

Promela - Statische Semantik

Stefan Schürmans

15. Juni 2005

1 Syntax von Promela

Um die statische Semantik von Promela formal definieren zu können, muss zunächst einmal die Syntax der Sprache definiert werden. Dazu wird im folgenden eine kontextfreie Grammatik angegeben.

1.1 Grammatik

Die folgende Promela-Grammatik (eine erweiterte kontextfreie Grammatik) basiert auf der Promela-Grammatik von Spin Version 4.

Es wurden kleine vereinfachende Änderungen vorgenommen, die jedoch ausnahmslos die beschriebene Sprache größer machen, d.h. alle gültigen Promela-Programme lassen nach wie vor mit dieser Grammatik ableiten.

Die Einschränkungen, die damit aus der Grammatik weggefallen sind, werden später anhand des Ableitungsbaumes überprüft und im Falle der Nicht-Einhaltung tritt ein Fehler auf.

Die kleineren Änderungen umfassen neben der Steichung der die Bezeichner und Zahlen aus einzelnen Zeichen aufbauenden Regeln die Zusammenfassung einiger Nichtterminalsymbole, die in der originalen Grammatik aus einander ableitbar waren (z.B. `name` und `uname`).

Die größte Änderung ist die Zusammenlegung von `expr` und `anyexpr`. Dadurch geht die Einschränkung verloren, dass die Kanal-Abfragen `full`, `nfull`, `empty` und `nempty` nur in mit `&&` und `||` konstruierten Ausdrücken vorkommen können. Diese Einschränkung ist aber viel leichter später anhand des Ableitungsbaums zu überprüfen als sie durch die Grammatik zu modellieren.

Die Deklarationslisten (`declst`) zwischen den Anweisungen wurden zu einzelnen Deklarationen (`onedecl`) umgebaut, da die Wiederholung von Deklarationen schon durch die Wiederholung von Anweisungen möglich wird.

Alle Terminalsymbole sind unterstrichen.

spec → module module *
 module → proctype | init | never
 | utype | mtype | onedecl
 proctype → active ? proctype name (declst ?)
 priority ? enabler ? { sequence }
 init → init priority ? { sequence }
 never → never { sequence }
 utype → typedef name { declst }
 mtype → mtype ≡ ? { name (; name) * }
 declst → onedecl (; onedecl) *
 onedecl → visible ? typename ivar (; ivar) *
 | visible ? unsigned uivar (; uivar) *
 typename → bit | bool | byte | pid | short | int | mtype | chan | name
 active → active ([const]) ?
 priority → priority const
 enabler → provided (expr)
 visible → hidden | show
 sequence → step (; step) *
 step → stmt (unless stmt) ?
 | onedecl
 | xr varref (; varref) *
 | xs varref (; varref) *
 ivar → name ([const]) ? (≡ expr | ≡ chinit) ?
 uivar → name ; const ([const]) ? (≡ expr) ?
 chinit → [const] of { typename (; typename) * }
 varref → name ([expr]) ? (. varref) ?
 send → varref ! sendargs
 | varref !! sendargs
 receive → varref ? recvargs
 | varref ?? recvargs
 | varref ? ≤ recvargs ≥
 | varref ?? ≤ recvargs ≥
 poll → varref ? [recvargs]
 | varref ?? [recvargs]
 sendargs → arglst | expr (arglst)
 arglst → expr (; expr) *
 recvargs → recvarg (; recvarg) * | recvarg (recvargs)
 recvarg → varref | eval (varref) | _ ? const
 assign → varref ≡ expr | varref ++ | varref --

```

stmnt → if options fi | do options od
      | atomic { sequence }
      | d\_step { sequence }
      | { sequence }
      | send | receive
      | assign | expr | assert expr
      | else | break
      | goto name
      | name ; stmnt
      | printf ( string ( , arglst )? )
      | printm ( expr )
options → :: sequence ( :: sequence )*
binarop → + | - | \* | / | % | & | ^ | | | && | ||
        | > | < | >= | <= | == | != | << | >>
unarop  → ~ | \_ | !
expr    → ( expr )
        | expr binarop expr
        | unarop expr
        | ( expr -> expr ; expr )
        | len ( varref )
        | poll | varref | const
        | timeout | np\_
        | run name ( arglst ? ) priority ?
        | pc\_value ( expr )
        | name [ expr ] @ name
        | name [ expr ] ; name
        | enabled ( expr )
        | chanpoll ( varref )
chanpoll → full | nfull | empty | nempty
const    → true | false | skip | number

```

Die nicht definierten Nichtterminalsymbole number, name und string stehen hier für eine dezimale Zahl, einen Bezeichner und eine in doppelte Anführungszeichen eingeschlossene Zeichenkette.

Die symbolischen Konstanten haben folgende Werte:

$$\llbracket \text{true} \rrbracket := 1 \quad , \quad \llbracket \text{false} \rrbracket := 0 \quad , \quad \llbracket \text{skip} \rrbracket := 1$$

1.1.1 Operator-Präzedenzen

Die Präzedenzen und Assoziativitäten der Operatoren in Ausdrücken (expr) sind in der obigen Grammatik nicht angegeben. Dies wird durch folgende Tabelle nachgeholt:

Operator	Stelligkeit	Assoziativität	Präzedenz
$\approx, =, !$	1	rechts	hoch
$*, /, \%$	2	links	
$+, -$	2	links	
\ll, \gg	2	links	
$\geq, \leq, \geq=, \leq=$	2	links	
$==, !=$	2	links	
$\&$	2	links	
\wedge	2	links	
\downarrow	2	links	
$\&\&$	2	links	
\parallel	2	links	niedrig

Bemerkung: Die Präzedenz der Operatoren in Promela entspricht also den Präzedenzen der Operatoren in C.

2 Gültigkeitsbereiche

Die statische Semantik besteht im wesentlichen aus den deklarierten Typen und Variablen in den einzelnen Gültigkeitsbereichen.

Außerhalb von Prozessen deklarierte Typen und Variablen sind global, sind also im gesamten Programm gültig. Auf Prozessebene deklarierte Bezeichner sind nur innerhalb dieses Prozesses gültig und innerhalb von Optionen einer `if`- oder `do`-Anweisung deklarierte Bezeichner gelten nur innerhalb dieser Option.

Dabei ist es erlaubt auf einem inneren Bereich den gleichen Bezeichner wie in einem höheren Bereich erneut zu deklarieren. In diesem Fall überdeckt der lokale Bezeichner den globalen, d.h. es ist im inneren Bereich nur der lokale Bezeichner sichtbar.

Der Vorteil dieser Gültigkeitsbereiche liegt in einem kleineren Speicherbedarf. Da die Variablen in den inneren Gültigkeitsbereichen nur kurz existieren (z.B. innerhalb einer Option in einer `do`-Anweisung), belegen sie auch nur kurz Speicher. Gültigkeitsbereiche auf gleicher Ebene, d.h. verschiedene Bereiche mit gleichem übergeordneten Bereich, können dabei den gleichen Speicher für ihre lokalen Variablen nutzen, da sich ein Prozess nie gleichzeitig in zwei Gültigkeitsbereichen auf gleicher Ebene befinden kann.

2.1 Baumstruktur der Gültigkeitsbereiche

Die Gültigkeitsbereiche bilden eine Baumstruktur, die direkt aus dem Promela-Quelltext aufgebaut werden kann. Dabei werden die Gültigkeitsbereiche, also die Knoten des Baums, mit Zahlen $s \in \mathbb{N}_0$ (scope) bezeichnet. Die gerichteten Kanten des Baums werden durch eine Menge $S \subseteq \mathbb{N}_0 \times \mathbb{N}_0$ (scopes) dargestellt, wobei $(p, c) \in S$ bedeutet, dass p (parent) der übergeordnete Bereich von c (child) ist.

In den folgenden Definitionen wird an einigen Stellen ein neuer Knoten für einen neuen Bereich angelegt. Dazu wird die Knotennummer des neuen Bereichs $\underline{\text{nsc}}(S)$ (next scope) aus der Kantenmenge S des bisherigen Baumes bestimmt:

$$\underline{\text{nsc}}(S) := (\max(\{p, c \mid (p, c) \in S\} \cup \{-1\})) + 1$$

Mit der Funktion $\underline{\text{scup}} : \text{Code} \times \mathbb{N}_0 \times \mathfrak{P}(\mathbb{N}_0 \times \mathbb{N}_0) \rightarrow \mathfrak{P}(\mathbb{N}_0 \times \mathbb{N}_0)$ (scopes update) wird für eine syntaktische Einheit C des Promela-Quelltexts, den aktuellen Gültigkeitsbereich s und den bisherigen Baum S eine Aktualisierung des Gültigkeitsbereich-Baums zu $\underline{\text{scup}}(C, s, S)$ durchgeführt.

Dabei werden die einzelnen Aktualisierungen aller Module eines Promela-Programms auf-
gesammelt:

$$\begin{aligned} \underline{\text{scup}}(\text{ module}_1 \ ; \ \dots \ ; \ \text{ module}_n, s, S_0) &:= S_n \\ \text{mit } S_k &= \underline{\text{scup}}(\text{ module}_k, s, S_{k-1}) \text{ f\"ur alle } k \in \{1, \dots, n\} \end{aligned}$$

Deklarationen von Typen, Konstanten und Variablen \u00e4ndern an den G\u00fcltigkeitsbereichen
nichts:

$$\underline{\text{scup}}((\text{ utype } | \text{ mtype } | \text{ onedekl}), s, S) := S$$

Prozessdeklarationen auf globaler Ebene bilden einen neuen G\u00fcltigkeitsbereich, der dem
globalen Bereich untergeordnet ist. F\u00fcr die formalen Parameter eines Prozesses muss da-
bei ein eigener G\u00fcltigkeitsbereich eingef\u00fchrt werden, der in der Hierarchie zwischen dem
globalen Bereich und dem Bereich mit den Anweisungen des Prozesses liegt:

$$\begin{aligned} \underline{\text{scup}}(\text{ active? } \underline{\text{proctype}} \text{ name } (\underline{\text{declst}}) \text{ priority? } \text{ enabler? } \{ \text{ sequence } \}, s, S) &:= S'' \\ \text{mit } S' &= \underline{\text{scup}}(\text{ declst}, s', S \cup \{(s, s')\}), \text{ wobei } s' = \underline{\text{nsc}}(S) \\ \text{und } S'' &= \underline{\text{scup}}(\text{ sequence}, s'', S' \cup \{(s', s'')\}), \text{ wobei } s'' = \underline{\text{nsc}}(S') \\ \underline{\text{scup}}(\text{ active? } \underline{\text{proctype}} \text{ name } () \text{ priority? } \text{ enabler? } \{ \text{ sequence } \}, s, S) &:= S'' \\ \underline{\text{scup}}((\text{ init } \text{ priority? } | \underline{\text{never}}) \{ \text{ sequence } \}, s, S) &:= S'' \\ \text{mit } S' &= S \cup \{(s, s')\}, \text{ wobei } s' = \underline{\text{nsc}}(S) \\ \text{und } S'' &= \underline{\text{scup}}(\text{ sequence}, s'', S' \cup \{(s', s'')\}), \text{ wobei } s'' = \underline{\text{nsc}}(S') \end{aligned}$$

In Befehlssequenzen k\u00f6nnen die einzelnen Befehle jeweils eine Aktualisierung bewirken.
Die einzelnen Aktualisierungen werden nacheinander abgearbeitet:

$$\begin{aligned} \underline{\text{scup}}(\text{ step}_1 \ ; \ \dots \ ; \ \text{ step}_n, s, S_0) &:= S_n \\ \text{mit } S_k &= \underline{\text{scup}}(\text{ step}_k, s, S_{k-1}) \text{ f\"ur alle } k \in \{1, \dots, n\} \end{aligned}$$

Einfache Befehle \u00e4ndern nichts am G\u00fcltigkeitsbereich:

$$\begin{aligned} \underline{\text{scup}}((\text{ xr } | \text{ xs}) \text{ varref } (\text{ _ varref})^*, s, S) &:= S \\ \underline{\text{scup}}(\text{ stmt}_1 \ \underline{\text{unless}} \ \text{ stmt}_2, s, S) &:= S'' \\ \text{mit } S' &= \underline{\text{scup}}(\text{ stmt}_1, s, S) \\ \text{und } S'' &= \underline{\text{scup}}(\text{ stmt}_2, s, S') \\ \underline{\text{scup}}(\text{ name } \ ; \ \text{ stmt}, s, S) &:= \underline{\text{scup}}(\text{ stmt}, s, S) \\ \underline{\text{scup}}((\text{ send } | \text{ receive } | \text{ assign } | \text{ expr}), s, S) &:= S \\ \underline{\text{scup}}((\text{ else } | \text{ break}), s, S) &:= S \\ \underline{\text{scup}}((\text{ goto } \text{ name } | \underline{\text{assert}} \ \text{ expr}), s, S) &:= S \\ \underline{\text{scup}}((\text{ printf } | \text{ printm}) (\dots), s, S) &:= S \end{aligned}$$

Am Eintritt in einen strukturierten Befehl beginnt ein neuer G\u00fcltigkeitsbereich:

$$\underline{\text{scup}}((\text{atomic} \mid \text{d_step})? \{ \text{sequence} \}, s, S) := S'$$

mit $S' = \underline{\text{scup}}(\text{sequence}, s', S \cup \{(s, s')\})$, wobei $s' = \underline{\text{nsc}}(S)$

Etwas komplizierter ist die Situation bei `if`- und `do`-Anweisungen, da hier bei jeder neuen Option auch ein neuer Bereich beginnt, aber alle Bereiche den gleichen übergeordneten Bereich besitzen.

$$\underline{\text{scup}}((\text{if} \mid \text{do}) :: \text{sequence}_1 \dots :: \text{sequence}_n (\text{fi} \mid \text{od}), s, S_0) := S_n$$

mit $S_k = \underline{\text{scup}}(\text{sequence}_k, s_k, S_{k-1} \cup \{(s, s_k)\})$ für alle $k \in \{1, \dots, n\}$
wobei $s_k = \underline{\text{nsc}}(S_{k-1})$ für alle $k \in \{1, \dots, n\}$

Man erhält damit den Gültigkeitsbereich-Baum eines Promela-Programms P bestehend aus den Knoten $\underline{\text{sc}}_P$ (scopes) und den gerichteten Kanten $\underline{\text{psc}}_P$ (parent scopes):

$$\underline{\text{sc}}_P := \{0\} \cup \{p, c \mid (p, c) \in \underline{\text{scup}}(P, 0, \emptyset)\}$$

$$\underline{\text{psc}}_P := \underline{\text{scup}}(P, 0, \emptyset)$$

2.2 Beispiel

Promela-Programm	s	S
<code>int x = 23;</code>	0	\emptyset
<code>init</code>	{	1 $\{(0, 1)\}$
<code>int y;</code>	{	1 $\{(0, 1)\}$
<code>{</code>	2	2 $\{(0, 1), (1, 2)\}$
<code>int z = 5, x[3]</code>	2	2 $\{(0, 1), (1, 2)\}$
<code>}</code>	1	1 $\{(0, 1), (1, 2)\}$
<code>}</code>	0	0 $\{(0, 1), (1, 2)\}$
<code>proctype myproc(int a)</code>	3	3 $\{(0, 1), (1, 2), (0, 3)\}$
<code>{</code>	4	4 $\{(0, 1), (1, 2), (0, 3), (3, 4)\}$
<code>int b</code>	4	4 $\{(0, 1), (1, 2), (0, 3), (3, 4)\}$
<code>}</code>	0	0 $\{(0, 1), (1, 2), (0, 3), (3, 4)\}$

$$\underline{\text{sc}}_P := \{0, 1, 2, 3, 4\}$$

$$\underline{\text{psc}}_P := \{(0, 1), (1, 2), (0, 3), (3, 4)\}$$

3 Symboltabelle

Die deklarierten Bezeichner werden in einer Symboltabelle verwaltet. Bevor nun die Symboltabelle eines Promela-Programms definiert werden kann, muss zunächst einmal die Struktur der Symboltabelle festgelegt werden.

3.1 Aufbau der Symboltabelle

Die dazu benutzte Funktion `st` ordnet jedem Element der Menge `Identifiers` der zulässigen Bezeichner und jeden Gültigkeitsbereich $s \in \mathbb{N}_0$ die Bedeutung dieses Bezeichners in diesem

Bereich zu. Dabei kann ein Bezeichner für eine Konstante, einen benutzerdefinierten Typ, eine Variable oder ein Feld stehen oder aber nicht definiert sein.

$$\underline{st} : \underline{\text{Idents}} \times \mathbb{N}_0 \rightarrow \underline{\text{ST}}$$

Die Menge $\underline{\text{ST}}$ enthält dabei alle möglichen Symboltabelleneinträge:

$$\begin{aligned} \underline{\text{ST}} := & \{ \underline{\text{mtype}} \} \times \mathbb{N} \\ & \cup \{ (\underline{\text{utype}}, \underline{\text{mb}}) \mid \underline{\text{mb}} : \underline{\text{Idents}} \rightarrow \underline{\text{MB}} \} \\ & \cup \{ \underline{\text{var}} \} \times \underline{\text{Idents}} \times \mathbb{N}_0 \times \mathbb{N}_0 \times (\mathbb{Z} \cup (\mathbb{N}_0 \times \underline{\text{Idents}}^+)) \\ & \cup \{ \underline{\text{arr}} \} \times \underline{\text{Idents}} \times \mathbb{N} \times \mathbb{N}_0 \times \mathbb{N}_0 \times (\mathbb{Z} \cup (\mathbb{N}_0 \times \underline{\text{Idents}}^+)) \\ & \cup \{ (\underline{\text{proc}}, s, \underline{\text{prm}}, n) \mid s \in \mathbb{N}_0, \underline{\text{prm}} : \mathbb{N} \rightarrow \underline{\text{Idents}}, n \in \mathbb{N}_0 \} \\ & \cup \{ \underline{\text{undef}} \} \end{aligned}$$

$\underline{st}(i, s)$	Bedeutung
$(\underline{\text{mtype}}, c)$	Konstante mit Wert c
$(\underline{\text{utype}}, \underline{\text{mb}})$	benutzerdefinierter Typ („struct“) mit Elementen $\underline{\text{mb}}$
$(\underline{\text{var}}, t, s', a, v)$	Variable vom Typ t aus Bereich s' an Adresse a mit initialem Wert v
$(\underline{\text{var}}, \underline{\text{chan}}, s', a, (l, T))$	Kanal-Variable aus Bereich s' an Adresse a für neuen Kanal der Länge l vom Typ T
$(\underline{\text{arr}}, t, n, s', a, v)$	Feld vom Typ t mit n Einträgen aus Bereich s' an Adresse a mit initialem Wert v für jeden Eintrag
$(\underline{\text{arr}}, \underline{\text{chan}}, n, s', a, (l, T))$	Kanal-Feld mit n Einträgen aus Bereich s' an Adresse a für neuen Kanal der Länge l vom Typ T für jeden Eintrag
$(\underline{\text{proc}}, s', \underline{\text{prm}}, n)$	Prozess-Typ mit oberstem Bereich s' und Parametern $\underline{\text{prm}}$ von dem initial n Prozesse aktiv sind
$\underline{\text{undef}}$	nichts mit diesem Bezeichner definiert

Die Initialisierungen $v \in \mathbb{Z}$ ergeben für Variablen eines benutzerdefinierten Typs keinen Sinn. Sie werden in Übereinstimmung mit Spin ignoriert und können daher in der Symboltabelle immer 0 sein.

Die Funktion $\underline{\text{prm}}$ ordnet jeder Parameter-Nummer (also den Zahlen $1, \dots, n$ für einen Prozess-Typ mit n Parametern) den Bezeichner dieses Parameters zu. Anhand dieses Bezeichners und dem obersten Bereich des Prozesses kann man dann die entsprechende Variable in der Symboltabelle finden.

3.1.1 Struktur-Einträge

Die Menge $\underline{\text{MB}}$ enthält alle möglichen Bedeutungen von Einträgen („members“) eines benutzerdefinierten Typs:

$$\begin{aligned} \underline{\text{MB}} := & \{ \underline{\text{var}} \} \times \underline{\text{Idents}} \times \mathbb{N}_0 \times (\mathbb{Z} \cup (\mathbb{N}_0 \times \underline{\text{Idents}}^+)) \\ & \cup \{ \underline{\text{arr}} \} \times \underline{\text{Idents}} \times \mathbb{N} \times \mathbb{N}_0 \times (\mathbb{Z} \cup (\mathbb{N}_0 \times \underline{\text{Idents}}^+)) \\ & \cup \{ \underline{\text{undef}} \} \end{aligned}$$

$\underline{\text{mb}}(i)$	Bedeutung
$(\underline{\text{var}}, t, o, v)$	Variable vom Typ t an Offset o mit initialem Wert v
$(\underline{\text{var}}, \underline{\text{chan}}, o, (l, T))$	Kanal-Variable an Offset o für neuen Kanal der Länge l vom Typ T
$(\underline{\text{arr}}, t, n, o, v)$	Feld vom Typ t mit n Elementen an Offset o mit initialem Wert v für jeden Eintrag
$(\underline{\text{arr}}, \underline{\text{chan}}, n, o, (l, T))$	Kanal-Feld mit n Einträgen an Offset o für neuen Kanal der Länge l vom Typ T für jeden Eintrag
$\underline{\text{undef}}$	kein Element mit diesem Bezeichner definiert

Die Initialisierungen $v \in \mathbb{Z}$ ergeben für Variablen eines benutzerdefinierten Typs auch hier keinen Sinn. Sie werden in Übereinstimmung mit Spin ignoriert und können daher auch hier immer 0 sein.

3.1.2 Initialisierung von Variablen und Struktur-Einträgen

Um zu Beginn eines Gültigkeitsbereichs die initialen Werte der Variablen und Felder zu kennen, werden diese auch in der Symboltabelle als eine Zahl aus \mathbb{Z} gespeichert.

Diese Initialisierungen ergeben für Variablen eines benutzerdefinierten Typs keinen Sinn. Solche Variablen werden aber sowieso mit den initialen Werten aus der Typ-Deklaration initialisiert. Daher wird in Übereinstimmung mit Spin der Initialisierungswert bei Variablen benutzerdefinierter Typen ignoriert.

Die Initialisierungen von Struktur-Einträgen in der Definition des benutzerdefinierten Typs (also im „typedef“) werden mit Hilfe der Funktion $\underline{\text{mb}}$ im Symboltabelleneintrag dieses Typs gespeichert.

Felder brauchen hier nicht speziell behandelt zu werden, da alle ihre Einträge stets auf den gleichen Wert initialisiert werden. Es ist also möglich den initialen Wert der Feld-Einträge genau wie bei einer Variable zu beschreiben.

Es ist notwendig die Initialisierungen in der Symboltabelle zu speichern, da z.B. folgendes Promela-Programm zur Ausgabe 23 führt:

```
typedef mytype
{
  int mymember = 23;
}

init
{
  mytype myvar;
  printf( "%d\n", myvar.mymember );
}
```

3.1.3 Sichtbarkeit von Bezeichnern

Bezeichner, die in einem höheren Gültigkeitsbereich deklariert wurden und nicht durch lokale Deklarationen überschattet werden, müssen auch im lokalen Bereich sichtbar sein.

Um dies zu erreichen, wird für jede Symboltabelle $\underline{\text{st}} : \underline{\text{Idents}} \times \mathbb{N}_0 \rightarrow \underline{\text{ST}}$ die erweiterte Symboltabelle $\widehat{\underline{\text{st}}} : \underline{\text{Idents}} \times \mathbb{N}_0 \rightarrow \underline{\text{ST}}$ definiert:

$$\widehat{\underline{\text{st}}}(i, 0) := \underline{\text{st}}(i, 0)$$

$$\widehat{\underline{\text{st}}}(i, s) := \begin{cases} \underline{\text{st}}(i, s) & \text{falls } \underline{\text{st}}(i, s) \neq \underline{\text{undef}} \text{ und } s > 0 \\ \widehat{\underline{\text{st}}}(i, p) & \text{falls } \underline{\text{st}}(i, s) = \underline{\text{undef}} \text{ und } (p, s) \in \underline{\text{psc}}_p \text{ und } s > 0 \end{cases}$$

D.h. falls ein Bezeichner i im Bereich s nicht deklariert ist, so wird die Bedeutung dieses Bezeichners i aus dem übergeordneten Gültigkeitsbereich $\text{psc}(s)$ übernommen.

3.2 Größe von Typen, Variablen und Gültigkeitsbereichen

Während der Deklaration von Variablen muss die nächste freie Adresse bestimmt werden. Dazu ist es notwendig die bisherige Größe des aktuellen Gültigkeitsbereiches zu ermitteln, wozu man wiederum Kenntnis über die Größe von Typen und Variablen benötigt.

Um im Nachfolgenden einfacher mit Basistypen umgehen zu können, wird die Menge basetypes der Basistypen definiert:

$$\begin{aligned} \text{basetypes} &:= \{\text{bit}, \text{bool}, \text{byte}, \text{pid}, \text{short}, \text{int}, \text{mtype}, \text{chan}\} \\ &\cup \{\text{unsigned}_c \mid 1 \leq c \leq 32\} \end{aligned}$$

Die Funktion size bestimmt dabei die Größe $\text{size}(i, s, \text{st})$ des Typs mit dem Bezeichner $i \in \text{Idents}$ im Bereich $s \in \mathbb{N}_0$ bzw. die Größe $\text{size}(\text{mb}, \text{st})$ des Typs mit den Einträgen mb anhand der Symboltabelle st:

$$\begin{aligned} \text{size}(i, s, \text{st}) &:= 1 && \text{falls } i \in \text{basetypes} \\ \text{size}(i, s, \text{st}) &:= \text{size}(\text{mb}, \text{st}) && \text{falls } \hat{\text{st}}(i, s) = (\text{utype}, \text{mb}) \end{aligned}$$

$$\begin{aligned} \text{size}(\text{mb}, \text{st}) &:= \max (\{ o + \text{size}(t, 0, \text{st}) \mid \exists i : \text{mb}(i) = (\text{var}, t, o, X) \} \\ &\cup \{ o + n \cdot \text{size}(t, 0, \text{st}) \mid \exists i : \text{mb}(i) = (\text{arr}, t, n, o, X) \} \\ &\cup \{0\}) \end{aligned}$$

Für die Größe einer Variablen mit dem Bezeichner i gilt:

$$\begin{aligned} \text{size}(i, s, \text{st}) &:= \text{size}(t, 0, \text{st}) && \text{falls } \hat{\text{st}}(i, s) = (\text{var}, t, s', a, X) \\ \text{size}(i, s, \text{st}) &:= n \cdot \text{size}(t, 0, \text{st}) && \text{falls } \hat{\text{st}}(i, s) = (\text{arr}, t, n, s', a, X) \end{aligned}$$

Damit lässt sich nun auch die Größe eines Gültigkeitsbereichs s in der Symboltabelle st definieren:

$$\begin{aligned} \text{size}(s, \text{st}) &:= \max (\{ a + \text{size}(t, 0, \text{st}) \mid \exists i : \text{st}(i, s) = (\text{var}, t, s', a, X) \} \\ &\cup \{ a + n \cdot \text{size}(t, 0, \text{st}) \mid \exists i : \text{st}(i, s) = (\text{arr}, t, n, s', a, X) \} \\ &\cup \{0\}) \end{aligned}$$

3.3 Aktualisierungen der Symboltabelle

Die Funktion stup führt mit dem Aufruf $\text{stup}(C, s, \text{st})$ eine Aktualisierung der Symboltabelle st aufgrund des Quelltextfragments C im Bereich s durch.

Die leere Symboltabelle wird mit st_\emptyset bezeichnet und ist definiert als:

$$\text{st}_\emptyset(i) := \text{undef} \text{ für alle } i \in \text{Idents}$$

In den nachfolgenden Definitionen wird an manchen Stellen die Nummer eines neu beginnenden Gültigkeitsbereichs benötigt. Dabei wird die Berechnung dieser Nummer nur mit $\underline{nsc}(\dots)$ angedeutet und das Argument S für die Funktion \underline{nsc} nicht angegeben. Es wäre möglich, parallel zur Berechnung der Symboltabelle mit der Funktion \underline{stup} die Berechnung von S wie in der oben definierten Funktion \underline{scup} durchzuführen. Da dies aber exakt gleich verlaufen und die Notation nur unnötig aufblähen würde, wird hier darauf verzichtet.

Eine Definition von zwei Variablen (Felder, Typen, Konstanten) unter dem gleichen Bezeichner im gleichen Gültigkeitsbereich ist verboten.

Um die Definitionen hier nicht unnötig kompliziert werden zu lassen, wird auf die vollständige formale Definition dieser Fehlersituation verzichtet.

Dieser Fehler tritt in den folgenden Definitionen genau dann auf, wenn $\underline{st}[(i, s)/x]$ verwendet wird und $\underline{st}(i, s) \neq \underline{undef}$ ist.

3.3.1 Deklaration von Konstanten

Bei der Deklaration von Konstanten wird einfach der nächste noch nicht von einer anderen Konstante belegte Wert benutzt:

$$\begin{aligned} \underline{stup}(\underline{mtype} \equiv? \{ \text{name}_1, \dots, \text{name}_n \}, s, \underline{st}) \\ &:= \underline{st}[(\text{name}_1, s)/(\underline{mtype}, c' + 1), \dots, (\text{name}_n, s)/(\underline{mtype}, c' + n)] \\ &\text{mit } c' = \max \{ c \mid \exists i, s' : \underline{st}(i, s') = (\underline{mtype}, c) \} \cup \{0\} \end{aligned}$$

3.3.2 Deklaration von Variablen

Für Variablen und Felder wird einfach die nächste noch nicht benutzte Adresse gesucht und diese für die neue Variable bzw. das neue Feld benutzt. Als Adresse wird hier einfach die Größe der bisherigen Symboltabelle des aktuellen Gültigkeitsbereiches benutzt, da dies die nächste freie Adresse ist:

$$\begin{aligned} \underline{stup}(\text{visible? typename name}, s, \underline{st}) \\ &:= \underline{st}[(\text{name}, s)/(\underline{var}, \text{typename}, s, \underline{size}(s, \underline{st}), 0)] \\ \underline{stup}(\text{visible? typename name} \equiv \text{anyexpr}, s, \underline{st}) \\ &:= \underline{st}[(\text{name}, s)/(\underline{var}, \text{typename}, s, \underline{size}(s, \underline{st}), \llbracket \text{anyexpr} \rrbracket_{\widehat{\underline{st}}(\cdot, s)})] \\ \underline{stup}(\text{visible? typename name} [\underline{const}], s, \underline{st}) \\ &:= \underline{st}[(\text{name}, s)/(\underline{arr}, \text{typename}, \llbracket \underline{const} \rrbracket, s, \underline{size}(s, \underline{st}), 0)] \\ \underline{stup}(\text{visible? typename name} [\underline{const}] \equiv \text{anyexpr}, s, \underline{st}) \\ &:= \underline{st}[(\text{name}, s)/(\underline{arr}, \text{typename}, \llbracket \underline{const} \rrbracket, s, \underline{size}(s, \underline{st}), \llbracket \text{anyexpr} \rrbracket_{\widehat{\underline{st}}(\cdot, s)})] \end{aligned}$$

wobei $\text{typename} \in \underline{\text{basetypes}}$

oder $\underline{st}(\text{typename}, 0) = (\underline{utype}, \underline{mb})$

Die Berechnung des initialen Werts schließt bei Basistypen eine Konvertierung auf den Typ der Variable mit ein. Der initiale Wert kann also für Basistypen immer ohne weitere Konvertierung der Variablen zugewiesen werden.

Es sei hier noch einmal darauf hingewiesen, dass der initiale Wert im Symboltabelleneintrag für Variablen benutzerdefinierter Typen ignoriert wird. Dieser Wert muss also nicht berechnet werden und kann zu 0 gesetzt werden.

Für den Typ `unsigned` erfolgt die Deklaration ebenso, allerdings muss hier noch die Anzahl der Bits berücksichtigt werden. Arrays dieses Typs scheint Spin nicht zu kennen, sie werden aber hier definiert:

$$\begin{aligned} \text{stup}(\text{visible? } \text{unsigned } \text{name}; \text{const}', s, \text{st}) & \\ & := \text{st}[(\text{name}, s) / (\text{var}, \text{unsigned}_{[\text{const}']}, s, \text{size}(s, \text{st}), 0)] \\ \text{stup}(\text{visible? } \text{unsigned } \text{name}; \text{const}' = \text{anyexpr}, s, \text{st}) & \\ & := \text{st}[(\text{name}, s) / (\text{var}, \text{unsigned}_{[\text{const}']}, s, \text{size}(s, \text{st}), [\text{anyexpr}]_{\widehat{\text{st}}(\cdot, s)})] \\ \text{stup}(\text{visible? } \text{unsigned } \text{name}; \text{const}' [\text{const}], s, \text{st}) & \\ & := \text{st}[(\text{name}, s) / (\text{arr}, \text{unsigned}_{[\text{const}']}, [\text{const}], s, \text{size}(s, \text{st}), 0)] \\ \text{stup}(\text{visible? } \text{unsigned } \text{name}; \text{const}' [\text{const}] = \text{anyexpr}, s, \text{st}) & \\ & := \text{st}[(\text{name}, s) / (\text{arr}, \text{unsigned}_{[\text{const}']}, [\text{const}], s, \text{size}(s, \text{st}), [\text{anyexpr}]_{\widehat{\text{st}}(\cdot, s)})] \end{aligned}$$

Wird bei der Deklaration einer Kanal-Variablen ein neuer Kanal erzeugt, so wird statt des initialen Werts der Kanal-Typ (l, T) bestehend aus der maximalen Länge l und dem Typ einer Nachricht T in der Symboltabelle abgelegt:

$$\begin{aligned} \text{stup}(\text{visible? } \text{chan } \text{name} = [\text{const}'] \text{ of } \{ \text{typename}_1, \dots, \text{typename}_n \}, s, \text{st}) & \\ & := \text{st}[(\text{name}, s) / (\text{var}, \text{chan}, s, \text{size}(s, \text{st}), ([\text{const}'], (\text{typename}_1, \dots, \text{typename}_n)))] \\ \text{stup}(\text{visible? } \text{chan } \text{name} [\text{const}] = [\text{const}'] \text{ of } \{ \text{typename}_1, \dots, \text{typename}_n \}, s, \text{st}) & \\ & := \text{st}[(\text{name}, s) / (\text{arr}, \text{chan}, [\text{const}], s, \text{size}(s, \text{st}), ([\text{const}'], (\text{typename}_1, \dots, \text{typename}_n)))] \end{aligned}$$

Nicht initialisierte Kanal-Variablen werden gemäß der weiter oben angegebenen Definition für nicht initialisierte Variablen mit dem initialen Wert 0 (ungültige Kanal-Nummer) in die Symboltabelle eingetragen.

Falls mehrere Variablen gleichzeitig definiert werden, so wird dies so behandelt, als ob alle diese Variablen nacheinander mit dem gleichen Typ deklariert worden wären:

$$\begin{aligned} \text{stup}(\text{visible? } \text{typename } \text{ivar}_1, \dots, \text{ivar}_n, s, \text{st}_0) & := \text{st}_n \\ \text{mit } \text{st}_k & = \text{stup}(\text{visible? } \text{typename } \text{ivar}_k, s, \text{st}_{k-1}) \text{ für alle } k \in \{1, \dots, n\} \\ \text{stup}(\text{visible? } \text{unsigned } \text{uivar}_1, \dots, \text{uivar}_n, s, \text{st}_0) & := \text{st}_n \\ \text{mit } \text{st}_k & = \text{stup}(\text{visible? } \text{unsigned } \text{uivar}_k, s, \text{st}_{k-1}) \text{ für alle } k \in \{1, \dots, n\} \end{aligned}$$

Eine Deklarationsliste wird Schritt für Schritt abgearbeitet:

$$\begin{aligned} \text{stup}(\text{onedekl}_1; \dots; \text{onedekl}_n, s, \text{st}_0) & := \text{st}_n \\ \text{mit } \text{st}_k & = \text{stup}(\text{onedekl}_k, s, \text{st}_{k-1}) \text{ für alle } k \in \{1, \dots, n\} \end{aligned}$$

3.3.3 Deklaration von Strukturen

Um Strukturen zu definieren, wird die Symboltabellen-Aktualisierung-Funktion `stup` ausgehend von der leeren Symboltabelle st_0 auf die Deklarationsliste der Strukturdefinition

angewand. Die so entstehenden lokalen Variablen werden anschließend in die Elemente der Struktur umgewandelt, so dass die Struktur in die aktuelle Symboltabelle \underline{st} eingetragen werden kann:

$$\begin{aligned} \underline{stup}(\underline{\text{typedef}} \text{ name}\{\underline{\text{declst}}\}, s, \underline{st}) &:= \underline{st}[(\text{name}, s)/(\underline{\text{u}}\text{type}, \underline{\text{mb}})] \\ \text{mit } \underline{\text{mb}}(i) &= \begin{cases} (\underline{\text{var}}, t, o, X) \text{ falls } \underline{st}_{tmp}(i, 0) = (\underline{\text{var}}, t, 0, o, X) \\ (\underline{\text{arr}}, t, n, o, X) \text{ falls } \underline{st}_{tmp}(i, 0) = (\underline{\text{arr}}, t, n, 0, o, X) \end{cases} \\ \text{wobei } \underline{st}_{tmp} &= \underline{stup}(\underline{\text{declst}}, 0, \underline{st}_0) \end{aligned}$$

3.3.4 Deklaration von Prozessen

In den Prozessdeklarationen werden die lokalen Bezeichner mit dem entsprechenden untergeordneten Gültigkeitsbereich in die Symboltabelle eingetragen. Zusätzlich wird die Symboltabelle um einen Eintrag für den Prozess ergänzt, wobei die speziellen Prozesse (init , never) unter dem jeweiligen Schlüsselwort in die Symboltabelle eingetragen werden. Für die formalen Parameter eines Prozesses wird der eigene Gültigkeitsbereich berücksichtigt:

$$\begin{aligned} \underline{stup}(\text{active? } \underline{\text{proctype}} \text{ name } (\underline{\text{declst}}) \text{ priority? enabler? } \{ \text{sequence} \}, s, \underline{st}) &:= \underline{st}'' \\ \text{mit } \underline{st}' &= \underline{stup}(\underline{\text{declst}}, s', \underline{st}[(\text{name}, s)/(\underline{\text{proc}}, s', \underline{\text{prm}}, n)]), \text{ wobei } s' = \underline{\text{nsc}}(\dots) \\ \text{und } \underline{st}'' &= \underline{stup}(\text{sequence}, s'', \underline{st}'), \text{ wobei } s'' = \underline{\text{nsc}}(\dots) \\ \text{und } \underline{\text{prm}} &= \underline{\text{prmup}}(\underline{\text{declst}}, \underline{\text{prm}}_0) \\ \text{und } n &= \underline{\text{active}}(\text{active?}) \\ \underline{stup}(\text{active? } \underline{\text{proctype}} \text{ name } () \text{ priority? enabler? } \{ \text{sequence} \}, s, \underline{st}) &:= \underline{st}'' \\ \text{mit } \underline{st}' &= \underline{st}[(\text{name}, s)/(\underline{\text{proc}}, s', \underline{\text{prm}}_0, n)], \text{ wobei } s' = \underline{\text{nsc}}(\dots) \\ \text{und } \underline{st}'' &= \underline{stup}(\text{sequence}, s'', \underline{st}'), \text{ wobei } s'' = \underline{\text{nsc}}(\dots) \\ \text{und } n &= \underline{\text{active}}(\text{active?}) \\ \underline{stup}(\text{ } (\underline{\text{init}} \text{ priority? } | \underline{\text{never}}) \{ \text{sequence} \}, s, \underline{st}) &:= \underline{st}'' \\ \text{mit } \underline{st}' &= \underline{st}[(\underline{\text{init}}|\underline{\text{never}}), s)/(\underline{\text{proc}}, s', \underline{\text{prm}}_0, 1)], \text{ wobei } s' = \underline{\text{nsc}}(\dots) \\ \text{und } \underline{st}'' &= \underline{stup}(\text{sequence}, s'', \underline{st}'), \text{ wobei } s'' = \underline{\text{nsc}}(\dots) \end{aligned}$$

Dabei ist $\underline{\text{prm}}_0(k) = \underline{\text{undef}}$ für alle $k \in \mathbb{N}$ und für $\underline{\text{prmup}}$ gelten folgende Definitionen:

$$\begin{aligned} \underline{\text{prmup}}(\text{visible? typename name}, \underline{\text{prm}}) &:= \underline{\text{prm}}[k/\text{name}] \text{ mit } k = \max(\{k' + 1 \mid \underline{\text{prm}}(k') \neq \underline{\text{undef}}\} \cup \{0\}) \\ \underline{\text{prmup}}(\text{visible? } \underline{\text{unsigned}} \text{ name;const}, \underline{\text{prm}}) &:= \underline{\text{prm}}[k/\text{name}] \text{ mit } k = \max(\{k' + 1 \mid \underline{\text{prm}}(k') \neq \underline{\text{undef}}\} \cup \{0\}) \\ \underline{\text{prmup}}(\text{visible? typename } \text{ivar}_1; \dots; \text{ivar}_n, \underline{\text{prm}}_0) &:= \underline{\text{prm}}_n \\ \text{mit } \underline{\text{prm}}_k &= \underline{\text{prmup}}(\text{visible? typename } \text{ivar}_k, \underline{\text{prm}}_{k-1}) \text{ für alle } k \in \{1, \dots, n\} \\ \underline{\text{prmup}}(\text{visible? } \underline{\text{unsigned}} \text{ uivar}_1; \dots; \text{uivar}_n, \underline{\text{prm}}_0) &:= \underline{\text{prm}}_n \\ \text{mit } \underline{\text{prm}}_k &= \underline{\text{prmup}}(\underline{\text{visible?}} \text{ unsigned } \text{uivar}_k, \underline{\text{prm}}_{k-1}) \text{ für alle } k \in \{1, \dots, n\} \\ \underline{\text{prmup}}(\text{onedec1}_1; \dots; \text{onedec1}_n, \underline{\text{prm}}_0) &:= \underline{\text{prm}}_n \end{aligned}$$

mit $\underline{\text{prm}}_k = \underline{\text{prmup}}(\text{onedec}_k, \underline{\text{prm}}_{k-1})$ für alle $k \in \{1, \dots, n\}$

Diese Definition lässt keine Felder und keine Initialisierungen in formalen Parametern eines Prozesses zu. Dies ist in Übereinstimmung mit der Beschreibung von Promela und mit dem Verhalten von Spin.

Die Funktion `active` wandelt ein eventuell vorhandenes `active`-Konstrukt in die Anzahl der initial aktiven Prozesse um:

$$\begin{aligned} \text{active}(\epsilon) &:= 0 \\ \text{active}(\text{active}) &:= 1 \\ \text{active}(\text{active}[\text{const}]) &:= \llbracket \text{const} \rrbracket \end{aligned}$$

3.4 Symboltabelle eines Promela-Programms

Die Symboltabelle des gesamten Programms kann erstellt werden, indem beginnend mit der leeren Symboltabelle $\underline{\text{st}}_\emptyset$ alle Deklarationen mit der Funktion `stup` aufgesammelt werden. Dabei muss die Funktion `stup` immer mit dem passenden Gültigkeitsbereich aufgerufen werden, um die Zuordnung der Variablen zu den einzelnen Bereichen zu gewährleisten.

Die einzelnen Module eines Promela-Programms werden sukzessiv bearbeitet und somit die darin deklarierten Bezeichner in die Symboltabelle eingetragen:

$$\begin{aligned} \underline{\text{stup}}(\text{module}_1 ; \dots ; \text{module}_n, s, \underline{\text{st}}_\emptyset) &:= \underline{\text{st}}_n \\ \text{mit } \underline{\text{st}}_k = \underline{\text{stup}}(\text{module}_k, s, \underline{\text{st}}_{k-1}) &\text{ für alle } k \in \{1, \dots, n\} \end{aligned}$$

Einfache Befehle ändern die Symboltabelle nicht:

$$\begin{aligned} \underline{\text{stup}}(\text{ } \underline{\text{xr}} \mid \underline{\text{xs}} \text{) } \underline{\text{varref}}(\text{ } \underline{\text{varref}} \text{)}^*, s, \underline{\text{st}} &:= \underline{\text{st}} \\ \underline{\text{stup}}(\text{stmnt}_1 \text{ unless } \text{stmnt}_2, s, \underline{\text{st}}) &:= \underline{\text{st}}'' \\ \text{mit } \underline{\text{st}}' = \underline{\text{stup}}(\text{stmnt}_1, s, \underline{\text{st}}) & \\ \text{und } \underline{\text{st}}'' = \underline{\text{stup}}(\text{stmnt}_2, s, \underline{\text{st}}') & \\ \underline{\text{stup}}(\text{ name } \underline{\text{stmnt}}, s, \underline{\text{st}}) &:= \underline{\text{stup}}(\text{stmnt}, s, \underline{\text{st}}) \\ \underline{\text{stup}}(\text{ } \underline{\text{send}} \mid \underline{\text{receive}} \mid \underline{\text{assign}} \mid \underline{\text{expr}} \text{)}, s, \underline{\text{st}} &:= \underline{\text{st}} \\ \underline{\text{stup}}(\text{ } \underline{\text{else}} \mid \underline{\text{break}} \text{)}, s, \underline{\text{st}} &:= \underline{\text{st}} \\ \underline{\text{stup}}(\text{ } \underline{\text{goto}} \text{ name } \mid \underline{\text{assert}} \text{ expr } \text{)}, s, \underline{\text{st}} &:= \underline{\text{st}} \\ \underline{\text{stup}}(\text{ } \underline{\text{printf}} \mid \underline{\text{printm}} \text{) } (\dots), s, \underline{\text{st}} &:= \underline{\text{st}} \end{aligned}$$

Innerhalb von strukturierten Befehlen müssen die Deklarationen aufgesammelt werden:

$$\begin{aligned} \underline{\text{stup}}(\text{ } \underline{\text{atomic}} \mid \underline{\text{d_step}} \text{)? } \{ \text{sequence } \underline{\text{st}} \}, s, \underline{\text{st}} &:= \underline{\text{st}}' \\ \text{mit } \underline{\text{st}}' = \underline{\text{stup}}(\text{sequence}, s', \underline{\text{st}}), \text{ wobei } s' = \underline{\text{nsc}}(\dots) & \end{aligned}$$

Bei `if`- und `do`-Anweisungen beginnt das Aufsammeln an jeder Option neu:

$\text{stup}(\text{ (if | do) :: sequence}_1 \dots \text{ :: sequence}_n \text{ (fi | od)}, s, \underline{\text{st}}_0) := \underline{\text{st}}_n$
 mit $\underline{\text{st}}_k = \text{stup}(\text{ sequence}_k, s_k, \underline{\text{st}}_{k-1})$ für alle $k \in \{1, \dots, n\}$
 wobei $s_k = \underline{\text{nsc}}(\dots)$ für alle $k \in \{1, \dots, n\}$

Die durch $\underline{\text{st}}_P := \text{stup}(P, 0, \underline{\text{st}}_0)$ aus einem Promela-Programm P entstehende Symbolta-
 belle hat aufgrund der Struktur der Gültigkeitsbereiche ebenfalls eine Baum-artige Struktur.
 Die Symboltabellen $\underline{\text{st}}_P(\cdot, s)$ der einzelnen Gültigkeitsbereiche s bilden die Knoten und die
 Nachfolgerrelation ist durch $\underline{\text{psc}}_P$ gegeben.

In der Symboltabelle $\widehat{\underline{\text{st}}}_P(\cdot, s)$ eines Bereichs s sind dann die Bezeichner in den Symbolta-
 bellen der Vorgängerknoten enthalten, sofern diese nicht überschattet werden.

3.5 Beispiel

Promela-Programm	s	S
<code>int x = 23;</code>	0	\emptyset
<code>init</code>		
<code>{</code>	1	$\{(0, 1)\}$
<code>int y;</code>	1	$\{(0, 1)\}$
<code>{</code>	2	$\{(0, 1), (1, 2)\}$
<code>int z = 5, x[3]</code>	2	$\{(0, 1), (1, 2)\}$
<code>}</code>	1	$\{(0, 1), (1, 2)\}$
<code>}</code>	0	$\{(0, 1), (1, 2)\}$
<code>proctype myproc(int a)</code>	3	$\{(0, 1), (1, 2), (0, 3)\}$
<code>{</code>	4	$\{(0, 1), (1, 2), (0, 3), (3, 4)\}$
<code>int b</code>	4	$\{(0, 1), (1, 2), (0, 3), (3, 4)\}$
<code>}</code>	0	$\{(0, 1), (1, 2), (0, 3), (3, 4)\}$

$\underline{\text{st}}_P(\text{init}, 0) = (\underline{\text{proc}}, 1, \underline{\text{prm}}_\emptyset)$	$\underline{\text{st}}_P(\text{myproc}, 0) = (\underline{\text{proc}}, 3, \underline{\text{prm}} : 1 \mapsto a)$
$\underline{\text{st}}_P(x, 0) = (\underline{\text{var}}, \text{int}, 0, 0, 23)$	$\underline{\text{st}}_P(y, 1) = (\underline{\text{var}}, \text{int}, 1, 0, 0)$
$\underline{\text{st}}_P(z, 2) = (\underline{\text{var}}, \text{int}, 2, 0, 5)$	$\underline{\text{st}}_P(x, 2) = (\underline{\text{arr}}, \text{int}, 3, 2, 1, 0)$
$\underline{\text{st}}_P(a, 3) = (\underline{\text{var}}, \text{int}, 3, 0, 0)$	$\underline{\text{st}}_P(b, 4) = (\underline{\text{var}}, \text{int}, 4, 0, 0)$
$\widehat{\underline{\text{st}}}_P(x, 1) = (\underline{\text{var}}, \text{int}, 0, 0, 23)$	$\widehat{\underline{\text{st}}}_P(x, 2) = (\underline{\text{arr}}, \text{int}, 3, 2, 1, 0)$

3.6 Sonderfälle

Durch das Aufsammeln aller deklarierten Bezeichner zur Symboltabelle des jeweiligen
 Bereichs ergibt sich die Möglichkeit, Variablen erst nach ihrer Benutzung zu deklarieren.

```

init
{
  i = 5;
  printf( "%d\n", i );
  int i
}

```

Hier existiert im Bereich des Init-Prozesses `i` als lokale Variable und somit ist die Verwen-
 dung von `i` in der ersten und zweiten Zeile des Prozessrumpfs erlaubt.

Solche Konstruktionen müssen vermieden werden, da es keinen Grund für deren Verwendung gibt und die Deklaration einer Variablen vor ihrer Benutzung die Lesbarkeit des Programms deutlich erhöht.

Weil die Initialisierung von Variablen komplett in der statischen Semantik behandelt wird, wird in folgendem Programm die Variable `j` mit dem initialen Wert von `i` initialisiert, so dass sich die Ausgabe 23 ergibt.

```
init
{
  int i = 23;
  i = 42;
  int j = i;
  printf( "%d\n", i )
}
```

Dieses Verhalten zeigt sich ebenfalls in Spin.

Jeder Option-Block in einer `if`- oder `do`-Anweisung stellt einen eignen Bereich mit einer eigenen lokalen Symboltabelle dar. Der übergeordnete Bereich ist der die `if`- oder `do`-Anweisung enthaltende Bereich.

Dies ist auch sinnvoll, da die Ausführung von unmittelbar vor der `if`-Anweisung durch den Nichtdeterminismus direkt zu jedem beliebigen Option-Block springen kann.

Es sei ausdrücklich darauf hingewiesen, dass dies ein anderes Verhalten als in Spin ist. Spin behandelt die Deklarationen auch in verschachtelten, nichtdeterministischen Anweisungen linear, so dass z.B. folgendes zulässig ist:

```
if
  :: false; int i = 5
  :: printf( "%d\n", i )
fi
```

Diese Konstruktion führt zur Ausgabe von 5, obwohl nicht intuitiv klar ist, warum die Deklaration und Initialisierung von `i` ausgeführt wurde.

Nach der hier gegebenen Semantik ist `i` in der `printf`-Anweisung nicht deklariert.

In jedem Falle muss man solche Konstruktionen vermeiden und stattdessen die saubere Variante mit Deklaration von `i` vor der Schleife benutzen.

```
int i;
if
  :: false; i = 5
  :: printf( "%d\n", i )
fi
```

Hier wird auch 0 ausgegeben, da die Zuweisung `i = 5` nicht ausgeführt wurde.

Inhaltsverzeichnis

1	Syntax von Promela	1
1.1	Grammatik	1
1.1.1	Operator-Präzedenzen	3
2	Gültigkeitsbereiche	4
2.1	Baumstruktur der Gültigkeitsbereiche	4
2.2	Beispiel	6

3	Symboltabelle	6
3.1	Aufbau der Symboltabelle	6
3.1.1	Struktur-Einträge	7
3.1.2	Initialisierung von Variablen und Struktur-Einträgen	8
3.1.3	Sichtbarkeit von Bezeichnern	8
3.2	Größe von Typen, Variablen und Gültigkeitsbereichen	9
3.3	Aktualisierungen der Symboltabelle	9
3.3.1	Deklaration von Konstanten	10
3.3.2	Deklaration von Variablen	10
3.3.3	Deklaration von Strukturen	11
3.3.4	Deklaration von Prozessen	12
3.4	Symboltabelle eines Promela-Programms	13
3.5	Beispiel	14
3.6	Sonderfälle	14