

Promela - Operationelle Semantik

Stefan Schürmans

29. August 2005

1 Informationen aus der Symboltabelle

Für die Definition der operationellen Semantik werden an vielen Stellen verschiedene Informationen aus der Symboltabelle $\widehat{\text{st}}_P$ des Promela-Programms benötigt. Nachfolgend werden einige Hilfsfunktionen definiert, die diese Informationen bereitstellen.

1.1 Typ einer Variablen-Referenz

Die Funktion vtype (variable type) bestimmt ausgehend von einer Variablen-Referenz varref und dem aktuellen Gültigkeitsbereich s den Typ der Variablen-Referenz vtype(varref, s):

$$\begin{aligned} \underline{\text{vtype}}(\underline{\text{name}}, s) &:= t && \text{falls } \widehat{\text{st}}_P(\underline{\text{name}}, s) = (\underline{\text{var}}, t, s', a, v) \\ \underline{\text{vtype}}(\underline{\text{name}}[\underline{\text{expr}}], s) &:= t && \text{falls } \widehat{\text{st}}_P(\underline{\text{name}}, s) = (\underline{\text{arr}}, t, n, s', a, v) \end{aligned}$$

Falls der erste Teil der Variablen-Referenz $r = (\underline{\text{name}}'(\underline{\text{expr}}')? \underline{\text{.}})^+$ schon bearbeitet wurde und somit sein Typ $t' = \underline{\text{vtype}}(r, s)$ bekannt ist, setzt sich das Feststellen des Typs wie folgt fort:

$$\begin{aligned} \underline{\text{vtype}}(r.\underline{\text{name}}, s) &:= t && \text{falls } \widehat{\text{st}}_P(t', 0) = (\underline{\text{utype}}, \underline{\text{mb}}) \\ &&& \text{und } \underline{\text{mb}}(\underline{\text{name}}) = (\underline{\text{var}}, t, o, v) \\ \underline{\text{vtype}}(r.\underline{\text{name}}[\underline{\text{expr}}], s) &:= t && \text{falls } \widehat{\text{st}}_P(t', 0) = (\underline{\text{utype}}, \underline{\text{mb}}) \\ &&& \text{und } \underline{\text{mb}}(\underline{\text{name}}) = (\underline{\text{arr}}, t, n, o, v) \end{aligned}$$

1.2 globale Variable oder lokale Variable

Da es für globale und lokale Variablen zwei verschiedene Adressräume gibt, ist es wesentlich zu wissen, ob eine Variable global ist oder nicht. Das ist mit Hilfe der Funktion global möglich, die dies anhand des einer Variablen-Referenz varref und des aktuellen Gültigkeitsbereichs s feststellt:

$$\begin{aligned} \underline{\text{global}}(\underline{\text{name}}(\underline{\text{name}}'(\underline{\text{expr}}')? \underline{\text{.}})^*, s) &:= \begin{cases} \underline{\text{G}} & \text{falls } \widehat{\text{st}}_P(\underline{\text{name}}, s) = (\underline{\text{var}}, t, 0, a, v) \\ \underline{\text{L}} & \text{falls } \widehat{\text{st}}_P(\underline{\text{name}}, s) = (\underline{\text{var}}, t, s', a, v) \text{ mit } s' > 0 \end{cases} \\ \underline{\text{global}}(\underline{\text{name}}[\underline{\text{expr}}](\underline{\text{name}}'(\underline{\text{expr}}')? \underline{\text{.}})^*, s) &:= \begin{cases} \underline{\text{G}} & \text{falls } \widehat{\text{st}}_P(\underline{\text{name}}, s) = (\underline{\text{arr}}, t, n, 0, a, v) \\ \underline{\text{L}} & \text{falls } \widehat{\text{st}}_P(\underline{\text{name}}, s) = (\underline{\text{arr}}, t, n, s', a, v) \text{ mit } s' > 0 \end{cases} \end{aligned}$$

1.3 Wertebereiche der Basistypen

Die Wertebereiche der Basistypen (basetypes) werden durch die Anzahl Bits für den Wert und für das Vorzeichen festgelegt. Um im folgenden einfacher auf diese Bit-Anzahlen zurückgreifen zu können, wird für die Basistypen $t \in \underline{\text{basetypes}}$ die Funktion bits(t) definiert:

$$\begin{aligned} \underline{\text{bits}}(\underline{\text{bit}}) &:= 1 \\ \underline{\text{bits}}(\underline{\text{bool}}) &:= 1 \\ \underline{\text{bits}}(\underline{\text{byte}}) &:= 8 \\ \underline{\text{bits}}(\underline{\text{chan}}) &:= 16 \\ \underline{\text{bits}}(\underline{\text{pid}}) &:= 8 \\ \underline{\text{bits}}(\underline{\text{short}}) &:= -15 \\ \underline{\text{bits}}(\underline{\text{int}}) &:= -31 \\ \underline{\text{bits}}(\underline{\text{unsigned}}_c) &:= c && \text{für } 1 \leq c \leq 32 \\ \underline{\text{bits}}(\underline{\text{mtype}}) &:= 8 \end{aligned}$$

Dabei gibt der Betrag die Anzahl der Bits zurück. Ein negativer Wert gibt an, dass der Typ ein Vorzeichen speichern kann.

1.4 Startadressen der Gültigkeitsbereiche

Um die Position der Variablen im lokalen Speicher bestimmen zu können, reicht es nicht aus, ihre Adressen im jeweiligen Bereich zu kennen. Es muss zu dieser Adresse noch die Startadresse des Gültigkeitsbereichs addiert werden. Diese Startadresse liegt genau hinter dem Ende des übergeordneten Gültigkeitsbereichs.

Der Gültigkeitsbereich 0 nimmt dabei eine Sonderstellung ein, da er die globalen Variablen enthält und somit einen eigenen Adressraum besitzt.

Daher kann man mit der Funktion scst (scope start) wie folgt die Startadressen der Gültigkeitsbereiche bestimmen:

$$\begin{aligned} \underline{\text{scst}}(0) &:= 0 \\ \underline{\text{scst}}(s) &:= 0 && \text{falls } s \neq 0 \text{ und } (0, s) \in \underline{\text{psc}}_P \\ \underline{\text{scst}}(s) &:= \underline{\text{scst}}(p) + \underline{\text{size}}(p, \underline{\text{st}}_P) && \text{falls } p, s \neq 0 \text{ und } (p, s) \in \underline{\text{psc}}_P \end{aligned}$$

1.5 Speichergrößen von Bereichen und Prozessen

Die maximale Größe des von lokalen Variablen eines Prozesses vom Typ p benötigten Speichers kann mit Hilfe der Startadressen der Gültigkeitsbereiche berechnet werden.

Dazu wird zunächst die Größe scsize(s) (scope size) eines Bereichs s inklusiv aller seiner Unterbereiche definiert. Der globale Bereich 0 bildet hier eine Ausnahme und schließt keine Unterbereiche mit ein, da er in einem separaten (dem globalen) Adressraum angesiedelt ist.

$$\begin{aligned} \underline{\text{scsize}}(0) &:= \underline{\text{size}}(0, \underline{\text{st}}_P) \\ \underline{\text{scsize}}(s) &:= \underline{\text{size}}(s, \underline{\text{st}}_P) + \max(\{\underline{\text{scsize}}(c) \mid (s, c) \in \underline{\text{psc}}_P\} \cup \{0\}) && \text{falls } s \neq 0 \end{aligned}$$

Somit kann die Größe des Speichers für die globalen Variablen durch $\underline{scsize}(0)$ berechnet werden.

Weiter kann die maximale Größe der Speichers für lokale Variablen eines Prozesses vom Typ p durch \underline{ptsize} (process type size) bestimmt werden, indem man den obersten Bereich dieses Prozesstyps in der Symboltabelle nachschlägt und dann dessen Größe berechnet:

$$\underline{ptsize}(p) := \underline{scsize}(s) \quad \text{falls } \underline{st}_P(p, 0) = (\underline{proc}, s, \underline{prm}, n)$$

1.6 flaches Äquivalent eines Typs

Bei der Verwendung von FIFO-Kanälen ist es in Spin möglich, andere Datentypen als die für den Kanal deklarierten in den Kanal zu senden oder aus dem Kanal zu empfangen. Dazu werden die Datentypen der Kanals und der zu sendenden bzw. zu empfangenden Variablen flach geklopft und dann Element für Element einander zugewiesen.

Hier wird zunächst das flache Äquivalent eines beliebigen elementaren bzw. benutzerdefinierten Typs definiert. Dazu wird ein Typ t mit Hilfe der Funktion \underline{ftype} (flat type) in ein Tupel von elementaren Typen $\underline{ftype}(t) = (e_1, \dots, e_n)$ überführt.

Für elementare Typen besteht dieses Tupel nur aus dem elementaren Typ:

$$\underline{ftype}(e) := (e) \quad \text{falls } e \in \underline{basetypes}$$

Für benutzerdefinierte Typen werden die Basistypen zusammen mit den schon flachen enthaltenen Typen nacheinander in das Tupel geschrieben:

$$\underline{ftype}(t) := (e_0, \dots, e_{s-1}) \quad \text{falls } \underline{st}_P(t, 0) = (\underline{utype}, \underline{mb}) \text{ und } s = \underline{size}(t, 0, \underline{st}_P)$$

$$\left. \begin{array}{l} \text{wobei gilt: } \underline{mb}(i) = (\underline{var}, t', o, v) \\ \underline{ftype}(t') = (e'_0, \dots, e'_{r-1}) \end{array} \right\} \quad \rightsquigarrow e_{o+j} := e'_j \text{ für } 0 < j \leq r$$

$$\left. \begin{array}{l} \underline{mb}(i) = (\underline{arr}, t', n, o, v) \\ \underline{ftype}(t') = (e'_0, \dots, e'_{r-1}) \end{array} \right\} \quad \rightsquigarrow e_{o+j+kr} := e'_j \text{ für } 0 < j \leq r, 0 < k < n$$

Um den flachen Typ eines Kanals zu erhalten, muss die Definition der Funktion \underline{ftype} noch auf die in Kanal-Initialisierungen vorkommenden Konstrukte $\underline{type_name} (\underline{type_name})^*$ ausgeweitet werden.

Dazu werden einfach die flachen Äquivalente der einzelnen Typen aneinander gehangen:

$$\underline{ftype}(\underline{type_name}_1 \underline{type_name}_2 \dots \underline{type_name}_n) := (e_{1,1}, \dots, e_{1,n_1}, \dots, e_{n,1}, \dots, e_{n,n_n})$$

$$\text{falls } \underline{ftype}(\underline{type_name}_k) = (e_{k,1}, \dots, e_{k,n_k}) \text{ für } 1 \leq k \leq n$$

2 Zwischencode

Die operationelle Semantik wird über einen Zwischenocode definiert. So können im ersten Schritt, der Übersetzung des vereinfachten Promela-Quelltextes in den Zwischenocode, Konstrukte höherer Ebenen bereits auf einfachere Konstrukte zurückgeführt werden.

Z.B. können die gleichzeitigen Initialisierungen von Variablen beim Eintritt in einen Gültigkeitsbereich in ein atomares und deterministisches Code-Stück übersetzt werden, in dem die einzelnen Variablen nacheinander ihre Werte zugewiesen bekommen.

2.1 Adressen von Variablen

Für den Zugriff auf Variablen muss deren Adresse im Speicher auf dem Datenkeller berechnet werden. Die Funktion $\underline{\text{vaddr}}$ (variable address) erzeugt mit $\underline{\text{vaddr}}(v, s)$ Zwischencode zur Berechnung der Adresse der Variable v im Bereich s :

Für elementare Variablen ist dies sehr einfach:

$$\underline{\text{vaddr}}(\text{name}, s) := \text{LDC } (\underline{\text{scst}}(s') + a) \text{ falls } \widehat{\text{st}}_P(\text{name}, s) = (\underline{\text{var}}, t, s', a, v)$$

Für Feld-Zugriffe muss zusätzlich der Index berücksichtigt werden. Dazu wird mit Hilfe der weiter unten definierten Funktion $\underline{\text{expr}}$ zunächst der Wert des Index bestimmt, dieser mit der Größe eines Feldelements multipliziert und dann die Anfangsadresse des Feldes dazuaddiert:

$$\begin{aligned} \underline{\text{vaddr}}(\text{name}[\underline{\text{expr}}], s) := & \underline{\text{expr}}(\underline{\text{expr}}, s) & \text{ falls } \widehat{\text{st}}_P(\text{name}, s) = (\underline{\text{arr}}, t, n, s', a, v) \\ & \text{ICHK } n \\ & \text{LDC } \underline{\text{size}}(t, 0, \underline{\text{st}}_P) \\ & \text{MUL} \\ & \text{LDC } \underline{\text{scst}}(s') + a \\ & \text{ADD} \end{aligned}$$

Bei Zugriff auf Elemente einer Struktur wird zunächst die Anfangs-Adresse der Struktur bestimmt und dann der Offset des Elements addiert. Hier sei wieder wie oben abkürzend $r = (\text{name}'([\underline{\text{expr}}])? \underline{\text{.}})^+$ und $t' = \underline{\text{vtype}}(r, s)$.

$$\begin{aligned} \underline{\text{vaddr}}(r.\underline{\text{name}}, s) := & \underline{\text{vaddr}}(r, s) & \text{ falls } \widehat{\text{st}}_P(t', 0) = (\underline{\text{utype}}, \underline{\text{mb}}) \\ & \text{LDC } o & \text{ und } \underline{\text{mb}}(\text{name}) = (\underline{\text{var}}, t, o, v) \\ & \text{ADD} \\ \underline{\text{vaddr}}(r.\underline{\text{name}}[\underline{\text{expr}}], s) := & \underline{\text{vaddr}}(r, s) & \text{ falls } \widehat{\text{st}}_P(t', 0) = (\underline{\text{utype}}, \underline{\text{mb}}) \\ & \underline{\text{expr}}(\underline{\text{expr}}, s) & \text{ und } \underline{\text{mb}}(\text{name}) = (\underline{\text{arr}}, t, n, o, v) \\ & \text{ICHK } n \\ & \text{LDC } \underline{\text{size}}(t, 0, \underline{\text{st}}_P) \\ & \text{MUL} \\ & \text{LDC } o \\ & \text{ADD} \\ & \text{ADD} \end{aligned}$$

2.2 Ausdrücke

Die Funktion $\underline{\text{expr}}$ (expression) erzeugt mit $\underline{\text{expr}}(e, s)$ Zwischencode zur Berechnung des Ausdrucks e im Bereich s :

Konstanten als einfachste Ausdrücke lassen sich durch eine einfache Load-Instruktion auf den Datenkeller legen:

$$\underline{\text{expr}}(\text{const}, s) := \text{LDC } [\text{const}]$$

Variablen eines Basistyps werden gelesen, indem ihre Adresse auf den Datenkeller gelegt wird und danach eine indirekte Load-Instruktion ausgeführt wird:

$$\underline{\text{expr}}(\text{varref}, s) := \underline{\text{vaddr}}(\text{varref}, s) \quad \text{falls } \underline{\text{vtype}}(\text{varref}, s) \in \underline{\text{basetypes}}$$

$$\text{LDV } g \quad \text{und } g = \underline{\text{global}}(\text{varref}, s)$$

Falls $\underline{\text{vaddr}}(\text{varref}, s)$ nur aus einer einzigen LDC-Instruktion besteht, kann diese mit der LDV-Instruktion zu einer LDVA-Instruktion zusammengefasst werden.

Die speziellen Variablen `timeout`, `_pid`, `_nr_pr`, `_last` und `np_` werden mit Hilfe der LDS-Instruktion geladen:

$$\underline{\text{expr}}(\underline{\text{timeout}}, s) := \text{LDS } \underline{\text{timeout}}$$

$$\underline{\text{expr}}(\underline{_pid}, s) := \text{LDS } \underline{\text{pid}}$$

$$\underline{\text{expr}}(\underline{_nr_pr}, s) := \text{LDS } \underline{\text{nrpr}}$$

$$\underline{\text{expr}}(\underline{_last}, s) := \text{LDS } \underline{\text{last}}$$

$$\underline{\text{expr}}(\underline{\text{np_}}, s) := \text{LDS } \underline{\text{np}}$$

Dabei sind `_last` und `np_` gemäß der Promela-Dokumentation nur im `never`-Prozess erlaubt. Hier werden diese Variablen auch für normale Prozesse zugelassen, allerdings liefert `_last` in der VM für normale Prozesse immer 0 zurück. Diese Information über den letzten ausgeführten Prozess ist nur im `never`-Prozess verfügbar, da sie nicht im Zustand gespeichert wird um die Zustandszahl nicht unnötig aufzublähen, wenn mehrere Prozesse nebenläufig ausgeführt werden.

Besteht ein Ausdruck aus einem unären Operator und einem Teilausdruck, so wird einfach der Teilausdruck berechnet und dann der unäre Operator angewandt:

$$\underline{\text{expr}}(\underline{_}\text{expr}, s) := \underline{\text{expr}}(\text{expr}, s)$$

$$\text{NEG}$$

Analog wird dies auch für $\underline{\sim}$, NOT und für $\underline{\!}$, BNOT definiert.

Besteht ein Ausdruck aus einem binären Operator mit zwei Teilausdrücken, so werden die beiden Teilausdrücke berechnet und anschließend mit dem binären Operator verknüpft:

$$\underline{\text{expr}}(\text{expr}_1 \underline{+} \text{expr}_2, s) := \underline{\text{expr}}(\text{expr}_1, s)$$

$$\underline{\text{expr}}(\text{expr}_2, s)$$

$$\text{ADD}$$

Analog wird dies auch für $\underline{-}$, SUB, für $\underline{*}$, MUL, für $\underline{/}$, DIV, für $\underline{\%}$, MOD, für $\underline{\&}$, AND, für $\underline{\^}$, XOR, für $\underline{|}$, OR, für $\underline{\geq}$, GT, für $\underline{\leq}$, LT, für $\underline{\geq\equiv}$, GTE, für $\underline{\leq\equiv}$, LTE, für $\underline{\equiv}$, EQ, für $\underline{\! \equiv}$, NEQ, für $\underline{\ll}$, SHL, für $\underline{\gg}$, SHR, für $\underline{\&\&}$, BAND und für $\underline{||}$, BOR definiert.

Ein geklammerter Ausdruck wird einfach berechnet. Die Reihenfolge der Operationen ist durch die Reihenfolge der Abarbeitung auf dem Datenkeller bereits festgelegt:

$$\underline{\text{expr}}(\underline{(\text{expr})}, s) := \underline{\text{expr}}(\text{expr}, s)$$

Der Operator $\underline{(_ _ _ : _ _ _)}$ wird wie folgt zusammengesetzt:

$$\begin{aligned} \underline{\text{expr}}(\underline{(_ \text{expr}_1 _ _ \text{expr}_2 _ : _ \text{expr}_3 _)}, s) &:= \underline{\text{expr}}(\text{expr}_1, s) \\ &\quad \text{JMPZ } a_1 \\ &\quad \underline{\text{expr}}(\text{expr}_2, s) \\ &\quad \text{JMP } a_2 \\ a_1 &: \underline{\text{expr}}(\text{expr}_3, s) \\ a_2 &: \end{aligned}$$

2.2.1 Kanal-Abfragen in Ausdrücken

Anhand einer Kanal-Kennzahl kann die Anzahl der Nachrichten in diesem Kanal ermittelt werden:

$$\underline{\text{expr}}(\underline{\text{len}}(\underline{\text{varref}}), s) := \underline{\text{expr}}(\text{varref}, s) \quad \text{falls } \underline{\text{vtype}}(\text{varref}, s) = \underline{\text{chan}}$$

CHLEN

Ähnlich lassen sich die Kanal-Abfragen `empty` und `nempty` realisieren. Es ist hier nur nach der Abfrage der aktuellen Länge noch ein Vergleich mit 0 notwendig:

$$\underline{\text{expr}}(\underline{\text{empty}}(\underline{\text{varref}}), s) := \underline{\text{expr}}(\text{varref}, s) \quad \text{falls } \underline{\text{vtype}}(\text{varref}, s) = \underline{\text{chan}}$$

CHLEN
LDC 0
LTE

$$\underline{\text{expr}}(\underline{\text{nempty}}(\underline{\text{varref}}), s) := \underline{\text{expr}}(\text{varref}, s) \quad \text{falls } \underline{\text{vtype}}(\text{varref}, s) = \underline{\text{chan}}$$

CHLEN
LDC 0
GT

Für die Kanal-Abfragen `full` und `nfull` muss die freie Kapazität des Kanals abgefragt werden:

$$\underline{\text{expr}}(\underline{\text{full}}(\underline{\text{varref}}), s) := \underline{\text{expr}}(\text{varref}, s) \quad \text{falls } \underline{\text{vtype}}(\text{varref}, s) = \underline{\text{chan}}$$

CHFEE
LDC 0
LTE

$$\underline{\text{expr}}(\underline{\text{nfull}}(\underline{\text{varref}}), s) := \underline{\text{expr}}(\text{varref}, s) \quad \text{falls } \underline{\text{vtype}}(\text{varref}, s) = \underline{\text{chan}}$$

CHFEE
LDC 0
GT

Anmerkung: Diese Definition von `full` entpricht der Dokumentation von Promela im Rahmen von Spin. Dort wird erklärt, dass `full` für synchrone Kanäle immer 0 ist. Dies ist jedoch in der Spin-Implementierung nicht der Fall, dort wird immer 1 zurückgegeben.

2.2.2 Kanal-Empfangs-Abfragen in Ausdrücken

Bevor die Kanal-Abfragen zum Test eines möglichen Empfangs definiert werden können, müssen zunächst einige Hilfsfunktionen definiert werden, die die benötigten Informationen zur Verfügung stellen.

Die erste dieser Hilfsfunktionen (rasize, receive argument size) ermittelt die Größe eines Empfangs-Arguments (recvarg):

$$\begin{aligned} \underline{\text{rasize}}(\text{ varref } , s) &:= n && \text{ falls } \underline{\text{ftype}}(\underline{\text{vtype}}(\text{ varref } , s)) = (e_1, \dots, e_n) \\ \underline{\text{rasize}}(\underline{\text{eval}}(\text{ varref }) , s) &:= n && \text{ falls } \underline{\text{ftype}}(\underline{\text{vtype}}(\text{ varref } , s)) = (e_1, \dots, e_n) \\ \underline{\text{rasize}}(\underline{\text{--}}? \text{ const } , s) &:= 1 \\ \underline{\text{rasize}}(\underline{\text{--}} , s) &:= 1 \end{aligned}$$

Die Funktion recvchk (receive check) erzeugt nun für zunächst ein Empfangs-Argument Zwischencode, der prüft, ob dieses Argument den Empfang vom Kanal zulässt. Am Beginn des erzeugten Codes muss die Kanal-Kennzahl auf dem Datenkeller liegen, damit am Ende der boolesche Wert auf dem Datenkeller liegt, der angibt, ob der Empfang möglich ist:

$$\begin{aligned} \underline{\text{recvchk}}(\text{ varref } , o, s) &:= \text{POP } r_0 \\ &\quad \text{LDC } 1 \\ \underline{\text{recvchk}}(\underline{\text{--}} , o, s) &:= \text{POP } r_0 \\ &\quad \text{LDC } 1 \end{aligned}$$

<u>recvchk</u> (<u>eval</u> (varref) , o, s)	mit $n = \underline{\text{rasize}}(\text{ varref } , s)$
:= <u>vaddr</u> (varref , s)	und $g = \underline{\text{global}}(\text{ varref } , s)$
POP r_1	// r_1 : Speicher-Adresse
POP r_0	// r_0 : Kanal-Kennzahl
LDC 1	// Ergebnis im Moment noch 1
PUSH r_0	// Kanal lesen (1. Wert)
CHGETO $o + 0$	
PUSH r_1	// Speicher lesen (1. Wert)
LDV g	
EQ	// vergleichen
AND	// mit bisherigem Ergebnis UNDEN
INC r_1	// nächste Adresse
:	
PUSH r_0	// Kanal lesen (n . Wert)
CHGETO $o + n - 1$	
PUSH r_1	// Speicher lesen (n . Wert)
LDV g	
EQ	// vergleichen
AND	// mit bisherigem Ergebnis UNDEN

$$\begin{aligned} \underline{\text{recvchk}}(\underline{\text{?}} \text{ const}, o, s) &:= \text{CHGETO } o \\ &\quad \text{LDC } [\underline{\text{?}} \text{ const}] \\ &\quad \text{EQ} \end{aligned}$$

Wie man sieht, lässt eine Zielvariable immer den Empfang zu. Eine mit `eval` ausgewertete Variable oder eine Konstante fordert den entsprechenden Wert im Kanal.

Nun kann `recvchk` auf mehrere Empfangs-Argumente ausgeweitet werden:

$$\begin{aligned} &\underline{\text{recvchk}}(\text{recvarg}_1, \dots, \text{recvarg}_n, o, s) \\ &:= \text{TOP } r_0 \quad // \text{Datenkeller: } D : c \rightarrow D : c : 1 : c \\ &\quad \text{LDC } 1 \\ &\quad \text{PUSH } r_0 \\ &\quad \underline{\text{recvchk}}(\text{recvarg}_1, o_1, s) \\ &\quad \text{AND} \\ &\quad \text{POP } r_1 \quad // \text{Datenkeller: } D : c : b \rightarrow D : c : b : c \\ &\quad \text{TOP } r_0 \\ &\quad \text{PUSH } r_1 \\ &\quad \text{PUSH } r_0 \\ &\quad \vdots \\ &\quad \underline{\text{recvchk}}(\text{recvarg}_{n-1}, o_{n-1}, s) \\ &\quad \text{AND} \\ &\quad \text{POP } r_1 \quad // \text{Datenkeller: } D : c : b \rightarrow D : c : b : c \\ &\quad \text{TOP } r_0 \\ &\quad \text{PUSH } r_1 \\ &\quad \text{PUSH } r_0 \\ &\quad \underline{\text{recvchk}}(\text{recvarg}_n, o_n, s) \\ &\quad \text{AND} \\ &\quad \text{POP } r_1 \quad // \text{Datenkeller: } D : c : b \rightarrow D : b \\ &\quad \text{POP } r_0 \\ &\quad \text{PUSH } r_1 \end{aligned}$$

dabei ist $o_i := o + \sum_{j=1}^{i-1} \text{rasize}(\text{recvarg}_j, s)$

Die andere Schreibweise der Argumente wird einfach auf die schon definierte Schreibweise zurückgeführt:

$$\begin{aligned} &\underline{\text{recvchk}}(\text{recvarg}_1, \dots, \text{recvarg}_k, (\text{recvarg}_{k+1}, \dots, \text{recvarg}_n), o, s) \\ &:= \underline{\text{recvchk}}(\text{recvarg}_1, \dots, \text{recvarg}_n, o, s) \end{aligned}$$

Nun kann die FIFO-Kanal-Abfrage einfach durch Leerheits-Test des Kanals mit anschließender Prüfung der Empfangs-Argumente realisiert werden:

```

expr( varref?[recvargs] , s)
:= expr(varref, s)           falls vtype(varref, s) = chan
TOP r0                       // Kanal-Kennzahl verdoppeln
PUSH r0
CHLEN
JMPNZ a1                       // springen, falls Kanal nicht leer
POP r0
LDC 0                         // falsch zurückgeben
JMP a2
a1 : recvchk( recvargs , 0, s)   // Empfangs-Möglichkeit prüfen
a2 :

```

Für die selektive Kanal-Abfrage muss jede Nachricht im Kanal auf mögliche Empfangbarkeit geprüft werden. Es wird dann der Wert 1 zurückgeliefert, wenn mindestens eine Nachricht empfangen werden kann.

Dies wird dadurch erreicht, dass nacheinander alle im Kanal enthaltenen Nachrichten überprüft werden. Da mit CHGET immer nur auf die erste Nachricht zugegriffen werden kann, werden die Nachrichten im Kanal mit CHROT zyklisch verschoben.

```

expr( varref??[recvargs] , s)
:= expr(varref, s)           falls vtype(varref, s) = chan
TOP r0                       // Kanal-Kennzahl verdoppeln
PUSH r0
CHLEN
TOP r1                       // Anzahl Nachrichten l in r1 zwischenspeichern
JMPZ a2                       // springen, falls Kanal leer
a1 : TOP r0                   // Datenkeller: D : c → D : c : l : c : c
PUSH r1
PUSH r0
PUSH r0
recvchk( recvargs , 0, s)   // Empfangs-Möglichkeit prüfen
JMPNZ a3                       // springen, falls Nachricht empfangbar
CHROT                       // nächste Nachricht
POP r1
LOOP r1, a1                   // Kanal insgesamt l mal rotieren
a2 : POP r0                   // Kanal-Kennzahl entfernen
LDC 0                       // falsch zurückgeben
JMP a5
a3 : POP r0                   // Datenkeller: D : c : l : c → D
POP r1

```

```

POP r0
a4 : PUSH r0           // nächste Nachricht
    CHROT
    LOOP r1, a4       // Kanal insgesamt l mal rotieren
    LDC 1             // wahr zurückgeben
a5 :

```

2.2.3 Zugriff auf andere Prozesse in Ausdrücken

Der Zugriff auf andere Prozesse ist gemäß Promela-Dokumentation nur vom `never`-Prozess aus möglich, wird hier allerdings wie in Spin auch aus anderen Prozessen heraus erlaubt. Eine Modellierung über globale Variablen sollte jedoch vorgezogen werden.

Das Auslesen des Programmzählers eines anderen Prozesses kann anhand der Kennzahl mit `pc_value` erfolgen:

$$\underline{\text{expr}}(\underline{\text{pc_value}}(\text{expr}), s) := \underline{\text{expr}}(\text{expr}, s)$$

PCVAL

Sehr ähnlich kann festgestellt werden, ob sich ein Prozess an einer bestimmten Sprungmarke befindet. Dazu muss nur der Programmzähler mit der Adresse der Sprungmarke des Prozesses verglichen werden:

$$\underline{\text{expr}}(\text{proc}[\underline{\text{expr}}]@\text{label}, s) := \underline{\text{expr}}(\text{expr}, s)$$

PCVAL
LDC label_{proc}
EQ

Zum Auslesen von lokalen Variablen anderer Prozesse muss man den Prozess und die Variable in der Symboltabelle nachschlagen. Dann kann der Zugriff auf Variablen von Basistypen aus dem Rumpf des Prozesses mit Hilfe der Prozess-Kennzahl und der Variablen-Adresse erfolgen:

$$\underline{\text{expr}}(\text{proc}[\underline{\text{expr}}]:\text{var}, s) := \underline{\text{expr}}(\text{expr}, s) \quad \text{falls } \underline{\text{st}}_P(\text{proc}, 0) = (\underline{\text{proc}}, s', \underline{\text{prm}}, n')$$

$$\text{LDC } \underline{\text{scst}}(s'') + a \quad \text{und } \underline{\text{st}}_P(\text{var}, s'') = (\underline{\text{var}}, t, s'', a, v)$$

$$\text{LVAR} \quad \text{und } (s', s'') \in \underline{\text{psc}}_P, t \in \underline{\text{basetypes}}$$

Es ist in Promela auch möglich zu prüfen, ob ein anderer Prozess ausführbar ist. Zu diesem Zweck steht die VM-Instruktion `ENAB` zur Verfügung, die diese Aufgabe anhand der Prozess-Kennzahl erledigt:

$$\underline{\text{expr}}(\underline{\text{enabled}}(\text{expr}), s) := \underline{\text{expr}}(\text{expr}, s)$$

ENAB

2.3 Initialisierungen

Der Zwischencode zur Initialisierung der Variablen in einem Bereich kann mit Hilfe der Funktionen init (initialization) erzeugt werden. Die einzelnen dabei verwendeten Funktionen unterscheiden sich in der Anzahl ihrer Parameter und führen unterschiedliche Aufgaben durch.

Die Funktion init(t, a, g, v) erzeugt Zwischencode für die Initialisierung einer Variablen vom Typ t an Adresse a , wobei g angibt, ob es sich um eine globale Adresse ($g = \underline{G}$) oder um eine lokale Adresse ($g = \underline{L}$) handelt. Der Parameter v beinhaltet den initialen Wert für Variablen und Feld-Elemente eines Basistyps:

$$\begin{aligned} \underline{\text{init}}(t, a, g, v) &:= \text{LDC } v && \text{falls } t \in \underline{\text{basetypes}} \\ &\quad \text{STVA } g, a \end{aligned}$$

Der initiale Wert muss hier nicht mit TRUNC abgeschnitten werden, da in der Symboltabelle nur initiale Werte aus dem Wertebereich des entsprechenden Typs enthalten sind.

Soll bei der Initialisierung einer Kanal-Variablen ein neuer Kanal erstellt werden, so erzeugt die Funktion init($s, \underline{\text{chan}}, a, g, (l, T)$) Zwischencode, der diesen Kanal mit der maximalen Länge l und dem Eintrags-Typ T anlegt:

$$\begin{aligned} \underline{\text{init}}(s, \underline{\text{chan}}, a, g, (l, T)) &:= \text{LDC } \underline{\text{bits}}(e_1) && \text{und } \underline{\text{ftype}}(T) = (e_1, \dots, e_n) \\ &\quad \vdots \\ &\quad \text{LDC } \underline{\text{bits}}(e_n) \\ &\quad \text{CHNEW } l, n \\ &\quad \text{STVA } g, a \end{aligned}$$

Nicht initialisierte Kanal-Variablen besitzen wie normale Variablen einen Symboltabelleintrag mit dem initialen Wert 0 (ungültige Kanal-Nummer) und werden gemäß der weiter oben angegebenen Definition mit 0 initialisiert.

Falls t kein Basistyp ist, werden die einzelnen Variablen und Felder der Struktur nacheinander initialisiert. Dabei wird der initiale Wert X aus dem Funktionsaufruf ignoriert:

$$\begin{aligned} \underline{\text{init}}(t, a, g, X) &:= \forall i : \underline{\text{mb}}(i) = (\underline{\text{var}}, t', o, X') && \text{falls } \widehat{\underline{\text{st}}_P}(t, 0) = (\underline{\text{utype}}, \underline{\text{mb}}) \\ &\quad \underline{\text{init}}(t', a + o, g, X') \\ &\quad \forall i : \underline{\text{mb}}(i) = (\underline{\text{arr}}, t', n, o, X') \\ &\quad \forall k : 0 \leq k < n \\ &\quad \underline{\text{init}}(t', a + o + k \cdot s', g, X') \quad \text{wobei } s' = \underline{\text{size}}(t', 0, \underline{\text{st}}_P) \end{aligned}$$

Der Zwischencode zur Initialisierung der Variablen im Bereich s kann nun mit der einstelligen Funktion init(s) generiert werden. Dazu wird für alle Variablen und Felder dieses Bereichs die vierstellige Funktion init(t, a, g, X) aufgerufen:

$$\begin{aligned}
\text{init}(s) &:= \forall i : \text{st}_P(i, s) = (\text{var}, t, a, s, X) \\
&\quad \text{init}(t, \text{scst}(s) + a, g, X) \quad \text{mit } g = \underline{G} \text{ falls } s = 0 \text{ bzw. } g = \underline{L} \text{ falls } s > 0 \\
&\quad \forall i : \text{st}_P(i, s) = (\text{arr}, t, n, a, s, X) \\
&\quad \forall k : 0 \leq k < n \\
&\quad \text{init}(t, \text{scst}(s) + a + k \cdot s', g, X) \quad \text{wobei } s' = \text{size}(t, 0, \text{st}_P)
\end{aligned}$$

2.4 einfache Anweisungen

Die Funktion sstmt (simple statement) erzeugt mit sstmt(S, s) Zwischencode zur Ausführung der Anweisung S im Bereich s .

Bei einer Wertzuweisung muss nur der Wert des Ausdrucks bestimmt und dieser dann der entsprechenden Variablen zugewiesen werden:

$$\begin{aligned}
\text{sstmt}(\text{varref} \underline{=} \text{expr}, s) &:= \text{expr}(\text{expr}, s) \quad \text{falls } \text{vtype}(\text{varref}, s) = t \\
&\quad \text{TRUNC } \underline{\text{bits}}(t) \quad \text{und } t \in \underline{\text{basetypes}} \\
&\quad \underline{\text{vaddr}}(\text{varref}, s) \quad \text{und } g = \underline{\text{global}}(\text{varref}, s) \\
&\quad \text{STV } g
\end{aligned}$$

Falls vaddr(varref, s) nur aus einer einzigen LDC-Instruktion besteht, kann diese mit der STV-Instruktion zu einer STVA-Instruktion zusammengefasst werden.

Schreibzugriffe auf die Variable $_$ werden ignoriert:

$$\text{sstmt}(_ \underline{=} \text{expr}, s) := \epsilon$$

Bei der Incrementierung einer Variablen wird ihr Wert gelesen, erhöht und direkt wieder abgespeichert:

$$\begin{aligned}
\text{sstmt}(\text{varref} \underline{++}, s) &:= \underline{\text{vaddr}}(\text{varref}, s) \quad \text{falls } \text{vtype}(\text{varref}, s) = t \\
&\quad \text{TOP } r_0 \quad \text{und } t \in \underline{\text{bastypes}} \\
&\quad \text{LDV } g \quad \text{und } g = \underline{\text{global}}(\text{varref}, s) \\
&\quad \text{LDC } 1 \\
&\quad \text{ADD} \\
&\quad \text{TRUNC } \underline{\text{bits}}(t) \\
&\quad \text{PUSH } r_0 \\
&\quad \text{STV } g
\end{aligned}$$

Analog dazu wird dies für die Decrementierung ($\text{varref} \underline{--}$) mit SUB statt ADD definiert.

Ausdrücke als Anweisungen brechen den aktuellen Programmpfad ab, wenn sie zu 0 ausgewertet werden:

$$\underline{\text{sstmnt}}(\text{expr}, s) := \underline{\text{expr}}(\text{expr}, s) \\ \text{NEXZ}$$

Bei Behauptungs-Anweisungen wird ebenfalls der Ausdruck ausgewertet, aber bei Resultat 0 tritt ein Laufzeitfehler auf:

$$\underline{\text{sstmnt}}(\text{assert expr}, s) := \underline{\text{expr}}(\text{expr}, s) \\ \text{BCHK}$$

2.4.1 Ausgabe

Bei der Ausgabe mit `printf` müssen die Argumente einzeln auf dem Datenkeller ausgewertet und dann mit Hilfe der PRINTV-Instruktion ausgegeben werden:

$$\underline{\text{sstmnt}}(\underline{\text{printf}}(„\%f_1 \dots \%f_n“ _1 \text{expr}_1 \dots _n \text{expr}_n _), s) \\ := \underline{\text{expr}}(\text{expr}_1, s) \\ \text{PRINTV „f}_1“ \\ \vdots \\ \underline{\text{expr}}(\text{expr}_n, s) \\ \text{PRINTV „f}_n“$$

Einfacher Text kann mit der PRINTS-Instruktion (auch zwischen den einzelnen Werten) ausgegeben werden, wenn die benötigten Zeichenketten parallel zum Bytecode in einer Tabelle abgelegt werden. Auf diese Definition wird hier verzichtet.

Die `printm`-Anweisung gibt nur ein Argument als `mtype`-Konstante aus:

$$\underline{\text{sstmnt}}(\underline{\text{printm}}(\text{expr}), s) \\ := \underline{\text{expr}}(\text{expr}, s) \\ \text{PRINTV „e“}$$

Die VM kennt keine symbolischen `mtype`-Konstanten, kann daher das Format „e“ nicht vollständig unterstützen. Falls der Benutzer die symbolischen Namen der `mtype`-Konstanten aber an der Position ihres Werts in einer Zeichenketten-Tabelle abgelegt hat, so kann bei PRINTV „e“ dort die Zeichenkette nachgeschlagen und ausgegeben werden.

2.5 Empfangen und Senden

2.5.1 Empfangen

Die Funktion `recvars` (receive arguments) erzeugt Zwischencode, der zunächst ein Empfangs-Argument aus der aktuellen Kanal-Nachricht in die entsprechende Variable ablegt. Am Beginn des erzeugten Codes muss die Kanal-Kennzahl auf dem Datenkeller liegen:

<u>recvars</u> (varref , o, s)	mit <u>f_{type}</u> (v _{type} (varref , s)) = (e ₁ , ..., e _n)
:= <u>vaddr</u> (varref , s)	und g = <u>global</u> (varref , s)
POP r ₁	// r ₁ : Speicher-Adresse
POP r ₀	// r ₀ : Kanal-Kennzahl
PUSH r ₀	// Kanal lesen (1. Wert)
CHGETO o + 0	
TRUNC <u>bits</u> (e ₁)	// Speicher beschreiben (1. Wert)
PUSH r ₁	
STV g	
INC r ₁	// nächste Adresse
⋮	
PUSH r ₀	// Kanal lesen (n. Wert)
CHGETO o + n - 1	
TRUNC <u>bits</u> (e _n)	// Speicher beschreiben (n. Wert)
PUSH r ₁	
STV g	

Für mit `eval` ausgewertete Empfangsargumente und Konstanten braucht nichts getan zu werden, außer die Kanal-Kennzahl wieder vom Datenkeller zu entfernen. Dies gilt auch für den Empfang in die Variable `_`:

<u>recvars</u> (<u>eval</u> (varref) , o, s)	:= POP r ₀
<u>recvars</u> (<u>_</u> ? const , o, s)	:= POP r ₀
<u>recvars</u> (<u>_</u> , o, s)	:= POP r ₀

Nun kann recvars auf mehrere Empfangs-Argumente ausgeweitet werden:

<u>recvars</u> (recvarg ₁ , ..., recvarg _n , o, s)	
:= TOP r ₀	// Datenkeller: D : c → D : c : c
PUSH r ₀	
<u>recvars</u> (recvarg ₁ , o ₁ , s)	
⋮	
TOP r ₀	// Datenkeller: D : c → D : c : c
PUSH r ₀	
<u>recvars</u> (recvarg _{n-1} , o _{n-1} , s)	
<u>recvars</u> (recvarg _n , o _n , s)	

$$\text{dabei ist } o_i := o + \sum_{j=1}^{i-1} \text{rasize}(\text{recvarg}_j, s)$$

Die andere Schreibweise der Argumente wird einfach auf die schon definierte Schreibweise zurückgeführt:

$$\begin{aligned} & \underline{\text{recvars}}(\text{recvarg}_1, \dots, \text{recvarg}_k, \underline{\text{recvarg}}_{k+1}, \dots, \text{recvarg}_n, o, s) \\ & := \underline{\text{recvars}}(\text{recvarg}_1, \dots, \text{recvarg}_n, o, s) \end{aligned}$$

Der normale FIFO-Empfang aus dem Kanal kann durch Leerheits-Test des Kanals, Prüfung der Empfangs-Argumente, Empfang der Argumente und anschließendes Löschen der Nachricht realisiert werden:

```

sstmnt( varref?recvars , s)
:= expr(varref, s)           falls vtype(varref, s) = chan
TOP r0                       // Kanal-Kennzahl merken
CHLEN
NEXZ                           // nicht ausführbar, falls Kanal leer
PUSH r0
PUSH r0
PUSH r0
recvchk( recvars , 0, s)      // Empfangs-Möglichkeit prüfen
NEXZ                           // nicht ausführbar, falls nicht empfangbar
recvars( recvars , 0, s)      // empfangen
CHDEL                           // Nachricht aus Kanal löschen

```

Beim FIFO-Empfangen aus dem Kanal ohne Löschen der Nachricht verzichtet man einfach auf das Löschen:

```

sstmnt( varref?<recvars> , s)
:= expr(varref, s)           falls vtype(varref, s) = chan
TOP r0                       // Kanal-Kennzahl merken
CHLEN
NEXZ                           // nicht ausführbar, falls Kanal leer
PUSH r0
PUSH r0
recvchk( recvars , 0, s)      // Empfangs-Möglichkeit prüfen
NEXZ                           // nicht ausführbar, falls nicht empfangbar
recvars( recvars , 0, s)      // empfangen

```

Die Erweiterung auf den selektiven Empfang erfolgt durch sukzessives Testen der Nachrichten im Kanal, bis eine empfangbare Nachricht gefunden wird:

```

sstmnt( varref??recvars , s)
:= expr(varref, s)           falls vtype(varref, s) = chan
    TOP  $r_0$                    // Kanal-Kennzahl verdoppeln
    PUSH  $r_0$ 
    CHLEN
    TOP  $r_1$                    // Anzahl Nachrichten  $l$  in  $r_1$  zwischenspeichern
    NEXZ                       // nicht ausführbar, falls Kanal leer
 $a_1$  : TOP  $r_0$                  // Datenkeller:  $D : c \rightarrow D : c : l : c : c$ 
    PUSH  $r_1$ 
    PUSH  $r_0$ 
    PUSH  $r_0$ 
    recvchk( recvars , 0, s)   // Empfangs-Möglichkeit prüfen
    JMPNZ  $a_2$                  // springen, falls Nachricht empfangbar
    CHROT                       // nächste Nachricht
    POP  $r_1$ 
    LOOP  $r_1, a_1$              // Kanal insgesamt  $l$  mal rotieren
    NEX                          // keine Nachricht empfangbar, nicht ausführbar
 $a_2$  : TOP  $r_0$                  // Datenkeller:  $D : c : l : c \rightarrow D : c : l : c : c$ 
    PUSH  $r_0$ 
    recvars( recvars , 0, s)   // empfangen
    CHDEL                       // Nachricht aus Kanal löschen
    POP  $r_1$ 
    DEC  $r_1$                    // Anzahl der Nachrichten  $l$  eins verkleinern
    PUSH  $r_1$ 
    JMPZ  $a_4$ 
    POP  $r_0$ 
 $a_3$  : PUSH  $r_0$                  // nächste Nachricht
    CHROT
    LOOP  $r_1, a_3$              // Kanal insgesamt  $l$  mal rotieren
 $a_4$  :

```

Der selektive Empfang ohne Löschen erfolgt ebenso, allerdings ohne Löschen der Nachricht:

```

sstmnt( varref??<recvars> , s)
:= expr(varref, s)           falls vtype(varref, s) = chan
    TOP  $r_0$                    // Kanal-Kennzahl verdoppeln
    PUSH  $r_0$ 
    CHLEN
    TOP  $r_1$                    // Anzahl Nachrichten  $l$  in  $r_1$  zwischenspeichern
    NEXZ                       // nicht ausführbar, falls Kanal leer

```

```

a1 : TOP r0 // Datenkeller: D : c → D : c : l : c : c
      PUSH r1
      PUSH r0
      PUSH r0
      rcvchk(rcvargs, 0, s) // Empfangs-Möglichkeit prüfen
      JMPNZ a2 // springen, falls Nachricht empfangbar
      CHROT // nächste Nachricht
      POP r1
      LOOP r1, a1 // Kanal insgesamt l mal rotieren
      NEX // keine Nachricht empfangbar, nicht ausführbar
a2 : rcvargs(rcvargs, 0, s) // empfangen (Datenkeller ist D : c : l : c)
      POP r1
      POP r0
a3 : PUSH r0 // nächste Nachricht
      CHROT
      LOOP r1, a3 // Kanal insgesamt l mal rotieren

```

2.5.2 Senden

Auch hier wird wieder eine Hilfsfunktion zur Ermittlung der Größe der Argumente benötigt. Diese Hilfsfunktionen (sasize, send argument size) ermittelt die Größe eines Sende-Arguments (sendarg).

Dabei steht hier expr für einen Ausdruck, der keine einfache Variablen-Referenz (varref) ist:

$$\begin{aligned} \text{sasize}(\text{varref}, s) &:= n && \text{falls } \text{ftype}(\text{vtype}(\text{varref}, s)) = (e_1, \dots, e_n) \\ \text{sasize}(\text{expr}, s) &:= 1 \end{aligned}$$

Die Funktion sendargs (send arguments) erzeugt Zwischencode, der zunächst ein Sende-Argument aus der entsprechenden Variable in aktuelle Kanal-Nachricht einträgt. Am Beginn des erzeugten Codes muss die Kanal-Kennzahl auf dem Datenkeller liegen:

```

sendargs(varref, o, s)           mit n = sasize(varref, s)
:= vaddr(varref, s)             und g = global(varref, s)
  POP r1                          // r1: Speicher-Adresse
  POP r0                          // r0: Kanal-Kennzahl
  PUSH r0                          // Kanal-Kennzahl auf Datenkeller
  PUSH r1                          // Speicher lesen (1. Wert)
  LDV g
  CHSETO o + 0                    // Kanal beschreiben (1. Wert)
  INC r1                          // nächste Adresse
  ⋮
  PUSH r0                          // Kanal-Kennzahl auf Datenkeller

```

```

PUSH  $r_1$                 // Speicher lesen ( $n$ . Wert)
LDV  $g$ 
CHSETO  $o + n - 1$         // Kanal beschreiben ( $n$ . Wert)

```

```

sendargs( expr ,  $o$  ,  $s$  )
:= expr( expr ,  $s$  )      // Ausdruck auswerten
   CHSETO  $o$                 // Kanal beschreiben

```

Nun kann sendargs auf mehrere Sende-Argumente ausgeweitet werden:

```

sendargs( sendarg1 . . . sendarg $n$  ,  $o$  ,  $s$  )
:= TOP  $r_0$                 // Datenkeller:  $D : c \rightarrow D : c : c$ 
   PUSH  $r_0$ 
   sendargs( sendarg1 ,  $o_1$  ,  $s$  )
   :
   TOP  $r_0$                 // Datenkeller:  $D : c \rightarrow D : c : c$ 
   PUSH  $r_0$ 
   sendargs( sendarg $n-1$  ,  $o_{n-1}$  ,  $s$  )
   sendargs( sendarg $n$  ,  $o_n$  ,  $s$  )

```

$$\text{dabei ist } o_i := o + \sum_{j=1}^{i-1} \text{sasize}(\text{sendarg}_j, s)$$

Die andere Schreibweise der Argumente wird einfach auf die schon definierte Schreibweise zurückgeführt:

```

sendargs( sendarg1 . . . sendarg $k$  ( sendarg $k+1$  . . . sendarg $n$  ) ,  $o$  ,  $s$  )
:= sendargs( sendarg1 . . . sendarg $n$  ,  $o$  ,  $s$  )

```

Das normale FIFO-Senden in einen Kanal kann durch einen Platz-Test des Kanals, das Anlegen einer neuen Nachricht und anschließendes Eintragen der Argumente realisiert werden:

```

sstmt( varref!sendargs , s)
:= expr(varref, s)           falls vtype(varref, s) = chan
TOP  $r_0$                        // Kanal-Kennzahl merken
CHFREE
NEXZ                           // nicht ausführbar, falls Kanal voll
PUSH  $r_0$ 
CHADD                           // neue Nachricht anlegen
PUSH  $r_0$ 
sendargs( sendargs , 0, s)     // senden

```

Beim sortierten Senden muss die neue Nachricht am Ende noch in den Kanal einsortiert werden:

```

sstmt( varref!!sendargs , s)
:= expr(varref, s)           falls vtype(varref, s) = chan
TOP  $r_0$                        // Kanal-Kennzahl merken
CHFREE
NEXZ                           // nicht ausführbar, falls Kanal voll
PUSH  $r_0$ 
CHADD                           // neue Nachricht anlegen
PUSH  $r_0$ 
PUSH  $r_0$ 
sendargs( sendargs , 0, s)     // senden
CHSORT                           // neue Nachricht einsortieren

```

2.6 Anweisungen

Die Funktion stmt (statement) erzeugt mit stmt(S, s, M, B_1, B, b, d) Zwischencode zur Berechnung der Anweisung S im Bereich s .

Der in den STEP-Instruktionen zwischen einzelnen Anweisungen zu verwendende Modus wird durch den Parameter M (mode) angegeben.

Der Zwischencode B_1 wird dabei vor der ersten Anweisung eingefügt. Falls weitere Anweisungen in der Übersetzung auftreten, wird vor deren Beginn der Zwischencode B eingefügt. Benötigt wird dieser einzufügende Code für `unless`-Blöcke, um vor jeder Anweisung zur Alternative zu verzweigen.

Für die Realisierung einer `break`-Anweisung muss die Ziel-Adresse des auszuführenden Sprungs bekannt sein. Diese Adresse wird in b übergeben und von normalen Anweisungen ignoriert. Enthält ein Konstrukt weitere Anweisungen, so wird diese Adresse normalerweise unverändert übergeben.

Das Flag d gibt im Falle $d = \underline{D}$ an, dass die Übersetzung innerhalb eines deterministischen Blocks stattfindet. Normalerweise ist $d = \underline{n}$.

Für die einfachen Anweisungen kann einfach der durch sstmt generierte Zwischencode benutzt werden:

$$\underline{\text{stmnt}}(S, s, M, B_1, B, b, d) := B_1 \quad \text{falls } \underline{\text{sstmnt}}(S, s) \text{ definiert ist}$$

$$\underline{\text{sstmnt}}(S, s)$$

Eine `break`-Anweisung muss nur einen Sprung zur angegebenen Ziel-Adresse ausführen. Falls diese `break`-Anweisung jedoch außerhalb einer `do`-Schleife steht, ist die Adresse undefiniert ($b < 0$), was zu einem Fehler während der Übersetzung führt:

$$\underline{\text{stmnt}}(\text{break}, s, M, B_1, B, b, d) := B_1 \quad \text{falls } b \geq 0$$

$$\text{JMP } b$$

Eine `else`-Anweisung ist einfach eine leere Anweisung, da die besondere Bedeutung bei der Übersetzung von `if` und `do` schon berücksichtigt wird:

$$\underline{\text{stmnt}}(\text{else}, s, M, B_1, B, b, d) := B_1$$

`else`-Anweisungen außerhalb von Optionen und im inneren von Anweisungs-Sequenzen sollten detektiert werden und Warnungen bei der Übersetzung erzeugen, da sie an solchen Positionen keinerlei Funktion erfüllen.

Im folgenden werden strukturierte Anweisungen betrachtet, die untergeordnete Gültigkeitsbereiche enthalten. Es wird dabei vorausgesetzt, dass `nsc(...)` die Nummer des neuen Gültigkeitsbereichs bezeichnet, der an der betrachteten Stelle beginnt.

Das Argument für die Funktion `nsc` wird hier der Vereinfachung halber nicht angegeben. Es wäre möglich, parallel zur Berechnung des Zwischencodes die Berechnung der Gültigkeitsbereiche wie mit `scup` durchzuführen, aber aus Gründen der Übersichtlichkeit wird hier darauf verzichtet.

Im Prinzip entspricht die Ausführung einer mit `{` und `}` geklammerten Sequenz einfach der Ausführung der einzelnen Anweisungen. Da hier jedoch ein neuer Gültigkeitsbereich beginnt, müssen vor der Ausführung der Anweisungen die Variablen dieses Bereichs initialisiert werden:

$$\underline{\text{stmnt}}(\{ \text{sequence} \}, s, M, B_1, B, b, d)$$

$$:= B_1$$

$$\underline{\text{init}}(s') \quad \text{wobei } s' = \underline{\text{nsc}}(\dots)$$

$$\underline{\text{sequence}}(\text{sequence}, s', M, B, b, d)$$

Sequenzen im atomaren oder deterministischen Modus werden ebenso in Zwischencode umgewandelt, allerdings muss hier der Modus und das Flag für die deterministische Übersetzung entsprechend angepasst werden.

$$\begin{aligned}
& \text{stmnt}(\underline{\text{atomic}} \{ \text{sequence} \}, s, M, B_1, B, b, \underline{n}) \\
& \quad := B_1 \\
& \quad \quad \underline{\text{init}}(s') \qquad \qquad \qquad \text{wobei } s' = \underline{\text{nsc}}(\dots) \\
& \quad \quad \underline{\text{sequence}}(\text{sequence}, s', \underline{A}, B, b, \underline{n}) \\
& \text{stmnt}(\underline{\text{atomic}} \{ \text{sequence} \}, s, M, B_1, B, b, \underline{D}) \\
& \quad := B_1 \\
& \quad \quad \underline{\text{init}}(s') \qquad \qquad \qquad \text{wobei } s' = \underline{\text{nsc}}(\dots) \\
& \quad \quad \underline{\text{sequence}}(\text{sequence}, s', \underline{I}, B, b, \underline{D}) \\
& \text{stmnt}(\underline{\text{d_step}} \{ \text{sequence} \}, s, M, B_1, B, b, d) \\
& \quad := B_1 \\
& \quad \quad \underline{\text{init}}(s') \qquad \qquad \qquad \text{wobei } s' = \underline{\text{nsc}}(\dots) \\
& \quad \quad \underline{\text{sequence}}(\text{sequence}, s', \underline{I}, \epsilon, b, \underline{D})
\end{aligned}$$

Weiter werden die aktuellen `unless`-Anweisungen im deterministischen Block außer Kraft gesetzt. Dies geschieht dadurch, dass der vor dem Beginn jeder Anweisung einzufügende Code im `d_step`-Block zu ϵ gesetzt wird.

Am Beginn einer `if`-Anweisung ohne `else` wird zunächst der einzufügende Code B_1 ausgeführt und dann nichtdeterministisch in die einzelnen Optionen gesprungen. Dort wird dann der neue Gültigkeitsbereich initialisiert und die darin enthaltenen Anweisungen werden ausgeführt:

$$\begin{aligned}
& \text{stmnt}(\underline{\text{if}} :: \text{sequence}_1 \dots :: \text{sequence}_n \underline{\text{fi}}, s, M, B_1, B, \underline{n}) \\
& \quad := B_1 \\
& \quad \quad \text{NDET } a_n \\
& \quad \quad \vdots \\
& \quad \quad \text{NDET } a_2 \\
& \quad a_1 : \underline{\text{init}}(s_1) \\
& \quad \quad \underline{\text{sequence}}(\text{sequence}_1, s_1, M, B, b, \underline{n}) \\
& \quad \quad \text{JMP } a \\
& \quad \quad \vdots \\
& \quad a_n : \underline{\text{init}}(s_n) \\
& \quad \quad \underline{\text{sequence}}(\text{sequence}_n, s_n, M, B, b, \underline{n}) \\
& \quad \quad \text{JMP } a \\
& \quad a : \\
& \quad \text{wobei } s_k = \underline{\text{nsc}}(\dots) \text{ für alle } k \in \{1, \dots, n\}
\end{aligned}$$

Falls die `if`-Anweisung eine Option enthält, die mit `else` beginnt, muss diese Option nur dann ausgeführt werden, wenn keine andere Option ausführbar ist:

$$\begin{array}{l}
\text{stmnt}(\text{if} :: \text{sequence}_1 \dots :: \text{sequence}_{n-1} :: \underline{\text{else}} \rightarrow \text{sequence}_n \underline{\text{fi}}, s, M, B_1, B, \underline{n}) \\
:= B_1 \\
\quad \text{ELSE } a_n \\
\quad \text{NDET } a_{n-1} \\
\quad \vdots \\
\quad \text{NDET } a_2 \\
a_1 : \underline{\text{init}}(s_1) \\
\quad \underline{\text{sequence}}(\text{sequence}_1, s_1, M, B, b, \underline{n}) \\
\quad \text{JMP } a \\
\quad \vdots \\
a_n : \underline{\text{init}}(s_n) \\
\quad \underline{\text{sequence}}(\underline{\text{else}} \rightarrow \text{sequence}_n, s_n, M, B, b, \underline{n}) \\
\quad \text{JMP } a \\
a : \\
\text{wobei } s_k = \underline{\text{nsc}}(\dots) \text{ f\"ur alle } k \in \{1, \dots, n\}
\end{array}$$

Innerhalb von `d_step`-Bl\"ocken erfolgt die \"Ubersetzung von `if`-Anweisungen anders. Es wird versucht die einzelnen Optionen nacheinander auszuf\"uhren, wobei nach der ersten ausf\"uhrbaren Option keine weitere Option mehr betrachtet wird:

$$\begin{array}{l}
\text{stmnt}(\text{if} :: \text{sequence}_1 \dots :: \text{sequence}_{n-1} :: (\underline{\text{else}} \rightarrow)? \text{sequence}_n \underline{\text{fi}}, s, M, B_1, B, \underline{D}) \\
:= B_1 \\
\quad \text{ELSE } a_n \\
\quad \vdots \\
\quad \text{ELSE } a_2 \\
a_1 : \underline{\text{init}}(s_1) \\
\quad \underline{\text{sequence}}(\text{sequence}_1, s_1, M, B, b, \underline{D}) \\
\quad \text{JMP } a \\
\quad \vdots \\
a_n : \underline{\text{init}}(s_n) \\
\quad \underline{\text{sequence}}((\underline{\text{else}} \rightarrow)? \text{sequence}_n, s_n, M, B, b, \underline{D}) \\
\quad \text{JMP } a \\
a : \\
\text{wobei } s_k = \underline{\text{nsc}}(\dots) \text{ f\"ur alle } k \in \{1, \dots, n\}
\end{array}$$

Falls die mit `else` beginnende Option nicht die letzte Option ist, erfolgt zun\"achst eine Verschiebung dieser Option ans Ende bevor die \"Ubersetzung durchgef\"uhrt wird. Das Auftreten von mehreren mit `else` beginnenden Optionen ist nicht zul\"assig und f\"uhrt zu einem Fehler bei der \"Ubersetzung.

F\"ur eine `do`-Anweisung wird die \"Ubersetzung der `if`-Anweisung verwendet. Hier wird jedoch am Ende ein Schritt und Sprungbefehl zum Anfang eingef\"ugt. Zus\"atzlich wird die

Ziel-Adresse für `break`-Anweisung für die Anweisungen in den Optionen auf das Ende der `do`-Anweisung gesetzt.

$$\begin{aligned} & \underline{\text{stmnt}}(\underline{\text{do}} :: \text{sequence}_1 \dots :: \text{sequence}_n \underline{\text{od}}, s, M, B_1, B, b, d) \\ & := B_1 \\ & a : \underline{\text{stmnt}}(\underline{\text{if}} :: \text{sequence}_1 \dots :: \text{sequence}_n \underline{\text{fi}}, s, M, \epsilon, B, b', d) \\ & \quad \text{STEP } \underline{M} \\ & \quad B \\ & \quad \text{JMP } a \\ & b' : \end{aligned}$$

Steht vor einer Anweisung eine Sprungmarke, so wird die Anweisung wie ohne Sprungmarke übersetzt und die Sprungmarke wird vor die erste Instruktion der Anweisung platziert. Diese Sprungmarke gilt dabei für den gesamten Prozess und ist nicht wie die bisher benutzten Sprungmarken auf die Anweisung beschränkt.

$$\begin{aligned} & \underline{\text{stmnt}}(\text{ name } : \text{stmnt}, s, M, B_1, B, b, d) \\ & := \text{ name } : \underline{\text{stmnt}}(\text{stmnt}, s, M, B_1, B, b, d) \end{aligned}$$

Beginnt der Name der Sprungmarke allerdings mit `progress` bzw. mit `accept`, so muss die Adresse als Fortschritts-Zustand bzw. als akzeptierender Zustand gekennzeichnet werden (`progress` \in `Flags[name]` bzw. `accept` \in `Flags[name]`).

Im Prinzip kann eine `goto`-Anweisung zu einer Sprungmarke in einem Prozess durch eine simple `JMP`-Instruktion realisiert werden. Wenn aber in neue Gültigkeitsbereiche gesprungen wird, müssen deren lokale Variablen vorher noch initialisiert werden:

$$\begin{aligned} & \underline{\text{stmnt}}(\underline{\text{goto}} \text{ name}, s, M, B_1, B, b, d) \\ & := a_1 : B_1 \\ & \quad \underline{\text{init}}(z_2) \\ & \quad \vdots \\ & \quad \underline{\text{init}}(z_n) \\ & \quad \text{JMP name} \end{aligned}$$

Hier sei z der Gültigkeitsbereich in dem die Ziel-Sprungmarke `name` definiert ist. Weiter seien z_1, \dots, z_n und s_1, \dots, s_k die kürzesten Folgen von Gültigkeitsbereichen mit $(z_i, z_{i+1}) \in \underline{\text{psc}}_P$ und $(s_i, s_{i+1}) \in \underline{\text{psc}}_P$ (stufenweiser Verschachtelung) mit $z_n = z$ und $s_k = s$ (Ende im Bereich der Sprungmarke bzw. des Spungbefehls) und mit $z_1 = s_1$ (gemeinsamem übergeordneten Bereich).

Damit wird mit diesem Sprung in die neuen Gültigkeitsbereiche z_2, \dots, z_n eingetreten und deshalb müssen die lokalen Variablen dieser Bereiche vor der `JMP`-Instruktion initialisiert werden.

Beginnt der Name der Ziel-Sprungmarke mit `progress` bzw. mit `accept`, so muss die Adresse a_1 in der `goto`-Anweisung als Fortschritts-Zustand bzw. als akzeptierender Zustand gekennzeichnet werden (`progress` \in `Flags[a_1]` bzw. `accept` \in `Flags[a_1]`), damit auch im Zustand vor dem Sprung zu einer solchen Marke das entsprechende Flag aktiv ist.

2.7 Schritte

Schritte werden mit $\underline{\text{step}}(S, s, M, B_1, B, b, d)$ in Zwischencode übersetzt. Die Parameter s, M, B_1, B, b, d haben dabei die gleiche Bedeutung wie bei $\underline{\text{stmnt}}$.

Falls ein Schritt aus einer einzigen Anweisung besteht, wird einfach der Zwischencode für diese Anweisung benutzt:

$$\underline{\text{step}}(\text{stmnt}, s, M, B_1, B, b, d) := \underline{\text{stmnt}}(\text{stmnt}, s, M, B_1, B, b, d)$$

In einer `unless`-Anweisung muss vor jeder Anweisung im vorderen Teil versucht werden, den hinteren Teil auszuführen. Nur wenn dies nicht möglich ist, kann die Ausführung im vorderen Teil fortgesetzt werden.

Um ein `unless`-Konstrukt in Zwischencode zu übersetzen, wird vor jeder Anweisung im vorderen Teil mittels `UNLESS` zunächst versucht, die erste Anweisung des hinteren Teils auszuführen.

$$\begin{aligned} \underline{\text{step}}(\text{stmnt}_1 \text{ \code{unless} } \text{stmnt}_2, s, M, B_1, B, b, d) \\ := B_1 \\ \text{UNLESS } a_1 \\ \underline{\text{stmnt}}(\text{stmnt}_1, s, M, \epsilon, B; \text{UNLESS } a_1, b, d) \\ \text{JMP } a_2 \\ a_1 : \underline{\text{stmnt}}(\text{stmnt}_2, s, M, \epsilon, B, b, d) \\ a_2 : \end{aligned}$$

So wird erreicht, dass unabhängig von der Art der in `unless` verschachtelten Anweisungen (z.B. weitere `unless`-Konstrukte oder komplizierte evtl. nicht ausführbare Anweisungen) immer sofort der hintere Teil ausgeführt wird, sobald er ausführbar wird.

Deklarationen sind bereits durch die statische Semantik erfasst und können ignoriert werden. Es erfolgt eine Kennzeichnung (\perp), dass es sich nicht um einen Schritt handelt.

$$\underline{\text{step}}(\text{onedekl} \mid \text{xr} \mid \text{xs}, s, M, B_1, B, b, d) := \perp$$

2.8 Sequenzen von Schritten

Der Zwischencode für eine Sequenz von Schritten kann einfach durch Zusammenfügen des Zwischencodes der einzelnen Schritte generiert werden. Dabei sind die Schritte auszulassen, deren Übersetzung \perp ist.

Die Funktion $\underline{\text{sequence}}(S, s, M, B, b, d)$ erzeugt den Zwischencode für eine Sequenz S im Bereich s . In den Schritten zwischen den Anweisungen wird der Modus M verwendet. Die Instruktionsfolge B wird zu Beginn jeder Anweisung innerhalb der Sequenz eingefügt, jedoch nicht vor der ersten Anweisung, da dies schon von der aufrufenden Funktion übernommen wurde. Die `break`-Adresse b und das Flag d für die deterministische Übersetzung werden einfach an alle Anweisungen weitergereicht.

$$\begin{aligned}
& \underline{\text{sequence}}(\text{step}_1; \dots; \text{step}_n, s, M, B, b, d) \\
& \quad := \underline{\text{step}}(\text{step}_1, s, M, \epsilon, B, b, d) \\
& \quad \quad \text{STEP } \underline{M} \\
& \quad \quad \underline{\text{step}}(\text{step}_2, s, M, B, B, b, d) \\
& \quad \quad \vdots \\
& \quad \quad \text{STEP } \underline{M} \\
& \quad \quad \underline{\text{step}}(\text{step}_n, s, M, B, B, b, d) \\
& \quad \text{falls } \underline{\text{step}}(\text{step}_i, s, M, B, B, b, d) \neq \perp \text{ für alle } 1 \leq i \leq n \\
& \underline{\text{sequence}}(\text{sequence}_1; \text{step}_2; \text{sequence}_3, s, M, B, b, d) \\
& \quad := \underline{\text{sequence}}(\text{sequence}_1; \text{sequence}_3, s, M, B, b, d) \\
& \quad \text{falls } \underline{\text{step}}(\text{step}_2, s, M, B, B, b, d) = \perp
\end{aligned}$$

2.9 Prozesse

2.9.1 Starten neuer Prozesse

Die Funktion `runarg` (run argument) erzeugt Zwischencode, der einen Parameter aus der entsprechenden Variable oder einen mit Hilfe eines Ausdrucks berechneten Parameter auf den Datenkeller legt. Der Typ des Parameters wird mit t und der aktuelle Bereich mit s angegeben.

$$\begin{aligned}
& \underline{\text{runarg}}(\text{expr}, t, s) && \text{falls } t \in \underline{\text{basetypes}} \\
& \quad := \text{LDC } \underline{\text{bits}}(t) \\
& \quad \quad \underline{\text{expr}}(\text{expr}, s)
\end{aligned}$$

$$\begin{aligned}
& \underline{\text{runarg}}(\text{varref}, t, s) && \text{falls } \underline{\text{ftype}}(t, s) = (e_1, \dots, e_n) \\
& \quad := \underline{\text{vaddr}}(\text{varref}, s) && \text{und } t = \underline{\text{vtype}}(\text{varref}, s) \\
& \quad \text{POP } r_0 && \text{und } g = \underline{\text{global}}(\text{varref}, s) \\
& \quad \text{LDC } \underline{\text{bits}}(e_1) && \quad \text{// Bitanzahl (1. Wert)} \\
& \quad \text{PUSH } r_0 && \quad \text{// Speicher lesen (1. Wert)} \\
& \quad \text{LDV } g && \\
& \quad \text{INC } r_0 && \quad \text{// nächste Adresse} \\
& \quad \vdots && \\
& \quad \text{LDC } \underline{\text{bits}}(e_n) && \quad \text{// Bitanzahl (n. Wert)} \\
& \quad \text{PUSH } r_0 && \quad \text{// Speicher lesen (n. Wert)} \\
& \quad \text{LDV } g &&
\end{aligned}$$

Das Erzeugen eines neuen Prozesses kann nun erfolgen, indem nacheinander alle Argumente und deren Bit-Anzahlen auf den Datenkeller gelegt werden und dann der Prozess

unter Angabe der Anzahl der elementaren Argumente und des initialen Programmzählers angelegt wird.

Da der `run`-Befehl in Ausdrücken verwendet werden kann, handelt es sich hier um eine Erweiterung der `expr`-Funktion:

$$\begin{aligned} &\underline{\text{expr}}(\underline{\text{run}} \text{ name } (\underline{\text{arg}}_1, \dots, \underline{\text{arg}}_n) \text{ priority? } , s) \\ &:= \underline{\text{runarg}}(\underline{\text{arg}}_1, t_1, s) \quad \text{falls } \underline{\text{st}}_P(\text{name}, s) = (\underline{\text{proc}}, s', \underline{\text{prm}}, n') \\ &\quad \vdots \\ &\quad \underline{\text{runarg}}(\underline{\text{arg}}_n, t_n, s) \quad \text{wobei } \underline{\text{st}}_P(\underline{\text{prm}}(i), s') = (\underline{\text{var}}, t_i, s', a_i, v_i) \text{ für alle } 1 \leq i \leq n \\ &\quad \text{und } \underline{\text{ftype}}(t_1, \dots, t_n) = (e_1, \dots, e_k) \\ &\quad \text{RUN } k, \text{ name} \end{aligned}$$

Anmerkung: Diese Definition lässt keine Felder als aktuelle Parameter zu. Dies ist erwünscht, da ein Prozess in Promela keine Felder als formale Parameter besitzen kann.

2.9.2 Zwischencode eines Prozesses

Die Funktion `proc` (process) erzeugt den Zwischencode eines Prozesses.

Für die Parameter eines Prozesses existiert ein eigener Gültigkeitsbereich s , der für Prozesse ohne Parameter leer ist und ansonsten durch die virtuelle Maschine beim Anlegen des Prozesses initialisiert wird. Deshalb erfolgt hier nur noch die Initialisierung der lokalen Variablen im Gültigkeitsbereich s' . Anschließend wird die Sequenz der Anweisungen ausgeführt, bevor der Prozess mit einem Schritt im Modus \underline{T} (terminated) endet:

$$\begin{aligned} &\underline{\text{proc}}(\text{ active? } \underline{\text{proctype}} \text{ name } (\underline{\text{declst}}?) \text{ priority? } \{ \text{ sequence } \}) \\ &:= \text{ name } : \underline{\text{init}}(s') \\ &\quad \underline{\text{sequence}}(\text{ sequence } , s', \underline{N}, \epsilon, -1, \underline{n}) \\ &\quad \text{STEP } \underline{T} \\ &\quad \text{wobei } \underline{\text{st}}_P(\text{name}, 0) = (\underline{\text{proc}}, s, \underline{\text{prm}}, n) \\ &\quad \text{und } (s, s') \in \underline{\text{psc}}_P \end{aligned}$$

Falls die Ausführung des Prozesses an eine Bedingung geknüpft ist, muss im ersten Schritt deren Erfüllung abgewartet werden:

$$\begin{aligned} &\underline{\text{proc}}(\text{ active? } \underline{\text{proctype}} \text{ name } (\underline{\text{declst}}?) \text{ priority? } \underline{\text{provided}} (\text{ expr }) \{ \text{ sequence } \}) \\ &:= \text{ name } : \underline{\text{sstmnt}}(\text{ expr } , 0) \\ &\quad \text{STEP } \underline{N} \\ &\quad \underline{\text{init}}(s') \\ &\quad \underline{\text{sequence}}(\text{ sequence } , s', \underline{N}, \epsilon, -1, \underline{n}) \\ &\quad \text{STEP } \underline{T} \\ &\quad \text{wobei } \widehat{\underline{\text{st}}}_P(\text{name}, 0) = (\underline{\text{proc}}, s, \underline{\text{prm}}, n) \\ &\quad \text{und } (s, s') \in \underline{\text{psc}}_P \end{aligned}$$

Der `init`- und der `never`-Prozess werden ebenso wie normale Prozesse übersetzt:

$$\begin{aligned} & \underline{\text{proc}}((\underline{\text{init}}|\underline{\text{never}}) \{ \text{sequence} \}) \\ & := (\underline{\text{init}}|\underline{\text{never}}) : \underline{\text{init}}(s') \\ & \quad \underline{\text{sequence}}(\text{sequence}, s', \underline{\mathbb{N}}, \epsilon, -1, \underline{n}) \\ & \quad \text{STEP } \underline{\mathbb{T}} \\ & \text{wobei } \underline{\text{st}}_P((\underline{\text{init}}|\underline{\text{never}}), 0) = (\underline{\text{proc}}, s, \underline{\text{prm}}, n) \\ & \quad \text{und } (s, s') \in \underline{\text{psc}}_P \end{aligned}$$

2.10 Programme

2.10.1 initial aktive Prozesse

Zu Beginn der Ausführung eines Promela-Programms müssen die initial aktiven Prozesse gestartet werden. Dabei erhalten sämtliche aktuellen Parameter dieser Prozesse den Wert 0.

Die Funktion `irun` erzeugt den Zwischencode für den Start der initial aktiven Kopien des Prozesses `name`:

$$\begin{aligned} & \underline{\text{irun}}(\text{name}) && \text{falls } \underline{\text{st}}_P(\text{name}, 0) = (\underline{\text{proc}}, s, \underline{\text{prm}}, 1) \\ & := \text{RUN } 0, \text{name} \\ & \quad \text{POP } r_0 \\ & \underline{\text{irun}}(\text{name}) && \text{falls } \underline{\text{st}}_P(\text{name}, 0) = (\underline{\text{proc}}, s, \underline{\text{prm}}, n) \text{ mit } n \geq 2 \\ & := \text{LDC } n \\ & \quad \text{POP } r_1 \\ & a_1 : \text{RUN } 0, \text{name} \\ & \quad \text{POP } r_0 \\ & \quad \text{LOOP } r_1, a_1 \end{aligned}$$

2.10.2 Zwischencode eines Promela-Programms

Zur formalen Definition des Zwischencodes eines Promela-Programms wird hier die Funktion `program` definiert, die syntaktische Einheiten eines Promela-Quelltexts in Zwischencode überführt.

Der Zwischencode beginnt dabei mit den Initialisierungen der globalen Variablen. Danach werden alle initial aktiven Prozesse gestartet und der erste Schritt ist abgeschlossen. Die Programmausführung wird dann mit dem `Init`-Prozess fortgesetzt.

Am Ende des Zwischencodes stehen die übersetzten Module eines Programms einfach nacheinander:

$$\begin{aligned}
& \underline{\text{program}}(\text{ module}_1 \ ; \dots \ ; \text{ module}_n) \\
& := \underline{\text{init}}(0) \\
& \quad \forall \text{name} : \underline{\text{st}}_P(\text{name}, 0) = (\underline{\text{proc}}, s, \underline{\text{prm}}, n) , \text{name} \notin \{\underline{\text{init}}, \underline{\text{never}}\} , n \geq 1 \\
& \quad \quad \underline{\text{irun}}(\text{name}) \\
& \quad \left. \begin{array}{l} \text{RUN } 0, \underline{\text{never}} \\ \text{MONITOR} \end{array} \right\} \text{ falls } \underline{\text{st}}_P(\underline{\text{never}}, 0) = (\underline{\text{proc}}, s, \underline{\text{prm}}, n) \\
& \quad \left. \begin{array}{l} \text{STEP } \underline{N} \\ \text{JMP } \underline{\text{init}} \end{array} \right\} \text{ falls } \underline{\text{st}}_P(\underline{\text{init}}, 0) = (\underline{\text{proc}}, s, \underline{\text{prm}}, n) \\
& \quad \left. \begin{array}{l} \text{STEP } \underline{T} \end{array} \right\} \text{ falls } \underline{\text{st}}_P(\underline{\text{init}}, 0) \neq (\underline{\text{proc}}, s, \underline{\text{prm}}, n) \\
& \quad \underline{\text{program}}(\text{ module}_1) \\
& \quad \vdots \\
& \quad \underline{\text{program}}(\text{ module}_n)
\end{aligned}$$

Der Init-Prozess übernimmt hier in seinem ersten Schritt die Aufgabe, die globalen Variablen zu initialisieren und die anderen Prozesse zu starten. Danach wird gegebenenfalls auch der `never`-Prozess erstellt und mit der `MONITOR`-Instruktion zum Monitor-Prozess gemacht. Erst im zweiten Schritt (nach der `STEP`-Instruktion) wird mit dem eigentlichen Init-Prozess begonnen.

Falls es keinen Init-Prozess im Promela-Quelltext gibt, so endet der Init-Prozess nach dem ersten Schritt mit der `STEP`-Instruktion im Modus `T` (terminated).

Deklarationen von Typen, Konstanten und Variablen sind bereits vollständig durch die statische Semantik erfasst und können somit wegfallen:

$$\underline{\text{program}}((\text{ utype } \mid \text{ mtype } \mid \text{ onedecl })) := \epsilon$$

Prozess-Typen werden einfach mit der Funktion `proc` in Zwischencode übersetzt:

$$\begin{aligned}
& \underline{\text{program}}((\text{ proctype } \mid \text{ init } \mid \text{ never })) \\
& \quad := \underline{\text{proc}}((\text{ proctype } \mid \text{ init } \mid \text{ never }))
\end{aligned}$$

Inhaltsverzeichnis

1	Informationen aus der Symboltabelle	1
1.1	Typ einer Variablen-Referenz	1
1.2	globale Variable oder lokale Variable	1
1.3	Wertebereiche der Basistypen	2
1.4	Startadressen der Gültigkeitsbereiche	2
1.5	Speichergrößen von Bereichen und Prozessen	2
1.6	flaches Äquivalent eines Typs	3

2	Zwischencode	3
2.1	Adressen von Variablen	4
2.2	Ausdrücke	4
2.2.1	Kanal-Abfragen in Ausdrücken	6
2.2.2	Kanal-Empfangs-Abfragen in Ausdrücken	7
2.2.3	Zugriff auf andere Prozesse in Ausdrücken	10
2.3	Initialisierungen	11
2.4	einfache Anweisungen	12
2.4.1	Ausgabe	13
2.5	Empfangen und Senden	13
2.5.1	Empfangen	13
2.5.2	Senden	17
2.6	Anweisungen	19
2.7	Schritte	24
2.8	Sequenzen von Schritten	24
2.9	Prozesse	25
2.9.1	Starten neuer Prozesse	25
2.9.2	Zwischencode eines Prozesses	26
2.10	Programme	27
2.10.1	initial aktive Prozesse	27
2.10.2	Zwischencode eines Promela-Programms	27