An incremental formal semantics for PROMELA

Carsten Weise

Lehrstuhl für Informatik I,
Aachen University of Technology, Germany
email: cweise@acm.org

Abstract. An approach to a formal semantics for PROMELA is presented. The approach uses SOS rules to define a labeled transition system model for a PROMELA program. The approach is a bottom-up, incremental approach with three basic steps (declarations, single processes, parallel processes). PROMELA before version 2.0 is treated nearly entirely. Especially assertions, never claims and correctness conditions are discussed.

1 Introduction

In the paper I present an new approach to a formal semantic for PROMELA. The approach is highly inspired by the approach given in [NH96]. The paper tries to cover all the aspects already present in [NH96], and will treat or at least discuss some more aspects. The presentation of the semantic however differs a lot from [NH96]. The semantic given here is an operational semantic given as a set of SOS rules. SOS stands for Structured Operational Semantics. SOS rules are a standard way to give formal semantics for process algebras, and are useful for any kind of operational semantics. Although this is also a matter of taste and habit, SOS rules are often thought to be more readable than other semantic definitions.

The approach presented here is *incremental*: the definition of the semantics is split into three main parts (or layers): variable and channel declarations, behavior of single processes and behavior of the complete system. These layers build upon each other: each layer defines its concepts in terms of the underlying layers. This leads to a bottom-up definition of the semantics.

The main advantage of such an incremental approach is that problems can be treated in isolation, which in general should make reasoning a lot easier. Hopefully it also makes understanding the semantics more simple. Thus the semantics presented here can be seen as an improvement over the approach presented in [NH96]. However, both semantics have a right on their own: in contrast to [NH96], the semantics given here is more abstract. The semantic of [NH96] is very concrete in how to handle constructs, and thus it should be easy to derive a compiler implementation from it. The incremental semantics presented here is less constructive: some definitions just rely on the existence of concepts instead of defining how to realize these concepts. However this also often helps to abstract away from technical details which sometimes can be more puzzling than explaining. The main strength of the incremental approach is its *compositionality*: many concepts can be defined in terms of simpler concepts. Among these concepts are non-determinism, parallelism, communication and also e.g. the `unless`-operator. Note that these are the concepts where process algebra methods traditionally have their strength.

The semantics given here tries to cover PROMELA before version 2.0 completely. Note that this includes arrays, parameter lists and structured channels (although not structures as types). Especially assert statements and never claims are treated, and correctness issues are discussed. Some aspects of version 2.0 (especially deterministic execution and the `unless` operator) will also be discussed at the end of the paper.

However the syntax has some restrictions. Message types and print statements were stripped as they are only of technical interest. Remote referencing is not allowed. Run-statements and channel enquiries ($x?[e]$) are not allowed within expressions. How to deal with these restrictions is discussed at the end of the paper.

2 Basic Definitions

The semantic model used here are labeled transition systems. A labeled transition system is a directed graph where edges have labels. The labels in the model are essentially the statements which are executed by the model. The nodes of the graph are the states of the system.

Instead of giving a specific labeled transitions system for a specific program, the paper follows the common approach in process algebra: there is only one labeled transition system which consists of all legal transition system. For a given program, this large (infinite!) transition system is interpreted with a start state. In the normal case, only a finite portion of the transition system will be reachable from a start state. Note that this way of modeling helps to get rid of necessity to speak about different systems when defining complex concepts from simpler ones.

The transition system is defined by SOS rules. An SOS rule has the form

$$\frac{\text{assumptions and requirements}}{\text{conclusion}}\ (\text{name})$$

While the conclusion is always a transition of the system, the assumptions and requirements can be quite complex conditions on all components appearing in the transition. To make these parts more readable, abbreviations are introduce via a let -construct. The let -construct is used in mainly two ways: explicit definition of x as in let $x := a + b$ or implicit definition of x as in let $c_1 = c_2 + x$ where c_1, c_2 are known from the context. The let can be conditional. In this case we write let $x := y$ if *condition*.

Our modeling makes use of *finite functions*. A function $f : \mathbb{N} \rightarrow M$ (where $\mathbb{N}$ are the natural numbers and M is some set including the undefined symbol $\perp$) is called finite if there is $n \in \mathbb{N}$ such that $f(i) \neq \perp$ for all $i < n$ and $f(i) = \perp$ for all $i \geq n$. The number n is the length of the finite function, written $|f|$. A special notation is used to "append" an element to a finite function: for $m \in M$ let $f' := f + m$ be the function which is equal to f on all $i \neq n$, and $f'(n) = m$. Finite functions are in fact infinite lists with an append function, but are easier to use in the formalism.

For functions $f : M \rightarrow M'$ we use a notation for replacement of values: $f' := f[a \mapsto b]$ is identical to f except that $f'(a) = b$. We also use a notation to replace the values of several arguments at once, in that case we write $f[a_i \mapsto b_i \mid cond(a_i, b_i)]$ which means the simultaneous setting all $f(a_i)$ to b_i if a_i, b_i satisfy condition $cond(a_i, b_i)$.

For a given set M the set $M_\perp$ is defined as $M \cup \{\perp\}$. If a function f is understood to have type $A \rightarrow B$ where $\perp \in B$, then $f_\perp$ is the function where for all $a \in A$, $f_\perp(a) = \perp$.

The boolean values true and false are written tt and ff.

The normal set theoretic concepts ($\cup, \in$, etc.) are used throughout the paper. If M is a set of sets, i.e. $M = \{M_1, m_2, \ldots\}$, then $\bigcup M$ is the union of these sets, i.e. $\bigcup M = M_1 \cup M_2 \cup \ldots$

3 Modeling the data

This section describes how the data (variables and channels) is modeled within the approach.

3.1 Basic Types, Arrays and Channel Identifier

To model variables we need to describe their type and their value. Types are modeled in a natural way: a type is just the set of all its possible values. Thus the basic PROMELA types `bit`, `byte`, `short`, `int` are defined to be sets $\mathrm{BIT} = \mathbb{B} = \{0, 1, \perp\}, \mathrm{BYTE} = \{0, \dots, 255, \perp\}, \mathrm{SHORT} = \{-2^{15}, \dots, 2^{15} - 1, \perp\}, \mathrm{INT} = \{-2^{31}, \dots, 2^{31} - 1, \perp\}$. As a variable can in principle by undefined, each type includes the value $\perp$ meaning undefined. Further it is useful to speak of variables which have not yet a type (i.e. which are yet undeclared), so we have a special type $\mathrm{UNDEF} = \{\perp\}$ which can take no meaningful value.

The type of a channel variable is just a natural number: this number is not the channel itself, but the *channel id* used to map a channel variable to the real channel carrying the stored information. The next subsection will explain how to model the "real" channels. Here we just define the type of channels to be $\mathrm{CHAN} := \mathbb{N} \cup \{\perp\}$.

The set of basic types is then $\mathbf{BASE} := \{\mathrm{BIT}, \mathrm{BYTE}, \mathrm{SHORT}, \mathrm{INT}, \mathrm{CHAN}, \mathrm{UNDEF}\}$.

Arrays are mappings from an index set (here: $\{0, \dots, n-1\}$) to a basic type. An array `int a[3]` is modeled as a mapping $f : \{0, 1, 2\} \to \mathrm{INT}$. After assignments `a[0] = 3; a[2] = 5; a[1] = -2;` the mapping f representing `a` will have the values $f(0) = 3, f(1) = -2$ and $f(2) = 5$. The type set $\mathbf{ARR}_n$ holds all types of arrays of length n, while $\mathbf{ARR}$ is the set of all array types:

$$\begin{aligned} \mathbf{ARR}_n &:= \{\{0, \dots, n-1\} \to T \mid T \in \mathbf{BASE}\} \\ \mathbf{ARR} &:= \bigcup_{n \in \mathbb{N}} \mathbf{ARR}_n \end{aligned}$$

These are all types a variable can have in our semantics. The set of all types is defined as $\mathbf{T} := \mathbf{BASE} \cup \mathbf{ARR}$. The set of all values a variable can take is defined as $\mathrm{V} := \bigcup \mathbf{BASE} \cup \bigcup \mathbf{ARR}$. Note that the latter is a set of values (which are either integers, or $\perp$, or a mapping from an index set into the integers), while the former is a set of set of values.

To model variables we use a predefined set NAME holding all legal variable names. We use functions $f : \mathrm{NAME} \to \mathbf{T} \times \mathrm{V}$, which assign to each variable `x` a pair $f(\mathtt{x}) = (T, v)$ consisting of the type T and the value v of the variable. A function $f : \mathrm{NAME} \to \mathbf{T} \times \mathrm{V}$ is called an *assignment function*. We will use the abbreviation $\mathsf{Asgn} := \mathrm{NAME} \to \mathbf{T} \times \mathrm{V}$.

As a simple example, assume the following short PROMELA program:

```
int x; int y = 3; byte a[100] = 10; x = 5;
```

After executing this program fragment, the assignment function f will have values $f(\mathtt{x}) = (\mathrm{INT}, 5), f(\mathtt{y}) = (\mathrm{INT}, 3), f(\mathtt{a}) = (\{0, \dots, 99\} \to \mathrm{BYTE}, g)$ where g is a mapping $g : \{0, \dots, 99\} \to \mathrm{BYTE}$ such that $g(i) = 10$ for all $i \in \{0, \dots, 99\}$.

As structures were not part of PROMELA before version 2.0, they are not modeled here. Structures with anonymous entries will however be described in the next subsection in order to describe the contents of channels.

3.2 Channels

A channel can be described by its capacity, the type of its messages and its content. A message is in general not only a base type, but a structure composed from base types. In contrast to structures in programming languages like C++, the elements of these structures are unnamed. Such an anonymous structure can be seen as a tuple $(v_1, \dots, v_n)$ where the elements v_i can have different types. Therefore the structure types $\mathbf{STR}_n, \mathbf{STR}$ are defined as:

$$\begin{aligned} \mathbf{STR}_n &:= \{T_1 \times \dots \times T_n \mid T_i \in \mathbf{BASE}\} \\ \mathbf{STR} &:= \bigcup_{n \in \mathbb{N}} \mathbf{STR}_n \end{aligned}$$

While the definition of a channel syntactically looks very similar to an array declaration, the access to channels differs fundamentally from arrays. Thus the contents of a channel must be described in a way different from arrays. If a channel entry has type $T = T_1 \times \dots \times T_n$, its content will be described by a word w over the alphabet T. Although most readers will be familiar with the concept of words and alphabets, here we repeat the basic definitions:

Let A be an arbitrary set called the *alphabet*. Then a *word* w over the alphabet A is a finite sequence of elements $a_i \in A$ written $w = a_1 \cdot a_2 \cdot \ldots \cdot a_n$. For the given word w, n is called the *length* of w, written $|w|$. There is a unique special word with length zero called the *empty word*, written ε. The set of all words over A of length n is written A^n, and the set of all words over A is written A^* and can be defined by $A^* := \bigcup_{i \in \mathbb{N}} A^i$. Note that $A^0 = \{\varepsilon\}$. Two words v, w can be concatenated via the operator $\circ$.

A channel can be described by a tupel consisting of the the type of entries, the content and the capacity. Let (T, w, k) be such a tuple, then clearly w must be a word over T of length less then or equal to k. Let $\mathrm{VSTR} := \bigcup \mathbf{STR}$ be the set of all values of structure types, then the channel type is the set $ch := \mathbf{STR} \times \mathrm{VSTR}^* \times \mathbb{N}_\perp$ with the side condition $(T, w, k) \in ch$ iff $w \in T^*$ and $|w| \le k$.

The set of all channels of a PROMELA program will be modeled by a mapping $\mathcal{C} : \mathbb{N} \to ch$. In the sequel, we will use $\mathsf{Chan} := \mathbb{N} \to ch$. For a given channel name `x` the value $f(\mathtt{x})$ gives the type of `x`'s entries, its content and its capacity. If `x` is yet uninitialized, then $f(\mathtt{x}) = (\mathrm{UNDEF}, \perp, 0)$. If there is no channel called `x`, then $f(\mathtt{x}) = \perp$.

The execution of the program fragment

```
chan x = [3] of { bit, int }; x!0, 3;
chan y = [1] of { byte }; y!5; x!1,-3;
```

will result in $\mathcal{C}(0) = (\mathrm{BIT} \times \mathrm{INT}, (0,3) \cdot (1,-3), 3)$, $\mathcal{C}(1) = (\mathrm{BYTE}, 5, 1)$ and $\mathcal{C}(i) = \perp$ for $i \notin \{0, 1\}$, assuming that `x` gets channel id 0 and `y` gets channel id 1.

3.3 Evaluating Expressions

For a specification of the behavior of a process it is necessary to define how to evaluate expressions which can be part of a statement (and even a statement by itself). This subsection defines the semantic of an expression. Fig. 1 gives the fragment of the formal syntax of PROMELA for expressions. The non-terminals *const* and *name* are assumed to be given and are not defined explicitly. Further several terms which are treated as expressions by the standard PROMELA syntax will be treated as statements within this paper and are therefore discussed in the section on local control: `timeout`, `run`-commands, and channel enquiries. Further remote referencing and access to structure elements have been dropped completely from the presentation. At the end of the paper, inclusion of all these features in expressions will be discussed.

An important step in the evaluation of expressions is the determination of the current values of variables. A PROMELA process has access to two scopes of variables: its own local variables and the global variables of the program. Local and global variables are modeled by two assignment functions $\mathcal{L}, \mathcal{G} : \mathrm{NAME} \to \mathbf{T} \times \mathrm{V}$. A locally defined variable hides

$$
\begin{array}{lll}
\textit{expr} & ::= & \textit{const} \mid \textit{var\_ref} \mid \texttt{len(}\textit{var\_ref}\texttt{)} \mid \texttt{(}\textit{expr}\texttt{)} \mid \textit{expr binop expr} \mid \textit{unop expr} \\
\textit{var\_ref} & ::= & \textit{name} \mid \textit{name}\texttt{[}\textit{expr}\texttt{]} \\
\textit{binop} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&} \mid \texttt{|} \mid \texttt{>} \mid \texttt{<} \mid \texttt{GE} \mid \texttt{LE} \mid \texttt{EQ} \mid \texttt{NE} \mid \texttt{AND} \mid \texttt{OR} \mid \texttt{LSHIFT} \mid \texttt{RSHIFT} \\
\textit{unop} & ::= & \sim \mid \texttt{-} \mid \texttt{!}
\end{array}
$$

Fig. 1. Expression fragment of the syntax

$$
\begin{array}{ll}
ev_{\mathcal{L},\mathcal{G}}[|\mathtt{n}|] & := n, \mathtt{n} \in \mathsf{CONST} \\
ev_{\mathcal{L},\mathcal{G}}[|e1{+}e2|] & := ev_{\mathcal{L},\mathcal{G}}[|e1|] + ev_{\mathcal{L},\mathcal{G}}[|e2|] \\
\ldots & \\
ev_{\mathcal{L},\mathcal{G}}[|(e)|] & := ev_{\mathcal{L},\mathcal{G}}[|e|] \\
ev_{\mathcal{L},\mathcal{G}}[|\mathtt{x}|] & := v, \text{where } \mathtt{x} \in \text{NAME}, (t,v) = (\mathcal{L},\mathcal{G})(\mathtt{x}) \\
ev_{\mathcal{L},\mathcal{G}}[|\mathtt{x}[e]|] & := f(i), \text{where } \mathtt{x} \in \text{NAME}, (t,f) = (\mathcal{L},\mathcal{G})(\mathtt{x}), t \in \mathbf{ARR}_n, i = ev_{\mathcal{L},\mathcal{G}}[|e|], 0 \leq i < n \\
ev_{\mathcal{L},\mathcal{G}}[|\mathtt{len}(\mathtt{x})|] & := |\alpha|, \text{where } (\text{CHAN}, chid) = ev_{\mathcal{L},\mathcal{G}}[|\mathtt{x}|], (T,\alpha,k) = \mathcal{C}(chid)
\end{array}
$$

Fig. 2. Semantics of an expression

the globally defined symbol. Thus the two assignment functions are coalesced into a new assignment function $(\mathcal{L},\mathcal{G}) : \text{NAME} \rightarrow \mathbf{T} \times \text{V}$ defined by

$$
(\mathcal{L},\mathcal{G})(\mathtt{x}) := \begin{cases} \mathcal{L}(\mathtt{x}) & \text{if } \mathcal{L}(\mathtt{x}) \neq (\text{UNDEF}, \bot) \\ \mathcal{G}(\mathtt{x}) & \text{else} \end{cases}
$$

Using this assignment function, the semantics of expressions is defined by the function $ev_{\mathcal{L},\mathcal{G}}[|.|] : \mathsf{EXPR} \rightarrow \text{V}$ as given in Fig. 2. Here the set EXPR is the set of all words derivable from the non-terminal *expr* of the syntax given in Fig.1. Note that the definition does not give much details for the simple cases of constants and operations, as these are just technical. In the case of operations it should be noted that all operations are required to be bottom preserving, i.e. if at least one argument is undefined, the result will be undefined.

Note that in the following we will not bother about static type checking, i.e. we assume that programs are correct with regard to static type checking. We will however comment an run time type checking where it is neccessary. To describe type checking within the formalism, an operation $type_{\mathcal{L},\mathcal{G}}$ will be used which for a given expressions returns its type. The operation can be defined along the same lines of $ev_{\mathcal{L},\mathcal{G}}[|.|]$, but we do not give details here.

4 Modeling Local Control

In this section the semantics for a single process is given. The syntax of a process' body is given in Fig. 3. This is essentially the syntax as given in [Hol91], but several changes had to be applied for the incremental approach. As already mentioned in the previous section, `timeout`, `run` and the channel enquiry have become statements. Further, for the incremental approach several kinds of statemant have to be distinguished: simple statements, control statements, and labeled statements. Further statements and declarations are treated equivalently instead of seeing a statement having an optional declaration list.

Additonally, the following conventions are used: non-terminals are given in italics, while terminals are given in teletype. Repetition of constructs is not modeled using meta-symbols but using tail recursion, i.e. we write $s ::= a \mid a;s$ instead of $s ::= a\{;a\}^*$. This reduces the use of meta-symbols to $::=$ and $\mid$ and thus should make the syntax more readable. For every

$$
\begin{array}{lll}
\textit{body} & ::= & \{\textit{seq}\} \\
\textit{seq} & ::= & \textit{step} \mid \textit{step}\texttt{;}\,\textit{seq} \\
\textit{step} & ::= & \textit{onedecl} \mid \textit{stmt} \\
\textit{stmt} & ::= & \textit{sstmt} \mid \textit{cstmt} \mid \textit{bstmt} \mid \textit{lstmt} \\
\textit{sstmt} & ::= & \textit{var\_ref}\,\texttt{:=}\ \textit{expr} \mid \textit{expr} \mid \textit{var\_ref}\texttt{?}\ \textit{margs} \mid \textit{var\_ref}\texttt{!}\ \textit{margs} \mid \texttt{assert}\ \textit{expr} \mid \texttt{timeout} \\
 & & \mid \texttt{run}\ \textit{name}\texttt{()}\ \mid \texttt{run}\ \textit{name}\texttt{(}\textit{arglist}\texttt{)} \\
\textit{cstmt} & ::= & \texttt{if}\ \textit{options}\ \texttt{fi}\ \mid \texttt{do}\ \textit{options}\ \texttt{od}\ \mid \texttt{break} \\
\textit{options} & ::= & \texttt{::}\ \textit{seq} \mid \texttt{::}\ \textit{seq}\ \textit{options} \\
\textit{arglist} & ::= & \textit{expr} \mid \textit{expr}\texttt{,}\ \textit{arglist} \\
\textit{margs} & ::= & \textit{arglist} \mid \textit{expr}\texttt{(}\textit{arglist}\texttt{)} \\
\textit{bstmt} & ::= & \texttt{atomic}\ \{\textit{seq}\} \\
\textit{lstmt} & ::= & \texttt{goto}\ \textit{name} \mid \textit{name}\texttt{:}\,\textit{stmt} \\
\textit{onedecl} & ::= & \textit{btype}\ \textit{ivarlist} \mid \texttt{chan}\ \textit{vardecl}\texttt{=}\ \textit{chinit} \\
\textit{btype} & ::= & \texttt{bit}\ \mid \texttt{int}\ \mid \ldots \\
\textit{ivarlist} & ::= & \textit{ivar} \mid \textit{ivar}\texttt{,}\ \textit{ivarlist} \\
\textit{ivar} & ::= & \textit{vardecl} \mid \textit{vardecl}\texttt{=}\ \textit{expr} \\
\textit{vardecl} & ::= & \textit{name} \mid \textit{name}\texttt{[}\textit{const}\texttt{]} \\
\textit{chinit} & ::= & \texttt{[}\textit{const}\texttt{]}\ \texttt{of}\ \{\textit{typelist}\} \\
\textit{typelist} & ::= & \textit{btype} \mid \textit{btype}\texttt{,}\ \textit{typelist}
\end{array}
$$

Fig. 3. Syntax of process bodies

non-terminal we assume a set of all words derivable from the non-terminal. If the name of the non-terminal is e.g. *seq*, then the name of the word set is SEQ.

This section concentrates on those parts of the semantic which are local to a specific process. A process will be modeled as a labeled transitions system. The labels will be the executed statements. The philosophy of labeled transition systems is that they describe an operational semantics where the labels are the observable behavior of the system. Under this philosophy, in many cases the labels are superfluous, as they describe internal behavior only. However, there are cases where the behavior of the process is connected with the global state of the system. In these cases the observable behavior (i.e. the labels) is used to infer the global behavior of the system. In the section on global control it can be seen how this is done.

4.1 Representation of Processes

A state of a process consists of the following: the (yet unexecuted) program body, the local variables, the global variables, the channels and a continuation stack. It should be obvius that the first four components are needed to describe the process behavior. The continuation stack will be explained in detail in the section on control statements and is essentially used with `break`. Local and global variables as well as channels are modeled as detailed in the previous section. The program is a word over the set STAT of all statements, while the continuation stack is a word over the set SEQ of all statement sequences. Summarizing, the type of a state of a process' labeled transition system is $\mathcal{S} := \mathsf{STAT}^* \times \mathsf{Asgn} \times \mathsf{Asgn} \times \mathsf{Chan} \times \mathsf{SEQ}^*$. A transition of the labeled transition system is a triple from $\mathcal{S} \times \mathsf{STAT} \times \mathcal{S}$. Let $\beta = \{\pi\}$ be the body of a processtype. Then the semantics of β are defined as a structure $(\mathcal{S}, S_0, \mathsf{STAT}, \rightarrow)$, where $\mathcal{S}$ is the set of states (as defined above), STAT is the set of labels, $\rightarrow \subseteq \mathcal{S} \times \mathsf{STAT} \times \mathcal{S}$ is the transition relation as defined in the followoing the subsections, and $S_0 \in \mathcal{S}$ is the start state. The start state depends on a given assignment function $\mathcal{G}$ modeling the global declarations and a channel function $\mathcal{C}$ defining the global channels. Then the start state is

$S_0 = (\pi, \mathcal{L}_\perp, \mathcal{G}, \mathcal{C}, \varepsilon)$, i.e. the state where the process is at the beginning of its body sequence π, all local variables are undefined, and the continuation stack is empty.

4.2 Execution of Simple Statements

This subsection gives the SOS-rules for simple statements, i.e. the process is in a state $S = (\pi = \mathit{stat} \cdot \pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)$ and $\mathit{stat} \in \mathsf{SSTMT}$. The rules are given one after the after with a short discussion.

Assignment: An assignment $x{:=}e$ evaluates the expression e and assigns the result to the variable x, which can either be a plain variable or an array access. The two cases will be modeled by two similar rules. The rule is straightforward: the expression is evaluated using $ev_{\mathcal{L},\mathcal{G}}[|.|]$, where $\mathcal{L}, \mathcal{G}$ are the current local and global assignment functions, and thereafter $\mathcal{L}, \mathcal{G}$ are updated according to the assignment. Note that $(\mathcal{L}', \mathcal{G}') = (\mathcal{L}, \mathcal{G})[x \mapsto v]$ is an abuse of notation. It is intended to mean that if x is defined in $\mathcal{L}$, then $\mathcal{L}'$ is $\mathcal{L}$ with the value of x replaced by v and $\mathcal{G}' = \mathcal{G}$, and if x is undefined in $\mathcal{L}$, then $\mathcal{L}' = \mathcal{L}$ and $\mathcal{G}'$ is $\mathcal{G}$ with the value of x replaced by v.

$$\frac{x \in \mathrm{NAME}, \mathrm{let}\ v := ev_{\mathcal{L},\mathcal{G}}[|e|], (\mathcal{L}', \mathcal{G}') := (\mathcal{L}, \mathcal{G})[x \mapsto v]}{(x{:=}e;\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{x:=e} (\pi', \mathcal{L}', \mathcal{G}', \mathcal{C}, \gamma)}\ (\mathrm{VASGN})$$

In case of an array access $x[e_1] :=e_2$, the statement is legal if e_1 is a valid index of the array x. This must be checked at run time and therefore is included in the assumptions here. As a run time error can occur, this must be modeled accordingly by an additional rule. Note that the rule does not check if x is an array, as this is already clear from static type checking. As x is an array, the outcome of $ev[|.|]$ is an element of $\mathbf{ARR}$.

$$\frac{\begin{array}{l} x \in \mathrm{NAME}, \mathrm{let}\ f := ev_{\mathcal{L},\mathcal{G}}[|x|], f \in \mathbf{ARR}_n \\ \mathrm{let}\ i := ev_{\mathcal{L},\mathcal{G}}[|e_1|], 0 \le i < n \\ \mathrm{let}\ v := ev_{\mathcal{L},\mathcal{G}}[|e_1|], \mathrm{let}\ f' := f[i \mapsto v] \\ \mathrm{let}\ (\mathcal{L}', \mathcal{G}') := (\mathcal{L}, \mathcal{G})[x \mapsto f'] \end{array}}{(x[e_1] :=e_2;\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{x[e_1]:=e_2} (\pi', \mathcal{L}', \mathcal{G}', \mathcal{C}, \gamma)}\ (\mathrm{AASGN1})$$

$$\frac{\begin{array}{l} x \in \mathrm{NAME}, \mathrm{let}\ f := ev_{\mathcal{L},\mathcal{G}}[|x|], f \in \mathbf{ARR}_n \\ \mathrm{let}\ i := ev_{\mathcal{L},\mathcal{G}}[|e_1|], i < 0 \vee i \ge n \end{array}}{(x[e_1] :=e_2;\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\texttt{run time error}} (\texttt{error}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)}\ (\mathrm{AASGN2})$$

Expression: Executing an expression has no effect on the state besides going to the next statement. It is well known that an expression can only be executed if it is non-zero.

$$\frac{e \in \mathsf{EXPR}, ev_{\mathcal{L},\mathcal{G}}[|e|] \neq 0}{(e \cdot \pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{e} (\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)}\ (\mathrm{EXPR})$$

Receive: A receive-statement reads a set of values from a channel. A channel can only be read if it is initialized (i.e. has a type) and is not empty. Further type compatabilty of the variables taking the values and the data of the channel must be checked. Initialization of the channel and type compatibility must be checked at run time, so it is included in the requirements of the rule. Further it is possible to use constants within receive-statement: these constants must match the values of the head entry of the channel in order to make the statement executable. Type compatibility and matching of constants can easily be checked with one function $compat : \mathsf{EXPR} \times \mathrm{V} \rightarrow \mathbb{B}$. This function takes as argument a PROMELA-expression and a value. The expression will be the i-th argument of the receive-statement, and the value will be the i-th value in the head of the channel. If the expression is a variable, $compat_{\mathcal{L},\mathcal{G}}$ checks if the value has the same type as the variable. If the expression is a constant, then the constant is compared to the value. The function can formally be defined as:

$$compat_{\mathcal{L},\mathcal{G}}(e,v) := \begin{cases} v \in type_{\mathcal{L},\mathcal{G}}(e) & e \in \mathsf{VAR\_REF} \\ ev_{\mathcal{L},\mathcal{G}}[|e|] == v & e \in \mathsf{CONST} \\ \mathsf{ff} & \text{else} \end{cases}$$

This leads to the following rule:

$$\frac{\begin{array}{c} ev_{\mathcal{L},\mathcal{G}}[|x|] \neq \perp, \text{let } id := ev_{\mathcal{L},\mathcal{G}}[|x|], \text{let } (T,\alpha,k) := \mathcal{C}(id) \\ |\alpha| > 0, \text{let } \alpha = (v_1,\ldots,v_r)\cdot\alpha' : \forall i \in \{1,\ldots,\min(r,s)\}.compat_{\mathcal{L},\mathcal{G}}(e_i,v_i) == \mathtt{tt} \\ \text{let } (\mathcal{L}',\mathcal{G}') := (\mathcal{L},\mathcal{G})[e_i \mapsto v_i \mid e_i \in \mathsf{VAR\_REF}], \text{let } \mathcal{C}' := \mathcal{C}[id \mapsto (T,\alpha',k)] \end{array}}{(x?e_1,\ldots,e_s;\pi',\mathcal{L},\mathcal{G},\mathcal{C},\gamma) \xrightarrow{x?e_1,\ldots,e_s} (\pi',\mathcal{L}',\mathcal{G}',\mathcal{C}',\gamma)} \text{(RCV)}$$

There are two problems with the receive-statement: the syntax of PROMELA allows arbitrary expressions e_i in $x?e_1,\ldots,e_n$, while the compiler will only accept constants or variables (as clear from the informal description of the command). It is possible to give a semantics to the more general case, but maybe it would be even better to specify explicitly if one wants a match is desired. This would lead to a construct `x?a,match(b),c` meaning that the receive may be carried out if the second element of the head of the channel has the same value as `b` (even if `b` is a variable).

The other problems is what happens if a variable occurs several times within the e_i. Due to the definition of simultaneous replacement the above rule can have several outcomes. SPIN seems to assign variables starting from e_1 and ending at e_n.

Send: A send-statement appends a set of values to a channel if the values are compatible with the type of the channel, and if the channel is already initialized. As with the receive command, compatibility must be checked at run time. If a channel entry has more elements than expressions are given in the send, the rest is filled with zeros by our rule. This seems to be the same with SPIN, while the reference manual claims the values will be undefined.

$$\frac{\begin{array}{c} ev_{\mathcal{L},\mathcal{G}}[|x|] \neq \perp, \text{let } id := ev_{\mathcal{L},\mathcal{G}}[|x|], \text{let } (T,\alpha,k) := \mathcal{C}(id) \\ |\alpha| < k, \text{let } T = T_1 \times \ldots \times T_r, \forall i \in \{1,\ldots,\min(r,s)\} : type_{\mathcal{L},\mathcal{G}}(e_i) = T_i \\ \text{let } \alpha' := \alpha\cdot(e_1,\ldots,e_r) \text{ if } r \leq s, \text{let } \alpha' := \alpha\cdot(e_1,\ldots,e_s,0,\ldots,0) \text{ if } r > s \\ \text{let } \mathcal{C}' := \mathcal{C}[id \mapsto (T,\alpha',k)] \end{array}}{(x!e_1,\ldots,e_s;\pi',\mathcal{L},\mathcal{G},\mathcal{C},\gamma) \xrightarrow{x!e_1,\ldots,e_s} (\pi',\mathcal{L},\mathcal{G},\mathcal{C}',\gamma)} \text{(SND)}$$

Note that with the above rules a single process can never use a channel with capacity zero, as the requirements $|\alpha| > 0$ resp. $|\alpha| < k = 0$ can never be true for the content of such a channel. This is exactly the intention here: a channel of capacity zero can only be used for hand-shake communication, and this can only occur when at least two processes are present. Within a process, hand-shake communication has no meaning. Hand-shake communication will be modeled in the section on global control.

The rules for the receive- and send-statements of the form $x?e(e_1, \ldots, e_s)$ and of the form $x!e(e_1, \ldots, e_s)$ are straight forward derived from the above rules and are not given explicitly here.

Channel Enquiry: A channel enquiry $x?[e_1, \ldots, e_s]$ can easily be modeled using the read operation: the enquiry is executable if the appropriate read-operation is executable, but the enquiry does not have any side effects:

$$\frac{(x?e_1, \ldots, e_s;\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{x?e_1,\ldots,e_s} (\pi', \mathcal{L}', \mathcal{G}', \mathcal{C}', \gamma)}{(x?[e_1, \ldots, e_s];\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{x?[e_1,\ldots,e_s]} (\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)} \text{(ENQ)}$$

assertions: Executing an assertion has no effect at all as long as the expression is non-zero. If the expression evaluates to zero, an `assertion violated` is issued by the process. Note that as remote referencing is not modeled here, the value of an assertion can be evaluated by looking at the executing process in isolation.

$$\frac{ev_{\mathcal{L},\mathcal{G}}[|e|] \neq 0}{(\texttt{assert } e;\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\texttt{assert } e} (\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)} \text{(ASSERT1)}$$

$$\frac{ev_{\mathcal{L},\mathcal{G}}[|e|] == 0}{(\texttt{assert } e;\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\texttt{assertion violation}} (\texttt{error}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)} \text{(ASSERT2)}$$

timeout- and run-commands: Neither a timeout nor a run command has a meaning within a single process. We will model them here by allowing them under all circumstances. The modeling of the global control will take care of the desired effect.

$$\frac{}{(\texttt{timeout};\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\texttt{timeout}} (\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)} \text{(TIMEOUT)}$$

$$\frac{}{(\texttt{run } x(e_1, \ldots, e_n);\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\texttt{run } x(e_1,\ldots,e_n)} (\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)} \text{(RUN)}$$

atomic execution: Similar observations hold for the case of atomic statement blocks. The modeling is a bit different yet as these are parenthesized blocks. As these blocks may not be nested, then can however quite easily be modeled using explicit end-statements for them. Further an atomic sequence is only executable if the sequence itself is executable:

$$\frac{(\pi;\pi_1, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{stat} (\pi';\pi_1, \mathcal{L}', \mathcal{G}', \mathcal{C}', \gamma')}{(\texttt{atomic}\{\pi\}\pi_1, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\texttt{atomic}} (\pi; \texttt{end\_atomic};\pi_1, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)} \text{(ATOM)}$$

Note that executing an `atomic`-transition explicitly does not disturb the executability of π. Additionally we also need a rule for the end-statements of the block:

$$\frac{}{(\texttt{end\_atomic};\pi, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\texttt{end\_atomic}} (\pi, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)} \text{(EATOM)}$$

4.3 Modeling Control Structures

This subsection explains how to model the control structures (i.e. branching and looping). This will be done in fairly standard way which however might not be very well known outside the field of formal semantics.

Branching: The `if`-construct of PROMELA is essentially a non-deterministic, guarded choice. In process algebras, choice between several alternatives can easily be described by going from the properties of the alternatives to the properties of the choice statement. Such an approach is often called compositional: the meaning of the choice operator is composed from the meaning of the operands. In a guarded choice, the branch π_i can be taken if it would have been possible to execute π_i by itself. This leads to the following rule:

$$\frac{(\pi_i \circ \pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{stat} (\pi_i' \circ \pi', \mathcal{L}', \mathcal{G}', \mathcal{C}', \gamma')}{(\texttt{if} :: \pi_1 :: \ldots :: \pi_n \texttt{fi} \circ \pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{stat} (\pi_i' \circ \pi', \mathcal{L}', \mathcal{G}', \mathcal{C}', \gamma')} \text{(IF)}$$

Note that this is in fact a rule for every possible $i \in \{1, \ldots, n\}$. Intuitively, the rule says: if component π_i can make a step in the current state all by itself, then in the non-deterministic choice this component can be chosen to be executed, and after the first step all other components of the choice are discarded.

Looping: Executing a loop can be seen as constantly unraveling its body. In PROMELA, a loop can only be left by executing a `break`. A `break` jumps behind the end of innermost loop. The unraveling of a loop `do forever` α can be easily modeled by replacing the loop with α; `do forever`α. This will happen again and again when the loop-statement is encountered. To be able to leave the loop and jump behind its end, additionally the program fragment which comes after the loop must be stored somewhere and will be activated when leaving the loop. Such program fragments are called *continuations*.

As loops can be nested, several continuations must be stored. This is best done on a stack. A stack of a type T can easily be modeled by words over T. This is what the continuation stack γ is used for: it stores the continuations whenever a loop is entered. Note that the elements of γ are words over program fragments, while the program fragment π is composed of statements. While unraveling a loop pushes a continuation onto γ, a `break` will pop the last continuation from γ and use it as the new program fragment. Note that unraveling a loop is an internal move of the process but does not execute any statement. Therefore unraveling is labeled with the silent action τ. This leads to the following rules:

$$\frac{}{(\texttt{do}\,\pi\,\texttt{od}; \pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\tau} (\texttt{if}\,\pi\,\texttt{fi}; \texttt{do}\,\pi\,\texttt{od}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \pi' \cdot \gamma)} \text{(DO)}$$

$$\frac{|\gamma| > 0, \text{let } \gamma = \pi \cdot \gamma'}{(\texttt{break}; \pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\texttt{break}} (\pi, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma')} \text{(BREAK1)}$$

$$\frac{\gamma == \varepsilon}{(\texttt{break}; \pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\texttt{illegal break}} (\texttt{error}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)} \text{(BREAK2)}$$

Note that a break occurring with an empty continuation stack is a run time error: the break must have been outside a loop. Note further that when unraveling a `do`-loop, every iteration will write a continuation to the stack. However, at all times but the first this is always the empty word, so for every loop its continuation is only written once to the stack.

4.4 Handling Declarations

Until now, we have seen how to evaluate expressions and how to execute statements. In both cases, assignment and channel functions have been used. But we have yet to see how these assignment and channel functions are constructed from the declarations in the program. It would be possible to introduce a separate mechanism to construct the appropriate functions. Here however we will see declarations as executable statements: a declaration changes the assignment or channel functions so that the declared variable is mapped to the correct type and value.

Variable Declarations: The basic case of a variable declaration is an expression $t\ x{=}e$, where t is a base type, x is a name and e is an expression with which the variable will be initialized. The other basic case is $t\ x[c]\ {=}e$, which is a declaration of an array (with size c). Both cases are modeled straightforwardly giving rules which extend the assignment function $\mathcal{L}$ by the newly introduce variables.

$$\frac{\text{let } v := ev_{\mathcal{L},\mathcal{G}}[|e|], \text{let } \mathcal{L}' := \mathcal{L}[x \mapsto (t, v)]}{(t\ x{=}e;\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{t\ x=e} (\pi', \mathcal{L}', \mathcal{G}, \mathcal{C}, \gamma)}\ \text{(VARD)}$$

$$\frac{\begin{array}{l}\text{let } v := ev_{\mathcal{L},\mathcal{G}}[|e|], \text{let } n := ev_{\mathcal{L},\mathcal{G}}[|c|], \text{let } I := \{0, \ldots, n-1\} \\ \text{let } f : I \to t : \forall i \in I. f(i) = v, \text{let } \mathcal{L}' := \mathcal{L}[x \mapsto (I \to t, f)]\end{array}}{(t\ x[c]\ {=}e;\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{t\ x[c]=e} (\pi', \mathcal{L}', \mathcal{G}, \mathcal{C}, \gamma)}\ \text{(ARRD)}$$

If no explicit initialization is given, variables are initialized to zero. Thus these cases are just special cases of the above:

$$\frac{(t\ x{=}\ \mathtt{0};\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{t\ x=\mathtt{0}} (\pi', \mathcal{L}', \mathcal{G}, \mathcal{C}, \gamma)}{(t\ x;\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{t\ x} (\pi', \mathcal{L}', \mathcal{G}, \mathcal{C}, \gamma)}\ \text{(UVARD)}$$

$$\frac{(t\ x[c] = \mathtt{0};\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{t\ x[c]=\mathtt{0}} (\pi', \mathcal{L}', \mathcal{G}, \mathcal{C}, \gamma)}{(t\ x[c];\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{t\ x[c]} (\pi', \mathcal{L}', \mathcal{G}, \mathcal{C}, \gamma)}\ \text{(UARRD)}$$

The case of multiple declarations $t\ iv_1, \ldots, iv_n$ can be derived as repetitive single declarations:

$$\frac{\begin{array}{l}\forall i \in \{1, \ldots, n\}. \\ (t\ iv_i; \ldots; t\ iv_n;\pi', \mathcal{L}_i, \mathcal{G}_i, \mathcal{C}_i, \gamma_i) \xrightarrow{t\ iv_i} (t\ iv_{i+1}; \ldots; t\ iv_n;\pi', \mathcal{L}_{i+1}, \mathcal{G}_{i+1}, \mathcal{C}_{i+1}, \gamma_{i+1})\end{array}}{(t\ iv_1, \ldots, iv_n;\pi', \mathcal{L}_1, \mathcal{G}_1, \mathcal{C}_1, \gamma_1) \xrightarrow{t\ iv_1, \ldots, iv_n} (\pi', \mathcal{L}_{n+1}, \mathcal{G}_{n+1}, \mathcal{C}_{n+1}, \gamma_{n+1})}\ \text{(MULD)}$$

Channel Declarations: Channel declarations are quite different. While the above declarations added information to the local assignment function, a channel declaration will in general add a new channel to the channel function $\mathcal{C}$. In the simple case of an uninitialized channel, the channel variable is however just set to $\perp$:

$$\frac{\text{let } \mathcal{L}' := \mathcal{L}[x \mapsto (\text{CHAN}, \perp)]}{(\mathtt{chan}\ x;\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\mathtt{chan}\ x} (\pi', \mathcal{L}', \mathcal{G}, \mathcal{C}, \gamma)}\ \text{(UCHAND)}$$

The case of an initialized channel has the generic form chan x= $[c]$of$\{t_1, \ldots, t_n\}$, where x is the channel name, c is the capacity of the channel and t_i are the types of the elements of the channel entries. Declaring an array of channels takes the form chan $x[c_1] = [c_2]$of$\{t_1, \ldots, t_n\}$. In each case, new channels are created by adding them to the function $\mathcal{C}$ and assigning the new channel ids to x or the $x[i]$:

$$\frac{\begin{array}{l}\text{let } k := ev_{\mathcal{L},\mathcal{G}}[|c|], \text{let } chid := |\mathcal{C}|, \text{let } \mathcal{C}' := \mathcal{C} + (t_1 \times \ldots \times t_n, \varepsilon, k),\\ \text{let } \mathcal{L}' := \mathcal{L}[x \mapsto (CHAN, chid)]\end{array}}{(\mathtt{chan}\ x\text{=}\ [c]\mathtt{of}\{t_1, \ldots, t_n\};\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\mathtt{chan}\ x=[c]\mathtt{of}\{t_1,\ldots,t_n\}} (\pi', \mathcal{L}', \mathcal{G}, \mathcal{C}, \gamma)} \text{(CHAND)}$$

The case of a channel array is just a combination of the cases of an array and a channel declaration. This leads to a complicated yet technical rule which is left out here.

4.5 Special Case: Labels and Goto

The goto command, while clearly useful especially within the field of modeling finite automata, is also a very harmful command. Not surprisingly, it does not fit very well into a compositional approach to formal semantics. This subsection discusses a simple way to extend the model so far with the possibility of labels and gotos.

First of all, it is a good idea to make labels of statements visible. A labeling of a transition will consist of a label and a statement from now on. All rules so far did not treat labeled commands. In the following a statement without label is treated as if it has the empty word as a label, i.e. what has been a transition $S \xrightarrow{stat} S'$ so far is now a transition $S \xrightarrow{\varepsilon, stat} S'$. If a statement has a label, the label is visible in the transition:

$$\frac{(stat;\pi, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\varepsilon, stat} (\pi, \mathcal{L}', \mathcal{G}', \mathcal{C}', \gamma')}{(\ell{:}stat;\pi, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\ell, stat} (\pi, \mathcal{L}', \mathcal{G}', \mathcal{C}', \gamma')} \text{(LSTAT)}$$

We will discuss some benefits of this extension in the section on global control. Until now, only the yet unexecuted program fragment of a process was stored in a state. As gotos can jump back in the process, the state must be extended to store information of the complete program. This is can be done by simply adding an element to the state which represents the complete body of the process, so that a state now has the form $(\pi, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma, \beta)$ where β is the process body. This element does never change during the execution of the process, so that the already defined rules can easily be extended from $(\pi, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\ell, stat} (\pi', \mathcal{L}', G', \mathcal{C}', \gamma')$ to $(\pi, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma, \beta) \xrightarrow{\ell, stat} (\pi', \mathcal{L}', G', \mathcal{C}', \gamma', \beta)$.

For the execution of a jump, one needs to identify the location within the process body where the jump goes to. We introduce a function $find : \mathsf{STAT}^* \times \mathsf{NAME} \to \mathsf{STAT}^*$. Then $find(\beta, \ell)$ finds the suffix β' of β which starts with the label ℓ, i.e. $\beta' = \ell{:}stat;\beta''$ for some β''. There is another twist with goto's: a jump can be an implicit break, if the process is currently within a do-loop. This is the case if the continuation-stack is not empty. In that case, the top of the continuation stack must be deleted. This leads to the following rule:

$$\frac{\text{let } \beta' := find(\beta, \ell), \text{let } \gamma = \pi' \cdot \gamma' \text{ if } \gamma \neq \varepsilon, \text{let } \gamma' := \gamma \text{ if } \gamma = \varepsilon}{(\mathtt{goto}\ \ell;\pi, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma, \beta) \xrightarrow{\varepsilon, \mathtt{goto}\ \ell} (\beta', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma', \beta)} \text{(GOTO)}$$

$program ::= unit \mid unit\ program$
$unit \quad ::=$ `proctype` $name()\ body \mid$ `proctype` $name(decl\_lst)\ body$
$\mid$ `never` $body \mid$ `init` $body \mid onedecl \mid$ `;`
$decl\_lst\ ::= btype\ name \mid btype\ name;\ decl\_lst$

Fig. 4. Syntax of PROMELA programs

5 Modeling Global Control

The global control is the semantics of a complete PROMELA program. The syntax of a PROMELA program is given in Fig. 4[1].

Global variables and channels are treated the same on the global and the local level: the only difference is that while a local declaration affected the local assignment function and possibly the channel function, a global declaration while change the global assignment function and possibly the channel function. Thus the rules for global variable and channel declarations are not given explicitly here.

Similar to the local control states, a global control states consists of a program fragment (from $\mathsf{PROGRAM}$), a global assignment function and a channel function. In addition, the global control needs to store information on the process types created, the instantiated processes and which processes are currently executing atomic or deterministic blocks.

Process Types: Each `proctype`-declaration defines a process type. A process type consists of the process body (in STAT^*) and a parameter function $p : \{1, \ldots, n\} \to \mathsf{NAME} \times \mathbf{T}$ which represents the name and the type of the i-th parameter. Let $\mathcal{P} := \{\{1, \ldots, n\} \to \mathsf{NAME} \times \mathbf{T} | n \in \mathbb{N}\}$, then the type of a process is $\mathsf{STAT}^* \times \mathcal{P} \cup \{\bot\}$, and a process assignment is a function $\mathsf{PA} := \mathsf{NAME} \to \mathsf{STAT}^* \times \mathcal{P} \cup \{\bot\}$ assigning a process type to each name (and bottom if the name is not a process type).

Process Instantiations: A process instantiation is represented by a tuple $(\pi, \beta, \mathcal{L}, \gamma)$, where π is the current program fragment, β is the body of the process, $\mathcal{L}$ is the current setting local assignment function and γ is the current continuation stack. The type of such an instantiation is $\mathsf{I} := \mathsf{STAT}^* \times \mathsf{STAT}^* \times \mathsf{Asgn} \times \mathsf{SEQ}^*$. The set of all active processes is modeled by a (finite) function $\mathsf{act} : \mathbb{N} \to \mathsf{I}$. The index i of an active process can be seen as its process identifier, although process ids are not further discussed within this paper.

Tracking atomic behavior: A global state has an additional entry at of type $\mathbb{N}_\bot$. If a process is executing atomically, then at will be set to the index of the process. If no process is executing atomically, at will be $\bot$.

The global state: A global state is a tupel $(\pi, \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at})$, where π is the (yet unexecuted) program fragment of the PROMELA program, $\mathcal{G}$ is the global assignment function, $\mathcal{C}$ is the channel function, pdef keeps the process definitions, act represents the active processes, and at and det are as above. The type of a global state is $\mathsf{UNIT}^* \times \mathsf{Asgn} \times \mathsf{Chan} \times \mathsf{PA} \times (\mathbb{N} \to$

[1] The syntax of *decl_lst* is different from what is given in [Hol91]: there seems to be an error in that the original syntax allows initialized channels and variables as parameters. SPIN will accept such parameters, but also requires explicit argument passing, so that any initialization will be overwritten when the process is created. Note that the original syntax also allows arrays as parameters, while SPIN will complain.

$\mathsf{I}) \times \mathbb{N}_\bot$. If $\mathsf{act}(i) = (\pi, \beta, \mathcal{L}, \gamma)$, then $(\pi, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma, \beta)$ is the local state of the process with identifier i.'

The start state of the global control is the state $(\pi, \mathcal{G}_\bot, \mathcal{C}_\bot, \mathsf{pdef}_\bot, \mathsf{act}_\bot, \bot)$, where π is the complete PROMELA program, global variables, channels, process types and active processes are all undefined and the atomic-execution-indicator is set to $\bot$.

5.1 Declarations in the global control

As already mentioned, the rules for channel and variable declarations are omitted. Note that the global part of a PROMELA consists of declarations only, which fall into the following categories: variable declarations, channel declarations, proctype declarations, init declaration, never claim declaration. The last three are all proctype declarations, but with a special meaning in the last two cases.

The simple case of a proctype declaration without formal parameters has the following rule, where f is the unique function into $\mathsf{NAME} \times \mathbf{T}$ taking no arguments:

$$\frac{\text{let } f : \{\} \to \mathsf{NAME} \times \mathbf{T}, \text{let } \mathsf{pdef}' := \mathsf{pdef}[x \mapsto (\pi, f)]}{\begin{array}{l}(\mathtt{proctype}\ x()\{\pi\} \circ \pi', \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \\ \xrightarrow{\mathtt{proctype}\ x()\{\pi\}} (\pi', \mathcal{G}, \mathcal{C}, \mathsf{pdef}', \mathsf{act}, \mathsf{at})\end{array}} \text{(PTYPE1)}$$

In the more complicated case of parameters, the function f needs to be defined in more detail:

$$\frac{\begin{array}{l}\text{let } f : \{1, \dots, n\} \to \mathsf{NAME} \times \mathbf{T}, \forall i \in \{1, \dots, n\}. f(i) = (x_i, t_i), \\ \text{let } \mathsf{pdef}' := \mathsf{pdef}[x \mapsto (\pi, f)]\end{array}}{\begin{array}{l}(\mathtt{proctype}\ x(t_1\ x_1, \dots, t_n\ x_n)\{\pi\} \circ \pi', \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \\ \xrightarrow{\mathtt{proctype}\ x(t_1\ x_1, \dots, t_n\ x_n)\{\pi\}} (\pi', \mathcal{G}, \mathcal{C}, \mathsf{pdef}', \mathsf{act}, \mathsf{at})\end{array}} \text{(PTYPE2)}$$

The case of the init process and the never claim are similar to rule PTYPE1, as these processes take no arguments. Additionally the init process and the never claim are always instantiated as process with index 0 and 1 resp.:

$$\frac{\begin{array}{l}\text{let } f : \{\} \to \mathsf{NAME} \times \mathbf{T}, \text{let } \mathsf{pdef}' := \mathsf{pdef}[\mathtt{init} \mapsto (\pi, f)], \\ \text{let } \mathsf{act}' := \mathsf{act}[0 \mapsto (\pi, \pi, \mathcal{L}_\bot, \varepsilon)]\end{array}}{\begin{array}{l}(\mathtt{init}\{\pi\} \circ \pi', \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \\ \xrightarrow{\mathtt{init}\{\pi\}} (\pi', \mathcal{G}, \mathcal{C}, \mathsf{pdef}', \mathsf{act}', \mathsf{at})\end{array}} \text{(INIT)}$$

$$\frac{\begin{array}{l}\text{let } f : \{\} \to \mathsf{NAME} \times \mathbf{T}, \text{let } \mathsf{pdef}' := \mathsf{pdef}[\mathtt{never} \mapsto (\pi, f)], \\ \text{let } \mathsf{act}' := \mathsf{act}[1 \mapsto (\pi, \pi, \mathcal{L}_\bot, \varepsilon)]\end{array}}{\begin{array}{l}(\mathtt{never}\{\pi\} \circ \pi', \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \\ \xrightarrow{\mathtt{never}\{\pi\}} (\pi', \mathcal{G}, \mathcal{C}, \mathsf{pdef}', \mathsf{act}', \mathsf{at})\end{array}} \text{(NEVER)}$$

5.2 Modeling execution of processes

The rules presented so far for the global control only set up the definitions of processes, variables and channels. Execution of processes starts once all these declarations are read in, which means that the program fragment has become the empty word. Thus all of the following rules will have ε as the first item of a state.

As outlined in the previous subsection, the init process and the never claim are always assigned the indexes 0 and 1. From now on we will assume that both processes exist. Note that a missing init process would be a run time error. If the never claim is missing, the process with index 1 should simply be process with an initial state and no transitions at all.

The execution of the global control will be modeled in two steps. The following rule simply expresses the fact that in general a step of the global control is a non-deterministic choice of steps from the active process. The derived transition relation is however not the one we are aiming at, therefore it is written $\rightarrow_P$. Further down the correct semantics of the global control will be derived using $\rightarrow_P$. The following rule expresses that a *stat*-step is possible in the global state if there is an active process which is capable of the *stat*-step, and that all occurring changes are due to the changes caused by this process. Further the index of the active process is visible in the rule:

$$\frac{\begin{array}{l}\mathsf{act}(i) \neq \perp, \text{let } \mathsf{act}(i) = (\pi, \beta, \mathcal{L}, \gamma), (\pi, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma, \beta) \xrightarrow{stat} (\pi', L', \mathcal{G}', \mathcal{C}', \gamma', \beta),\\ \text{let } \mathsf{act}' := \mathsf{act}[i \mapsto (\pi', \beta, \mathcal{L}', \gamma')]\end{array}}{(\varepsilon, \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \xrightarrow{stat,i}_P (\varepsilon, \mathcal{G}', \mathcal{C}', \mathsf{pdef}, \mathsf{act}', \mathsf{at})} \text{(SPEC)}$$

The above rule can be directly used as semantic of the global control if neither process is executing atomically, and if the executed statement is neither a `run`-command, nor a `timeout`, `atomic`, `end_atomic`. This is reflected by the following rule:

$$\frac{\begin{array}{l}\mathsf{at} = \perp, stat \notin \{\mathtt{run}\ x(\dots), \mathtt{timeout}, \mathtt{atomic}, \mathtt{end\_atmoic}\}\\ (\varepsilon, \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \xrightarrow{stat}_P (\varepsilon, \mathcal{G}', \mathcal{C}', \mathsf{pdef}, \mathsf{act}', \mathsf{at})\end{array}}{(\varepsilon, \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \xrightarrow{stat} (\varepsilon, \mathcal{G}', \mathcal{C}', \mathsf{pdef}, \mathsf{act}', \mathsf{at})} \text{(EXEC)}$$

In the left cases, there will be additional side effects on the global state which are all described in the following rules. The first and most complicated rule is the run-command. A run-command is executed if an active process (often the init process) executes the statement. A new instance of a process type x is created. To do this, the definition of x must be fetched from pdef. Further the actual values for the parameters must be evaluated. Note that this evaluation is dependent on the local variables of the process issuing the run. These local variables are found in $\mathcal{L}_0$. Once the actual values have been evaluated (represented here by v_i), the local variables of the new process instance can be constructed: all variables are undefined despite the parameters, which are assigned their types and values. The new instance is added to the set act of active processes. All this is formally expressed by:

$$\frac{\begin{array}{l}(\varepsilon, \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \xrightarrow{\mathtt{run}\ x(e_1,\dots,e_n),i}_P (\varepsilon, \mathcal{G}', \mathcal{C}', \mathsf{pdef}, \mathsf{act}', \mathsf{at})\\ \text{let } (\pi, f) := \mathsf{pdef}(x), \text{let } \mathsf{act}(i) = (\pi_0, \beta_0, \mathcal{L}_0, \gamma_0),\\ \forall i \in \{1, \dots, n\}.\text{let } v_i := ev_{\mathcal{L}_0,\mathcal{G}}[|e_i|], (t_i, x_i) := f(i)\\ \text{let } \mathcal{L}' := \mathcal{L}_\perp[x_i \mapsto (t_i, v_i) \mid i \in \{1, \dots, n\}]\text{let } \mathsf{act}'' := \mathsf{act}' + (\pi, \pi, \mathcal{L}', \varepsilon)\end{array}}{(\varepsilon, \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \xrightarrow{\mathtt{run}\ x(e_1,\dots,e_n)} (\varepsilon, \mathcal{G}', \mathcal{C}', \mathsf{pdef}, \mathsf{act}'', \mathsf{at})} \text{(RUN)}$$

Note that for simplicity we did not give the conditions when the run-statement leads to a run time error.

5.3 Timeout

The rule for timeout is quite simple: a timeout may occur in the global state if no active process can engage in a transition which is not a timeout. This is described by the following rule:

$$\frac{\begin{array}{l}(\varepsilon, \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \xrightarrow{\mathtt{timeout},i}_P (\varepsilon, \mathcal{G}', \mathcal{C}', \mathsf{pdef}, \mathsf{act}', i),\\ \forall \mathit{stat} \neq \mathtt{timeout}, i \in \mathbb{N}.\text{there is no transition } (\varepsilon, \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \xrightarrow{\mathit{stat},i}_P\end{array}}{(\varepsilon, \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \xrightarrow{\mathtt{timeout}} (\varepsilon, \mathcal{G}', \mathcal{C}', \mathsf{pdef}, \mathsf{act}', i)} \text{(GTIMEOUT)}$$

5.4 Atomic Execution

The rules for atomic execution are quite simple: the first rule covers the case where a process executes `atomic`, and sets the appropriate entry the global state to the index of the process:

$$\frac{\mathsf{at} = \perp, (\varepsilon, \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \xrightarrow{\mathtt{atomic},i}_P (\varepsilon, \mathcal{G}', \mathcal{C}', \mathsf{pdef}, \mathsf{act}', \mathsf{at})}{(\varepsilon, \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \xrightarrow{\mathtt{atomic}} (\varepsilon, \mathcal{G}', \mathcal{C}', \mathsf{pdef}, \mathsf{act}', i)} \text{(AT1)}$$

The next rule describe the fact that during atomic steps no other processes may execute:

$$\frac{\mathsf{at} = i, \mathit{stat} \notin \{\mathtt{atomic}, \mathtt{end\_atomic}\}, (\varepsilon, \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \xrightarrow{\mathit{stat},i}_P (\varepsilon, \mathcal{G}', \mathcal{C}', \mathsf{pdef}, \mathsf{act}', \mathsf{at})}{(\varepsilon, \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \xrightarrow{\mathit{stat}} (\varepsilon, \mathcal{G}', \mathcal{C}', \mathsf{pdef}, \mathsf{act}', \mathsf{at})} \text{(AT2)}$$

If the atomically executing process executes `end_atomic`, then at is reset to bottom, allowing all others process to participate:

$$\frac{\mathsf{at} = i, (\varepsilon, \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \xrightarrow{\mathtt{end\_atomic},i}_P (\varepsilon, \mathcal{G}', \mathcal{C}', \mathsf{pdef}, \mathsf{act}', \mathsf{at})}{(\varepsilon, \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \xrightarrow{\mathtt{end\_atomic}} (\varepsilon, \mathcal{G}', \mathcal{C}', \mathsf{pdef}, \mathsf{act}', \perp)} \text{(AT3)}$$

While we will not discuss all possible errors during atomic execution, we give a rule for a run-time error if nested atomic blocks occur:

$$\frac{\mathsf{at} = i, (\varepsilon, \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \xrightarrow{\mathtt{atomic},i}_P (\varepsilon, \mathcal{G}', \mathcal{C}', \mathsf{pdef}, \mathsf{act}', \mathsf{at})}{(\varepsilon, \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \xrightarrow{\mathtt{run\ time\ error}} (\mathtt{error}, \mathcal{G}', \mathcal{C}', \mathsf{pdef}, \mathsf{act}', \perp)} \text{(AT4)}$$

5.5 Handshake

As mentioned before, handshake communication must be modeled explicitly in the global state, while "real" channel communication is modeled already within the local states. The following rule expresses the fact that if a process i can write to a channel of capacity 1, and process j can then read from this channel without other processes interacting, then a handshake communication can happen over the same channel if it has capacity 0, and the effect of both transitions happening simultaneously is the same as with the communication over the channel with capacity 1.

$$\frac{\begin{array}{l} i \neq j, \mathcal{C}(x) = (T, \varepsilon, 0), \text{let } \mathcal{C}_1 := \mathcal{C}[x \mapsto (T, \varepsilon, 1)], \\ (\varepsilon, \mathcal{G}, \mathcal{C}_1, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \xrightarrow{x!e,i}_P (\varepsilon, \mathcal{G}', \mathcal{C}_1', \mathsf{pdef}', \mathsf{act}', \mathsf{at}'), \\ (\varepsilon, \mathcal{G}', \mathcal{C}_1', \mathsf{pdef}', \mathsf{act}', \mathsf{at}') \xrightarrow{x?e,j}_P (\varepsilon, \mathcal{G}'', \mathcal{C}_1'', \mathsf{pdef}'', \mathsf{act}'', \mathsf{at}''), \text{let } \mathcal{C}'' := \mathcal{C}_1''[x \mapsto (T, \varepsilon, 0)] \end{array}}{(\varepsilon, \mathcal{G}, \mathcal{C}, \mathsf{pdef}, \mathsf{act}, \mathsf{at}) \xrightarrow{\texttt{handshake}\ i,j} (\varepsilon, \mathcal{G}'', \mathcal{C}'', \mathsf{pdef}'', \mathsf{act}'', \mathsf{at}''),} \text{(HANDSH)}$$

Note that from a capacity zero it follows that the channel's content is always ε. The above rule needs to removes this restriction by going from $\mathcal{C}$ to $\mathcal{C}_1$, and restores it when going from $\mathcal{C}_1''$ to $\mathcal{C}''$.

6 Executability, Correctness Issues and Termination

While in [NH96] executability was modeled directly, the concept of executability is implicit in the model here. Note that the executability of a statement depends on the context, e.g. the executability of `x == 3` depends on the value of `x`. It is easy to define context-dependent executability within the given semantics: a statement *stat* is executable in the context $(\pi, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)$ if there is a transition $(\pi, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{stat}$ in the defined semantics. From such a definition of executabilty, we can e.g. derive laws like that `x == 3` is executable in any context where `x` has the value 3.

Correctness issues like never claims, accept and progress labels depend mainly on the existence or non-existence of loops with labels of a special format. In the section on goto's it was explained how to make labels visible. In the section on global control at was also seen how to make process indexes visible. Using these methods, one can derive a semantics where all transitions are labeled with a triple $(\ell, stat, i)$. Then progress can be described in the model by the existence of a loop which at least one transition labeled $(\ell, stat, i)$ where ℓ is a progress label. Note that if process indices are not made visible, this definition would not be correct as potentially the progress label could appear in different processes throughout the loop.

A process terminates when it reaches the end of its body and its father has terminated. To define such a concept within the semantics, it would be necessary to store the father information within process instances. Within this framework a process reaches the end of its body when it reaches a state $(\varepsilon, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)$.

7 Extendability

Due to the incremental, i.e. modular approach, extending the given semantics with new concepts should be no problem. This is especially true for structured features like e.g. the `unless`-operator. The following rules for the `unless`-operator could be added to the rules of local control. The first rule models the case if π_2 is executable:

$$\frac{(\pi_2;\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{stat} (\pi_2';\pi', \mathcal{L}', \mathcal{G}', \mathcal{C}', \gamma')}{(\pi_1 \texttt{ unless } \pi_2;\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{stat} (\pi_2';\pi', \mathcal{L}', \mathcal{G}', \mathcal{C}', \gamma')} \text{(UNL1)}$$

The second models the case where π_2 is not executable:

```
chan x = [1] of { bool, int };
int g = 1;

proctype B(chan b; int m) {
   int v;
   g = m;
   do
      :: b ? 0,v;
      :: timeout -> break;
   od;
   g = 1;
}
```

```
proctype A(chan a) {
   a ! 0,3;
   atomic {
      g == 0 -> g = g+1;
   }
}

init {
   run A(x);
   run B(x, 0);
}
```

Fig. 5. Example PROMELA Program

$\pi_1 :=$ `chan x = [1] of {bool, int}`; π_2
$\pi_2 :=$ `int g = 1`; π_3
$\pi_3 :=$ `proctype A(chan a)` $\{\pi_4\}$ π_7
$\pi_4 :=$ `a ! 0,3`; π_5
$\pi_5 :=$ `atomic` $\{\pi_6\}$
$\pi_6 :=$ `g == 0; g = g + 1;`
$\pi_7 :=$ `proctype B(chan b; int m)`$\{\pi_8\}$ π_{13}
$\pi_8 :=$ `int v`; π_9
$\pi_9 :=$ `g = m`; π_{10}
$\pi_{10} :=$ `do` π_{11} `od; g = 1;`
$\pi_{11} :=$ `:: b ? 0,v; ::` π_{12}
$\pi_{12} :=$ `timeout; break`
$\pi_{13} :=$ `init` $\{\pi_{14}\}$
$\pi_{14} :=$ `run A(x); run B(x, 0);`
$\pi_{15} := \pi_6$; `end_atomic;`
$\pi_{16} :=$ `g = g + 1; end_atomic;`
$\pi_{17} :=$ `do` π_{11} `od;`
$\pi_{18} :=$ `if` π_{11} `fi`; π_{17}

Fig. 6. Abbreviations

$$\frac{\text{there is no } \mathit{stat} \text{ such that } (\pi_2;\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{stat}, \quad (\pi_1, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{stat'} (\pi_1', \mathcal{L}', \mathcal{G}', \mathcal{C}', \gamma')}{(\pi_1 \texttt{ unless } \pi_2;\pi', \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{stat'} (\pi_1' \texttt{ unless } \pi_2';\pi', \mathcal{L}', \mathcal{G}', \mathcal{C}', \gamma')} \text{(UNL2)}$$

Also deterministic execution can be added similar to the rules given for atomic execution.

A larger problem with extensions are those features which are connected to expressions. Expressions were not given an operational semantics in the paper, but a denotational semantics was defined for them. Thus it is not easy to incorporate changes to expressions into the current framework. This is the main reason why `run`-statements and channel enquiries are not allowed in expression within this approach. The choice for a denotational semantics for expressions was due to a more elegant approach. In the light of extendability, an operational semantics for expressions could be the better choice. This should also simplify the inclusion of remote referencing.

8 Example

This section gives a simple example to illustrate the approach. The PROMELA code of the example is given in Fig. 5. The labeled transition systems given in the following pictures use the abbreviations as given in Fig 6.

Fig. 7 and 8 gives all the transitions for the process types A and B induced by the source code. Note that there are no restrictions on $\pi, \mathcal{L}, \mathcal{G}, \mathcal{C}$ and γ despite those given so the Figures

$(\pi_4, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\texttt{a ! 0,3;}} (\pi_5, \mathcal{L}, \mathcal{G}, \mathcal{C}', \gamma)$
  if $|w| < k$, where $chid := (\mathcal{L}, \mathcal{G})(a), (T, w, k) := \mathcal{C}(chid), \mathcal{C}' := \mathcal{C}[chid \mapsto (T, w \cdot (0,3), k)]$
$(\pi_5, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\texttt{atomic}} (\pi_6;\ \texttt{end\_atomic};, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)$, if $ev_{\mathcal{L},\mathcal{G}}[|g|] == 0$
$(\pi_{15}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\texttt{g==0}} (\pi_{16}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)$, if $ev_{\mathcal{L},\mathcal{G}}[|g|] == 0$
$(\pi_{16}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\texttt{g = g+1;}} (\texttt{end\_atomic};, \mathcal{L}', \mathcal{G}', \mathcal{C}, \gamma)$, let $(\mathcal{L}', \mathcal{G}') := (\mathcal{L}, \mathcal{G})[g \mapsto ev_{\mathcal{L},\mathcal{G}}[|g|] + 1]$
$(\texttt{end\_atomic};, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\texttt{end\_atomic}} (\varepsilon, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)$

Fig. 7. Semantics for `proctype A`

$(\pi_8, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\texttt{int v;}} (\pi_9, \mathcal{L}', \mathcal{G}, \mathcal{C}, \gamma)$, let $\mathcal{L}' := \mathcal{L}[v \mapsto (\mathrm{INT}, 0)]$
$(\pi_9, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\texttt{g = m;}} (\pi_{10}, \mathcal{L}', \mathcal{G}', \mathcal{C}, \gamma)$, let $(\mathcal{L}', \mathcal{G}') := (\mathcal{L}, \mathcal{G})[g \mapsto ev_{\mathcal{L},\mathcal{G}}[|m|]]$
$(\pi_{10}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\tau} (\pi_{18}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \texttt{g = 1;} \cdot \gamma)$
$(\pi_{18}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\texttt{b ? 0,v;}} (\pi_{17}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)$
$(\pi_{17}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\tau} (\pi_{18}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)$
$(\pi_{18}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\texttt{timeout}} (\texttt{break};, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)$
$(\texttt{break};, \mathcal{L}, \mathcal{G}, \mathcal{C}, \texttt{g = 1;} \cdot \gamma) \xrightarrow{\texttt{break;}} (\texttt{g = 1;}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma)$
$(\texttt{g = 1;}, \mathcal{L}, \mathcal{G}, \mathcal{C}, \gamma) \xrightarrow{\texttt{g = 1;}} (\varepsilon, \mathcal{L}', \mathcal{G}', \mathcal{C}, \gamma)$, let $(\mathcal{L}', \mathcal{G}') := (\mathcal{L}, \mathcal{G})[g \mapsto 1]$

Fig. 8. Semantics for `proctype B`

really represent an infinite number of transitions, but most of the states are not reachable from the start state. For the unravelling of the do-loop, Fig. 8 assumes that `g=1;` is the top element of the continuation stack (which is the only possibility for states reached via the do-loop).

In Fig. 9 a sample path through the complete program is given as an example of global control. All but the two last lines give the complete global state and the transitions between global states. Let statements introduce necessary definitions for the global states in the lines preceding each transition. In the last two lines, after processes of type A and B have been created, only the transitions are given, but not the details of the global state. Note that in these steps $\mathcal{L}, \mathcal{G}$ and $\mathcal{C}$ will change according to the semantics of the individual processes as given in the previous figures. For readability, the process identifiers have been added to the labels: 0 is the init process, 2 is the process of type A and 3 is the process of type B. Note that in states S_6, S_7 and S_8, the atomic-execution-indicator of the global state will be set to the process id 2.

In the step where the inital process is created for simplicity an empty process for the (non-existent) never claim is created as well.

9 Conclusion

I have presented a new approach to a formal semantics for PROMELA. The approach is able to handle all features of PROMELA. It is a modular approach which makes reasoning and understanding the semantics easy. The approach is more abstract than the approach given in [NH96]. The both approaches should not be seen as competing but as approaches on different levels: the approach here should be more suitable for reasoning about the semantics of PROMELA, while the approach in [NH96] should more suitable for reasoning about

$$\text{let } \mathcal{G}_1 := \mathcal{G}_\perp[\texttt{x} \mapsto (\mathrm{CHAN}, |\mathcal{C}_\perp|)], \mathcal{C}_1 := \mathcal{C}_\perp + (\mathrm{I\!B} \times \mathrm{INT}, \varepsilon, 1)$$
$$(\pi_1, \mathcal{G}_\perp, \mathcal{C}_\perp, \mathsf{pdef}_\perp, \mathsf{act}_\perp, \perp) \xrightarrow{\texttt{chan x = [1] of\{bool, int\};}} (\pi_2, \mathcal{G}_1, \mathcal{C}_1, \mathsf{pdef}_\perp, \mathsf{act}_\perp, \perp)$$
$$\text{let } \mathcal{G}_2 := \mathcal{G}_1[\mathsf{g} \mapsto (\mathrm{INT}, 1)]$$
$$\xrightarrow{\texttt{int g = 1;}} (\pi_3, \mathcal{G}_2, \mathcal{C}_1, \mathsf{pdef}_\perp, \mathsf{act}_\perp, \perp)$$
$$\text{let } \mathsf{pdef}_1 := \mathsf{pdef}[\texttt{A} \mapsto (\pi_4, f_{\texttt{A}})], f_{\texttt{A}} : \{1\} \to \mathrm{NAME} \times \mathbf{T}, f_{\texttt{A}}(1) = (\mathsf{a}, \mathrm{CHAN})$$
$$\xrightarrow{\texttt{proctype A(chan a) }\{\pi_4\}} (\pi_7, \mathcal{G}_2, \mathcal{C}_1, \mathsf{pdef}_1, \mathsf{act}_\perp, \perp)$$
$$\text{let } \mathsf{pdef}_2 := \mathsf{pdef}_1[\texttt{B} \mapsto (\pi_8, f_{\texttt{B}})], f_{\texttt{B}} : \{1, 2\} \to \mathrm{NAME} \times \mathbf{T}, f_{\texttt{B}}(1) := (\mathsf{b}, \mathrm{CHAN}), f_{\texttt{B}}(2) := (\mathtt{m}, \mathrm{INT})$$
$$\xrightarrow{\texttt{proctype B(chan b; int m)}\{\pi_8\}} (\pi_{13}, \mathcal{G}_2, \mathcal{C}_1, \mathsf{pdef}_2, \mathsf{act}_\perp, \perp)$$
$$\text{let } \mathsf{pdef}_3 := \mathsf{pdef}_2[\mathtt{init} \mapsto (\pi_{14}, f_0)][\mathtt{never} \mapsto (\varepsilon, f_0)], f_0 : \{\} \to \mathrm{NAME} \times \mathbf{T},$$
$$\mathsf{act}_1 := \big(\mathsf{act}_\perp + (\pi_{14}, \pi_{14}, \mathcal{L}_\perp, \varepsilon)\big) + (\varepsilon, \varepsilon, \mathcal{L}_\perp, \varepsilon)$$
$$\xrightarrow{\pi 13} (\varepsilon, \mathcal{G}_2, \mathcal{C}_1, \mathsf{pdef}_3, \mathsf{act}_1, \perp)$$
$$\text{let } \mathsf{act}_2 := \mathsf{act}_1 + (\pi_4, \pi_4, \mathcal{L}_{\texttt{A}}, \varepsilon), \mathcal{L}_{\texttt{A}} := \mathcal{L}_\perp[\mathsf{a} \mapsto ev[|\mathtt{x}|]]$$
$$\xrightarrow{\texttt{run A(x);}} (\varepsilon, \mathcal{G}_2, \mathcal{C}_1, \mathsf{pdef}_3, \mathsf{act}_2, \perp)$$
$$\text{let } \mathcal{C}_2 := \mathcal{C}_1[0 \mapsto (\mathrm{I\!B} \times \mathrm{INT}, (0, 3), 1)]$$
$$\xrightarrow{\texttt{a ! 0,3;,2}} (\varepsilon, \mathcal{G}_2, \mathcal{C}_2, \mathsf{pdef}_3, \mathsf{act}_1, \perp)$$
$$\text{let } \mathsf{act}_2 := \mathsf{act}_1 + (\pi_9, \pi_8, \mathcal{L}_{\texttt{B}}, \varepsilon), \mathcal{L}_{\texttt{B}} := \mathcal{L}_\perp[\mathsf{b} \mapsto ev[|\mathtt{x}|]]$$
$$\xrightarrow{\texttt{run B(x);,0}} (\varepsilon, \mathcal{G}_2, \mathcal{C}_2, \mathsf{pdef}_3, \mathsf{act}_2, \perp)$$
$$\xrightarrow{\texttt{int v,3}} S_1 \xrightarrow{\texttt{g = m,3}} S_2 \xrightarrow{\tau,3} S_3 \xrightarrow{\texttt{b ? 0,v;,3}} S_4 \xrightarrow{\tau,3} S_5 \xrightarrow{\texttt{atomic,2}} S_6 \xrightarrow{\texttt{== 0;,2}} S_7 \xrightarrow{\texttt{g = g+1;,2}} S_8$$
$$\xrightarrow{\texttt{end\_atomic,2}} S_9 \xrightarrow{\texttt{timeout,3}} S_{10} \xrightarrow{\texttt{break;,3}} S_{11} \xrightarrow{\texttt{g = 1;,3}} S_{12}$$

Fig. 9. Sample Path through the Program Example

the implementation of a PROMELA compiler. It would be interesting to investigate a translation from one approach to the other.

Note that a simple, correct and complete formal semantics has several advantages. For once it can be used as a reference model for correctness questions. Even more important a sound semantics is only possible when the language definition is sound. Very often special cases are not treated in informal language definitions, while this incompleteness is often detected within a formal framework. Examples for such situations in PROMELA have been addressed in the comments on the semantics of the send- and receive-statement and in the footnote in the section on the global control.

The presented approach is neither complete nor error free. However it should be possible to gain a complete and error free semantics for PROMELA within short time. Further there are a lot a spots where the elegance of the approach could be improved.

References

[Hol91] G.J. Holzmann. Design and Validation of computer protocols. Prentice hall Software series, Englewood Cliffs 1991.

[NH96] V. Natarajan, G.J. Holzmann. Outline for an Operational Semantics of PROMELA.