,T1] , [P2,T2,P2T2Plus,P2]++expP1MoreAlt)]
23 check3_18_2 = equalTranslations result3_18_2 newCase3_18_2
24
25 test3_18_3 = testPattern newCase3_18_2
26 result3_18_3 = map createTranslation [([T1,T1,T1] , [P1,P2T2Plus,P1,More]),
27                                       ([T1,T1,T1] , [P1,P2,P1,More]),
28                                       ([T1,T1,T1] , [P1,P2T2Plus,P2,P1,More]),
29                                       ([T1,T1,T1] , [P2,T2,P1,P1,More]),
30                                       ([T1,T1,T1] , [P2,T2,P1,P2,More]),
31                                       ([T1,T1,T1] , [P2,T2,P2T2Plus,P1,P1,More]),
32                                       ([T1,T1,T1] , [P2,T2,P2T2Plus,P1,P2,More]),
33                                       ([T1,T1,T1] , [P2,P1,P1,More]),
34                                       ([T1,T1,T1] , [P2,P1,T2,More]),
35                                       ([T1,T1,T1] , [P2,T2,P2,P1,P1,More]),
36                                       ([T1,T1,T1] , [P2,T2,P2,P1,T2,More]),
37                                       ([T1,T1,T1] , [P2,T2,P2T2Plus,P2,P1,P1,More]),
38                                       ([T1,T1,T1] , [P2,T2,P2T2Plus,P2,P1,T2,More])]
```

```
39  newCase3_18_3 = map createTranslation [([T1,T1,T1] , [P1,P2T2Plus]++expP1MoreLongAlt),
40                                         ([T1,T1,T1] , [P1,P2,P1,someSteps]),
41                                         ([T1,T1,T1] , [P1,P2T2Plus,P2,P1,someSteps]),
42                                         ([T1,T1,T1] , [P2,T2,P1]++expP1MoreLongAlt),
43                                         ([T1,T1,T1] , [P2,T2,P1]++expP2More),
44                                         ([T1,T1,T1] , [P2,T2,P2T2Plus,P1]++expP1MoreLongAlt),
45                                         ([T1,T1,T1] , [P2,T2,P2T2Plus,P1]++expP2More),
46                                         ([T1,T1,T1] , [P2,P1]++expP1MoreLongAlt),
47                                         ([T1,T1,T1] , [P2,P1]++expT2MoreLongAlt),
48                                         ([T1,T1,T1] , [P2,T2,P2,P1]++expP1MoreLongAlt),
49                                         ([T1,T1,T1] , [P2,T2,P2,P1]++expT2MoreLongAlt),
50                                         ([T1,T1,T1] , [P2,T2,P2T2Plus,P2,P1]++
                                               expP1MoreLongAlt),
51                                         ([T1,T1,T1] , [P2,T2,P2T2Plus,P2,P1]++
                                               expT2MoreLongAlt)]
52  check3_18_3 = equalTranslations result3_18_3 newCase3_18_3
53
54  test3_18_4 = testPattern newCase3_18_3
55  result3_18_4 = map createTranslation [([T1,T1,T1] , [P2,T2,P1,P2T2Plus,P1,More]),
56                                        ([T1,T1,T1] , [P2,T2,P1,P2,P1,More]),
57                                        ([T1,T1,T1] , [P2,T2,P1,P2T2Plus,P2,P1,More]),
58                                        ([T1,T1,T1] , [P2,T2,P2T2Plus,P1,P2T2Plus,P1,More]),
59                                        ([T1,T1,T1] , [P2,T2,P2T2Plus,P1,P2,P1,More]),
60                                        ([T1,T1,T1] , [P2,T2,P2T2Plus,P1,P2T2Plus,P2,P1,More])
                                              ,
61                                        ([T1,T1,T1] , [P2,P1,P1,P1,P1,More]),
62                                        ([T1,T1,T1] , [P2,P1,P1T1Plus,P1,P1,P1,More]),
63                                        ([T1,T1,T1] , [P2,P1,P1,P1,T2,More]),
64                                        ([T1,T1,T1] , [P2,P1,P1T1Plus,P1,P1,T2,More]),
65                                        ([T1,T1,T1] , [P2,P1,T2,P2,P1,T2,More]),
66                                        ([T1,T1,T1] , [P2,P1,T2,P2,T2P2Plus,P1,T2,More]),
67                                        ([T1,T1,T1] , [P2,T2,P2,P1,P1,P1,P1,More]),
68                                        ([T1,T1,T1] , [P2,T2,P2,P1,P1T1Plus,P1,P1,P1,More]),
69                                        ([T1,T1,T1] , [P2,T2,P2,P1,P1,P1,T2,More]),
70                                        ([T1,T1,T1] , [P2,T2,P2,P1,P1T1Plus,P1,P1,T2,More]),
71                                        ([T1,T1,T1] , [P2,T2,P2,P1,T2,P2,P1,T2,More]),
72                                        ([T1,T1,T1] , [P2,T2,P2,P1,T2,P2,T2P2Plus,P1,T2,More])
                                              ,
73                                        ([T1,T1,T1] , [P2,T2,P2T2Plus,P2,P1,P1,P1,P1,More]),
74                                        ([T1,T1,T1] , [P2,T2,P2T2Plus,P2,P1,P1T1Plus,P1,P1,P1,
                                              More]),
75                                        ([T1,T1,T1] , [P2,T2,P2T2Plus,P2,P1,P1,P1,T2,More]),
76                                        ([T1,T1,T1] , [P2,T2,P2T2Plus,P2,P1,P1T1Plus,P1,P1,T2,
                                              More]),
77                                        ([T1,T1,T1] , [P2,T2,P2T2Plus,P2,P1,T2,P2,P1,T2,More])
                                              ,
78                                        ([T1,T1,T1] , [P2,T2,P2T2Plus,P2,P1,T2,P2,T2P2Plus,P1,
                                              T2,More])]
```

```
79  newCase3_18_4 = map createTranslation [([T1,T1,T1] , [P2,T2,P1,P2T2Plus]++expP1MoreLongAlt),
80                                         ([T1,T1,T1] , [P2,T2,P1,P2,P1,someSteps]),
81                                         ([T1,T1,T1] , [P2,T2,P1,P2T2Plus,P2,P1,someSteps]),
82                                         ([T1,T1,T1] , [P2,T2,P2T2Plus,P1,P2T2Plus]++
                                               expP1MoreLongAlt),
83                                         ([T1,T1,T1] , [P2,T2,P2T2Plus,P1,P2,P1,someSteps]),
84                                         ([T1,T1,T1] , [P2,T2,P2T2Plus,P1,P2T2Plus,P2]++
                                               expP1MoreLongAlt),
85                                         ([T1,T1,T1] , [P2,P1,P1,P1,P1,someSteps]),
86                                         ([T1,T1,T1] , [P2,P1,P1T1Plus,P1,P1,P1,someSteps]),
87                                         ([T1,T1,T1] , [P2,P1,P1,P1,T2,someSteps]),
88                                         ([T1,T1,T1] , [P2,P1,P1T1Plus,P1,P1,T2,someSteps]),
89                                         ([T1,T1,T1] , [P2,P1,T2,P2,P1,T2,someSteps]),
90                                         ([T1,T1,T1] , [P2,P1,T2,P2,T2P2Plus,P1,T2,someSteps])
                                               ,
91                                         ([T1,T1,T1] , [P2,T2,P2,P1,P1,P1,P1,someSteps]),
92                                         ([T1,T1,T1] , [P2,T2,P2,P1,P1T1Plus,P1,P1,P1,
                                               someSteps]),
93                                         ([T1,T1,T1] , [P2,T2,P2,P1,P1,P1,T2,someSteps]),
94                                         ([T1,T1,T1] , [P2,T2,P2,P1,P1T1Plus,P1,P1,T2,
                                               someSteps]),
95                                         ([T1,T1,T1] , [P2,T2,P2,P1,T2,P2,P1,T2,someSteps]),
96                                         ([T1,T1,T1] , [P2,T2,P2,P1,T2,P2,T2P2Plus,P1,T2,
                                               someSteps]),
97                                         ([T1,T1,T1] , [P2,T2,P2T2Plus,P2,P1,P1,P1,P1,
                                               someSteps]),
98                                         ([T1,T1,T1] , [P2,T2,P2T2Plus,P2,P1,P1T1Plus,P1,P1,P1
                                               ,someSteps]),
99                                         ([T1,T1,T1] , [P2,T2,P2T2Plus,P2,P1,P1,P1,T2,
                                               someSteps]),
100                                        ([T1,T1,T1] , [P2,T2,P2T2Plus,P2,P1,P1T1Plus,P1,P1,T2
                                               ,someSteps]),
101                                        ([T1,T1,T1] , [P2,T2,P2T2Plus,P2,P1,T2,P2,P1,T2,
                                               someSteps]),
102                                        ([T1,T1,T1] , [P2,T2,P2T2Plus,P2,P1,T2,P2,T2P2Plus,P1
                                               ]++expT2MoreAlt)]
103 check3_18_4 = equalTranslations result3_18_4 newCase3_18_4
104
105 test3_18_5 = testPattern newCase3_18_4
```

B.19. Übersetzungen mit $\tau(?) = T_1T_1P_2$

```
1  test3_19_1 = testPattern $ map createTranslation [([T1,T1,P2],[step,More])]
2  result3_19_1 = map createTranslation [([T1,T1,P2] , [P1,More]),
3                                        ([T1,T1,P2] , [P2,More])]
4  newCase3_19_1 = map createTranslation [([T1,T1,P2] , expP1MoreLongAlt),
5                                         ([T1,T1,P2] , expP2MoreLongAlt)]
6  check3_19_1 = equalTranslations result3_19_1 newCase3_19_1
7
8  test3_19_2 = testPattern newCase3_19_1
9  result3_19_2 = map createTranslation [([T1,T1,P2] , [P2,P1,P1,More]),
10                                       ([T1,T1,P2] , [P2T2Plus,P2,P1,P1,More]),
11                                       ([T1,T1,P2] , [P2,P1,T2,More]),
12                                       ([T1,T1,P2] , [P2T2Plus,P2,P1,T2,More])]
13 newCase3_19_2 = map createTranslation [([T1,T1,P2] , [P2,P1,P1,someSteps]),
14                                        ([T1,T1,P2] , [P2T2Plus,P2,P1,P1,someSteps]),
15                                        ([T1,T1,P2] , [P2,P1,T2,someSteps]),
16                                        ([T1,T1,P2] , [P2T2Plus,P2,P1,T2,someSteps])]
17 check3_19_2 = equalTranslations result3_19_2 newCase3_19_2
18
19 test3_19_3 = testPattern newCase3_19_2
20 result3_19_3 = map createTranslation [([T1,T1,P2] , [P2,P1,P1,P1,T2,More]),
21                                       ([T1,T1,P2] , [P2T2Plus,P2,P1,P1,P1,T2,More])]
22 newCase3_19_3 = map createTranslation [([T1,T1,P2] , [P2,P1,P1,P1,T2,someSteps]),
23                                        ([T1,T1,P2] , [P2T2Plus,P2,P1,P1,P1,T2,someSteps])]
24 check3_19_3 = equalTranslations result3_19_3 newCase3_19_3
25
26 test3_19_4 = testPattern newCase3_19_3
```

B.20. Übersetzungen mit $\tau(?) = T_1T_1T_2$

```
1  test3_20_1 = testPattern $ map createTranslation [([T1,T1,T2],[step,More])]
2  result3_20_1 = map createTranslation [([T1,T1,T2] , [P1,More]),
3                                        ([T1,T1,T2] , [P2,More])]
4  newCase3_20_1 = map createTranslation [([T1,T1,T2] , expP1MoreLongAlt),
5                                         ([T1,T1,T2] , expP2MoreLongAlt)]
6  check3_20_1 = equalTranslations result3_20_1 newCase3_20_1
7
8  test3_20_2 = testPattern newCase3_20_1
9  result3_20_2 = map createTranslation [([T1,T1,T2] , [P2,T2,P1,P2,More]),
10                                       ([T1,T1,T2] , [P2,T2,P2T2Plus,P1,P2,More]),
11                                       ([T1,T1,T2] , [P2,P1,P1,More]),
12                                       ([T1,T1,T2] , [P2T2Plus,P2,P1,P1,More]),
13                                       ([T1,T1,T2] , [P2,P1,T2,More]),
14                                       ([T1,T1,T2] , [P2T2Plus,P2,P1,T2,More])]
15 newCase3_20_2 = map createTranslation [([T1,T1,T2] , [P2,T2,P1]++expP2More),
16                                        ([T1,T1,T2] , [P2,T2,P2T2Plus,P1]++expP2More),
17                                        ([T1,T1,T2] , [P2,P1,P1,someSteps]),
18                                        ([T1,T1,T2] , [P2T2Plus,P2,P1,P1,someSteps]),
19                                        ([T1,T1,T2] , [P2,P1]++expT2MoreAlt),
20                                        ([T1,T1,T2] , [P2T2Plus,P2,P1]++expT2MoreAlt)]
21 check3_20_2 = equalTranslations result3_20_2 newCase3_20_2
22
23 test3_20_3 = testPattern newCase3_20_2
```

B.21. Übersetzungen mit $\tau(?) = T_1P_2P_1$

```
1  test3_21_1 = testPattern $ map createTranslation [([T1,P2,P1],[step,More])]
2  result3_21_1 = map createTranslation [([T1,P2,P1] , [P1,More]),
3                                        ([T1,P2,P1] , [P2,More])]
4  newCase3_21_1 = map createTranslation [([T1,P2,P1] , expP1MoreLongAlt),
5                                         ([T1,P2,P1] , expP2MoreLongAlt)]
6  check3_21_1 = equalTranslations result3_21_1 newCase3_21_1
7
8  test3_21_2 = testPattern newCase3_21_1
9  result3_21_2 = map createTranslation [([T1,P2,P1] , [P2,P1,P1,More]),
10                                       ([T1,P2,P1] , [P2T2Plus,P2,P1,P1,More]),
11                                       ([T1,P2,P1] , [P2,P1,T2,More]),
12                                       ([T1,P2,P1] , [P2T2Plus,P2,P1,T2,More])]
13 newCase3_21_2 = map createTranslation [([T1,P2,P1] , [P2,P1,P1,someSteps]),
14                                        ([T1,P2,P1] , [P2T2Plus,P2,P1,P1,someSteps]),
15                                        ([T1,P2,P1] , [P2,P1,T2,someSteps]),
16                                        ([T1,P2,P1] , [P2T2Plus,P2,P1,T2,someSteps])]
17 check3_21_2 = equalTranslations result3_21_2 newCase3_21_2
18
19 test3_21_3 = testPattern newCase3_21_2
20 result3_21_3 = map createTranslation [([T1,P2,P1] , [P2,P1,P1,T2,T1,More]),
21                                       ([T1,P2,P1] , [P2,P1,P1,T2,T2,More]),
22                                       ([T1,P2,P1] , [P2T2Plus,P2,P1,P1,T2,T1,More]),
23                                       ([T1,P2,P1] , [P2T2Plus,P2,P1,P1,T2,T2,More])]
24 newCase3_21_3 = map createTranslation [([T1,P2,P1] , [P2,P1,P1,T2,T1,someSteps]),
25                                        ([T1,P2,P1] , [P2,P1,P1,T2,T2,someSteps]),
26                                        ([T1,P2,P1] , [P2T2Plus,P2,P1,P1,T2,T1,someSteps]),
27                                        ([T1,P2,P1] , [P2T2Plus,P2,P1,P1,T2,T2,someSteps])]
28 check3_21_3 = equalTranslations result3_21_3 newCase3_21_3
29
30 test3_21_4 = testPattern newCase3_21_3
```

B.22. Übersetzungen mit $\tau(?) = T_1P_2T_1$

```
1  test3_22_1 = testPattern $ map createTranslation [([T1,P2,T1],[step,More])]
2  result3_22_1 = map createTranslation [([T1,P2,T1] , [P1,More]),
3                                        ([T1,P2,T1] , [P2,More])]
4  newCase3_22_1 = map createTranslation [([T1,P2,T1] , expP1MoreLongAlt),
5                                         ([T1,P2,T1] , expP2MoreLongAlt)]
6  check3_22_1 = equalTranslations result3_22_1 newCase3_22_1
7
8  test3_22_2 = testPattern newCase3_22_1
9  result3_22_2 = map createTranslation [([T1,P2,T1] , [P2,T2,P1,T2,More]),
10                                       ([T1,P2,T1] , [P2,T2,P2T2Plus,P1,T2,More]),
11                                       ([T1,P2,T1] , [P2,P1,P1,More]),
12                                       ([T1,P2,T1] , [P2T2Plus,P2,P1,P1,More]),
13                                       ([T1,P2,T1] , [P2,P1,T2,More]),
14                                       ([T1,P2,T1] , [P2T2Plus,P2,P1,T2,More])]
15 newCase3_22_2 = map createTranslation [([T1,P2,T1] , [P2,T2,P1]++expT2More),
16                                        ([T1,P2,T1] , [P2,T2,P2T2Plus,P1]++expT2More),
17                                        ([T1,P2,T1] , [P2,P1,P1,someSteps]),
18                                        ([T1,P2,T1] , [P2T2Plus,P2,P1,P1,someSteps]),
19                                        ([T1,P2,T1] , [P2,P1]++expT2MoreLongAlt),
20                                        ([T1,P2,T1] , [P2T2Plus,P2,P1]++expT2MoreLongAlt)]
21 check3_22_2 = equalTranslations result3_22_2 newCase3_22_2
22
23 test3_22_3 = testPattern newCase3_22_2
24 result3_22_3 = map createTranslation [([T1,P2,T1] , [P2,P1,P1,T2,T2,More]),
25                                       ([T1,P2,T1] , [P2T2Plus,P2,P1,P1,T2,T2,More])]
26 newCase3_22_3 = map createTranslation [([T1,P2,T1] , [P2,P1,P1,T2,T2,someSteps]),
27                                        ([T1,P2,T1] , [P2T2Plus,P2,P1,P1,T2,T2,someSteps])]
28 check3_22_3 = equalTranslations result3_22_3 newCase3_22_3
29
30 test3_22_4 = testPattern newCase3_22_3
```

B.23. Übersetzungen mit $\tau(?) = T_1P_2P_2$

```
1  test3_23_1 = testPattern $ map createTranslation [([T1,P2,P2],[step,More])]
2  result3_23_1 = map createTranslation [([T1,P2,P2] , [P1,More]),
3                                        ([T1,P2,P2] , [P2,More])]
4  newCase3_23_1 = map createTranslation [([T1,P2,P2] , expP1MoreLongAlt),
5                                         ([T1,P2,P2] , expP2MoreLongAlt)]
6  check3_23_1 = equalTranslations result3_23_1 newCase3_23_1
7
8  test3_23_2 = testPattern newCase3_23_1
9  result3_23_2 = map createTranslation [([T1,P2,P2] , [P2,T2,P1,T2,More]),
10                                       ([T1,P2,P2] , [P2,T2,P2T2Plus,P1,T2,More]),
11                                       ([T1,P2,P2] , [P2,P1,P1,More]),
12                                       ([T1,P2,P2] , [P2T2Plus,P2,P1,P1,More]),
13                                       ([T1,P2,P2] , [P2,P1,T2,More]),
14                                       ([T1,P2,P2] , [P2T2Plus,P2,P1,T2,More])]
15 newCase3_23_2 = map createTranslation [([T1,P2,P2] , [P2,T2,P1,T2,someSteps]),
16                                        ([T1,P2,P2] , [P2,T2,P2T2Plus,P1,T2,someSteps]),
17                                        ([T1,P2,P2] , [P2,P1,P1,someSteps]),
18                                        ([T1,P2,P2] , [P2T2Plus,P2,P1,P1,someSteps]),
19                                        ([T1,P2,P2] , [P2,P1,T2,someSteps]),
20                                        ([T1,P2,P2] , [P2T2Plus,P2,P1,T2,someSteps])]
21 check3_23_2 = equalTranslations result3_23_2 newCase3_23_2
22
23 test3_23_3 = testPattern newCase3_23_2
24 result3_23_3 = map createTranslation [([T1,P2,P2] , [P2,P1,P1,T2,T2,More]),
25                                       ([T1,P2,P2] , [P2T2Plus,P2,P1,P1,T2,T2,More])]
26 newCase3_23_3 = map createTranslation [([T1,P2,P2] , [P2,P1,P1,T2,T2,someSteps]),
27                                        ([T1,P2,P2] , [P2T2Plus,P2,P1,P1,T2,T2,someSteps])]
28 check3_23_3 = equalTranslations result3_23_3 newCase3_23_3
29
30 test3_23_4 = testPattern newCase3_23_3
```

B.24. Übersetzungen mit $\tau(?) = T_1 P_2 T_2$

```
test3_24_1 = testPattern $ map createTranslation [([T1,P2,T2],[step,More])]
result3_24_1 = map createTranslation [([T1,P2,T2] , [P1,More]),
                                      ([T1,P2,T2] , [P2,More])]
newCase3_24_1 = map createTranslation [([T1,P2,T2] , expP1MoreLongAlt),
                                       ([T1,P2,T2] , expP2MoreLongAlt)]
check3_24_1 = equalTranslations result3_24_1 newCase3_24_1

test3_24_2 = testPattern newCase3_24_1
result3_24_2 = map createTranslation [([T1,P2,T2] , [P2,T2,P1,P2,More]),
                                      ([T1,P2,T2] , [P2,T2,P2T2Plus,P1,P2,More]),
                                      ([T1,P2,T2] , [P2,P1,P1,More]),
                                      ([T1,P2,T2] , [P2T2Plus,P2,P1,P1,More]),
                                      ([T1,P2,T2] , [P2,P1,T2,More]),
                                      ([T1,P2,T2] , [P2T2Plus,P2,P1,T2,More])]
newCase3_24_2 = map createTranslation [([T1,P2,T2] , [P2,T2,P1]++expP2MoreAlt),
                                       ([T1,P2,T2] , [P2,T2,P2T2Plus,P1]++expP2MoreAlt),
                                       ([T1,P2,T2] , [P2,P1,P1,someSteps]),
                                       ([T1,P2,T2] , [P2T2Plus,P2,P1,P1,someSteps]),
                                       ([T1,P2,T2] , [P2,P1,T2,someSteps]),
                                       ([T1,P2,T2] , [P2T2Plus,P2,P1,T2,someSteps])]
check3_24_2 = equalTranslations result3_24_2 newCase3_24_2

test3_24_3 = testPattern newCase3_24_2
```

B.25. Übersetzungen mit $\tau(?) = T_1 T_2 P_1$

```
test3_25_1 = testPattern $ map createTranslation [([T1,T2,P1],[step,More])]
result3_25_1 = map createTranslation [([T1,T2,P1] , [P1,More]),
                                      ([T1,T2,P1] , [P2,More])]
newCase3_25_1 = map createTranslation [([T1,T2,P1] , expP1MoreLongAlt),
                                       ([T1,T2,P1] , expP2MoreLongAlt)]
check3_25_1 = equalTranslations result3_25_1 newCase3_25_1

test3_25_2 = testPattern newCase3_25_1
result3_25_2 = map createTranslation [([T1,T2,P1] , [P2,T2,P1,P2,More]),
                                      ([T1,T2,P1] , [P2,T2,P2T2Plus,P1,P2,More])]
newCase3_25_2 = map createTranslation [([T1,T2,P1] , [P2,T2,P1,P2,someSteps]),
                                       ([T1,T2,P1] , [P2,T2,P2T2Plus,P1,P2,someSteps])]
check3_25_2 = equalTranslations result3_25_2 newCase3_25_2

test3_25_3 = testPattern newCase3_25_2
```

B.26. Übersetzungen mit $\tau(?) = T_1T_2T_1$

```
1  test3_26_1 = testPattern $ map createTranslation [([T1,T2,T1],[step,More])]
2  result3_26_1 = map createTranslation [([T1,T2,T1] , [P1,More]),
3                                        ([T1,T2,T1] , [P2,More])]
4  newCase3_26_1 = map createTranslation [([T1,T2,T1] , expP1MoreLongAlt),
5                                         ([T1,T2,T1] , expP2MoreLongAlt)]
6  check3_26_1 = equalTranslations result3_26_1 newCase3_26_1
7
8  test3_26_2 = testPattern newCase3_26_1
9  result3_26_2 = map createTranslation [([T1,T2,T1] , [P2,T2,P1,P2,More]),
10                                       ([T1,T2,T1] , [P2,T2,P2T2Plus,P1,P2,More]),
11                                       ([T1,T2,T1] , [P2,P1,P1,More]),
12                                       ([T1,T2,T1] , [P2T2Plus,P2,P1,P1,More])]
13 newCase3_26_2 = map createTranslation [([T1,T2,T1] , [P2,T2,P1,P2,someSteps]),
14                                        ([T1,T2,T1] , [P2,T2,P2T2Plus,P1,P2,someSteps]),
15                                        ([T1,T2,T1] , [P2,P1,P1,someSteps]),
16                                        ([T1,T2,T1] , [P2T2Plus,P2,P1,P1,someSteps])]
17 check3_26_2 = equalTranslations result3_28_2 newCase3_28_2
18
19 test3_26_3 = testPattern newCase3_26_2
```

B.27. Übersetzungen mit $\tau(?) = T_1T_2P_2$

```
1  test3_27_1 = testPattern $ map createTranslation [([T1,T2,P2],[step,More])]
2  result3_27_1 = map createTranslation [([T1,T2,P2] , [P1,More]),
3                                        ([T1,T2,P2] , [P2,More])]
4  newCase3_27_1 = map createTranslation [([T1,T2,P2] , expP1MoreLongAlt),
5                                         ([T1,T2,P2] , expP2MoreLongAlt)]
6  check3_27_1 = equalTranslations result3_27_1 newCase3_27_1
7
8  test3_27_2 = testPattern newCase3_27_1
9  result3_27_2 = map createTranslation [([T1,T2,P2] , [P2,T2,P1,P2,More]),
10                                       ([T1,T2,P2] , [P2,T2,P2T2Plus,P1,P2,More]),
11                                       ([T1,T2,P2] , [P2,P1,T2,More]),
12                                       ([T1,T2,P2] , [P2T2Plus,P2,P1,T2,More])]
13 newCase3_27_2 = map createTranslation [([T1,T2,P2] , [P2,T2,P1]++expP2MoreAlt),
14                                        ([T1,T2,P2] , [P2,T2,P2T2Plus,P1]++expP2MoreAlt),
15                                        ([T1,T2,P2] , [P2,P1,T2,someSteps]),
16                                        ([T1,T2,P2] , [P2T2Plus,P2,P1,T2,someSteps])]
17 check3_27_2 = equalTranslations result3_27_2 newCase3_27_2
18
19 test3_27_3 = testPattern newCase3_27_2
```

B.28. Übersetzungen mit $\tau(?) = T_1T_2T_2$

```
1  test3_28_1 = testPattern $ map createTranslation [([T1,T2,T2],[step,More])]
2  result3_28_1 = map createTranslation [([T1,T2,T2] , [P1,More]),
3                                        ([T1,T2,T2] , [P2,More])]
4  newCase3_28_1 = map createTranslation [([T1,T2,T2] , expP1MoreLongAlt),
5                                         ([T1,T2,T2] , expP2MoreLongAlt)]
6  check3_28_1 = equalTranslations result3_28_1 newCase3_28_1
7
8  test3_28_2 = testPattern newCase3_28_1
9  result3_28_2 = map createTranslation [([T1,T2,T2] , [P1,P2,P2,More]),
10                                       ([T1,T2,T2] , [P1T1Plus,P1,P2,P2,More]),
11                                       ([T1,T2,T2] , [P2,T2,P1,P2,More]),
12                                       ([T1,T2,T2] , [P2,T2,P2T2Plus,P1,P2,More])]
13 newCase3_28_2 = map createTranslation [([T1,T2,T2] , [P1,P2,P2,someSteps]),
14                                        ([T1,T2,T2] , [P1T1Plus,P1,P2,P2,someSteps]),
15                                        ([T1,T2,T2] , [P2,T2,P1,P2,someSteps]),
16                                        ([T1,T2,T2] , [P2,T2,P2T2Plus,P1,P2,someSteps])]
17 check3_28_2 = equalTranslations result3_28_2 newCase3_28_2
18
19 test3_28_3 = testPattern newCase3_28_2
```