Promela - Virtuelle Maschine

Stefan Schürmans

7. November 2005

1 Konzept der virtuellen Maschine

Die virtuelle Maschine muss ein dynamisches Konzept für Prozesse und Kanäle bieten, wobei Kanal-Bezeichner über Kanäle verschickbar sein müssen. Dabei muss die Benutzung von synchroner und asynchroner Kommunikation nur vom Kanal und nicht von den kommunizierenden Prozessen abhängig sein, um die Kompatibilität zu Promela zu gewährleisten.
Prozesse können von anderen Prozessen unter Anagbe der Startadresse ihres Codes und ihrer Parameter gestartet werden und können sich am Ende ihres Codes selbst beenden. Zu Beginn ist ein initialer Prozess aktiv, der dann gegebebenfalls weitere Prozesse starten kann.

1.1 Prozess-Schritte

Innerhalb der Prozesse wird der Zugriff auf globale und lokale Variablen sowie die Auswertung von Ausdrücken unterstützt. Daher ist das Grundkonzept eine Keller-Maschine, die allerdings um acht Register ergänzt ist, um ein Zwischenspeichern von mehrfach verwendeten Werten zu ermöglichen.
Die Ausführung darf nur an speziellen Stellen im Code zu anderen Prozessen wechseln, um Anweisungen einer höheren Sprache atomar realisieren zu können. Da in höheren Sprachen an diesen Stellen eine komplette Anweisung abgeschlossen ist, befindet sich kein Ausdruck in der Auswertung und die Werte auf dem Datenkeller und in den Registern müssen dort nicht gespeichert werden.
Der nach außen hin sichtbare Zustandsraum enthält nur die Zustände zwischen diesen atomaren Schritten, was dazu führt, dass der Datenkeller und die Register nach außen hin unsichtbar sind.

1.2 Nichtdeterminismus

Nichtdeterminismus wird sowohl auf lokaler als auch auf globaler Ebene unterstützt. Auf lokaler Ebene wird dies durch nichtdeterministische Sprünge erreicht, an denen sich der Ausführungspfad gabelt. Ein Pfad fährt mit der folgenden Anweisung fort und der andere Pfad an der Ziel-Adresse des Sprunges. Eine Verzweigung in mehr als zwei Pfade lässt sich dabei durch eine Kette von nichtdeterministischen Sprüngen beschreiben.
Weiter ist es möglich die Ausführung eines Befehls zu beginnen, aber nicht zu beenden. Dies ist sinnvoll bei Wächter- und Kommunikations-Anweisungen. Dazu wird einfach der aktuelle Ausführungspfad abgebrochen, ohne dass ein nach außen hin sichtbarer Folgezustand entsteht.

Auf globaler Ebene besteht der Nichtdeterminismus in der Auswahl des nächsten auszuführenden Prozesses. Diese Wahlfreiheit kann dadurch eingeschränkt werden, dass dem

aktuellen Prozess das exklusive Ausführungsrecht zugeteilt wird. In diesem Fall sind andere Prozesse so lange von der Ausführung ausgeschlossen, bis der Prozess dieses Recht explizit aufgibt oder aber keinen Schritt mehr ausführen kann.

1.3 Kanäle

Die Kanäle werden mit ihrer FIFO-Puffer-Tiefe und dem Typ der Nachrichten verwaltet und mit Hilfe ihrer eindeutigen Kennzahl angesprochen. So kann in jeder Sende- und Empfangsoperation ein beliebiger Kanal benutzt werden, wie es z.B. für Promela nötig ist.
Die Konvertierung des Nachrichtentyps wird teilweise von der VM übernommen, da der Code des Prozesses den Kanal-Typ nicht kennen kann, wenn er den Kanal z.B. selbst über einen Kanal empfangen hat.
Auch die Entscheidung zwischen synchroner und asynchroner Kommunikation hängt dabei nur von den Eigenschaften des Kanals (FIFO-Puffer-Tiefe $= 0$ oder > 0) und nicht von den beteiligten Prozessen ab.

2 lokaler Zustand eines Prozesses

Im lokalen Zustand $\Lambda = (L, m, R, D)$ eines Prozesses sind die lokalen Variablen L, der Befehlszähler m, die Register R und der Datenkeller D gespeichert.
In einem Schritt-Zustand (d.h. einem Zustand zwischen den einzelnen Promela-Anweisungen, an dem die Ausführung möglicherweise zu einem anderen Prozess wechseln kann) hat der lokale Zustand die Gestalt $\Lambda = (L, m)$, d.h. Register und Datenkeller fehlen.
Der Befehlszähler wird einfach als eine Zahl $m \in \mathbb{N}_0$ gespeichert, die die Nummer der als nächsten auszuführenden Instruktion angibt.

2.1 lokale Variablen

Zur formalen Darstellung der lokalen Variablen L (local variables) wird eine Abbildung $L[\cdot] : \mathbb{N}_0 \to \mathbb{Z}$ verwendet.
Einzelne Elemente des lokalen Speichers können mit der Schreibweise $L[i/v]$ verändert werden:

$$L[i/v] := L' \qquad \text{mit } L'[j] := \begin{cases} v & \text{für } j = i \\ L[j] & \text{sonst} \end{cases}$$

Der initiale Zustand des lokalen Speichers ist L_0 mit $L_0[i] := 0$ für alle $i \in \mathbb{N}_0$.

2.2 Register

Die 8 Register $(r_0, \ldots, r_7)$ können Werte aus $\mathbb{Z}$ annehmen und werden sinnvollerweise überall dort eingesetzt, wo die zu erledigenden Aufgaben nicht zur Struktur eines Datenkellers passen. Da die Register nur zur Zwischenspeicherung von mehrfach benutzen Werten gedacht sind, werden sie vom Datenkeller geladen oder ihr Wert wird auf dem Datenkeller abgelegt. Berechnungen mit Werten aus Registern erfolgen mit Ausnahme von Incrementierung und Decrementierung nur über den Umweg des Datenkellers.
Zusätzlich gibt es noch das sogenannte Flag-Register r_F, in dem Werte aus $\mathbb{N}_0$ gespeichert werden können. Dieses Register wird dazu benutzt, boolsche Informationen vom Benutzer entgegenzunehmen und boolsche Informationen an den Benutzer zurückzugeben. (Z.B. könnte ein Model-Checker in diesem Register Korrektheits-Informationen ablegen

und diese dann zusammen mit dem Nachfolgezustand entgegennehmen und auswerten.) Der Zugriff auf dieses Register erfolgt daher bitweise.
Formal werden die Register als eine Abbildung $R[\cdot] : \{0, \ldots, 7, F\} \to \mathbb{Z}$ dargestellt, auf die mit der Schreibweise $R[i/v]$ auch schreibend zugegriffen werden kann:

$$R[i/v] := R' \qquad \text{mit } R'[j] := \begin{cases} v & \text{für } j = i \\ R[j] & \text{sonst} \end{cases}$$

R_0 mit $R_0[i] := 0$ für alle $i \in \{0, \ldots, 7\}$ und $R_0[F] = \mathit{flag_reg}$ ist der initiale Wert der Register. Der Wert *flag_reg* des Flag-Registers wird dabei zu Beginn eines Schritts vom Benutzer angegeben.

2.3 Datenkeller, Auswertung von Ausdrücken

Die Auswertung von Ausdrücken findet auf dem Datenkeller statt. Dazu stehen Instruktionen zum Laden von Konstanten und Variablen, sowie zur Ausführung von arithmetischen und boolschen Operationen zur Verfügung.
Um nicht zu viele verschiedene Instruktionen zu benötigen, erfolgen alle Operationen auf dem Datenkeller in $\mathbb{Z}$. Eine Konvertierung von den bzw. in die entsprechenden wertbeschränkten Datentypen erfolgt beim Laden und Speichern von Variablen.
Formal wird der Datenkeller als eine möglicherweise leere Folge von Elementen aus $\mathbb{Z}$ dargestellt, die durch : getrennt werden. Mehrere Elemente des Datenkellers oder der gesamte Datenkeller werden durch groß geschriebene Variablen (z.B. D) bezeichnet.

2.4 lokaler Zustand nach einem Schritt

Am Ende eines kompletten Schritts, muss die Auswertung aller Ausdrücke abgeschlossen und somit der Datenkeller leer sein.
Daher hat der vollständige lokale Zustand (L, m, D, R) zu diesen Zeitpunkten die Form (L, m, ϵ, R), wobei die Registerwerte R nicht mehr benötigt werden. Der lokale Zustand kann daher in der Form (L, m) dargestellt und gespeichert werden.
Sollte der Datenkeller an einem Schritt-Ende nicht leer sein, so wird der Inhalt verworfen. Der neue Schritt beginnt somit auf jeden Fall wieder mit einem leeren Datenkeller und mit Registerwerten 0.
Da die Anzahl der lokalen Variablen bekannt und konstant ist, kann der lokale Zustand (L, m) nach einem Schritt in einem Speicherbereich konstanter Größe verwaltet werden.

3 globaler Zustand

Der globale Zustand $\Gamma = (\Pi, e, o, G, \Phi)$ setzt sich aus den aktiven Prozessen Π, der Kennzahl e des exklusiv auszuführenden Prozesses, der Kennzahl o des aktuellen Monitor-Prozesses, den globalen Variablen G und den Kanälen Φ zusammen.
Die Kennzahl e gibt die Nummer des exklusiv auzuführenden Prozesses an. Ist $e = \perp$, bedeutet dies, dass kein Prozess exklusiv ausgeführt wird.
Ebenso gibt o die Nummer des aktuellen Monitor-Prozesses an, oder es ist $o = \perp$, falls kein solcher existiert. Der Monitor-Prozess wird synchron im Wechsel mit dem übrigen System ausgeführt und kann dazu benutzt werden, Eigenschaften des Systems zu testen.

3.1 aktive Prozesse

Ein Prozess $\pi = (p, M, \Lambda)$ besteht aus einer eindeutigen Prozess-Kennzahl $p \in \mathbb{N}$, seinem aktuellen Ausführungsmodus $M \in \{\underline{\mathrm{N}}, \underline{\mathrm{A}}, \underline{\mathrm{I}}, \underline{\mathrm{T}}\}$ (normal, atomar, unsichtbar (invisible), beendet (terminated)) und dem lokalen Zustand Λ.
Dabei werden p und M nicht im lokalen Zustand gespeichert, um diese Parameter nicht bei der Definition aller Instruktion betrachten zu müssen.
Der unsichtbare Modus verhält sich dabei exakt wie der atomare Modus, jedoch werden die erreichten Zustände bei der Übergabe an den Benutzer als unsichtbar markiert, so dass der Benutzer sich entscheiden kann, diese nicht zu speichern.
Alle aktiven Prozesse werden in der Menge Π verwaltet. Dabei darf es zu keinem Zeitpunkt zwei Prozesse $\pi, \pi' \in \Pi$ geben, die verschieden sind ($\pi \neq \pi'$) und die gleiche Prozess-Kennzahl besitzen ($\pi = (p, M, \Lambda)$ und $\pi' = (p, M', \Lambda')$).

3.2 globale Variablen

Die formale Darstellung der globalen Variablen G (global variables) erfolgt völlig analog zu den lokalen Variablen durch eine Abbildung $G[\cdot] : \mathbb{N}_0 \to \mathbb{Z}$ verwendet.
Einzelne Elemente des globalen Speichers können auch hier mit der Schreibweise $G[i/v]$ verändert werden:

$$G[i/v] := G' \qquad \text{mit } G'[j] := \begin{cases} v & \text{für } j = i \\ G[j] & \text{sonst} \end{cases}$$

Der initiale Zustand des globalen Speichers ist G_0 mit $G_0[i] := 0$ für alle $i \in \mathbb{N}_0$.

3.3 Kanäle

Ein Kanal $\varphi = (c, l, t, C)$ besteht aus einer eindeutigen Kanal-Kennzahl $c \in \mathbb{N}$, der maximalen Anzahl Einträge l (length), dem Typ der Nachrichten t (type) und dem aktuellen Inhalt C (contents).
Die Kennzahl c basiert auf der Prozess-Kennzahl des erzeugenden Prozesses und wird so erweitert, dass eindeutige Kanal-Kennzahlen entstehen. Der Grund für dieses Verfahren liegt in der Reduktion der Zustandsanzahl des Gesamtsystems. Würde nur eine fortlaufende Nummer zur Identifizierung der Kanäle benutzt, so würden bei der nebenläufigen Erzeugung von Kanälen in mehreren Prozessen Zustände entstehen, die sich nur in den Kanal-Kennzahlen unterscheiden. Durch die Verwendung der Prozess-Kennzahl wird dieser Effekt verhindert und die Benennung der Kanäle erfolgt gleich, so dass in jedem Fall der gleiche Zustand erreicht wird.
Der Typ t eines Kanals ist dabei ein Tupel $t = (b_1, \ldots, b_n)$ mit den Bit-Anzahlen der einzelnen elementaren Typen, wie sie von der Funktion $\underline{\text{bits}}$ zurückgegeben werden.
Der Kanal-Inhalt C ist ein Wort mit einer Länge von 0 bis l, das aus den einzelnen Nachrichten $\in \mathbb{Z}^n$ im Kanal besteht. Wird der Kanal-Inhalt durch die enthaltenen Nachrichten angegeben, so werden die einzelnen Nachrichten durch : getrennt. Die zuletzt eingefügte Nachricht steht dabei ganz rechts in C.

Synchrone Kanäle besitzen die Länge 0. Daher darf normalerweise der Inhalt C nur das leere Wort ϵ sein. Während einer synchronen Kommunikation ist es aber möglich, dass eine Nachricht im Kanal liegt, d.h. $C \in \mathbb{Z}^n$. Allerdings dürfen solche Zustände nicht nach außen sichtbar werden.

Alle existierenden Kanäle werden in der Menge Φ verwaltet. Dabei darf es zu keinem Zeitpunkt zwei Kanäle $\varphi, \varphi' \in \Phi$ geben, die verschieden sind ($\varphi \neq \varphi'$) und die gleiche Kanal-Kennnung besitzen ($\varphi = (c, l, t, C)$ und $\varphi' = (c, l', t', C')$).

3.4 Nichtdeterminismus

Aufgrund des Nichtdeterminismus gibt es Gabelungen der Programmausführung. An einer solchen Stelle muss der Zustand verdoppelt werden.
Ein Fall ist die Ausführung eines nichtdeterministischen Sprungs, der zwei Kopien des Zustands erzeugt, die sich nur im Wert des Befehlszählers des aktuellen Prozesses unterscheiden. In der einen Kopie steht der Befehlszähler einfach auf dem nächsten Befehl, in der anderen Kopie auf der Ziel-Adresse des Sprungbefehls.
Der andere Fall ist ein zufälliger Empfang auf einem Kanal mit mehreren Nachrichten. Die entsprechende Instruktion muss hier für jede Nachricht, die empfangen werden kann, einen Nachfolgezustand generieren. Diese Zustände unterscheiden sich dabei im Inhalt des Datenkellers des aktuellen Prozesses, da die verschiedenen Nachrichten dort abgelegt werden.

4 Beginn und Ende eines Schritts

4.1 initialer Zustand

Zu Beginn der Ausführung gibt es nur einen einzigen Prozess π mit Kennzahl 1, der im normalen Modus $\underline{\mathrm{N}}$ ist und als nächstes die Instruktion an Adresse 0 ausführt. Die globalen Variablen sind noch nicht initialisiert und es gibt keinen Kanal:

$$\Gamma_{init} \ := \ (\ \{(1, \underline{\mathrm{N}}, (L_0, 0))\}, \ \bot, \ \bot, \ G_0, \ \emptyset \ )$$

4.2 Zwischenzustände bei synchroner Kommunikation

Synchrone Kommunikation geschieht, indem in einem Kanal der Länge 0 in einem Schritt vom Sender eine Nachricht ablegt wird und unmittelbar danach in einem zweiten Schritt vom Empfänger wieder aus dem Kanal entfernt wird.
Der zwischen diesen Schritten auftretende Zwischenzustand beinhaltet einen Kanal, der mehr Nachrichten enthält, als gemäß seiner maximalen Länge zugelassen sind.
Um die Erkennung solcher Zustände in den folgenden Definitionen zu vereinfachen, wird die Funktion $\underline{\mathrm{sync}}(\Gamma)$ definiert, die prüft, ob der Zustand Γ ein Zwischenzustand synchroner Kommunikation ist (in diesem Falle ist $\underline{\mathrm{sync}}(\Gamma) = \underline{\mathrm{true}}$):

$$\underline{\mathrm{sync}}(\ (\Pi, e, o, G, \Phi) \ ) \ := \ \begin{cases} \underline{\mathrm{false}} & \text{falls } \forall \varphi = (c, l, t, C) \in \Phi : \ |C| \leq l \\ \underline{\mathrm{true}} & \text{sonst} \end{cases}$$

4.3 Beginn eines Schritts

Zu Beginn eines Schritts muss der nächste auszuführende Prozess ermittelt werden. Alle Prozesse $\pi \in \Pi$ befinden sich zu diesem Zeitpunkt in einem Schritt-Zustand: $\pi = (p, M, (L, m))$ Der lokaleZustand des ausgewählten Prozesses wird dann um mit 0 initialisierte Register, ein mit *flag_reg* initialisiertes Flag-Register und einen leeren Datenkeller ergänzt, so dass die Zustandsübergänge, die durch die Instruktionen definiert sind, anwendbar werden.
Falls ein Prozess exklusiv ausgeführt wird, so muss dieser für den nächsten Schritt gewählt werden. Ansonsten ($e = \bot$) kann jeder noch nicht beendete Prozess ($M \neq \underline{\mathrm{T}}$) ausgewählt werden:

$$(\{(p,M,(L,m)),\pi_1,\ldots,\pi_n\},e,o,G,\Phi) \xrightarrow[act]{p} (\{(p,M,(L,m,R_0,\epsilon)),\pi_1,\ldots,\pi_n\},e,o,G,\Phi)$$
$$\text{falls}\quad \pi_i = (p_i,M_i,(L_i,m_i)) \text{ für alle } 1 \le i \le n \quad \text{und} \quad e \in \{p,\bot\},\quad p \neq o,\quad M \neq \underline{\mathrm{T}}$$

Dies beschreibt die Aktivierung von normalen Prozessen, wenn das eigentliche System läuft. In diesem Fall ist die Variable $monitor = \underline{\text{false}}$. Falls aber der Monitor-Prozess ausgeführt werden soll ($monitor = \underline{\text{true}}$), findet die folgende Definition Anwendung:

$$(\{(o,M,(L,m)),\pi_1,\ldots,\pi_n\},e,o,G,\Phi) \xrightarrow[act]{o} (\{(o,M,(L,m,R_0,\epsilon)),\pi_1,\ldots,\pi_n\},e,o,G,\Phi)$$
$$\text{falls}\quad \pi_i = (p_i,M_i,(L_i,m_i)) \text{ für alle } 1 \le i \le n \quad \text{und} \quad M \neq \underline{\mathrm{T}}$$

In jedem Fall führt der aktive Prozess ausgehend vom aktivierten Zustand eine Folge von Transitionen aus, die wieder in einem (oder mehreren möglichen) nicht aktivierten Zustand endet. Diese Transitionen, die die einzelnen Instruktionen abarbeiten, werden weiter unten aufgeführt.

4.4 lokale Schritte

Wenn ein Ausführungsschritt beendet ist, hat der lokale Zustand des aktuellen Prozesses wieder die Form (L,m). D.h. die Registerwerte und der Datenkeller sind bereits verworfen. Man betrachtet nun Zustands-Folgen von einem Schritt-Zustand zum nächsten und bestimmt darauf aufbauend zunächst die Relation $\xrightarrow[step']{p,M,I}$ der lokalen Einzel-Schritte des Prozesses p, die noch mit einer Information $I = (l,f)$ für den Benutzer (bestehend aus einer Kennzeichung l (label) der ausgeführen Transition und dem Wert des Flag-Registers f) versehen sind:

$$\Gamma \xrightarrow[step']{p,M,I} \Gamma' \quad \text{wenn} \quad \Gamma \xrightarrow[act]{p} \Gamma_0 \xrightarrow[in]{*} \Gamma_1 \xrightarrow[end]{M,I} \Gamma'$$

Ein Schritt kann in einem unsichtbaren Zustand enden. In diesem Fall führt der gleiche Prozess sofort den nächsten Schritt aus, bis wieder ein sichtbarer Zustand erreicht ist. Lokale Schritte ergeben sich dann aus nichtleeren Folgen lokaler Einzel-Schritte eines Prozesses p von einem Zustand in den nächsten sichtbaren Zustand und werden durch die Relation $\xrightarrow[step]{p,I}$ beschrieben.

Falls die weitere Ausführung in einem unsichtbaren Zustand nicht weiter möglich ist oder synchrone Kommunikation in Bearbeitung ist, d.h. ein Kanal mehr Nachrichten als erlaubt enthält (siehe dazu auch globale Schritte), so wird dieser unsichtbare Zustand sichtbar:

$$\Gamma \xrightarrow[step]{p,I} \Gamma' \quad \text{wenn} \quad \Gamma \xrightarrow[step']{p,\underline{\mathrm{I}},I_1} \Gamma_1 \xrightarrow[step']{p,\underline{\mathrm{I}},I_2} \ldots \xrightarrow[step']{p,\underline{\mathrm{I}},I_n} \Gamma_n \xrightarrow[step']{p,M,I} \Gamma'\,, \quad n \in \mathbb{N}_0$$
$$\text{mit} \quad \underline{\text{sync}}(\Gamma_k) = \underline{\text{false}} \quad \text{für alle } 1 \le k \le n$$
$$\text{und} \quad M \neq \underline{\mathrm{I}} \quad \text{oder} \quad \Gamma' \;\not\xrightarrow[step']{p} \quad \text{oder} \quad \underline{\text{sync}}(\Gamma') = \underline{\text{true}}$$

Auch die Ausgabe mit `PRINTx` in einem Schritt lässt einen folgenden unsichtbaren Zustand sichtbar werden. Dies wird hier allerdings aus Gründen der Übersichtlichkeit nicht formal modelliert.

4.4.1 Nicht-Ausführbarkeit eines Schritts

Dadurch, dass Anweisungen nicht ausführbar sein können, ist es auch möglich, dass ein exklusiv ausgeführter Prozess keinen Schritt mehr ausführen kann. In diesem Fall verliert der Prozess sein exklusives Ausführungsrecht:

$$(\Pi, e, o, G, \Phi) \xrightarrow[step]{p,I} \Gamma' \quad \text{wenn} \quad (\Pi, e, o, G, \Phi) \underset{step}{\overset{e}{\not\longrightarrow}}, \quad (\Pi, \perp, G, \Phi) \xrightarrow[step]{p,I} \Gamma', \quad e \neq \perp$$

4.5 globale Schritte

Die globalen Schritte werden durch die Relation $\xrightarrow[glob]{I}$ festgelegt, wobei I wieder die Information für den Benutzer bezeichnet.
Zunächst wird dazu die Transition $\xrightarrow[glob']{I}$ definiert, die bis auf Berücksichtung von Timeout-Situtationen und des Monitor-Prozesses schon die globalen Schritte festlegt.

Falls ein Schritt keine synchrone Kommunikation durchführt, ist der erreichte Zustand kein Zwischenzustand synchroner Kommunikation (d.h. $\underline{\text{sync}}(\cdot) = \underline{\text{false}}$). Der Schritt kann damit zu einem globalen Schritt werden:

$$\Gamma \xrightarrow[glob']{I} \Gamma' \quad \text{wenn} \quad \Gamma \xrightarrow[step]{p,I} \Gamma' \quad \text{mit} \quad \underline{\text{sync}}(\Gamma') = \underline{\text{false}}$$
$$\text{dabei } exec := p$$

Die Prozess-Kennzahl des ausgeführten Prozesses wird dabei der Variable $exec$ zugewiesen.

4.5.1 synchrone Kommunikation

Bei synchroner Kommunikation ist zunächst nur der Sendebefehl ein möglicher Schritt. Dieser Schritt endet allerdings in einem Zwischenzustand synchroner Kommunikation (d.h. $\underline{\text{sync}}(\cdot) = \underline{\text{true}}$).
In diesem Fall muss es noch einen anderen Prozess geben, der anschließend unabhängig vom exklusiven Ausführungsrecht ausgeführt wird und diese Nachricht synchron empfängt, damit sich ein globaler Schritt ergibt:

$$\Gamma \xrightarrow[glob']{sync,I,I'} \Gamma' \quad \text{wenn} \quad \Gamma \xrightarrow[step]{p,I} (\Pi, e, o, G, \Phi) \quad \text{mit} \quad \underline{\text{sync}}(\ (\Pi, e, o, G, \Phi)\ ) = \underline{\text{true}}$$
$$\text{und} \quad (\Pi, \perp, o, G, \Phi) \xrightarrow[step]{p',I'} \Gamma' \quad \text{mit} \quad \underline{\text{sync}}(\Gamma') = \underline{\text{false}} \quad \text{und} \quad p \neq p'$$
$$\text{dabei} \quad exec := p'$$

Die Information für den Benutzer wird hier aus beiden Schritten aufgesammelt und mit der Zusatz-Information $sync$ versehen. Als ausgeführter Prozess wird der Empfänger in $exec$ zurückgegeben.

4.6 Timeout-Situationen

Während eines Schritts ist die globale Timeout-Variable $timeout$ zunächst false. Falls unter dieser Voraussetzung kein Nachfolgezustand existiert, so wird diese Variable für diesen einen Schritt auf true gesetzt und die Nachfolgezustände werden neu bestimmt:

$$
\begin{aligned}
\Gamma \xrightarrow[glob]{I} \Gamma' \quad & \text{wenn} \quad \Gamma \xrightarrow[glob']{I} \Gamma' \quad \text{mit} \quad timeout := \underline{\text{false}} \\
\Gamma \xrightarrow[glob]{timeout,I} \Gamma' \quad & \text{wenn} \quad \Gamma \underset{glob'}{\not\longrightarrow} \quad \text{mit} \quad timeout := \underline{\text{false}} \\
& \text{und} \quad \Gamma \xrightarrow[glob']{I} \Gamma' \quad \text{mit} \quad timeout := \underline{\text{true}}
\end{aligned}
$$

Das Auftreten eines Timeouts wird in der Information für den Benutzer verzeichnet.

4.7 Monitor-Prozess und Nachfolgezustände

Neben den Prozessen des eigentlichen Systems kann ein Monitor-Prozess existieren, der synchron im Wechsel mit dem System ausgeführt wird um das System zu beobachten und Eigenschaften des Systems zu ermitteln.
Bereits weiter oben wurde definiert, dass mit Hilfe der Variable *monitor* festgelegt werden kann, ob das System oder der Monitor-Prozess einen Schritt ausführt. Des weiteren wird die Variable *flag_reg* als intialer Wert für das Flag-Register und die Variable *last* (im Monitor: Kennzahl des Prozesses, der den letzten Schritt ausführte) in der Instruktion `LDS` für einen Schritt benötigt.
Der Benutzer stellt einen initialen Wert des Flag-Registers für das System in der Variable *flags* zusammen mit einem Ausgangszustand Γ zur Verfügung. Die Transition $\underset{succ}{\longrightarrow}$ liefert mit jedem Nachfolgezustand Γ' eine Information I zurück.

Falls es keinen Monitor-Prozess gibt, wird einfach die $\underset{glob}{\longrightarrow}$-Relation weitergegeben:

$$
\begin{aligned}
\Gamma \xrightarrow[succ]{I} \Gamma' \quad & \text{wenn} \quad \Gamma \xrightarrow[glob]{I} \Gamma' \quad \text{mit} \quad monitor := \underline{\text{false}}, flag_reg := flags, last := 0 \\
& \text{und} \quad \Gamma' = (\Pi, e, \bot, G, \Phi)
\end{aligned}
$$

Gibt es einen Monitor-Prozess, wird versucht diesen nach dem System auszuführen. Dabei wird das Flag-Register zu 0 gesetzt und die Kennzahl des letzten ausgeführten Prozesses in *last* übergeben.
Wenn der Monitor-Prozess ausführbar ist, wird der Nachfolgezustand dieses Prozesses zurückgegeben. Die Informationen werden aber dem Schritt des Systems entnommen:

$$
\begin{aligned}
\Gamma \xrightarrow[succ]{I} \Gamma'' \quad & \text{wenn} \quad \Gamma \xrightarrow[glob]{I} \Gamma' \quad \text{mit} \quad monitor := \underline{\text{false}}, flag_reg := flags, last := 0 \\
& \text{und} \quad \Gamma' = (\Pi, e, o, G, \Phi), \quad o \neq \bot \\
& \text{und} \quad \Gamma' \xrightarrow[glob]{I'} \Gamma'' \quad \text{mit} \quad monitor := \underline{\text{true}}, flag_reg := 0, last := exec
\end{aligned}
$$

Zusammen mit dem Nachfolger-Zustand und der Information I wird dabei an den Benutzer übergeben, ob es einen Monitor-Prozess gibt, ob er einen Schritt ausführen konnte, und ob er sich in einem akzeptierenden Zustand befindet ($\underline{\text{accept}} \in \underline{\text{Flags}}[m]$ für $(o', M', (L', m')) \in \Pi'$ mit $\Gamma'' = (\Pi', e', o', G', \Phi')$).

Ist die Ausführung des Systems nicht möglich, d.h. wenn alle Prozesse blockieren oder terminiert sind, wird der Ausgangs-Zustand als Nachfolger-Zustand an den Benutzer gemeldet und diese Situation durch die Information *sys_block* gekennzeichnet.

$$\Gamma \overset{sys_block}{\underset{succ}{\longrightarrow}} \Gamma \quad \text{wenn} \quad \Gamma \underset{glob}{\not\longrightarrow} \quad \text{mit} \quad monitor := \underline{\text{false}}, flag_reg := flags, last := 0$$

Falls das System in einem Zustand mit akzeptierendem Monitor-Prozess blockiert, entsteht durch diese Definition ein unendlicher Pfad aus akzeptierenden Zuständen, so dass der Model-Checker die Fehlersituation erkennen kann.

Falls die Ausführung des Monitors nicht möglich ist, entsteht kein Nachfolger-Zustand. Dies ist immer der Fall, wenn der Monitor-Prozess auf Grund einer nicht ausführbaren Anweisung blockiert ist.

Ist der Monitor-Prozess im Ausgangs-Zustand bereits terminiert (oder nicht vorhanden), so wird ebenfalls der Ausgangs-Zustand als Nachfolger-Zustand an den Benutzer gemeldet. Dabei wird diese Situation mit mon_term markiert.

$$\begin{aligned} \Gamma \overset{mon_term}{\underset{succ}{\longrightarrow}} \Gamma \quad & \text{wenn} \quad \Gamma = (\Pi, e, o, G, \Phi), \quad o \neq \bot \\ & \text{und} \quad (o, \underline{\text{T}}, \Lambda) \in \Pi \quad \text{oder} \quad (o, M, \Lambda) \notin \Pi \end{aligned}$$

Durch dieses Verhalten erzeugt ein terminierender Monitor-Prozess einen unendlichen Pfad von Zuständen mit mon_term-Information. Wenn der Benutzer diese Information als akzeptierenden Zustand wertet, ergibt sich so ein unendlich oft akzeptierender Pfad wie bei einem nicht terminieren Monitor-Prozess, der einen Fehler aufgedeckt hat.

5 Instruktionen

Die Instruktionen werden in einer Liste $\underline{\text{Instr}}$ mit Indices $m \in \mathbb{N}_0$ gespeichert. Der Programmzähler m im lokalen Zustand eines Prozesses bestimmt somit die nächste auszuführende Anweisung $\underline{\text{Instr}}[m]$.
Für jede Adresse m gibt es weiter einige Flags, die besondere Eigenschaften dieser Adresse angeben. Diese Flags werden in einer Liste $\underline{\text{Flags}}$ gespeichet, wobei $\underline{\text{Flags}}[m] \subseteq \{\underline{\text{progress}}, \underline{\text{accept}}\}$ (Fortschritts-Zustand, akzeptierender Zustand).

5.1 abkürzende Schreibweisen

Bei der Definition der Semantik der Instruktionen wäre es aufwändig, immer den gesamten globalen Zustand betrachten zu müssen. Daher werden einige abkürzende Schreibweisen für den Übergang von einem globalen Zustand Γ_1 in einen globalen Zustand Γ_2 definiert.
Dabei ist es immer so, dass kein Prozess außer dem betrachteten sich in der Ausführung eines Schritts befindet. D.h. alle anderen Prozesse haben einen lokalen Zustand der Form (L, m).
Viele Instruktionen haben nur eine lokale Wirkung in einem Prozess und transformieren lediglich den lokalen Zustand Λ_1 in einen lokalen Zustand Λ_2. Oft werden dabei auch die globalen Variablen (G_1, G_2) oder die Kanäle (Φ_1, Φ_2) gelesen oder verändert.
Daher wird nun eine abkürzende Schreibweise definiert:

$$\Lambda_1, G_1, \Phi_1 \underset{x}{\longrightarrow} \Lambda_2, G_2, \Phi_2$$

$$\text{bedeutet } (\{(p, M, \Lambda_1), \pi_1, \ldots, \pi_n\}, e, o, G_1, \Phi_1) \underset{x}{\longrightarrow} (\{(p, M, \Lambda_2), \pi_1, \ldots, \pi_n\}, e, o, G_2, \Phi_2)$$

$$\text{mit } \pi_i = (p_i, M_i, (L_i, m_i)) \text{ für alle } 1 \leq i \leq n$$

Es ist dabei zulässig, die globalen Variablen (G_1, G_2) und/oder die Kanäle (Φ_1, Φ_2) wegzulassen, falls sie in der entsprechenden Instruktion nicht benutzt werden.

5.2 aktuelle Instruktion

Im folgenden wird nun immer nur die aktuelle Instruktion des aktuellen Prozesses betrachtet. Der aktuelle Prozess ist automatisch dadurch gegeben, dass in einem globalen Zustand immer nur ein Prozess mit lokalem Zustand der Form (L, m, R, D) enthalten sein kann.
Die aktuelle Instruktion wird mit Hilfe der Bedingung festgelegt, dass der Befehlszähler $m \in \mathbb{N}_0$ des betrachteten Prozesses auf die beschriebene Instruktion verweist. D.h. ein durch die Instruktion $\texttt{INSTR}\, x_1, \ldots, x_n$ beschriebener Zustandsübergang $(L, m, R, D) \underset{in}{\longrightarrow} (L', m', R', D')$ setzt voraus, dass $\underline{\text{Instr}}[m] = \texttt{INSTR}\, x_1, \ldots, x_n$.

5.3 Laden und Speichern

Zunächst wird die Funktion $\underline{\text{trunc}}(v, b)$ definiert, die einen Wert v so abschneidet, dass nur die letzten $|b|$ Bits übrig bleiben. Falls $b < 0$ ist, bleibt das Vorzeichen des Werts v erhalten:

$$\underline{\text{trunc}}(v, b) := \begin{cases} v \mod 2^{|b|} & \text{falls } b \geq 0 \text{ oder } v \geq 0 \\ (v \mod 2^{-b}) - 2^{-b} & \text{falls } b < 0 \text{ und } v < 0 \end{cases}$$

Diese Funktion kann also dazu verwendet werden, einen Wert v so zu kürzen, dass er in einer Variablen vom Typ t gespeichert werden kann: $\underline{\text{trunc}}(v, \underline{\text{bits}}(t))$

`LDC` c	load constant lege eine konstante Zahl c auf den Datenkeller $(L, m, R, D) \underset{in}{\longrightarrow} (L, m+1, R, D : c)$
`LDV` g	load variable lade eine lokale ($g = \underline{\text{L}}$) oder globale ($g = \underline{\text{G}}$) Variable auf den Datenkeller $(L, m, R, D : a) \underset{in}{\longrightarrow} (L, m+1, R, D : L[a])$ falls $g = \underline{\text{G}}$ $(L, m, R, D : a), G \underset{in}{\longrightarrow} (L, m+1, R, D : G[a]), G$ falls $g = \underline{\text{L}}$
`LDS` s	load special variable lade eine spezielle Variable auf den Datenkeller $(\{\pi, \pi_1, \ldots, \pi_n\}, e, o, G, \Phi) \underset{in}{\longrightarrow} (\{\pi', \pi_1, \ldots, \pi_n\}, e, o, G, \Phi)$ mit $\pi = (p, M, (L, m, R, D))$ und $\pi_i = (p_i, M_i, (L_i, m_i))$ für alle $1 \leq i \leq n$ und $\pi' = (p, M, (L, m+1, R, D : v))$ $v = \begin{cases} 0 & \text{falls } s = \underline{\text{timeout}} \text{ und } \mathit{timeout} = \underline{\text{false}} \\ 1 & \text{falls } s = \underline{\text{timeout}} \text{ und } \mathit{timeout} = \underline{\text{true}} \\ p & \text{falls } s = \underline{\text{pid}} \\ \lvert I_{act} \rvert + 1 & \text{falls } s = \underline{\text{nrpr}} \text{ mit } I_{act} = \{i \mid M_i \neq \underline{\text{T}}\} \\ \mathit{last} & \text{falls } s = \underline{\text{last}} \\ v & \text{falls } s = \underline{\text{np}} \text{ mit } v = \begin{cases} 0 & \text{falls } \exists i : \underline{\text{progress}} \in \underline{\text{Flags}}[m_i] \\ 1 & \text{sonst} \end{cases} \end{cases}$
`STV` g	store variable speichere eine lokale ($g = \underline{\text{L}}$) oder globale ($g = \underline{\text{G}}$) Variable vom Datenkeller $(L, m, R, D : v : a) \underset{in}{\longrightarrow} (L[a/v], m+1, R, D)$ falls $g = \underline{\text{L}}$ $(L, m, R, D : v : a), G \underset{in}{\longrightarrow} (L, m+1, R, D), G[a/v]$ falls $g = \underline{\text{G}}$
`TRUNC` b	truncate value schneide den Wert einer Variablen ab $(L, m, R, D : v) \underset{in}{\longrightarrow} (L, m+1, R, D : w)$ $w := \underline{\text{trunc}}(v, b)$ b enthält die Anzahl der Bits der Ziel-Variable

Da oft ein Zugriff auf elementare Variablen erfolgt, werden dafür eigene Instruktionen eingeführt:

`LDVA` g, a	load variable at address lade eine lokale ($g = \underline{\text{L}}$) oder globale ($g = \underline{\text{G}}$) Variable auf den Datenkeller $(L, m, R, D) \underset{in}{\longrightarrow} (L, m+1, R, D : L[a])$ falls $g = \underline{\text{L}}$ $(L, m, R, D), G \underset{in}{\longrightarrow} (L, m+1, R, D : G[a]), G$ falls $g = \underline{\text{G}}$
`STVA` g, a	store variable at address speichere eine lokale ($g = \underline{\text{L}}$) oder globale ($g = \underline{\text{G}}$) Variable vom Datenkeller $(L, m, R, D : v) \underset{in}{\longrightarrow} (L[a/v], m+1, R, D)$ falls $g = \underline{\text{L}}$ $(L, m, R, D : v), G \underset{in}{\longrightarrow} (L, m+1, R, D), G[a/v]$ falls $g = \underline{\text{G}}$

Diese beide Instruktionen lassen sich auch als Makro implementieren:

$$\texttt{LDVA/STVA}\ g, a \ := \ \begin{array}{l} \texttt{LDC}\ a \\ \texttt{LDV/STV}\ g \end{array}$$

5.4 arithmetische und boolsche Operationen

`ADD`	add addiere zwei Werte auf dem Datenkeller $(L, m, R, D : u : v) \underset{in}{\longrightarrow} (L, m+1, R, D : u+v)$ analog für andere arithmetische Operationen: `SUB`, `MUL`, `DIV`, `MOD`
`NEG`	negate ändere das Vorzeichen eines Werts auf dem Datenkeller $(L, m, R, D : v) \underset{in}{\longrightarrow} (L, m+1, R, D : -v)$
`NOT`	bitwise not invertiere einen Wert auf dem Datenkeller bitweise $(L, m, R, D : v) \underset{in}{\longrightarrow} (L, m+1, R, D : \neq v)$
`AND`	bitwise and verUNDe zwei Werte auf dem Datenkeller bitweise $(L, m, R, D : u : v) \underset{in}{\longrightarrow} (L, m+1, R, D : u \wedge v)$ analog für andere bitweise Operationen: `OR`, `XOR`, `SHL`, `SHR`
`EQ`	check if equal prüfe ob zwei Werte auf dem Datenkeller gleich sind $(L, m, R, D : u : v) \underset{in}{\longrightarrow} (L, m+1, R, D : w)$ $w := 1$ falls $u = v$, $w := 0$ sonst analog für andere Relations-Operatoren: `NEQ`, `LT`, `LTE`, `GT`, `GTE`
`BNOT`	boolean not negiere einen Wert auf dem Datenkeller boolsch $(L, m, R, D : v) \underset{in}{\longrightarrow} (L, m+1, R, D : w)$ $w := 1$ falls $v = 0$, $w := 0$ sonst
`BAND`	boolean and verUNDe zwei Werte auf dem Datenkeller boolsch $(L, m, R, D : u : v) \underset{in}{\longrightarrow} (L, m+1, R, D : w)$ $w := 1$ falls $u \neq 0$ und $v \neq 0$, $w := 0$ sonst analog für andere boolsche Operationen: `BOR`

5.5 Test-Befehle für Laufzeitfehler

`ICHK` n	index check prüfe, dass Feld-Index zulässig für Feld der Größe n ist $(L, m, R, D : i) \underset{in}{\longrightarrow} (L, m+1, R, D : i)$ falls $0 \leq i < n$ sonst Laufzeitfehler (Index außerhalb des zulässigen Bereichs)
`BCHK`	boolean check prüfe, dass Wert auf Datenkeller nicht 0 ist $(L, m, R, D : v) \underset{in}{\longrightarrow} (L, m+1, R, D)$ falls $v \neq 0$ sonst Laufzeitfehler (Behauptung nicht erfüllt)

5.6 deterministische Sprünge

`JMP` a	jump fahre mit Programmausführung an Adresse a fort $(L, m, R, D) \underset{in}{\longrightarrow} (L, a, R, D)$
`JMPZ` a	jump if zero fahre mit Programmausführung an Adresse a fort, falls Wert $= 0$ $(L, m, R, D : 0) \underset{in}{\longrightarrow} (L, a, R, D)$ $(L, m, R, D : v) \underset{in}{\longrightarrow} (L, m + 1, R, D)$ für $v \neq 0$
`JMPNZ` a	jump if not zero fahre mit Programmausführung an Adresse a fort, falls Wert $\neq 0$ $(L, m, R, D : 0) \underset{in}{\longrightarrow} (L, m + 1, R, D)$ $(L, m, R, D : v) \underset{in}{\longrightarrow} (L, a, R, D)$ für $v \neq 0$

5.7 Register-Instruktionen

In einigen Situationen ist es notwendig einen Wert vom Datenkeller zwischenzuspeichern, um ihn anschließend mehrfach zu verwenden. Dazu stehen folgende Befehle für den Datenaustausch zwischen Datenkeller und Registern zur Verfügung:

`TOP` r_i	place topmost value into register kopiere den obersten Wert vom Datenkeller in ein Register $(L, m, R, D : v) \underset{in}{\longrightarrow} (L, m + 1, R[i/v], D : v)$
`POP` r_i	move value into register lege den obersten Wert vom Datenkeller in ein Register $(L, m, R, D : v) \underset{in}{\longrightarrow} (L, m + 1, R[i/v], D)$
`PUSH` r_i	get value from register lege einen Wert aus einem Register auf den Datenkeller $(L, m, R, D) \underset{in}{\longrightarrow} (L, m + 1, R, D : R[i])$

Anmerkung: Die `TOP`-Instruktion kann mit Hilfe von `POP` und `PUSH` als Makro implementiert werden:

$$\texttt{TOP}\ r_i \;:=\; \begin{array}{l}\texttt{POP}\ r_i\\ \texttt{PUSH}\ r_i\end{array}$$

Für Schleifen eignen sich Register ebenfalls wesentlich besser als der Datenkeller. Deshalb gibt es diese Befehle zu Realisierung von Schleifen:

`INC` r_i	increment register erhöhe Wert in Register um 1 $(L, m, R, D) \underset{in}{\longrightarrow} (L, m+1, R[i/R[i]+1], D)$
`DEC` r_i	decrement register erniedrige Wert in Register um 1 $(L, m, R, D) \underset{in}{\longrightarrow} (L, m+1, R[i/R[i]-1], D)$
`LOOP` r_i, a	loop instruction erniedrige Wert in Register um 1 und springe falls danach größer als 0 $(L, m, R, D) \underset{in}{\longrightarrow} (L, m+1, R[i/R[i]-1], D)$ falls $R[i]-1 \leq 0$ $(L, m, R, D) \underset{in}{\longrightarrow} (L, a, R[i/R[i]-1], D)$ falls $R[i]-1 > 0$

Anmerkung: Diese drei Instruktionen können als Makro mit Hilfe von `POP` und `PUSH` sowie den Instruktionen zur Auswertung der Ausdrücke auf dem Datenkeller realisiert werden:

```
INC/DEC r_i := PUSH r_i
               LDC 1
               ADD/SUB
               POP r_i
LOOP r_i, a := DEC r_i
               PUSH r_i
               LDC 0
               GT
               JMPNZ a
```

Zur Manipulation der einzelnen Bits des Flag-Registers r_F stehen folgende Instruktionen zur Verfügung:

`FCLR`	clear flags lösche alle Flags $(L, m, R, D) \underset{in}{\longrightarrow} (L, m+1, R[F/0], D)$
`FGET` f	get flag lege Flag f auf den Datenkeller $(L, m, R, D) \underset{in}{\longrightarrow} (L, m+1, R, D : v)$ mit $R[F] = \sum b_i 2^i$ für $b_i \in \{0, 1\}$ $v := 1$ falls Bit $b_f = 1$, $v := 0$ sonst
`FSET` f	set flag setzte Flag f abhängig von Wert auf Datenkeller $(L, m, R, D : v) \underset{in}{\longrightarrow} (L, m+1, R[F/u], D)$ mit $R[F] = \sum b_i 2^i$ für $b_i \in \{0, 1\}$ und $u = R[F] + (1 - b_f) 2^f$ falls $v \neq 0$ bzw. $u = R[F] - b_f 2^f$ falls $v = 0$

5.8 Unterprogramm-Instruktionen

Zur Vereinfachung immer wiederkehrender Berechnungen, wie z.B. die Auswertung bestimmter Eigenschaften von Zuständen vor jedem Schritt-Ende, stehen die Instruktionen

`CALL` und `RET` zur Realisierung von Unterprogrammen zur Verfügung. Da die Rücksprungadresse dabei auf dem Datenkeller liegt, darf in einem solchen Unterprogramm kein Schritt-Ende liegen, da dadurch der Inhalt des Datenkellers und somit auch die Rücksprungadresse verloren geht:

`CALL` a	call subroutine Unterprogrammaufruf $(L, m, R, D) \underset{in}{\longrightarrow} (L, a, R, D : m + 1)$
`RET`	return from subroutine Rücksprung aus Unterprogramm $(L, m, R, D : a) \underset{in}{\longrightarrow} (L, a, R, D)$

5.9 Nichtdeterminismus

`NDET` a	non-determinism fahre mit Programmausführung an Adresse a und an folgender Adresse fort $(L, m, R, D) \underset{in}{\longrightarrow} (L, m + 1, R, D)$ $(L, m, R, D) \underset{in}{\longrightarrow} (L, a, R, D)$

Diese Instruktion verdoppelt den Zustand. In der einen Kopie wird der Programmzähler erhöht, in der andern Kopie wird er auf a gesetzt. Im folgenden müssen beide Kopien des Zustands betrachtet werden.

`ELSE` a	else case fahre mit Programmausführung an Adresse a fort, falls Ausführung an folgender Adresse keinen Schritt beendet $(L, m, R, D) \underset{in}{\longrightarrow} (L, m + 1, R, D)$ $(L, m, R, D) \underset{in}{\longrightarrow} (L, a, R, D)$ falls **nicht** $(L, m + 1, R, D) \overset{*}{\underset{in}{\longrightarrow}} X \underset{end}{\longrightarrow} Y$
`UNLESS` a	unless executable fahre mit Programmausführung an folgender Adresse fort, falls Ausführung an Adresse a keinen Schritt beendet $(L, m, R, D) \underset{in}{\longrightarrow} (L, a, R, D)$ $(L, m, R, D) \underset{in}{\longrightarrow} (L, m + 1, R, D)$ falls **nicht** $(L, a, R, D) \overset{*}{\underset{in}{\longrightarrow}} X \underset{end}{\longrightarrow} Y$

Diese Instruktionen verhalten sich wie `NDET`, jedoch wird einer der beiden Ausführungspfade nur betrachtet, wenn im anderen Ausführungspfad kein Schritt-Zustand erreicht wird.
Mit Hilfe dieser Instruktion können deterministische Alternativen wie `else` in `if`-Anweisungen oder `unless`-Konstrukte realisiert werden.

5.10 Kanal-Instruktionen

Es werden eine Instruktion zur Erstellung eines Kanals und Instruktionen zur Zustandansabfrage benötigt:

`CHNEW` l, n	channel new erzeuge einen neuen (und leeren Kanal) mit maximaler Länge l der Nachrichten-Typ des Kanals besteht aus n Basistypen $(L, m, R, D : b_1 : \cdots : b_n), \Phi$ $\underset{in}{\longrightarrow} (L, m+1, R, D : c), \Phi \cup \{(c, l, (b_1, \ldots, b_n), \epsilon)\}$ mit $(p, M, (L, m, R, D : b_1 : \cdots : b_n)) \in \Pi$ und c basierend auf p mit $c \notin \{c' \| (c', l', t', C') \in \Phi\} \cup \{0\}$
`CHMAX`	get maximum channel length liefere die maximale Länge des Kanals zurück $(L, m, R, D : c), \Phi \underset{in}{\longrightarrow} (L, m+1, R, D : \max(1, l)), \Phi$ falls $(c, l, t, C) \in \Phi$
`CHLEN`	get channel length liefere die aktuelle Länge des Kanals zurück $(L, m, R, D : c), \Phi \underset{in}{\longrightarrow} (L, m+1, R, D : k), \Phi$ falls $(c, l, t, c_1 : \cdots : c_k) \in \Phi$

Um die Sende-Möglichkeit schneller abprüfen zu können, wird eine eigene Instruktionen definiert:

`CHFREE`	get free space in channel liefere den freien Platz im Kanal zurück $(L, m, R, D : c), \Phi \underset{in}{\longrightarrow} (L, m+1, R, D : \max(1, l) - k), \Phi$ falls $(c, l, t, c_1 : \cdots : c_k) \in \Phi$

Falls eine unbenutztes Register r_i zur Verfügung steht, kann diese Instruktion auch als Makro implementiert werden:

```
CHFREE := TOP r_i
          CHMAX
          PUSH r_i
          CHLEN
          SUB
```

FIFO-Senden und -Empfangen kann mit folgenden Instruktionen realisiert werden:

`CHADD`	add message to channel lege eine neue Nachricht in den Kanal $(L, m, R, D : c), \Phi \underset{in}{\longrightarrow} (L, m+1, R, D), (\Phi \setminus \{\varphi\}) \cup \{\varphi'\}$ falls $\varphi = (c, l, t, c_1 : \cdots : c_k)$, $\varphi' = (c, l, t, c_1 : \cdots : c_{k+1})$, $\varphi \in \Phi$ und $k < \max(l, 1)$, $t = (b_0, \ldots, b_{n-1})$, $c_{k+1} = 0^n$
`CHSET`	set value in channel message setze einen Wert in der letzten Nachricht im Kanal $(L, m, R, D : c : o : v), \Phi \underset{in}{\longrightarrow} (L, m+1, R, D), (\Phi \setminus \{\varphi\}) \cup \{\varphi'\}$ falls $\varphi = (c, l, t, c_1 : \cdots : c_k)$, $\varphi' = (c, l, t, c_1 : \cdots : c_{k-1} : c'_k)$, $\varphi \in \Phi$ und $k > 0$, $t = (b_0, \ldots, b_{n-1})$, $c_k = (v_0, \ldots, v_{n-1})$, $c'_k = (v'_0, \ldots, v'_{n-1})$ $v'_o = \underline{\text{trunc}}(v, b_o)$ falls $o < n$, $v'_i = v_i$ für $i \neq o$
`CHGET`	get value from channel message hole einen Wert aus der ersten Nachricht im Kanal $(L, m, R, D : c : o), \Phi \underset{in}{\longrightarrow} (L, m+1, R, D : v), \Phi$ falls $\varphi = (c, l, t, c_1 : \cdots : c_k)$, $\varphi \in \Phi$ und $k > 0$, $t = (b_0, \ldots, b_{n-1})$, $c_1 = (v_0, \ldots, v_{n-1})$ $v = v_o$ falls $o < n$, $v = 0$ sonst
`CHDEL`	delete message from channel lösche die erste Nachricht aus dem Kanal $(L, m, R, D : c), \Phi \underset{in}{\longrightarrow} (L, m+1, R, D), (\Phi \setminus \{\varphi\}) \cup \{\varphi'\}$ falls $\varphi = (c, l, t, c_1 : C)$, $\varphi' = (c, l, t, C)$, $\varphi \in \Phi$

Beim Senden wird zunächst mit `CHADD` eine neue Nachricht in den Kanal gelegt und dann werden die einzelnen Einträge mit `CHSET` gesetzt. Beim Empfangen werden zunächst die einzelnen Einträge der Nachricht mit `CHGET` gelesen und die Bedingungen überprüft. Dann können die Einträge erneut gelesen werden, um ihre Werte den Ziel-Variablen zuzuweisen, bevor die Nachricht dann gegebenenfalls mit `CHDEL` aus dem Kanal entfernt wird.

Dadurch, dass in der Definition von `CHADD` die Anzahl der Nachrichten im Kanal mit $k < \max(l, 1)$ abgefragt wird, ist es auch möglich, eine Nachricht in einem synchronen Kanal (d.h. mit $l = 0$) abzulegen. Der dadurch entstehende Schritt-Zustand ist allerdings nicht zulässig und darf nicht nach außen hin sichtbar werden. Wenn einen solcher Zustand auftritt, wird sofort danach geprüft, ob es einen Prozess gibt, der die Nachricht aus diesem Kanal empfängt. Sollte dies nicht der Fall sein, wird der unzulässige Zustand verworfen.

Oft erfolgt ein Zugriff auf fest Elemente einer Kanal-Nachricht. Daher gibt es folgende Instruktionen zur Beschleunigung des Zugriffs:

`CHSETO` o	set value in channel message at offset setze einen Wert an festem Offset in der letzten Nachricht im Kanal $(L, m, R, D : c : v), \Phi \underset{in}{\longrightarrow} (L, m+1, R, D), (\Phi \setminus \{\varphi\}) \cup \{\varphi'\}$ falls $\varphi = (c, l, t, c_1 : \cdots : c_k)$, $\varphi' = (c, l, t, c_1 : \cdots : c_{k-1} : c'_k)$, $\varphi \in \Phi$ und $k > 0$, $t = (b_0, \ldots, b_{n-1})$, $c_k = (v_0, \ldots, v_{n-1})$, $c'_k = (v'_0, \ldots, v'_{n-1})$ $v'_o = \underline{\text{trunc}}(v, b_o)$ falls $o < n$, $v'_i = v_i$ für $i \neq o$
`CHGETO` o	get value from channel message at offset hole einen Wert an festem Offset aus der ersten Nachricht im Kanal $(L, m, R, D : c), \Phi \underset{in}{\longrightarrow} (L, m+1, R, D : v), \Phi$ falls $\varphi = (c, l, t, c_1 : \cdots : c_k)$, $\varphi \in \Phi$ und $k > 0$, $t = (b_0, \ldots, b_{n-1})$, $c_1 = (v_0, \ldots, v_{n-1})$ $v = v_o$ falls $o < n$, $v = 0$ sonst

Anmerkung: Diese beiden Instruktionen können als Makro mit Hilfe von `CHGET` und `CHSET` implmentiert werden, falls ein unbenutztes Register r_i zur Verfügung steht:

$$\begin{aligned}
\texttt{CHSETO}\ o &:= \texttt{POP}\ r_i \\
&\ \texttt{LDC}\ o \\
&\ \texttt{PUSH}\ r_i \\
&\ \texttt{CHSET} \\
\texttt{CHGETO}\ o &:= \texttt{LDC}\ o \\
&\ \texttt{CHGET}
\end{aligned}$$

Für sortiertes Senden wird erst wie beim FIFO-Senden eine Nachricht am Ende des Kanals erstellt. Diese wird dann anschließend mit `CHSORT` in den Kanal einsortiert.
Die Überprüfung, ob sich eine selektiv empfangbare Nachricht im Kanal befindet, kann dadurch realisiert werden, dass die Nachrichten im Kanal zyklisch rotiert werden und jeweils die erste Nachricht auf Empfangbarkeit überprüft wird:

`CHSORT`	sort last message into channel sortiere die letzte Nachricht in den Kanal ein $(L, m, R, D : c), \Phi \underset{in}{\longrightarrow} (L, m+1, R, D), (\Phi \setminus \{\varphi\}) \cup \{\varphi'\}$ falls $\varphi = (c, l, t, c_1 : \cdots : c_k)$, $\varphi' = (c, l, t, c_1 : \cdots : c_{j-1} : c_k : c_j : \cdots : c_{k-1})$, $\varphi \in \Phi$ und $k > 0$, $j = \min(\{j' \mid 1 \leq j' < k, c_{j'} > c_k\} \cup \{k\})$
`CHROT`	rotate messages in channel verschiebe Nachrichten im Kanal zyklisch $(L, m, R, D : c), \Phi \underset{in}{\longrightarrow} (L, m+1, R, D), (\Phi \setminus \{\varphi\}) \cup \{\varphi'\}$ falls $\varphi = (c, l, t, c_1 : \cdots : c_k)$, $\varphi' = (c, l, t, : c_2 : \cdots : c_k : c_1)$, $\varphi \in \Phi$

5.11 Ausgabe

Für die Realisierung der Ausgabe müssen Zeichen und Werte einzeln ausgegeben werden:

`PRINTS` s	print string schreibe die Zeichenkette Nr. s auf die Ausgabe $(L, m, R, D) \underset{in}{\longrightarrow} (L, m+1, R, D)$
`PRINTV` f	print value schreibe den Wert v formatiert als f auf die Ausgabe $(L, m, R, D : v) \underset{in}{\longrightarrow} (L, m+1, R, D)$

Die Zeichenketten werden parallel zum Bytecode in einer Tabelle verwaltet. So ist anhand der Nummer s der Zugriff auf die Zeichenkette und deren Ausgabe möglich.
Das Format f gibt an, in welcher Art der Wert ausgegeben werden soll: als Zeichen („c"), dezimal („d"), oktal („o"), vorzeichenlos dezimal („u") oder hexadezimal („x").
Die erfolgten Ausgaben werden parallel zum Zustand gespeichert und zusammen mit dem Nachfolgezustand an den Benutzer übergeben.

5.12 Ende eines Ausführungsschritts

`STEP` $M'[,l]$ step completed

ein Ausführungsschritt wurde abgeschlossen

Prozess ist nun in Ausführungmodus $M' \in \{\underline{\mathrm{N}}, \underline{\mathrm{A}}, \underline{\mathrm{I}}, \underline{\mathrm{T}}\}$

(normal, atomar, unsichtbar (invisible), beendet (terminated))

Bezeichner l kennzeichnet die Transition ($l = \epsilon$ wenn nicht angegeben)

$$(\{(p, M, (L, m, R, D)), \pi_1, \ldots, \pi_n\}, e, o, G, \Phi) \xrightarrow[end]{M',I} (\{(p, M', (L, m+1)), \pi_1, \ldots, \pi_n\}, e', o, G, \Phi)$$

mit $e' := \begin{cases} p \text{ falls } M' \in \{\underline{\mathrm{A}}, \underline{\mathrm{I}}\} \\ \perp \text{ sonst} \end{cases}$

und $\pi_i = (p_i, M_i, (L_i, m_i))$ für alle $1 \leq i \leq n$, $I = (l, R[F])$

Mit Hilfe dieser Instruktion wird der aktuelle Zustand als ein Nachfolgezustand des Ausgangszustandes mit Ausfürhungsmodus M' markiert. Die Programmausführung hält hier zunächst an und wechselt gegebenenfalls zum nächsten lokalen Ausführungspfad.
Die Kennzeichnung l (label) des Schritts und der Wert $R[F]$ des Flag-Registers werden zu einer Information I zusammengefasst und an den Benutzer weitergemeldet. Der Benutzer (z.B. ein Model-Checker) kann die Kennzeichnung und die Flags zur Überprüfung von Korrektheits-Kriterien verwenden.
Der Inhalt des Datenkellers und der Register wird nicht mehr im Zustand gespeichert und steht damit nach der `STEP`-Instruktion nicht mehr zur Verfügung. Der Wert des Flag-Registers r_F wird allerdings zusammen mit dem Zustand an den Benutzer (z.B. einen Model-Checker) übergeben und kann dazu benutzt werden, vor der `STEP`-Instruktion in diesem Register abgelegte Korrektheits-Informationen zu übermitteln.

Falls die Ausführung eines Pfades nicht weiter möglich ist (z.B. wegen einer nicht erfüllten Wächter-Anweisung oder bei Empfang aus einem leeren Kanal), muss der Ausführungspfad abgebrochen werden. Dies geschieht dadurch, dass einfach kein Zustandsübergang für `NEX` definiert wird:

`NEX`	step not executable Ausführungsschritt ist nicht ausführbar $(L, m, R, D) \not\xrightarrow[in]{}$ und $(L, m, R, D) \not\xrightarrow[end]{}$
`NEXZ`	step not executable if zero Ausführungsschritt ist nicht ausführbar, falls Wert auf Datenkeller $= 0$ $(L, m, R, D : 0) \not\xrightarrow[in]{}$ und $(L, m, R, D : 0) \not\xrightarrow[end]{}$ $(L, m, R, D : v) \xrightarrow[in]{} (L, m+1, R, D)$ für $v \neq 0$
`NEXNZ`	step not executable if not zero Ausführungsschritt ist nicht ausführbar, falls Wert auf Datenkeller $\neq 0$ $(L, m, R, D : 0) \xrightarrow[in]{} (L, m+1, R, D)$ $(L, m, R, D : v) \not\xrightarrow[in]{}$ und $(L, m, R, D : v) \not\xrightarrow[end]{}$ für $v \neq 0$

Die beiden bedingten Instruktionen `NEXZ` und `NEXNZ` können als Makro implementiert werden:

`NEXZ/NEXNZ` := `JMPNZ/JMPZ` a
`NEX`

$a:$

5.13 Start eines Prozesses

Zum Start eines neuen Prozesses müssen dessen aktuelle Parameter v_i und deren Bit-Anzahlen b_i auf den Datenkeller gelegt werden. Dann kann unter Angabe der Anzahl der Parameter k und der Start-Adresse a der Prozesses erstellt werden:

`RUN` k, a run process
starte einen neuen Prozess
beginne dabei mit der Ausführung an Adresse a
$(\{\pi, \pi_1, \ldots, \pi_n\}, e, o, G, \Phi) \underset{in}{\longrightarrow} (\{\pi', \pi_1, \ldots, \pi_n, \pi''\}, e, o, G, \Phi)$
mit $\pi = (p, M, (L, m, R, D : b_0 : v_0 : \cdots : b_{k-1} : v_{k-1}))$
und $\pi_i = (p_i, M_i, (L_i, m_i))$ für alle $1 \leq i \leq n$
und $\pi' = (p, M, (L, m+1, R, D : p''))$
und $\pi'' = (p'', \underline{\mathrm{N}}, (L_0[0/w_0, \ldots, k-1/w_{k-1}], a))$
und $p'' := (\ \max(\ \{p' | (p', M', \Lambda') \in \{\pi, \pi_1, \ldots, \pi_n\}\}\ )\ ) + 1$
und $w_i := \underline{\mathrm{trunc}}(v_i, b_i)$ für $0 \leq i < k$

5.14 Zugriff auf andere Prozesses

`PCVAL` get program counter value
lese den Programmzähler eines anderen Prozesses
$(\{\pi, \pi_1, \ldots, \pi_n\}, e, o, G, \Phi) \underset{in}{\longrightarrow} (\{\pi', \pi_1, \ldots, \pi_n\}, e, o, G, \Phi)$
mit $\pi = (p, M, (L, m, R, D : p_j))$ mit $1 \leq j \leq n$
und $\pi_i = (p_i, M_i, (L_i, m_i))$ für alle $1 \leq i \leq n$
und $\pi' = (p, M, (L, m+1, R, D : m_j))$

`LVAR` get local variable
lese eine lokale Variable eines anderen Prozesses
$(\{\pi, \pi_1, \ldots, \pi_n\}, e, o, G, \Phi) \underset{in}{\longrightarrow} (\{\pi', \pi_1, \ldots, \pi_n\}, e, o, G, \Phi)$
mit $\pi = (p, M, (L, m, R, D : p_j : a))$ mit $1 \leq j \leq n$
und $\pi_i = (p_i, M_i, (L_i, m_i))$ für alle $1 \leq i \leq n$
und $\pi' = (p, M, (L, m+1, R, D : L_j[a]))$

`ENAB` check if enabled
prüfe, ob ein anderer Prozess ausgeführt werden kann
$(\{\pi, \pi_1, \ldots, \pi_n\}, e, o, G, \Phi) \underset{in}{\longrightarrow} (\{\pi', \pi_1, \ldots, \pi_n\}, e, o, G, \Phi)$
mit $\pi = (p, M, (L, m, R, D : p_j))$ mit $1 \leq j \leq n$
und $\pi_i = (p_i, M_i, (L_i, m_i))$ für alle $1 \leq i \leq n$
und $\pi' = (p, M, (L, m+1, R, D : v))$
und $v = 1$ falls $(\{\pi_1, \ldots, \pi_n\}, 0, G, \Phi) \underset{step'}{\overset{p_j, M, I}{\longrightarrow}} \Gamma$, $v = 0$ sonst

5.15 Monitor-Prozess

`MONITOR` set monitor process
wähle einen Prozess als Monitor-Prozess aus
$(\{\pi, \pi_1, \ldots, \pi_n\}, e, o, G, \Phi) \underset{in}{\longrightarrow} (\{\pi', \pi_1, \ldots, \pi_n\}, e, o', G, \Phi)$
mit $\pi = (p_0, M, (L, m, R, D : p))$
und $\pi_i = (p_i, M_i, (L_i, m_i))$ für alle $1 \leq i \leq n$
und $\pi' = (p_0, M, (L, m+1, R, D))$
und $o' = p_j$ falls $p = p_j$ mit $0 \leq j \leq n$, $o' = \perp$ falls $p = 0$

6 Beispiel

6.1 Promela-Programm

```
init
{
  int x;
  x = 5;
  do
    :: x == 23; x = 42
    :: else; x = 23
  od
}
```

6.2 VM-Instruktionen

	Instruktion	Kommentar
0 :	STEP $\underline{N}$	
1 :	LDC 0	// `int x`
2 :	STVA $\underline{L}$, 0	
3 :	LDC 5	// `x = 5`
4 :	TRUNC -31	
5 :	STVA $\underline{L}$, 0	
6 :	STEP $\underline{N}$	
7 :	ELSE 17	// start of `do`
8 :	LDVA $\underline{L}$, 0	// `x == 23`
9 :	LDC 23	
10 :	EQ	
11 :	NEXZ	
12 :	STEP $\underline{N}$	
13 :	LDC 42	// `x = 42`
14 :	TRUNC -31	
15 :	STVA $\underline{L}$, 0	
16 :	JMP 22	
17 :	STEP $\underline{N}$	// `else`
18 :	LDC 23	// `x = 23`
19 :	TRUNC -31	
20 :	STVA $\underline{L}$, 0	
21 :	JMP 22	
22 :	STEP $\underline{N}$	// `od`
23 :	JMP 7	

7 Implementierung der Virtuellen Maschine

Die Ziele bei der Implementierung der Virtuellen Maschine waren neben der möglichst exakten Umsetzung des formalen Modells eine portable Implementierung und eine im Vergleich zu Spin gute Leistung in Bezug auf Speicherbedarf und Geschwindigkeit.

Daher wurde die Programmiersprache C gewählt, die sich auf fast allen Architekturen übersetzen und an nahezu jede andere Programmiersprache anbinden lässt und dabei eine herausragende Geschwindigkeit bei geringem Speicherbedarf zeigt.

Bedingt durch die nichtdeterministische Struktur der Virtuellen Maschine und die Möglichkeit zur spekulativen Ausführung wurde eine Implementierung „von Hand“ gewählt, da VM-Generatoren wie z.B Vmgen oder LLVM für solche Maschinen-Strukturen nicht vorgesehen sind und eine Implementierung mit Hilfe solcher Werkzeuge an den genannten Punkten auf zusätzliche Schwierigkeiten stoßen würde.

7.0.1 Abweichungen vom formalen Modell

Es ist nicht ohne große Einbußen in der Geschwindigkeit und im Speicherverbrauch möglich, in einer Implementierung mit dem Datentyp $\mathbb{Z}$ zu arbeiten. Daher stellen verschiedene Datentypen den wesentlichen Unterschied zum formalen Modell dar.

Globale und lokale Variablen, sowie die elementare Typen in Kanal-Nachrichten haben eine Länge von 8, 16 oder 32 Bit und sind eventuell vorzeichenbehaftet. Da allerdings eine Unterscheidung in Bezug auf das Vorzeichen für den Typ mit 32 Bit nicht notwendig ist, ergibt sich eine Gesamtzahl von 5 Datentypen in der Virtuellen Maschine.
Werte auf dem Datenkeller und in Registern haben grundsätzlich 32 Bit, eine Konvertierung erfolgt implizit in den Instruktionen zum Laden und Speichern. Diese interne Wortbreite ist aber im Quelltext sehr leicht zu ändern, da hierfür ein eigener Typ-Name verwendet wurde.

Da auch die Realisierung der globalen und lokalen Variablen durch unendliche Funktionen $\mathbb{N}_0 \rightarrow \mathbb{Z}$ in einer Implementierung nicht möglich ist, haben sowohl globale als auch lokale Variablen eine Maximalgröße. Zugriffe auf nicht existierende Adressen führen dadurch zu Laufzeitfehlern.
Die Größe der lokalen Variablen wird als zusätzlicher Parameter bei der Prozess-Erzeugung angegeben. Zusätzlich ist eine Änderung dieser Größe sowie der Größe der globalen Variablen jeweils mit Hilfe einer speziellen Instruktion möglich.

Die Anzahl der Prozesse ist wie in Spin auf 255 beschränkt, und sowohl auf globaler Ebene wie auch für jeden Prozess sind maximal 255 Kanäle zugelassen. Diese Grenzwerte können jedoch einfach durch Konstanten verändert werden, wenn gleichzeitig im Compiler die Größen der Typen für Prozess- und Kanal-Kennzahlen angepasst werden.

7.1 Zustand als Speicherblock

Um dem Benutzer der Virtuellen Maschine eine möglichst einfache Handhabung der Zustände zu ermöglichen, wurde der Zustand als flacher Speicherblock ohne Zeiger realisiert. Somit kann ein Zustand im Speicher verschoben werden, auf einen Massenspeicher bis zur weiteren Verwendung ausgelagert werden oder über ein Netzwerk versendet werden, ohne dass eine Serialisierung erforderlich ist.
Da in einem Model Checker sehr viele Zustände verwaltet werden müssen und der Zugriff auf gespeicherte Zustände schnell erfolgen muss, wird oft Hashing angewendet. Auch hier erweist sich das flache Format der Zustände als vorteilhaft, da eine einfache Hashfunktion für Speicherbereiche direkt angewendet werden kann.

7.1.1 Aufbau eines Zustands

Um im vorhandenen Speicher so viele Zustände wie möglich unterbringen zu können, muss ein Zustand so klein wie möglich sein. Deshalb beinhaltet ein Zustand keine redundanten Informationen, die auch berechnet werden können, wie z.B. die Gesamtlänge.

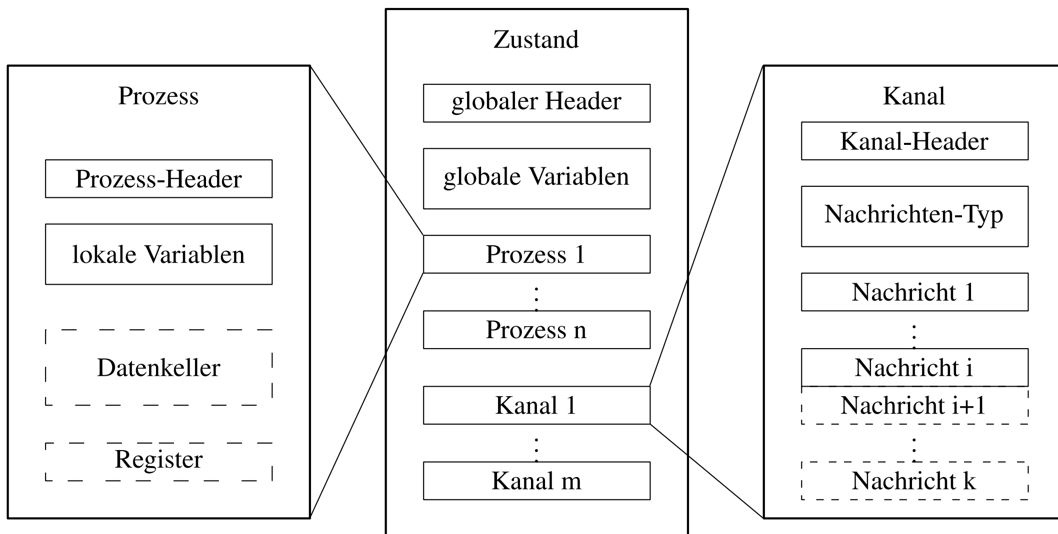

Abbildung 1: Aufbau eines Zustands

Wie Abbildung 1 zeigt, besteht ein Zustand aus einem globalen Header, den globalen Variablen, den Prozessen und den Kanälen. Der globale Header enthält die Kennzahl des exklusiv ausgeführten Prozesses, die Kennzahl des Monitor-Prozesses, die Größe der globalen Variablen, die Anzahl der Prozesse und die Anzahl der Kanäle.
Prozesse innerhalb des Zustands beginnen mit dem Prozess-Header, worin die Kennzahl des Prozesses, der aktuelle Ausführungsmodus, der Befehlszähler und die Größe der lokalen Variablen enthalten ist. Es folgen die lokalen Variablen und im Falle eines aktiven Prozesses (durch ein Flag im Prozess-Header angezeigt) zusätzlich noch der Datenkeller und die Register.
Ebenso beginnen auch Kanäle mit einem Kanal-Header, der die Kennzahl, die aktuelle und maximale Länge des Kanals, die Länge einer Nachricht und die Anzahl elementarer Typen in einer Nachricht enthält. Nach dem Nachrichten-Typ, dargestellt als die Bit-Anzahlen der elementaren Typen, folgen die Nachrichten. Um nicht bei jeder Sende- und Empfangs-Operation die Größe des Zustands ändern zu müssen, ist in jedem Kanal stets Speicher für die maximale Anzahl Nachrichten enthalten. Die unbenutzten Nachrichten-Plätze sind dabei mit 0 gefüllt, damit sich die Zustände nicht in diesen Plätzen unterscheiden können.

7.1.2 Maschinen-unabhängiger Zustand

Um die Virtuelle Maschine ohne größeren Aufwand in einem *parallelen Model Checker* auf *heterogener Hardware* einsetzen zu können, muss das Zustands-Format für inaktive Zustände, d.h. Zustände, in denen kein Prozess aktiv ist, auf jeder Maschine identisch sein. Erreicht wird dieses Ziel durch Packen der für die Header verwendeten Strukturen und die Ablage aller Werte in *network byte order*, d.h. mit dem höchstwertigsten Byte zuerst. Nur für die Werte auf dem Datenkeller und in den Registern wird eine Ausnahme gemacht und das native Format der Maschine verwendet. Dies spart Zeit für die Konvertierung ein und ist möglich, da nur aktive Zustände, die nicht nach außen hin sichtbar werden, einen Datenkeller und Register besitzen.

7.2 Veränderte und zusätzliche Instruktionen

Da die Variablen unterschiedliche Größen besitzen und die Gesamtgröße der globalen und lokalen Variablen beschränkt ist, sind einige Änderungen an den Instruktionen des formalen Modells sowie einige zusätzliche Instruktionen notwendig.

7.2.1 Zugriff auf Variablen

Beim Zugriff auf eine Variable muss deren Größe bekannt sein. Dies kann dadurch erreicht werden, indem man in den Instruktionen zum Zugriff auf Variablen (`LDV`, `LDVA`, `LDV`, `LVAR`, `STV` und `STVA`) einen zusätzlichen Parameter einführt, der die Größe der angesprochenen Variable sowie eine Information über die Existenz eines Vorzeichen-Bits enthält.
Dieser zusätzliche Größen-Parameter kann $1u$ für vorzeichenlose bzw. $1s$ für vorzeichenbehaftete 8-Bit-Variablen, $2u$ für vorzeichenlose bzw. $2s$ für vorzeichenbehaftete 16-Bit-Variablen oder 4 für 32-Bit-Variablen sein.

7.2.2 Code-Adressen

In Instruktionen, die einen Adress-Parameter besitzen, wie beispielsweise `JMP`, werden in der Implementierung relative Adressen verwendet. Die relative Adresse ist dabei ein vorzeichenbehafteter 16-Bit-Wert und bezieht sich auf die Adresse hinter der Instruktion.
Von den Instruktionen `JMP`, `CALL` und `RUN` existieren zusätzlich lange Varianten, die anstelle der relativen Adresse einen vorzeichenlosen 32-Bit-Wert als absolute Adresse verwenden. Diese Variante wird jeweils durch ein vorangestelltes `L` (long) gekennzeichnet: `LJMP`, `LCALL`, `LRUN`

7.2.3 Gesamtgröße der Variablen

Bei der Erzeugung eines neuen Prozesses mit `RUN` oder `LRUN` muss die Größe der lokalen Variablen des neuen Prozesses bekannt gegeben werden. Dazu wurden die Instruktionen um einen Parameter ergänzt, der die initiale Gesamtgröße der lokalen Variablen festlegt.
Eine Änderung der Gesamtgröße der lokalen Variablen des aktuellen Prozesses ist noch mit `LOCSZ` l möglich, wobei l die neue Größe angibt. Neu angelegte Variablen werden mit 0 initialisiert.
Ebenso existiert für die Größenänderung der globalen Variablen die Instruktion `GLOBSZ` g mit g als neue Größe der globalen Variablen.
Diese beiden Instruktionen werden üblicherweise nur während der Initialisierung eingesetzt, weil der initiale Zustand der Virtuellen Maschine keine globalen Variablen und keine lokalen Variablen im initialen Prozess besitzt und diese beiden Parameter somit nachträglich noch geändert werden müssen. Andere Prozesse werden direkt mit der gewünschten Anzahl lokaler Variablen erzeugt.

7.3 Berechnung der Nachfolger-Zustände mit Callbacks

Nach außen hin zeigt die Virtuelle Maschine prinzipiell eine sehr einfache Schnittstelle zur Berechnung der Nachfolger-Zustände. Ausgehend von einem Zustand im Speicher, der von der Virtuellen Maschine nur gelesen wird, werden die Nachfolger-Zustände intern generiert und dann mittels einer Callback-Funktion an den Benutzer übergeben, der diese Zustände ebenfalls nur lesen darf und für eine eventuelle Speicherung selbst verantwortlich ist.

Zusammen mit einem Nachfolger-Zustand erhält der Benutzer im Callback weitere Informationen, wie z.B. den Wert des Flag-Registers, die eventuell vorhandene in der `STEP`-Instruktion verwendete Marke, Informationen über das Auftreten von Rendezvous-Kommunikation oder eines Timeouts sowie Informationen über den Zustand des Monitor-Prozesses.

Die während der Berechnung der Nachfolger-Zustände durch die `PRINTx`-Instruktionen ausgelösten Ausgabe-Ereignisse werden parallel zum Zustand verwaltet und in Form einer linearen Liste an das Nachfolger-Callback übergeben. Damit ist die Zuordnung der Ausgaben zum entsprechenden Nachfolger-Zustand sichergestellt und der Benutzer kann entscheiden, ob die Ausgabe durchgeführt wird oder die Ausgabe-Ereignisse ignoriert werden.

Über eine zweite Callback-Funktion wird der Benutzer beim Auftreten von Laufzeitfehlern informiert. Mit Hilfe des Rückgabewerts des Fehler-Callbacks (sowie des Nachfolger-Callbacks) kann der Benutzer festlegen, ob die Generierung von Nachfolger-Zuständen abgebrochen oder fortgesetzt werden soll.

7.3.1 Interna der Berechnung von Nachfolger-Zuständen

Intern läuft die Berechnung von Nachfolger-Zuständen eng angelehnt an das formale Modell der Virtuellen Maschine ab.

Es erfolgt auf unterster Ebene zunächst die Aktivierung eines Prozesses, der im Anschluss daran ausgeführt wird, bis er wieder einen inaktiven Zustand erreicht oder den Ausführungspfad abbricht. Wird während der Ausführung ein neuer Ausführungspfad angelegt, so wird dieser zwischengespeichert und nach der Deaktivierung bzw. nach dem Abbruch des vorherigen Ausführungspfads bearbeitet.
Von allen berechneten Schritt-Zuständen werden die sichtbaren an die nächsthöhere Ebene übergeben, die unsichtbaren in eine Warteschlange eingetragen.
Im ersten Zustand dieser Warteschlange wird dann der ausgeführte Prozess erneut aktiviert und bis zum Erreichen des nächsten Schritts ausgeführt. Dies wird so lange wiederholt, bis sich keine unsichtbaren Zustände mehr in der Warteschlange befinden. Es dürfen somit keine Zyklen ohne sichtbare Schritte existieren, da dies zu einer Endlosschleife führt.

Die höheren Ebenen sind als Callback-Funktionen implementiert, die die Nachfolger-Zustände der unteren Ebene erhalten und gegebenenfalls erneut an die untere Ebene zur Erledigung weiterer Prozess-Ausführungen übergeben, wie z.B. für die Abwicklung von synchroner Kommunikation oder die Ausführung des Monitor-Prozesses.
Ist ein Nachfolger-Zustand erreicht, der nach außen hin sichtbar werden soll, so wird er an das vom Benutzer zur Verfügung gestellte Callback übergeben.

7.4 Zerlegung von Zuständen zur Speichereinsparung

Bei der Ausführung eines Schritts verändert sich meist nur der Zustand eines System-Prozesses und eventuell der Zustand des Monitor-Prozesses. Daher enthalten viele Zustände gleiche Informationen in Form von gleichen Prozessen und gleichen Kanälen.
Durch Ausnutzung dieses Wissens über diese redundante Information in den Zuständen ist es möglich Speicher einzusparen, indem verschiedene Zustände den selben Speicherplatz für gleiche Prozesse bzw. Kanäle verwenden.

Um einem Benutzer der Virtuellen Maschine den Zugriff auf einzelne Prozesse und Kanäle zur Reduzierung des Speicherbedarfs nach obiger Idee zu erlauben, wird die Möglichkeit der Zerlegung von Zuständen geboten.
Dazu übergibt der Benutzer einen kompletten Zustand an eine Funktion, die dann drei verschiedene Callbacks mit dem globalen Teil des Zustands, den Prozessen und den Kanälen als flache Speicherblöcke aufruft.
Diese Speicherblöcke kann der Benutzer nun in geeigneter Form abspeichern und so die Redundanzen gleicher Prozesse und gleicher Kanäle ausnutzen.
Für die Wiederherstellung eines Zustands wird eine andere Funktion bereitgestellt, die drei verschiedene Callbacks zum Abruf des globalen Teil des Zustands, der Prozesse und der Kanäle benutzt.

7.5 Assembler

Die Virtuelle Maschine erwartet aus Effizienzgründen ein binäres Bytecode-Format als Eingabe, welches nur sehr schwer von Hand erzeugbar ist. Um die Erstellung zu vereinfachen, wurde der Assembler implementiert. Somit ist es möglich, den Zwischencode in einem eng an das formale Modell angelehnten, menschenlesbaren Text-Format zu erstellen und dann automatisiert in die Binär-Darstellung zu übersetzen.
Da eine zeilenweise Übersetzung der Instruktionen von der Text- in die Binär-Darstellung keine interessante Aufgabe ist, wurde Wert auf eine kurze Entwicklungszeit gelegt. Dies führte zur Auswahl der Sprache Perl, die sich u.a. durch die exzellente Unterstützung von regulären Ausdrücken bestens zum Erkennen von Sprungmarken, Instruktionen und deren Parametern eignet. Die relativ geringe Ausführungsgeschwindigkeit dieser interpretierten Sprache im Vergleich zu kompilierten Sprachen ist hier auf Grund der geringen Komplexität der Aufgabe nicht entscheidend.

7.5.1 Eingabe-Format

Die Eingabe des Zwischencodes in den Assembler erfolgt als Text in der im formalen Modell verwendeten Darstellung der Instruktionen. Weiter können mit Hilfe der Direktive `!flags` die Flags `progress` und `accept` für die aktuelle Adresse gesetzt werden.
Strings, die parallel zum Zwischencode für die `PRINTx`-Instruktionen benötigt werden, können mit Hilfe von `!string` unter Angabe der String-Nummer und des in Anführungszeichen eingeschlossenen Textes in die String-Tabelle eingetragen werden.
Weitere Direktiven erlauben das Ablegen mehrerer Module in einer Datei sowie die Ergänzung des Zwischencodes mit Zusatzinformationen aus dem Quelltext (Position im Quelltext, Beginn und Ende syntaktischer Einheiten), um diese für den eventuellen Einsatz eines Optimierers zu erhalten.
Eine detaillierte Beschreibung des Eingabe-Formats sowie einige Beispiele liegen dem Assembler bei.

7.5.2 Ausgabe-Format

Das Ausgabe-Format des Assemblers ist natürlich das Bytecode-Format der Virtuellen Maschine und enthält mindestens ein Modul, kann aber auch mehrere Module enthalten.
Jedes Modul besteht neben seinem Namen im wesentlichen aus dem Bytecode, einer Tabelle mit den Flags für einige Adressen und der String-Tabelle. Zusätzlich können noch Zusatzinformationen aus dem Quelltext enthalten sein.
Auch für das binäre Ausgabe-Format existiert eine Beschreibung, die der Virtuellen Maschine beiliegt.

Inhaltsverzeichnis

1 Konzept der virtuellen Maschine **1**
- 1.1 Prozess-Schritte . . . 1
- 1.2 Nichtdeterminismus . . . 1
- 1.3 Kanäle . . . 2

2 lokaler Zustand eines Prozesses **2**
- 2.1 lokale Variablen . . . 2
- 2.2 Register . . . 2
- 2.3 Datenkeller, Auswertung von Ausdrücken . . . 3
- 2.4 lokaler Zustand nach einem Schritt . . . 3

3 globaler Zustand 3
- 3.1 aktive Prozesse 4
- 3.2 globale Variablen 4
- 3.3 Kanäle 4
- 3.4 Nichtdeterminismus 5

4 Beginn und Ende eines Schritts 5
- 4.1 initialer Zustand 5
- 4.2 Zwischenzustände bei synchroner Kommunikation 5
- 4.3 Beginn eines Schritts 5
- 4.4 lokale Schritte 6
  - 4.4.1 Nicht-Ausführbarkeit eines Schritts 7
- 4.5 globale Schritte 7
  - 4.5.1 synchrone Kommunikation 7
- 4.6 Timeout-Situationen 7
- 4.7 Monitor-Prozess und Nachfolgezustände 8

5 Instruktionen 9
- 5.1 abkürzende Schreibweisen 9
- 5.2 aktuelle Instruktion 10
- 5.3 Laden und Speichern 10
- 5.4 arithmetische und boolsche Operationen 12
- 5.5 Test-Befehle für Laufzeitfehler 12
- 5.6 deterministische Sprünge 13
- 5.7 Register-Instruktionen 13
- 5.8 Unterprogramm-Instruktionen 14
- 5.9 Nichtdeterminismus 15
- 5.10 Kanal-Instruktionen 15
- 5.11 Ausgabe 18
- 5.12 Ende eines Ausführungsschritts 19
- 5.13 Start eines Prozesses 20
- 5.14 Zugriff auf andere Prozesses 20
- 5.15 Monitor-Prozess 20

6 Beispiel 21
- 6.1 Promela-Programm 21
- 6.2 VM-Instruktionen 21

7 Implementierung der Virtuellen Maschine 21
  - 7.0.1 Abweichungen vom formalen Modell 22
- 7.1 Zustand als Speicherblock 22
  - 7.1.1 Aufbau eines Zustands 22
  - 7.1.2 Maschinen-unabhängiger Zustand 23
- 7.2 Veränderte und zusätzliche Instruktionen 24
  - 7.2.1 Zugriff auf Variablen 24
  - 7.2.2 Code-Adressen 24
  - 7.2.3 Gesamtgröße der Variablen 24
- 7.3 Berechnung der Nachfolger-Zustände mit Callbacks 24
  - 7.3.1 Interna der Berechnung von Nachfolger-Zuständen 25
- 7.4 Zerlegung von Zuständen zur Speichereinsparung 25
- 7.5 Assembler 26
  - 7.5.1 Eingabe-Format 26
  - 7.5.2 Ausgabe-Format 26