# Process Algebra Semantics of $\varphi$SDL*

J.A. Bergstra[1,3] C.A. Middelburg[2,3]

[1]*Programming Research Group, University of Amsterdam*
*P.O. Box 41882, 1009 DB Amsterdam, The Netherlands*

[2]*Department of Network & Service Control, KPN Research*
*P.O. Box 421, 2260 AK Leidschendam, The Netherlands*

[3]*Department of Philosophy, Utrecht University*
*P.O. Box 80126, 3508 TC Utrecht, The Netherlands*

*E-mail:* janb@fwi.uva.nl - keesm@phil.ruu.nl

**Abstract**

A new semantics of an interesting subset of the specification language SDL is given by a translation to a discrete-time variant of process algebra in the form of ACP extended with data as in $\mu$CRL. The strength of the chosen subset, called $\varphi$SDL, is its close connection with full SDL, despite its dramatically reduced size. Thus, we are able to concentrate on solving the basic semantic issues without being in danger of having to turn the results inside out in order to deal with full SDL. Novel to the presented semantics is that it relates the time used with timer setting to the time involved in waiting for signals and delay of signals.

*Keywords & Phrases:* discrete time, process algebra, semantics, specification language, asynchronous communication, delay, timers, ACP, SDL.

*1994 CR Categories:* D.2.1, D.3.1, F.3.1.

*This paper is to be presented at the 2nd Workshop on Algebra of Communicating Processes, May 1995.

# Contents

# 1 Introduction

A process algebra semantics of $\varphi$SDL is presented. $\varphi$SDL is roughly a subset of Basic SDL.[1] The following simplifications have been made:

- blocks are removed and consequently channels and signal routes are merged – making channel to route connections obsolete;
- variables are treated more liberal: all variables are revealed and they can be viewed freely;
- timer setting is regarded as just a special use of signals;
- timer setting is based on discrete time.

Besides, $\varphi$SDL does not deal with the specification of abstract data types. An algebraic specification of all data types used in an $\varphi$SDL specification is assumed as well as an initial algebra semantics for it. The pre-defined data types Boolean and Natural, with the obvious interpretation, should be included; and besides, PId and Time should be included as copies of Natural.

We decided to focus in $\varphi$SDL on the behavioural aspects of SDL. We did so for the following two reasons. Firstly, the structural aspects of SDL are mostly of a static nature and therefore not very relevant from a semantic point of view. Secondly, the part of SDL that deals with the specification of abstract data types is well understood – besides, it can easily be isolated and treated as a parameter.[2] Because it will largely be a routine matter, we also chose to postpone the addition of procedures, syntypes with a range condition and process types with a bound on the number of instances that may exist simultaneously. For similar reasons, the any expression is omitted. Services are not supported by $\varphi$SDL for other reasons: the semantics of services is hard to understand, ETSI forbids for this reason their use in European telecommunication standards (see [19]), and the SDL community currently discusses its usefulness (see [16]).

Apart from the data type definitions, all SDL system definitions without usage of procedures, services, syntypes with a range condition, process types with a bound on the number of instances that may exist simultaneously, and the any expression can be transformed to $\varphi$SDL system definitions. The transformation concerned has, apart from some minor adaptations, already been given. The first part of the transformation is the mapping for the shorthand notations of SDL which is given informally in the ITU/TS Recommendation Z.100 [21] and defined in a fully precise manner in its Annex F.2 [23]. The second and final part is essentially the mapping *extract-dict* which is defined in its Annex F.3 [24]; $\varphi$SDL system definitions can actually be viewed as textual presentations of the

[1] This subset is called $\varphi$SDL, where $\varphi$ stands for flat, as it does not cover the structural aspects of SDL. Throughout the paper, we will write SDL for the version of SDL defined in [21], the ITU/TS Recommendation Z.100 published in 1992.

[2] The following is also worth noticing: (1) ETSI discourages the use of abstract data types other than the pre-defined ones in European telecommunication standards (see [19]); (2) ASN.1 [20] is widely used for data type specification in the telecommunications field, and there is an emerging ITU/TS Recommendation, Z.105, for combining SDL and ASN.1 (see [25]).

extracted *Entity-dict*s which are interpreted instead of the SDL system definitions proper.

The semantics of $\varphi$SDL agrees with the semantics of SDL as far as reasonably possible. This means in the first place that obvious errors in [24] have not been taken over. For example, the intended effect of SDL's create and output actions may sometimes be reached with interruption according to [24] – allowing amongst other things that a process ceases to exist while a signal is sent to it without any delay. Secondly, the way of dealing with time is considered to be unnecessarily complex and inadequate in SDL and has been adapted as explained below.

In SDL,Time and Duration, the pre-defined sorts of absolute time and relative time, are both copies of the pre-defined sort Real (intended to stand for the real numbers, but in fact standing for the rational numbers, see [22]). When a timer is set, a real expiration time must be given. However, the time considered is the system time which proceeds actually in a discrete manner: the system receives ticks from the environment which increase the system time with a certain amount (how much real time they represent is left open). Therefore, the timer is considered to expire when the system receives the first tick that indicates that its expiration time has passed. So nothing is lost by adopting in $\varphi$SDL a discrete time approach, using copies of Natural for Time and Duration, where the time unit can be viewed as the time between two ticks but does not really rely upon the environment. This much simpler approach also allows us to remove the original inadequacy to relate the time used with timer setting to the time involved in waiting for signals by processes and in delay of signals in channels.

We had to make our own choices with respect to time in $\varphi$SDL, because the time related aspects of SDL are virtually left out completely in the ITU/TS recommendation Z.100. Our choices were based on communications with various practitioners from the telecommunications field using SDL. In particular the communications with Leonard Pruitt [18] provided convincing practical justification for the premise of our choices: provided time is divided into sufficiently large time slices, an SDL process will only enter a next time slice if there are no more signals to consume for it in the current time slice. Ease of adaptation to other viewpoints on time in SDL is guaranteed relatively well by using a discrete-time variant of process algebra, essentially $\mathrm{ACP}_{\mathrm{dt}}$ (see [2]), as the basis of the presented semantics.

The language $\varphi$SDL and the presented semantics for it are primarily intended for work on advanced analysis tools for systems modelled using SDL. However, it can also serve to gain a better insight into the semantic aspects of proposed simplifications, and other future changes, of SDL.

The structure of this paper is as follows. First of all, we give an overview of $\varphi$SDL (Section 2). Next, we give a brief summary of the ingredients of process algebra which make up the basis for the semantics of $\varphi$SDL presented in this paper (Section 3). Then, we describe specifics on the operator used to formalize execution of a process in a state (Section 4). After that, we present the process algebra semantics of $\varphi$SDL (Section 5). Finally, we make some additional remarks about the work reported on in this paper as well as some remarks about

related work (Section 6). Besides, there are appendices about notational conventions used (Appendix A) and details about the contexts used to model scope in the presented semantics (Appendix B).

# 2 Overview of $\varphi$SDL

This section gives an overview of $\varphi$SDL. Its syntax is described by means of production rules in the form of an extended BNF grammar (the extensions are explained in Appendix A). The meaning of the language constructs of the various forms distinguished by these production rules is explained informally. Some peculiar details, inherited from full SDL, are left out to improve the comprehensibility of the overview. These details will, however, be made mention of in Section 5, where a process algebra semantics of $\varphi$SDL is presented.

## 2.1 System definition

First of all, the $\varphi$SDL view of a system is explained in broad outline.

Basically, a system consists of *processes* which communicate with each other and the environment by sending and receiving *signals* via *signal routes*. A process proceeds in parallel with the other processes in the system and communicates with these processes in an asynchronous manner. This means that a process sending a signal does not wait until the receiving process consumes it, but it proceeds immediately. A process may also use local *variables* for storage of values. A variable is associated with a value that may change by assigning a new value to it. A variable can only be assigned new values by the process to which it is local, but it may be viewed by other processes. Processes can be distinguished by unique addresses, called *pid values* (process identification values), which they get with their creation.

A signal can be sent from the environment to a process, from a process to the environment or from a process to a process. A signal may carry values to be passed from the sender to the receiver; on consumption of the signal, these values are assigned to local variables of the receiver. A signal route is a unidirectional connection between the processes of two types, or between the processes of one type and the environment, for conveying signals. A signal route may contain a *channel*.[3] Signals that must pass through a channel are delayed, but signals always leave a channel in the order in which they have entered it. Thus a signal route is a communication path for sending signals, with or without a delay, from the environment to a process, from one process to another process or from a process to the environment. If a signal is sent to a process via a signal route that does not contain a channel, it can be instantaneously delivered to that process. Otherwise there can be an arbitrary delay. A channel may be contained in more than one signal route.

[3]The original channels have been merged with signal routes, but the term channel is reused in $\varphi$SDL (see also Section 2.4).

**Syntax:**

```
<system definition> ::=
   system <system nm> ; {<definition>}+ endsystem ;

<definition> ::=
    dcl <variable nm> <sort nm> ;
  | signal <signal nm> [ ( <sort nm> {, <sort nm>}* ) ] ;
  | channel <channel nm> ;
  | signalroute <signalroute nm>
      from {<process nm> | env} to {<process nm> | env}
      with <signal nm> {, <signal nm>}* [ delayed by <channel nm> ] ;
  | process <process nm> ( <natural ground expr> ) ;
      [ fpar <variable nm> {, <variable nm>}* ; ]
      start ; <transition> {<state def>}*
    endprocess ;
```

A system definition consists of definitions of the types of processes present in the system, the local variables used by the processes for storage of values, the types of signals used by the processes for communication, the signal routes via which the signals are conveyed and the channels contained in signal routes to delay signals.

A variable definition dcl $v$ $T$; defines a variable $v$ that may be assigned values of sort $T$.

A signal definition signal $s(T_1, \ldots, T_n)$; defines a type of signals $s$ of which the instances carry values of the sorts found in $T_1, \ldots, T_n$. If $(T_1, \ldots, T_n)$ is absent, the signals of type $s$ do not carry any value.

A channel definition channel $c$ defines a channel that delays signals that pass through it.

A signal route definition signalroute $r$ from $X_1$ to $X_2$ with $s_1, \ldots, s_n$; defines a signal route $r$ that delivers without a delay signals sent by processes of type $X_1$ to processes of type $X_2$, for signals of types found in $s_1, \ldots, s_n$. The process types $X_1$ and $X_2$ are called the sender type of $r$ and the receiver type of $r$, respectively. A signal route from the environment can be defined by replacing from $X_1$ by from env. A signal route to the environment can be defined analogously. A signal route delivering signals with an arbitrary delay can be defined by adding delayed by $c$, where $c$ is the channel causing the delay.

A process definition process $X(k)$; fpar $v_1, \ldots, v_m$; start; $tr$ $d_1$ ... $d_n$ endprocess; defines a type of processes $X$ of which $k$ instances will be created during the start-up of the system. On creation of a process of type $X$ after the start-up, the creating process passes values to it which are assigned to the local variables found in $v_1, \ldots, v_m$. If fpar $v_1, \ldots, v_m$ is absent, no values are passed on creation. The process body start; $tr$ $d_1, \ldots, d_n$ describes the behaviour of the processes of type $X$ in terms of states and transitions (see further Section 2.2). Each process will start by making the transition $tr$, called its start transition, to enter one of its states. The state definitions found in $d_1$ ... $d_n$ define all the states in which the process may come while it proceeds.

## 2.2 Process behaviours

First of all, the $\varphi$SDL view of a process is briefly explained.

To begin with, a process is either in a *state* or making a *transition* to another state. Besides, when a signal arrives at a process, it is put into the unique *input queue* associated with the process until it is consumed by the process. The states of a process are the points in its behaviour where a signal may be consumed. However, a state may have signals that have to be saved, i.e. withhold from being consumed in that state. The signal consumed in a state of a process is the first one in its input queue that has not to be saved for that state. If there is no signal to consume, the process waits until there is a signal to consume. So if a process is in a state, it is either waiting to consume a signal or consuming a signal.

A transition from a state of a process is initiated by the consumption of a signal, unless it is a spontaneous transition. The start transition is not initiated by the consumption of a signal either. A transition is made by performing certain actions: signals may be sent, variables may be assigned new values, new processes may be created and *timers* may be set and reset. A transition may at some stage also take one of a number of branches, but it will eventually come to an end and bring the process to a next state or to its termination.

A timer can be set which sends at its expiration time a signal to the process setting it. A timer is identified with the type and carried values of the signal it sends on expiration. Thus an active timer can be set to a new time or reset; if this is done between the sending of the signal noticing expiration and its consumption, the signal is removed from the input queue concerned. A timer is de-activated when it is reset or the signal it sends on expiration is consumed.

**Syntax:**

```
<state def> ::=
   state <state nm> ;
      [ save <signal nm> {, <signal nm>}* ; ] {<transition alt>}*

<transition alt> ::=
   {<input guard> | input none ;} <transition>

<input guard> ::=
   input <signal nm> [ ( <variable nm> {, <variable nm>}* ) ] ;

<transition> ::=
   {<action>}* {nextstate <state nm> | stop | <decision>} ;

<action> ::=
    output <signal nm> [ ( <expr> {, <expr>}* ) ]
      [ to <pid expr> ] via <signalroute nm> {, <signalroute nm>}* ;
   | set ( <time expr> , <signal nm> [ ( <expr> {, <expr>}* ) ] ) ;
   | reset ( <signal nm> [ ( <expr> {, <expr>}* ) ] ) ;
   | task <variable nm> := <expr> ;
   | create <process nm> [ ( <expr> {, <expr>}* ) ] ;
```

<decision> ::=
  decision {<expr> | any} ;
    ( [ <ground expr> ] ) : <transition>
    {( [ <ground expr> ] ) : <transition>}$^+$
  enddecision

A state definition state *st*; save $s_1, \ldots, s_m$; $alt_1 \ \ldots \ alt_n$ defines a state *st* in which certain signals may be consumed and subsequently certain transitions must be made. The signals of the types found in $s_1, \ldots, s_m$ are saved for the state. Each input guard occurring in $alt_1 \ \ldots \ alt_n$ gives a type of signals that may be consumed in the state; the corresponding transition is the one that is initiated on consumption of a signal of that type. The transitions with input none; instead of an input guard are the spontaneous transitions that may be made from the state. No signals are saved for the state if save $s_1, \ldots, s_m$; is absent.

An input guard input $s(v_1, \ldots, v_n)$; may consume a signal of type $s$ and, on consumption, it assigns the carried values to the variables found in $v_1, \ldots, v_n$. If the signals of type $s$ carry no value, $(v_1, \ldots, v_n)$ is left out.

A transition $a_1 \ \ldots \ a_n$ nextstate *st*; performs the actions found in $a_1 \ \ldots \ a_n$ in sequential order and ends with entering the state *st*. Replacing nextstate *st* by the keyword stop yields a transition ending with process termination. Replacing it by the decision *dec* leads instead to transfer of control to one of two or more transition branches.

An output action output $s(e_1, \ldots, e_n)$ to $e$ via $r_1, \ldots, r_m$; sends a signal of type $s$ carrying the current values of the expressions in $e_1, \ldots, e_n$ to the process with the current (pid) value of the expression $e$ as its address, via one of the usable signal routes found in via $r_1, \ldots, r_m$. If the signals of type $s$ carry no value, $(e_1, \ldots, e_n)$ is left out. If to $e$ is absent, the signal is sent via one of the signal routes found in via $r_1, \ldots, r_m$ to an arbitrary process of its receiver type. The output action is called an output action with explicit addressing if to $e$ is present. Otherwise, it is called an output action with implicit addressing.

A set action set $(e, s(e_1, \ldots, e_n))$; sets a timer that expires, unless it is set again or reset, at the current (time) value of the expression $e$ with sending a signal of type $s$ that carries the current values of the expressions in $e_1, \ldots, e_n$.

A reset action reset $(s(e_1, \ldots, e_n))$; de-activates the timer identified with the signal type $s$ and the current values of the expressions in $e_1, \ldots, e_n$.

An assignment task action task $v{:=}e$; assigns the current value of the expression $e$ to the local variable $v$.

A create action create $X(e_1, \ldots, e_n)$; creates a process of type $X$ and passes the current values of the expressions in $e_1, \ldots, e_n$ to the newly created process. If no values are passed on creation of processes of type $X$, $(e_1, \ldots, e_n)$ is left out.

A decision decision $e$;$(e_1)$:$tr_1 \ \ldots \ (e_n)$:$tr_n$ enddecision transfers control to the transition branch $tr_i$ $(1 \leq i \leq n)$ for which the value of the expression $e_i$ equals the current value of the expression $e$. Non-existence and non-uniqueness of such a branch result in an error. A non-deterministic choice can be obtained by replacing the expression $e$ by the keyword any and removing all the expressions $e_i$.

## 2.3 Values

The value of expressions in $\varphi$SDL may vary according to the last values assigned to variables, including local variables of other processes. It may also depend on the system state, e.g. on timers being active or the system time.

**Syntax:**

```
<expr> ::=
    <operator nm> [ ( <expr> {, <expr>}* ) ]
  | if <boolean expr> then <expr> else <expr> fi
  | <variable nm>
  | view ( <variable nm> , <pid expr> )
  | active ( <signal nm> [ ( <expr> {, <expr>}* ) ] )
  | now | self | parent | offspring | sender
```

An operator application $op(e_1, \ldots, e_n)$ evaluates to the value yielded by applying the operation $op$ to the current values of the expressions in $e_1, \ldots, e_n$.

A conditional expression if $e_1$ then $e_2$ else $e_3$ fi evaluates to the current value of the expression $e_2$ if the current (Boolean) value of the expression $e_1$ is true, and the current value of the expression $e_3$ otherwise.

A variable access $v$ evaluates to the current value of the local variable $v$ of the process evaluating the expression.

A view expression view $(v,e)$ evaluates to the current value of the local variable $v$ of the process with the current (pid) value of the expression $e$ as its address.

An active expression active $(s(e_1, \ldots, e_n))$ evaluates to the Boolean value true if the timer identified with the signal type $s$ and the current values of the expressions in $e_1, \ldots, e_n$ is currently active, and false otherwise.

The expression now evaluates to the current system time.

The expressions self, parent, offspring and sender evaluate to the pid values of the process evaluating the expression, the process by which it was created, the last process created by it, and the sender of the last signal consumed by it.

## 2.4 Differences with SDL

Syntactically, $\varphi$SDL is not exactly a subset of SDL. The syntactic differences are as follows:

- variable definitions occur at the system level instead of inside process definitions;
- signal route definitions and process definitions occur at the system level instead of inside block definitions;
- channel paths in channel definitions are absent;
- the option delayed by $c$ in signal route definitions is new;
- formal parameters in process definitions are variable names instead of pairs of variable names and sort names;
- signal names are used as timer names.

These differences are all due to the simplifications mentioned in Section 1.

Recall that channels and signal routes have been merged. Because the resulting communication paths connect processes with one another or with the enviroment, like the original signal routes, we chose to call them signal routes as well. However, the new signal routes may have delaying parts which are reminiscent of the original channels. Therefore, we chose to reuse their name for these delaying parts.

# 3 Process algebra preliminaries

This section gives a brief summary of the ingredients of process algebra which make up the basis for the semantics of $\varphi$SDL presented in Section 5. We will suppose that the reader is familiar with them. Appropriate references to the literature are included.

We will make use of the Algebra of Communicating Processes (ACP),[4] introduced in [8], extended with the silent step $\tau$ and the abstraction operator $\tau_I$ for abstraction. Semantically, we adopt the approach to abstraction, originally proposed for ACP in [9], which is based on weak bisimulation due to Milner [15]. For a systematic introduction to ACP, the reader is referred to [5].

Further we will use the following extensions:

**state operator** We will use the state operator $\lambda_S$, added to ACP in [1]. This operator formalizes execution of a process in a state. Basic is the execution of actions: the action $a'$ that occurs as the result of executing an action $a$ in a state $S$, and the state $S'$ that results when executing $a$ in $S$. This leads to defining equations of the form $\lambda_S(a \cdot P) = a' \cdot \lambda_{S'}(P)$.

**process creation** We will also use the process creation mechanism, added to ACP in [6]. The process creation operator $E_\phi$ introduced there allows, given a mapping $\phi$ from process names to process expressions, the use of actions of the form $cr(X)$ to create processes $\phi(X)$. The most crucial equation from the defining equations of this operator is $E_\phi(cr(X) \cdot P) = \overline{cr}(X) \cdot E_\phi(\phi(X) \parallel P)$. Note that the process creation operator leaves a trace of actions of the form $\overline{cr}(X)$.

**conditionals** Besides, we will use the one-armed conditional operator $:\!\rightarrow$ as in [3]. The expression $b :\!\rightarrow P$, is to be read as $\mathsf{if}\, b\, \mathsf{then}\, P$; it can only be performed if $b \neq \mathsf{false}$. It is often referred to as a guarded command.

**iteration** We will also use the binary version of Kleene's star operator $^*$, added to ACP in [7], with the defining equation $P^*Q = P \cdot (P^*Q) + Q$. The behaviour of $P^*Q$ is zero or more repetitions of $P$ followed by $Q$.

**discrete time** We need a relative time version of discrete time process algebra in the form of ACP. We will use the extension of ACP that can be found

[4] We will actually use ACP without communication, also known as $\mathrm{PA}_\delta$.

in [2], which is quite similar to ATP [17]. Here we briefly survey discrete time processes in an informal way.

Time is divided into slices indexed by natural numbers. These time slices represent time intervals of a length which corresponds to the time unit used. If the current time is $t$, $t \in \mathbb{R}_{\geq 0}$, the current time slice is the time interval $[\lfloor t \rfloor, \lfloor t \rfloor + 1)$ (where $\lfloor t \rfloor$ denotes the floor of $t$). We will use the constants $\underline{\underline{a}}$ (one for each action $a$) and $\underline{\underline{\delta}}$,[5] as well as the delay operator $\sigma_{\mathsf{rel}}$. $\underline{\underline{a}}$ is $a$ performed within the current time slice and $\sigma_{\mathsf{rel}}(P)$ is $P$ delayed till the next time slice. In a parallel composition $P_1 \parallel \ldots \parallel P_n$ the transition to the next time slice is a simultaneous transition of each of the $P_i$s. For example, $\underline{\underline{\delta}} \parallel \sigma_{\mathsf{rel}}(\underline{\underline{b}})$ will never perform $\underline{\underline{b}}$ because $\underline{\underline{\delta}}$ can neither be delayed nor performed, so $\underline{\underline{\delta}} \parallel \sigma_{\mathsf{rel}}(\underline{\underline{b}}) = \underline{\underline{\delta}}$. However, $\underline{\underline{a}} \parallel \sigma_{\mathsf{rel}}(\underline{\underline{b}}) = \underline{\underline{a}} \cdot \sigma_{\mathsf{rel}}(\underline{\underline{b}})$.

**summation over data domains** We will in addition use actions parametrized by data and summation over a data domain as in $\mu$CRL [13, 14]. The notation $a(t_1, \ldots, t_n)$, where the $t_i$s denote data values, is used for instances of parametrized actions. In $\sum_{x:D} P$, the scope of the variable $x$ is exactly $P$. The behaviour of $\sum_{x:D} P$ is a choice between the instances of $P$ for the different values that $x$ can take, i.e. the values from the data domain $D$.

The above-mentioned extensions of ACP with a state operator and a process creation mechanism are also presented in [5]. In ACP with abstraction, the operators $\lambda_S$ and $E_\phi$ can be defined.

We will also use some abbreviations. Let $(P_i)_{i \in I}$ be a indexed set of process expressions where $I = \{i_1, \ldots, i_n\}$. Then, we write:

$\sum_{i \in I} P_i$ for $P_{i_1} + \ldots + P_{i_n}$
$\|_{i \in I} P_i$ for $P_{i_1} \parallel \ldots \parallel P_{i_n}$

Let $P$ be a process expression and let $n \in \mathbb{N}$. Then, we write:

$\|^n P$ for $\underbrace{P \parallel \ldots \parallel P}_{n\times}$

If conditionals are present, the definition of the state operator needs in addition an evaluation function $eval_S$. The additional equation is $\lambda_S(b :\rightarrow P) = eval_S(b) :\rightarrow \lambda_S(P)$. Thus, execution of $P$ is disabled in state $S$ if $b$ evaluates to $\mathsf{false}$ in $S$. The state operator used for the semantics of $\varphi$SDL is a slight adaptation of the state operator described in [1], due to the highly state dependent nature of the SDL mechanisms for storage, communication, timing and process creation. Actions parametrized by domains that are built on expressions denoting values, in contrast with values, are needed as state transforming actions. The reason for this is that, in general, the values concerned depend on the state in which the actions are executed. Consequently, the equations become somewhat more involved than suggested above, as is witnessed by Section 4. The evaluation function $eval_S$ is also needed in these equations.

[5] In [4], a revision of [2], a different notation for $\underline{\underline{a}}$ is used, viz. $\mathsf{cts}(a)$.

The process creation operator used for the semantics of $\varphi$SDL is a slight adaptation of the process creation operator described in [6], due to the following details of the process creation mechanism of SDL:

- formal parameters are local variables and parameter passing amounts to assigning initial values to local variables of a newly created process when its execution starts;
- the pid value of the creating process is passed to a newly created process when its execution starts.

Consequently, the process creation action needs, in addition to the name of a process type, parameters to be used by the state operator described in Section 4. So the defining equations have to be reformulated. This is, however, trivial because these additional parameters of the process creation action are ignored by the process creation operator. For example, the most crucial equation becomes

$$E_\phi(\underline{\underline{cr}}(X, \langle v_1, \ldots, v_n\rangle, \langle u_1, \ldots, u_n\rangle, i) \cdot P) = \\ \underline{\underline{cr}}(X, \langle v_1, \ldots, v_n\rangle, \langle u_1, \ldots, u_n\rangle, i) \cdot E_\phi(\phi(X) \parallel P)$$

where $v_1, \ldots, v_n$ are the formal parameters and $u_1, \ldots, u_n$ are the corresponding actual parameters.

# 4 Processes with states

The input guards, the SDL actions and the terminator stop constitute the SDL mechanisms for storage, communication, timing and process creation. In the process algebra semantics of $\varphi$SDL, which will be presented in Section 5, the state operator mentioned in Section 3 is used to describe these mechanisms in whole or in part. This means that input guards, SDL actions and stop correspond to ACP actions that interact with a global state. In this section, we will describe the state space, the actions that transform states, and the result of executing processes, built up from these actions, in a state from this state space.

## 4.1 Preliminaries

We mentioned before that $\varphi$SDL does not deal with the specification of abstract data types. We assume a fixed algebraic specification covering all data types used and an initial algebra semantics, denoted by $\mathcal{A}$, for it. We will write $Sort_\mathcal{A}$ and $Op_\mathcal{A}$ for the set of all sort names and the set of all operation names, respectively, in the signature of $\mathcal{A}$. We will write $U$ for $\bigcup_{T \in sort(\mathcal{A})} T^\mathcal{A}$, where $T^\mathcal{A}$ is the interpretation of the sort name $T$ in $\mathcal{A}$. We will assume that $\mathsf{nil} \notin U$. In the sequel, we will use for each $op \in Op_\mathcal{A}$ an extension to $U$, also denoted by $op$, such that $op(t_1, \ldots, t_n) = \mathsf{nil}$ if at least one the $t_i$s is not of the appropriate sort. Thus, we can change over from the many-sorted case to the one-sorted case for the description of the meaning of $\varphi$SDL constructs. We can do so without loss of generality, because it can (and should) be statically checked that only terms of appropriate sorts occur.

Uncustomary notation concerning sets, functions and sequences, used in this section, is explained in Appendix A.

## 4.2 Basic domains and functions, the state space

The state space, used to describe the meaning of system definitions, depends upon the specific variables, types of signals, channels and types of processes introduced in the system definition concerned. They largely make up the contextual information extracted from the system definition by means of the function $\{\!|\bullet|\!\}$ defined in Appendix B. For convenience, we define these state space parameters for arbitrary contexts $\kappa$ (the notation concerning contexts introduced in Appendix B is used):

$$\begin{aligned} V_\kappa &= \mathit{vars}(\kappa) \\ S_\kappa &= \mathit{sigs}(\kappa) \\ C_\kappa &= \mathit{chans}(\kappa) \\ P_\kappa &= \mathit{procs}(\kappa) \end{aligned}$$

First, we define the set $\mathit{Sig}_\kappa$ of signals and the set $\mathit{ExtSig}_\kappa$ of extended signals, which fit into the picture of the communication mechanism. A signal consist of the name of its type and the sequence of values that it carries. An extended signal contains, in addition to a signal, the pid values of its sender and receiver. The pid value of the sender is needed seeing that the identity of the sender may otherwise get lost; a delivered signal need not be consumed immediately, but may be put into an input queue instead. In case a signal must pass through a channel, the pid value of the receiver is also essential because of the possible loss of identity due to queueing or delaying.

$$\begin{aligned} \mathit{Sig}_\kappa &= S_\kappa \times U^* \\ \mathit{ExtSig}_\kappa &= \mathit{Sig}_\kappa \times \mathbb{N} \times \mathbb{N} \end{aligned}$$

We write $\mathit{snm}(\mathit{sig})$ and $\mathit{vals}(\mathit{sig})$, where $\mathit{sig} = (s, vs) \in \mathit{Sig}_\kappa$, for $s$ and $vs$, respectively. We write $\mathit{sig}(\mathit{esig})$, where $\mathit{esig} = (\mathit{sig}, i, i') \in \mathit{ExtSig}_\kappa$, for $\mathit{sig}$.

The local state of a process includes a storage which associates local variables with the values assigned to them, an input queue where delivered signals are kept until they are consumed, and a component keeping track of the expiration times of active timers. We define the set $\mathit{Stg}_\kappa$ of storages, the set $\mathit{InpQ}_\kappa$ of input queues and the set $\mathit{Timers}_\kappa$ of timers as follows:

$$\begin{aligned} \mathit{Stg}_\kappa &= \bigcup_{V \subseteq V_\kappa} (V \xrightarrow{\mathit{fin}} U) \\ \mathit{InpQ}_\kappa &= \mathit{ExtSig}_\kappa{}^* \\ \mathit{Timers}_\kappa &= \bigcup_{T \subseteq \mathit{Sig}_\kappa} (T \xrightarrow{\mathit{fin}} \mathbb{N} \cup \{\mathsf{nil}\}) \end{aligned}$$

We will follow the convention that the domain of a function from $\mathit{Stg}_\kappa$ does not contain variables with which no value is associated because a value has never been assigned to them. Consequently, the absence of a value need not to be represented by nil. We will also follow the convention that the domain of a function from

$Timers_\kappa$ contains precisely the active timers. While an expired timer is still active, its former expiration time will be replaced by nil. The basic operations on $Stg_\kappa$ and $Timers_\kappa$ are general operations on functions: function application, overriding ($\oplus$) and domain subtraction ($\lhd$). Overriding and domain subtraction are defined in Appendix A. In so far as the communication mechanism of SDL is concerned, the basic operations on $InpQ_\kappa$ are the functions

$$\begin{array}{ll} getnxt & : InpQ_\kappa \times \mathcal{P}_{fin}(S_\kappa) \to ExtSig_\kappa \cup \{\mathsf{nil}\}, \\ rmvfirst & : InpQ_\kappa \times Sig_\kappa \to InpQ_\kappa, \\ merge & : \mathcal{P}_{fin}(InpQ_\kappa) \to \mathcal{P}_{fin}(InpQ_\kappa) \end{array}$$

defined below. The value of $getnxt(\sigma, ss)$ is the first (extended) signal in $\sigma$ that is of a type different from the ones in $ss$. The value of $rmvfirst(\sigma, sig)$ is the input queue $\sigma$ from which the first occurrence of the signal $sig$ has been removed. Both functions are used to describe the consumption of signals by SDL processes. The function $getnxt$ is recursively defined by

$$\begin{array}{l} getnxt(\langle\,\rangle, ss) = \mathsf{nil} \\ getnxt((sig, i, i') \,\&\, \sigma, ss) = (sig, i, i') \qquad \textbf{if } snm(sig) \notin ss \\ getnxt((sig, i, i') \,\&\, \sigma, ss) = getnxt(\sigma, ss) \textbf{ if } snm(sig) \in ss \end{array}$$

and the function $rmvfirst$ is recursively defined by

$$\begin{array}{l} rmvfirst(\langle\,\rangle, sig) = \langle\,\rangle \\ rmvfirst((sig, i, i') \,\&\, \sigma, sig) = \sigma \\ rmvfirst((sig, i, i') \,\&\, \sigma, sig') = (sig, i, i') \,\&\, rmvfirst(\sigma, sig') \textbf{ if } sig \neq sig' \end{array}$$

For each process, sequences of signals coming from different channels as well as signals noticing timer expiration have to be merged when time progresses to the next time slice. The function $merge$ is used to describe this precisely. It is inductively defined by

$$\begin{array}{l} \sigma \in merge(\{\sigma\}) \\ \langle\,\rangle \in merge(\{\langle\,\rangle, \langle\,\rangle\}) \\ \sigma \in merge(\{\sigma_1, \sigma_2\}) \Rightarrow (sig, i, i') \,\&\, \sigma \in merge(\{(sig, i, i') \,\&\, \sigma_1, \sigma_2\}) \\ \sigma \in merge(\{\sigma_1, \sigma_2\}) \wedge \sigma_2 \in merge(\Sigma) \Rightarrow \sigma \in merge(\{\sigma_1\} \cup \Sigma) \end{array}$$

We define now the set $\mathcal{L}_\kappa$ of local states. The local state of a process contains, in addition to the above-mentioned components, the name of its type. Thus, the type of the process concerned will not get lost. This is important, because a signal may be sent to an arbitrary process of a process type.

$$\mathcal{L}_\kappa \;=\; Stg_\kappa \times InpQ_\kappa \times Timers_\kappa \times P_\kappa$$

We write $stg(L)$, $inpq(L)$, $timers(L)$ and $ptype(L)$, where $L = (\rho, \sigma, \theta, X) \in \mathcal{L}_\kappa$, for $\rho$, $\sigma$, $\theta$ and $X$, respectively.

The global state of a system contains, besides a local state for each existing process, components keeping track of the system time and the pid value issued last, and also a queue for each channel where signals presented to the channel are kept until it is their turn to pass through it. To keep track of the system time and the pid value issued last, natural numbers suffice. We define the set $ChQ_\kappa$ of channel queues as follows:

$$ChQ_\kappa \;=\; (ExtSig_\kappa \times \mathbb{N})^*$$

Each element in a channel queue contains, in addition to an (extended) signal, a natural number presenting the duration of the delay that it experiences when it does pass through the channel; the arbitrary choice between all possible durations of this delay is made before the signal is put into the channel queue – by means of alternative composition. Global states can be transformed by actions as well as by progress of time. As mentioned above, there may be signals leaving channels and entering the input queues of processes when time progresses to the next time slice, and there may be timers expiring and corresponding signals entering the input queues as well. In so far as channels are concerned, the functions that are used to describe this precisely are the following ones:

$$\begin{array}{l} \mathit{unitdelay}\colon ChQ_\kappa \to ChQ_\kappa, \\ \mathit{arriving} \;\colon ChQ_\kappa \times \mathbb{N} \to InpQ_\kappa, \\ \mathit{coming} \;\;\colon ChQ_\kappa \to ChQ_\kappa \end{array}$$

The value of $\mathit{unitdelay}(\gamma)$ is the channel queue $\gamma$ in which the delay duration of the first signal is decreased by one time unit. The value of $\mathit{arriving}(\gamma, i)$ is the longest prefix of $\gamma$ that consists of signals with delay duration zero, weeded of signals with other receivers than $i$ and stripped of delay durations. The value of $\mathit{coming}(\gamma)$ is the longest suffix of $\gamma$ that does not start with a signal with delay duration zero. These functions are used to describe the delivery of signals by channels. The function *unitdelay* is defined by the following equations:

$$\begin{array}{l} \mathit{unitdelay}(\langle\,\rangle) = \langle\,\rangle \\ \mathit{unitdelay}(((sig, i, i'), 0)\;\&\;\gamma) = ((sig, i, i'), 0)\;\&\;\gamma \\ \mathit{unitdelay}(((sig, i, i'), d+1)\;\&\;\gamma) = ((sig, i, i'), d)\;\&\;\gamma \end{array}$$

The function *arriving* and *coming* are recursively defined by

$$\begin{array}{ll} \mathit{arriving}(\langle\,\rangle, i) = \langle\,\rangle & \\ \mathit{arriving}(((sig, i, i'), 0)\;\&\;\gamma, i') = (sig, i, i')\;\&\;\mathit{arriving}(\gamma, i') & \\ \mathit{arriving}(((sig, i, i'), 0)\;\&\;\gamma, j') = \mathit{arriving}(\gamma, j') & \textbf{if } i' \neq j' \\ \mathit{arriving}(((sig, i, i'), d+1)\;\&\;\gamma, j') = \langle\,\rangle & \end{array}$$

$$\begin{array}{l} \mathit{coming}(\langle\,\rangle) = \langle\,\rangle \\ \mathit{coming}(((sig, i, i'), 0)\;\&\;\gamma) = \mathit{coming}(\gamma) \\ \mathit{coming}(((sig, i, i'), d+1)\;\&\;\gamma) = ((sig, i, i'), d+1)\;\&\;\gamma \end{array}$$

We define now a set $\mathcal{M}_\kappa$ of global states which contains proper as well as improper states. Recall that the global state of a system contains a component keeping track of the pid value issued last, a component keeping track of the system time, a channel queue for each channel and a local state for each existing process. The channel queues are indexed by the fixed set of channel names and the local states are indexed by a variable set of pid values, which contains the pid values of the currently existing processes. The improper states are the ones that does not keep the last issued pid value up to date.

$$\mathcal{M}_\kappa \;=\; \mathbb{N} \times \mathbb{N} \times (C_\kappa \xrightarrow{fin} ChQ_\kappa) \times \bigcup_{I \subseteq \mathbb{N}_1} (I \to \mathcal{L}_\kappa)$$

We write $cnt(G)$, $now(G)$, $chs(G)$ and $lsts(G)$, where $G = (c, n, \Gamma, \Sigma) \in \mathcal{M}_\kappa$, for $c$, $n$, $\Gamma$ and $\Sigma$, respectively. Note that the local states are indexed by a subset of $\mathbb{N}_1$. This means that 0 will never serve as the pid value of a process that exists within the system. But 0 is not excluded from being used as a pid value; it is reserved for the environment.

Last, we define the state space $\mathcal{G}_\kappa$:

$$\mathcal{G}_\kappa \;= \{G \in \mathcal{M}_\kappa \mid \forall i \in dom(lsts(G)) \cdot i \leq cnt(G)\}$$

We write $exists(i, G)$, where $i \in \mathbb{N}$ and $G \in \mathcal{G}_\kappa$, for $i \in dom(lsts(G))$. The state space $\mathcal{G}_\kappa$ consists exactly of the proper states in $\mathcal{M}_\kappa$.

## 4.3 Actions

In this subsection, we will introduce the actions that are used for the semantics of $\varphi$SDL. We will make a distinction between the state transforming actions and the actions that do not transform states. For each action $a$ from the latter kind, the action that appears as the result of executing $a$ in a state is always the action $a$ itself; i.e. $\lambda_G(a \cdot P) = a \cdot \lambda_G(P)$. These actions are called *inert* actions.

We mentioned before that we will use actions parametrized by domains that are built on expressions denoting values, in contrast with values, as state transforming actions. These expressions are needed because, in general, the values concerned depend on the state in which the actions are executed. The syntax of these expressions, called value expressions, is as follows:

<vexpr> ::=
  <operator nm> [ ( <vexpr> {, <vexpr>}* ) ]
| *cond* ( <boolean vexpr> , <vexpr> , <vexpr> )
| *value* ( <variable nm> , <pid vexpr> )
| *active* ( <signal nm> [ ( <vexpr> {, <vexpr>}* ) ] )
| *now*
| <value nm>
| <vexpr> = <vexpr>
| *cnt*
| *waiting* ( <signal nm> {, <signal nm>}* , <pid vexpr> )
| *type* ( <pid vexpr> )
| *hasinst* ( <process nm> )

We assume a fixed set of terminal productions of <value nm> including the special value name *self*. We will write $VExpr_\kappa$ for the set of all terminal productions of <vexpr> where the set of terminal productions of <operator nm>, <variable nm>, <signal nm> and <process nm> are $Op_\mathcal{A}$, $V_\kappa$, $S_\kappa$ and $P_\kappa$, respectively. We will write $NExpr_\kappa$ for $\{u \in VExpr_\kappa \mid \forall G \in \mathcal{G}_\kappa \cdot eval_G(u) \in \mathbb{N} \cup \{\mathsf{nil}\}\}$.

The first five cases correspond to operator applications, conditional expressions, view expressions, active expressions and the expression now, respectively,

in SDL. The SDL expressions parent, offspring and sender are regarded as variables accesses, and variable accesses are treated as a special case of view expressions. The sixth case includes *self*, which corresponds to the SDL expressions self.

The remaining five cases are needed to reflect the intended meaning of various other SDL construct exactly. The expression *cnt* is used to associate a unique pid value with each created process. Expressions of the form $waiting(s_1, \ldots, s_n, u)$ are used to give meaning to SDL's state definitions. They are needed to model that signal consumption is not delayed till the next time slice when there is a signal to consume. Expressions of the forms $type(u)$ and $hasinst(X)$ are used to give meaning to SDL's output actions. They are needed to check (dynamically) if a receiver with a given pid value is of the appropriate type for a given signal route and to check if a receiver of the appropriate type for a given signal route exists. Expressions of the form $u_1 = u_2$ are, as a matter of course, used to give meaning to SDL's decisions. Furthermore, they are used with expressions of the form *cnt* or $type(u)$ as left-hand sides where the latter expressions are used.

The state transforming actions are parametrized by several domains that are built on $VExpr_\kappa$:

$$\begin{aligned} SigD_\kappa &= S_\kappa \times {VExpr_\kappa}^* \\ ExtSigD_\kappa &= SigD_\kappa \times NExpr_\kappa \times NExpr_\kappa \\ ExtSigP_\kappa &= (S_\kappa \times {V_\kappa}^*) \times \{\mathsf{nil}\} \times NExpr_\kappa \end{aligned}$$

The domains *SigD* and *ExtSigD* are like *Sig* and *ExtSig*, respectively, but with $U$ and $\mathbb{N}$ replaced by $VExpr_\kappa$ and $NExpr_\kappa$, respectively. The domain *ExtSigP* differs slightly from *ExtSigD*, because it represents signal patterns, with variables used for the unknown values and nil for "don't care".

The following state transforming actions are used:

$$\begin{aligned} input &: ExtSigP_\kappa \times \mathcal{P}_{fin}(S_\kappa) \\ output &: ExtSigD_\kappa \times (C_\kappa \cup \{\mathsf{nil}\}) \times NExpr_\kappa \\ set &: NExpr_\kappa \times SigD_\kappa \times NExpr_\kappa \\ reset &: SigD_\kappa \times NExpr_\kappa \\ ass &: V_\kappa \times VExpr_\kappa \times NExpr_\kappa \\ \overline{cr} &: P_\kappa \times {V_\kappa}^* \times {VExpr_\kappa}^* \times (NExpr_\kappa \cup \{\mathsf{nil}\}) \\ stop &: NExpr_\kappa \\ inispont &: NExpr_\kappa \end{aligned}$$

These are the ACP actions that correspond to input guards, SDL actions, stop and input none. The second parameter of an *input* action is the save set being in force. The third parameter of an *output* action denotes the delay that the signal experiences if it must pass through a channel. The last parameter of the remaining actions denotes the pid value of the process from which the action originates. Recall that the second and third parameter of an $\overline{cr}$ action are the formal parameters and the actual parameters, respectively, of the process to be created. The presence of nil needs some further explanation. The second parameter of an *output* action is a channel if the signal to be sent must pass through a channel, and nil otherwise. The last parameter of a $\overline{cr}$ action is a value expression denoting the pid value of the creating process if it exists, and

nil otherwise – a creating process does not exist for the processes created during system start-up. Similar remarks also apply to the corresponding actions after execution, and to a $cr$ action (see below).

The following inert actions are used:

$$
\begin{array}{ll}
cr & : P_\kappa \times {V_\kappa}^* \times {VExpr_\kappa}^* \times (NExpr_\kappa \cup \{\mathsf{nil}\}) \\
input' & : ExtSig_\kappa \times {V_\kappa}^* \\
output' & : ExtSig_\kappa \times (C_\kappa \cup \{\mathsf{nil}\}) \\
set' & : \mathbb{N} \times Sig_\kappa \times \mathbb{N} \\
reset' & : Sig_\kappa \times \mathbb{N} \\
ass' & : V_\kappa \times U \times \mathbb{N} \\
cr' & : P_\kappa \times {V_\kappa}^* \times U^* \times (\mathbb{N} \cup \{\mathsf{nil}\}) \\
stop' & : \mathbb{N} \\
\mathbb{t} & :
\end{array}
$$

They do not transform states. They are the actions that appear as the result of executing a state transforming action, except for $cr$. The instances of $cr$ are used for process creation, leaving instances of $\overline{cr}$ as a trace. The action $\mathbb{t}$ is a special action with no observable effect whatsoever. It appears as the result of executing an instance of *inispont* as well as during system start-up as explained in Section 5.2.

The second parameter of a create action ($cr$, $\overline{cr}$ or $cr'$) is the sequence of formal parameters for the relevant process type. This is convenient in two ways. Firstly, the alternative to make the association between process types and their formal parameters itself a parameter of the state operator is very unattractive. Secondly, that association is not fully immutable. Recall that the formal parameters are variables and that parameter passing amounts to assigning initial values to these variables – as part of a process creation action. During the start-up of the system, such values are not available and no parameter passing takes place, which correponds to a different association between process types and formal parameters. This can simply be accomplished in the approach adopted here by using an empty sequence.

## 4.4 State transformers and observers

In the process algebra semantics of $\varphi$SDL, which will be presented in Section 5, ACP actions that transform states from $\mathcal{G}_\kappa$ are used to describe the meaning of input guards, SDL actions and stop. State transforming actions are also needed to initiate spontaneous transitions (indicated by input none). In the next subsection, we will define the result of executing a process, built up from these actions, in a state from $\mathcal{G}_\kappa$. That is, we will define the relevant state operator. This will, for the most part, boil down to describing how the actions, and the progress of time (modelled by the delay operator $\sigma_{\mathsf{rel}}$), transform states. For the sake of comprehensibility, we will first define matching state transforming operations, and also some state observing operations.

A few of the state observing operations are used directly to define the state operator; the others are used to define the evaluation function for the expressions

being used in case the values concerned depend on a state. First of all, these expressions are needed as constituents of the actions because the values concerned depend on the state in which these actions are executed. Besides, they are needed as conditions to describe processes that may proceed conditionally, dependent on the state in which they are executed. Various SDL constructs, as a matter of course including decisions, give rise to such processes. In the next subsection, we will define, in addition to the state operator, the above-mentioned evaluation function.

**State transformers:**

In general, the state transformers change one or two components of the local state of one process. The notable exception is *rcvsig*, which is defined first. It may change all components except the process type. This is a consequence of the fact that the storage, communication and timing mechanisms are rather intertwined on the consumption of signals in SDL. For each state transformer it holds that everything remains unchanged if an attempt is made to transform the local state of a non-existing process. This will not be explicitly mentioned in the explanations given below.

The function $rcvsig : ExtSig_\kappa \times {V_\kappa}^* \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's input guards transform states.

$$
\begin{array}{l}
rcvsig((sig, i, i'), \langle v_1, \ldots, v_n \rangle, G) \;=\; \\
\quad \begin{array}{ll}
(cnt(G), now(G), chs(G), lsts(G) \oplus \{i' \mapsto (\rho, \sigma, \theta, X)\}) & \textbf{if } exists(i', G) \\
G & \textbf{otherwise}
\end{array} \\
\\
\text{where } \begin{array}{rcl}
\rho & = & stg(lsts(G)_{i'}) \oplus \{v_1 \mapsto vals(sig)_1, \ldots, v_n \mapsto vals(sig)_n, \mathsf{sender} \mapsto i\}, \\
\sigma & = & rmvfirst(inpq(lsts(G)_{i'}), sig), \\
\theta & = & \{sig\} \lhd timers(lsts(G)_{i'}), \\
X & = & ptype(lsts(G)_{i'})
\end{array}
\end{array}
$$

$rcvsig((sig, i, i'), \langle v_1, \ldots, v_n \rangle, G)$ deals with the consumption of signal *sig* sent from $i$ to $i'$. It transforms the local state of the receiver as follows:

- the values carried by *sig* are assigned to the local variables $v_1, \ldots, v_n$ of the receiver and the sender's pid value ($i$) is assigned to sender;
- the first occurrence of *sig* in the input queue of the receiver is removed;
- if *sig* is a timer signal, it is removed from the active timers.

Everything else is left unchanged.

The function $sndsig : ExtSig_\kappa \times (C_\kappa \cup \{\mathsf{nil}\}) \times \mathbb{N} \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's output actions transform states.

$$
\begin{array}{l}
sndsig((sig, i, i'), c, d, G) \;=\; \\
\quad \begin{array}{ll}
(cnt(G), now(G), chs(G), lsts(G) \oplus \{i' \mapsto (\rho, \sigma, \theta, X)\}) & \\
\qquad & \textbf{if } exists(i', G) \wedge (c = \mathsf{nil} \vee (chs(G)_c = \langle\,\rangle \wedge d = 0)) \\
(cnt(G), now(G), chs(G) \oplus \{c \mapsto \gamma\}, lsts(G)) & \\
\qquad & \textbf{if } \neg(c = \mathsf{nil} \vee (chs(G)_c = \langle\,\rangle \wedge d = 0)) \\
G & \textbf{otherwise}
\end{array}
\end{array}
$$

$$
\begin{array}{lcl}
\text{where } \rho & = & \mathit{stg}(\mathit{lsts}(G)_{i'}), \\
\sigma & = & \mathit{inpq}(\mathit{lsts}(G)_{i'}) \frown \langle(\mathit{sig}, i, i')\rangle, \\
\theta & = & \mathit{timers}(\mathit{lsts}(G)_{i'}), \\
X & = & \mathit{ptype}(\mathit{lsts}(G)_{i'}), \\
\gamma & = & \mathit{chs}(G)_c \frown \langle((\mathit{sig}, i, i'), d)\rangle
\end{array}
$$

$\mathit{sndsig}((\mathit{sig}, i, i'), c, d, G)$ deals with passing signal $\mathit{sig}$ from $i$ to $i'$, through channel $c$ with a delay $d$ if $c \neq \mathsf{nil}$. If $c = \mathsf{nil}$, or the queue of $c$ is empty and $d = 0$, it transforms the local state of the receiver as follows:

- $\mathit{sig}$ is put into the input queue of the receiver, unless $i' = 0$ (indicating that the environment is the receiver of the signal).

Otherwise, it transforms the queue of the delaying channel as follows:

- $\mathit{sig}$ is put into the queue of the delaying channel.

Everything else is left unchanged.

The function $\mathit{settimer} : \mathbb{N} \times \mathit{Sig}_\kappa \times \mathbb{N} \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's set actions transform states.

$$
\begin{array}{l}
\mathit{settimer}(t, \mathit{sig}, i, G) \; = \\
\quad \begin{array}{ll}
(\mathit{cnt}(G), \mathit{now}(G), \mathit{chs}(G), \mathit{lsts}(G) \oplus \{i \mapsto (\rho, \sigma, \theta, X)\}) & \textbf{if } \mathit{exists}(i, G) \\
G & \textbf{otherwise}
\end{array} \\
\\
\begin{array}{lcll}
\text{where } \rho & = & \mathit{stg}(\mathit{lsts}(G)_i), & \\
\sigma & = & \mathit{rmvfirst}(\mathit{inpq}(\mathit{lsts}(G)_i), \mathit{sig}) & \textbf{if } t > \mathit{now}(G) \\
 & & \mathit{rmvfirst}(\mathit{inpq}(\mathit{lsts}(G)_i), \mathit{sig}) \frown \langle(\mathit{sig}, i, i)\rangle & \textbf{otherwise}, \\
\theta & = & \mathit{timers}(\mathit{lsts}(G)_i) \oplus \{\mathit{sig} \mapsto t\} & \textbf{if } t > \mathit{now}(G) \\
 & & \mathit{timers}(\mathit{lsts}(G)_i) \oplus \{\mathit{sig} \mapsto \mathsf{nil}\} & \textbf{otherwise}, \\
X & = & \mathit{ptype}(\mathit{lsts}(G)_i) &
\end{array}
\end{array}
$$

$\mathit{settimer}(t, \mathit{sig}, i, G)$ deals with setting a timer, identified with signal $\mathit{sig}$, to time $t$. If $t$ has not yet passed, it transforms the local state of the process with pid value $i$, the process to be notified of the timer's expiration, as follows:

- the occurrence of $\mathit{sig}$ in the input queue originating from an earlier setting, if any, is removed;
- $\mathit{sig}$ is included among the active timers with expiration time $t$; thus overriding an earlier setting, if any.

Otherwise, it transforms the local state of the process with pid value $i$ as follows:

- $\mathit{sig}$ is put into the input queue after removal of its occurrence originating from an earlier setting, if any;
- $\mathit{sig}$ is included among the active timers without expiration time.

Everything else is left unchanged.

The function $\mathit{resettimer} : \mathit{Sig}_\kappa \times \mathbb{N} \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's reset actions transform states.

$$
\begin{array}{l}
\mathit{resettimer}(\mathit{sig}, i, G) \;=\; \\
\quad \begin{array}{ll}
(\mathit{cnt}(G), \mathit{now}(G), \mathit{chs}(G), \mathit{lsts}(G) \oplus \{i \mapsto (\rho, \sigma, \theta, X)\}) & \textbf{if } \mathit{exists}(i, G) \\
G & \textbf{otherwise}
\end{array} \\
\text{where } \rho \;=\; \mathit{stg}(\mathit{lsts}(G)_i), \\
\qquad\quad \sigma \;=\; \mathit{rmvfirst}(\mathit{inpq}(\mathit{lsts}(G)_i), \mathit{sig}), \\
\qquad\quad \theta \;=\; \{\mathit{sig}\} \lhd \mathit{timers}(\mathit{lsts}(G)_i), \\
\qquad\quad X \;=\; \mathit{ptype}(\mathit{lsts}(G)_i)
\end{array}
$$

$\mathit{resettimer}(\mathit{sig}, i, G)$ deals with resetting a timer, identified with signal $\mathit{sig}$. It transforms the local state of the process with pid value $i$, the process that would otherwise have been notified of the timer's expiration, as follows:

- the occurrence of $\mathit{sig}$ in the input queue originating from an earlier setting, if any, is removed;
- if $\mathit{sig}$ is an active timer, it is removed from the active timers.

Everything else is left unchanged.
Notice that $\mathit{settimer}(t, \mathit{sig}, i, G)$ and $\mathit{settimer}(t, \mathit{sig}, i, \mathit{resettimer}(\mathit{sig}, i, G))$ have the same effect. In other words, *settimer* resets implicitly. In this way, at most one signal from the same timer will ever occur in an input queue. Furthermore, SDL keeps timer signals and other signals apart: not a single signal can originate from both timer setting and customary signal sending. Thus, resetting, either explicitly or implicitly, will solely remove signals from input queues that originate from timer setting.

The function $\mathit{assignvar} : V_\kappa \times U \times \mathbb{N} \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's assignment task actions transform states.

$$
\begin{array}{l}
\mathit{assignvar}(v, t, i, G) \;=\; \\
\quad \begin{array}{ll}
(\mathit{cnt}(G), \mathit{now}(G), \mathit{chs}(G), \mathit{lsts}(G) \oplus \{i \mapsto (\rho, \sigma, \theta, X)\}) & \textbf{if } \mathit{exists}(i, G) \\
G & \textbf{otherwise}
\end{array} \\
\text{where } \rho \;=\; \mathit{stg}(\mathit{lsts}(G)_i) \oplus \{v \mapsto t\}, \\
\qquad\quad \sigma \;=\; \mathit{inpq}(\mathit{lsts}(G)_i), \\
\qquad\quad \theta \;=\; \mathit{timers}(\mathit{lsts}(G)_i), \\
\qquad\quad X \;=\; \mathit{ptype}(\mathit{lsts}(G)_i)
\end{array}
$$

$\mathit{assignvar}(v, t, i, G)$ deals with assigning value $t$ to variable $v$. It transforms the local state of the process with pid value $i$, the process to which the variable is local, as follows:

- $t$ is assigned to the local variable $v$, i.e. $v$ is included among the variables in the storage with value $t$; thus overriding an earlier assignment, if any.

Everything else is left unchanged.

The function $\mathit{createproc} : P_\kappa \times {V_\kappa}^* \times U^* \times (\mathbb{N} \cup \{\mathsf{nil}\}) \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's create actions transform states.

$$
\begin{array}{l}
\mathit{createproc}(X, \langle v_1, \ldots, v_n \rangle, \langle t_1, \ldots, t_n \rangle, i, G) \;= \\
\quad (\mathit{cnt}(G) + 1, \mathit{now}(G), \mathit{chs}(G), \\
\quad\; \mathit{lsts}(G) \oplus \{\mathit{cnt}(G) + 1 \mapsto (\rho, \sigma, \theta, X), i \mapsto (\rho', \sigma', \theta', X')\}) \;\textbf{if}\; \mathit{exists}(i, G) \\
\quad (\mathit{cnt}(G) + 1, \mathit{now}(G), \mathit{chs}(G), \\
\quad\; \mathit{lsts}(G) \oplus \{\mathit{cnt}(G) + 1 \mapsto (\rho, \sigma, \theta, X)\}) \qquad\qquad\qquad\qquad \textbf{if}\; i = \mathsf{nil} \\
\quad G \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
\text{where } \rho & = & \{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n, \mathsf{parent} \mapsto i\}, \\
\sigma & = & \langle\,\rangle, \\
\theta & = & \{\,\}, \\
\rho' & = & \mathit{stg}(\mathit{lsts}(G)_i) \oplus \{\mathsf{offspring} \mapsto \mathit{cnt}(G) + 1\}, \\
\sigma' & = & \mathit{inpq}(\mathit{lsts}(G)_i), \\
\theta' & = & \mathit{timers}(\mathit{lsts}(G)_i), \\
X' & = & \mathit{ptype}(\mathit{lsts}(G)_i)
\end{array}
$$

$\mathit{createproc}(X, \langle v_1, \ldots, v_n \rangle, \langle t_1, \ldots, t_n \rangle, i, G)$ deals with creating a process of type $X$. It increments the last issued pid value – which will be used as the pid value of the created process. In addition, it transforms the local state of the process with pid value $i$, the parent of the created process, as follows:

- the pid value of the created process is assigned to offspring.

Besides, it creates a new local state for the created process which is initiated as follows:

- the values $t_1, \ldots, t_n$ are assigned to the local variables $v_1, \ldots, v_n$ of the created process and the parent's pid value ($i$) is assigned to parent;
- $X$ is made the process type.

Everything else is left unchanged.

The function $\mathit{stopproc} : \mathbb{N} \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's stop transform states.

$$
\mathit{stopproc}(i, G) \;= (\mathit{cnt}(G), \mathit{now}(G), \mathit{chs}(G), \{i\} \lhd \mathit{lsts}(G))
$$

$\mathit{stopproc}(i, G)$ deals with terminating the process with pid value $i$. It disposes of the local state of the process with pid value $i$. Everything else is left unchanged.

The function $\mathit{inispont} : \mathbb{N} \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how ACP actions used to initiate spontaneous transitions transform states.

$$
\begin{array}{l}
\mathit{inispont}(i, G) \;= \\
\quad (\mathit{cnt}(G), \mathit{now}(G), \mathit{chs}(G), \mathit{lsts}(G) \oplus \{i \mapsto (\rho, \sigma, \theta, X)\}) \;\textbf{if}\; \mathit{exists}(i, G) \\
\quad G \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
\text{where } \rho & = & \mathit{stg}(\mathit{lsts}(G)_i) \oplus \{\mathsf{sender} \mapsto i\}, \\
\sigma & = & \mathit{inpq}(\mathit{lsts}(G)_i), \\
\theta & = & \mathit{timers}(\mathit{lsts}(G)_i), \\
X & = & \mathit{ptype}(\mathit{lsts}(G)_i)
\end{array}
$$

$\mathit{inispont}(i, G)$ deals with initiating spontaneous transitions. It transforms the local state of the process with pid value $i$, the process for which a spontaneous transition is initiated, by assigning $i$ to sender. Everything else is left unchanged.

The function $unitdelay : \mathcal{G}_\kappa \to \mathcal{P}_{fin}(\mathcal{G}_\kappa)$ is used to describe how progress of time transforms states. In general, these transformations are non-deterministic – how signals from channels and expiring timers enter input queues is not uniquely determined. Therefore, this function yields for each state a set of possible states.

$$
\begin{array}{l}
G' \in unitdelay(G) \Leftrightarrow \\
\quad cnt(G') = cnt(G) \wedge \\
\quad now(G') = now(G) + 1 \wedge \\
\quad \forall c \in dom(chs(G)) \cdot chs(G')_c = coming(unitdelay(chs(G)_c)) \wedge \\
\quad \forall i \in dom(lsts(G)) \cdot \\
\qquad stg(lsts(G')_i) = stg(lsts(G)_i) \wedge \\
\qquad (\exists \sigma \in InpQ \cdot \\
\qquad\quad inpq(lsts(G')_i) = inpq(lsts(G)_i) \frown \sigma \wedge \\
\qquad\quad \sigma \in merge(\{arriving(unitdelay(chs(G)_c), i) \mid c \in dom(chs(G))\} \cup \\
\qquad\qquad\qquad \{\langle (sig, i, i) \rangle \mid timers(lsts(G)_i)(sig) \leq now(G)\})) \wedge \\
\qquad timers(lsts(G')_i) = \\
\qquad\quad timers(lsts(G)_i) \oplus \{sig \mapsto \mathsf{nil} \mid timers(lsts(G)_i)(sig) \leq now(G)\} \wedge \\
\qquad ptype(lsts(G')_i) = ptype(lsts(G)_i)
\end{array}
$$

$unitdelay(G)$ transforms the global state as follows:

- the last issued pid value is left unchanged;
- the system time is incremented with one unit;
- for each channel, the signals leaving the channel within one time unit are removed from its queue;
- for the local state of each process:
  - its storage is left unchanged;
  - the signals leaving any channel within one time unit and having the process as receiver, as well as the signals that notify expiration of any of its timers within one time unit, are put into its input queue in a merging, order preserving, manner;
  - for each of its timers that expire within one time unit, the expiration time is removed;
  - its process type is left unchanged.

**State observers:**

In general, the state observers examine one component of the local state of one process. The only exception is *has-instance*, which may even examine the process type component of all processes. If an attempt is made to observe the local state of a non-existing process, each non-boolean-valued state observer yields nil and each boolean-valued state observer yields false. This will not be explicitly mentioned in the explanations given below. The functions $nxtsig : \mathcal{P}_{fin}(S_\kappa) \times \mathbb{N} \times \mathcal{G}_\kappa \to ExtSig_\kappa \cup \{\mathsf{nil}\}$ and $nxtsignm : \mathcal{P}_{fin}(S_\kappa) \times \mathbb{N} \times \mathcal{G}_\kappa \to S_\kappa \cup \{\mathsf{nil}\}$ are used to define the result of executing ACP actions corresponding to SDL's input guards in a state.

$$nxtsig(ss, i, G) \;=\; \begin{array}{ll} getnxt(inpq(lsts(G)_i), ss) & \textbf{if } exists(i, G) \\ \mathsf{nil} & \textbf{otherwise} \end{array}$$

$nxtsig(ss, i, G)$ yields the first signal in the input queue of the process with pid value $i$ that is of a type different from the ones in $ss$.

$$nxtsignm(ss, i, G) \;=\; \begin{array}{ll} snm(sig(nxtsig(ss, i, G))) & \textbf{if } nxtsig(ss, i, G) \neq \mathsf{nil} \\ \mathsf{nil} & \textbf{otherwise} \end{array}$$

$nxtsignm(ss, i, G)$ yields the type of the first signal in the input queue of the process with pid value $i$ that is of a type different from the ones in $ss$.

The function $contents : V_\kappa \times \mathbb{N} \times \mathcal{G}_\kappa \to U \cup \{\mathsf{nil}\}$ is used to describe the value of expressions of the form $value(v, u)$ which correspond to SDL's variable accesses and view expressions.

$$contents(v, i, G) \;=\; \begin{array}{ll} \rho(v) & \textbf{if } exists(i, G) \wedge v \in dom(\rho) \\ \mathsf{nil} & \textbf{otherwise} \end{array}$$

where $\rho \;=\; stg(lsts(G)_i)$

$contents(v, i, G)$ yields the current value of the variable $v$ that is local to the process with pid value $i$.

The function $is\text{-}active : Sig_\kappa \times \mathbb{N} \times \mathcal{G}_\kappa \to \mathbb{B}$ is used to describe the value of expressions of the form $active(sig, u)$ which correspond to SDL's active expressions.

$$is\text{-}active(sig, i, G) \;=\; \begin{array}{ll} \mathsf{true} & \textbf{if } exists(i, G) \wedge sig \in dom(timers(lsts(G)_i)) \\ \mathsf{false} & \textbf{otherwise} \end{array}$$

$is\text{-}active(sig, i, G)$ yields true iff $sig$ is an active timer signal of the process with pid value $i$.

The function $is\text{-}waiting : \mathcal{P}_{fin}(S_\kappa) \times \mathbb{N} \times \mathcal{G}_\kappa \to \mathbb{B}$ is used to describe the value of expressions of the form $waiting(s_1, \ldots, s_n, u)$ which are used to give meaning to SDL's state definitions.

$$is\text{-}waiting(ss, i, G) \;=\; \begin{array}{ll} \mathsf{true} & \textbf{if } exists(i, G) \wedge nxtsig(ss, i, G) \neq \mathsf{nil} \\ \mathsf{false} & \textbf{otherwise} \end{array}$$

$is\text{-}waiting(ss, i, G)$ yields true iff there is a signal in the input queue of the process with pid value $i$ that is of a type different from the ones in $ss$.

The function $type : \mathbb{N} \times \mathcal{G}_\kappa \to P_\kappa \cup \{\mathsf{env}, \mathsf{nil}\}$ is used to describe the value of expressions of the form $type(u)$ which are used to give meaning to SDL's output actions with explicit addressing.

$$type(i, G) \;=\; \begin{array}{ll} ptype(lsts(G)_i) & \textbf{if } exists(i, G) \\ \mathsf{env} & \textbf{if } i = 0 \\ \mathsf{nil} & \textbf{otherwise} \end{array}$$

$type(i, G)$ yields the type of the process with pid value $i$. Different from the other state observers, it yields a result if $i = 0$ as well, viz. $\mathsf{env}$.

The function $has\text{-}instance : (P_\kappa \cup \{\mathsf{env}\}) \times \mathcal{G}_\kappa \to \mathbb{B}$ is used to describe the value of expressions of the form $hasinst(X)$, where $X$ is a process name, which are used to give meaning to SDL's output actions with implicit addressing.

$$\mathit{has\text{-}instance}(X, G) = \mathsf{true} \textbf{ if } \exists i \in \mathbb{N} \cdot (i = 0 \vee \mathit{exists}(i, G)) \wedge \mathit{type}(i, G) = X$$
$$\mathsf{false} \textbf{ otherwise}$$

*has-instance*$(X, G)$ yields true iff there exists a process of type $X$.

## 4.5 State operator and evaluation function

In this subsection, we will finally define the state operator that is used to describe, in whole or in part, the SDL mechanisms for storage, communication, timing and process creation. We will not define the *action* and *effect* functions explicitly, as in [1]. Instead we will define, for each state transforming action $a$, the result of executing a process of the form $a \cdot P$ in a state from $\mathcal{G}_\kappa$.[6] Because progress of time transforms states as well, we will also define the result of executing a process of the form $\sigma_{\mathsf{rel}}(P)$ in a state. In addition, we will define the evaluation function that is used to describe the value of an expression $u$ in a state $G$.

**State operator:**

The state transformers defined in Section 4.4 are used below to describe the state $G'$ resulting from executing a state transforming action $a$ in a state $G$. In general, the action $a'$ that appears as the result of executing a state transforming action $a$ in a state $G$ is the action $a$ with the expressions occurring in it replaced by their values in state $G$. However, there are exception to this rule for the input actions and the output actions. For output actions, the delay duration is additionally stripped of. Input actions deviate more. The constituents of an input action $a$ are a pattern of an (extended) signal and a set of signal types, and the constituents of the corresponding action $a'$ are a signal matching this pattern and the sequence of variables occurring in the pattern. That a signal pattern is replaced by a matching signal is to be expected, the sequence of variables is added because it shows to which variables the values carried by the signal have been assigned, and the set of signal types is removed because there is no use to retain it after execution. There is still another exception for the actions used to initiate spontaneous transitions. As mentioned before, $\not t$ appears as the result of executing these actions.

We will first define the result of executing a process of the form $a \cdot P$ in a state from $\mathcal{G}_\kappa$ for the state transforming ACP actions corresponding to SDL's input guards, output actions, set actions, reset actions, assignment task actions, create actions and the terminator stop, and for the state transforming ACP actions of the form $\overline{\overline{\mathit{inispont}}}(u)$ which will be used to set sender properly when spontaneous transitions take place. All this is rather straightforward with the state transformers defined in Section 4.4; only the case of the ACP actions corresponding to SDL's input guards needs further explanation. If the value of at least one of the expressions occurring in an ACP action is undefined in the state concerned,

[6] We follow the convention that, for each equation $\lambda_G(a \cdot P) = a' \cdot \lambda_{G'}(P)$, the equation $\lambda_G(a) = a'$ is implicit.

the action will fail, i.e. yield deadlock. Different from the other cases, the execution of an action $\underline{\underline{input}}(((s, \langle v_1, \ldots, v_n \rangle), \mathsf{nil}, u), ss)$ may fail in certain states for other reasons as well. It fails if the type of the first signal in the input queue of the process referred to by $u$ with a type not occurring in $ss$ is different from $s$. Otherwise, it succeeds, the values carried by this signal are assigned to the local variables $v_1, \ldots, v_n$ of the process concerned, and the signal is removed from the input queue.

$$\lambda_G(\underline{\underline{input}}(((s, \langle v_1, \ldots, v_n \rangle), \mathsf{nil}, u'), ss) \cdot P) \;=\; \underline{\underline{\underline{input'}}}(nxt, \langle v_1, \ldots, v_n \rangle) \cdot \lambda_{rcvsig(nxt, \langle v_1, \ldots, v_n \rangle, G)}(P) \;\textbf{if}\; i' \neq \mathsf{nil} \wedge nxtsignm(ss, i', G) = s \;\underline{\underline{\delta}}\; \textbf{otherwise}$$

where $nxt = nxtsig(ss, i', G)$,
$i' = eval_G(u')$

$$\lambda_G(\underline{\underline{output}}(((s, \langle u_1, \ldots, u_n \rangle), u, u'), c, u'') \cdot P) \;=\; \underline{\underline{\underline{output'}}}((sig, i, i'), c) \cdot \lambda_{sndsig((sig,i,i'),c,d,G)}(P) \;\textbf{if}\; t_1 \neq \mathsf{nil} \wedge \ldots \wedge t_n \neq \mathsf{nil} \wedge i \neq \mathsf{nil} \wedge i' \neq \mathsf{nil} \wedge d \neq \mathsf{nil} \;\underline{\underline{\delta}}\; \textbf{otherwise}$$

where $sig = (s, \langle t_1, \ldots, t_n \rangle)$,
$t_j = eval_G(u_j)$ ( for $1 \leq j \leq n$),
$i = eval_G(u)$,
$i' = eval_G(u')$,
$d = eval_G(u'')$

$$\lambda_G(\underline{\underline{set}}(u, (s, \langle u_1, \ldots, u_n \rangle), u') \cdot P) \;=\; \underline{\underline{\underline{set'}}}(t, sig, i) \cdot \lambda_{settimer(t,sig,i,G)}(P) \;\textbf{if}\; t \neq \mathsf{nil} \wedge t_1 \neq \mathsf{nil} \wedge \ldots \wedge t_n \neq \mathsf{nil} \wedge i \neq \mathsf{nil} \;\underline{\underline{\delta}}\; \textbf{otherwise}$$

where $t = eval_G(u)$,
$sig = (s, \langle t_1, \ldots, t_n \rangle)$,
$t_j = eval_G(u_j)$ ( for $1 \leq j \leq n$),
$i = eval_G(u')$

$$\lambda_G(\underline{\underline{reset}}((s, \langle u_1, \ldots, u_n \rangle), u') \cdot P) \;=\; \underline{\underline{\underline{reset'}}}(sig, i) \cdot \lambda_{resettimer(sig,i,G)}(P) \;\textbf{if}\; t_1 \neq \mathsf{nil} \wedge \ldots \wedge t_n \neq \mathsf{nil} \wedge i \neq \mathsf{nil} \;\underline{\underline{\delta}}\; \textbf{otherwise}$$

where $sig = (s, \langle t_1, \ldots, t_n \rangle)$,
$t_j = eval_G(u_j)$ ( for $1 \leq j \leq n$),
$i = eval_G(u')$

$$\lambda_G(\underline{\underline{ass}}(v, u, u') \cdot P) \;=\; \underline{\underline{\underline{ass'}}}(v, t, i) \cdot \lambda_{assignvar(v,t,i,G)}(P) \;\textbf{if}\; t \neq \mathsf{nil} \wedge i \neq \mathsf{nil} \;\underline{\underline{\delta}}\; \textbf{otherwise}$$

where $t = eval_G(u)$,
$i = eval_G(u')$

$$\lambda_G(\underline{\underline{cr}}(X, fpars, \langle u_1, \ldots, u_n\rangle, u) \cdot P) \;=\; \begin{cases} \underline{\underline{cr'}}(X, fpars, apars, i) \cdot \lambda_{createproc(X,fpars,apars,i,G)}(P) & \textbf{if } t_1 \neq \mathsf{nil} \wedge \ldots \wedge t_n \neq \mathsf{nil} \\ \underline{\underline{\delta}} & \textbf{otherwise} \end{cases}$$

where $apars = \langle t_1, \ldots, t_n\rangle$,
$t_j = eval_G(u_j)$ ( for $1 \leq j \leq n$),
$i = eval_G(u)$

$$\lambda_G(\underline{\underline{\underline{stop}}}(u) \cdot P) \;=\; \begin{cases} \underline{\underline{\underline{stop'}}}(i) \cdot \lambda_{stopproc(i,G)}(P) & \textbf{if } i \neq \mathsf{nil} \\ \underline{\underline{\delta}} & \textbf{otherwise} \end{cases}$$

where $i = eval_G(u)$

$$\lambda_G(\underline{\underline{\underline{inispont}}}(u) \cdot P) \;=\; \begin{cases} \underline{\underline{\boldsymbol{t}}} \cdot \lambda_{inispont(i,G)}(P) & \textbf{if } i \neq \mathsf{nil} \\ \underline{\underline{\delta}} & \textbf{otherwise} \end{cases}$$

where $i = eval_G(u)$

Here $eval_G$ is used to describe the value of expressions, occurring in a state transforming action, in state $G$. This evaluation function will be defined later on.

Recall that for each inert action $a$, we simply have

$$\lambda_G(a \cdot P) = a \cdot \lambda_G(P)$$

We will now proceed with defining the result of executing a process of the form $\sigma_{\mathsf{rel}}(P)$ in a state from $\mathcal{G}_\kappa$. This case is quite different from the preceding ones. Executing a process that is delayed till the next time slice in some state means that the execution is delayed till the next time slice and, in general, that it takes place in another state due to the progress of time. Usually, it is not uniquely determined how progress of time transforms states. This leads to the following equation:

$$\lambda_G(\sigma_{\mathsf{rel}}(P)) \;=\; \sigma_{\mathsf{rel}}(\textstyle\sum_{G' \in unitdelay(G)} \lambda_{G'}(P))$$

**Evaluation function:**

We will end this section with defining the evaluation function that was already used to describe the value of an expression $u$ in a state $G$. Most state observers defined in Section 4.4 are used to define this function. If the value of at least one of the subexpressions occurring in an expression is undefined in the state concerned, the expression will be undefined, i.e. yield $\mathsf{nil}$.

The SDL expressions are covered by the first six cases, as explained in Section 4.3. These cases do not need any further explanation except the remark that the meta-variable $x$ ranges over a set of variables in the sense of $\mu$CRL that includes *self*, a special variable corresponding to the SDL expression $\mathsf{self}$.

$$eval_G(op(u_1, \ldots, u_n)) \;=\; \begin{cases} op(eval_G(u_1), \ldots, eval_G(u_n)) & \textbf{if } eval_G(u_1) \neq \mathsf{nil} \wedge \ldots \wedge eval_G(u_n) \neq \mathsf{nil} \\ \mathsf{nil} & \textbf{otherwise} \end{cases}$$

$$eval_G(cond(u_1,u_2,u_3)) \;=\; \begin{array}{ll} eval_G(u_2) & \textbf{if } eval_G(u_1) = \mathsf{true} \\ eval_G(u_3) & \textbf{if } eval_G(u_1) = \mathsf{false} \\ \mathsf{nil} & \textbf{otherwise} \end{array}$$

$$eval_G(value(v,u)) \;=\; \begin{array}{ll} contents(v, eval_G(u), G) & \textbf{if } eval_G(u) \neq \mathsf{nil} \\ \mathsf{nil} & \textbf{otherwise} \end{array}$$

$$eval_G(active((s,\langle u_1,\ldots,u_n\rangle),u)) \;=\; \begin{array}{ll} is\text{-}active(sig, eval_G(u), G) & \textbf{if } eval_G(u_1) \neq \mathsf{nil} \wedge \ldots \wedge eval_G(u_n) \neq \mathsf{nil} \wedge \\ & \quad eval_G(u) \neq \mathsf{nil} \\ \mathsf{nil} & \textbf{otherwise} \end{array}$$

where $sig \;=\; (s, \langle eval_G(u_1), \ldots, eval_G(u_n)\rangle)$

$$eval_G(now) \;=\; now(G)$$

$$eval_G(x) \;=\; x$$

The remaining cases are about expressions which are used in Section 5, as explained in Section 4.3 as well. They are very straightforward.

$$eval_G(u_1 = u_2) \;=\; \begin{array}{ll} \mathsf{true} & \textbf{if } eval_G(u_1) = eval_G(u_2) \\ \mathsf{false} & \textbf{if } eval_G(u_1) \neq eval_G(u_2) \\ \mathsf{nil} & \textbf{otherwise} \end{array}$$

$$eval_G(cnt) \;=\; cnt(G)$$

$$eval_G(waiting(s_1,\ldots,s_n,u)) \;=\; \begin{array}{ll} is\text{-}waiting(\{s_1,\ldots,s_n\}, eval_G(u), G) & \textbf{if } eval_G(u) \neq \mathsf{nil} \\ \mathsf{nil} & \textbf{otherwise} \end{array}$$

$$eval_G(type(u)) \;=\; \begin{array}{ll} type(eval_G(u), G) & \textbf{if } eval_G(u) \neq \mathsf{nil} \\ \mathsf{nil} & \textbf{otherwise} \end{array}$$

$$eval_G(hasinst(X)) \;=\; has\text{-}instance(X, G)$$

# 5 Process algebra semantics

In this section, we will present a process algebra semantics of $\varphi$SDL. It relies heavily upon the specifics of the state operator defined in Section 4.5. Here, all peculiar details of the semantics, inherited from full SDL, become visible.

The semantics of $\varphi$SDL is defined by interpretation functions, one for each syntactic category, which are all written in the form $[\![\ ]\!]^\kappa$. The superscript $\kappa$ is used to provide contextual information where required. The exact interpretation function is always clear from the context. We will be lazy about specifying the range of each interpretation function, since this is usually clear from the context as well. Many of the interpretations are expressions, equations, etc. They will simply be written in their display form. We will in addition assume that the interpretation of a name is the same name.

## 5.1 System definition

The meaning of a system definition is a quadruple $(P, \phi, E, \mathcal{G})$ where:

- $P$ is a process expression describing the behaviour of the system from its start-up;
- $\phi$ is the mapping from process names to process expressions that is to be associated with the process creation operator used in $P$;
- $E$ is the set of recursive process-equations defining the processes corresponding to the SDL states referred to in the process expressions in the range of $\phi$;
- $\mathcal{G}$ is the state space that is to be associated with the state operator used in $P$.

The first component depends on the names introduced by the definitions of channels and process types, and on the given numbers of processes to be created during the start-up of the system for the process types defined. The second and third component depend heavily on the process definitions proper. The last component depends simply on the names introduced by the definitions of variables, signal types, channels and process types – this means that the state space depends solely on purely syntactic aspects of the system.

The meaning of each definition occurring in a system definition is a pair $(\phi, E)$ where:

- $\phi$ is singleton mapping from process names to process expressions if it is the definition of a process type, and an empty mapping otherwise;
- $E$ is the set of recursive process-equations defining the processes corresponding to the SDL states referred to in the single process expression in the range of $\phi$ if it is the definition of a process type, and an empty set otherwise.

In case of a process definition, the first component is expressed in terms of the meaning of its start transition and the second component in terms of the meaning of its state definitions. We write $[\![D]\!]^\kappa_\phi$ and $[\![D]\!]^\kappa_E$, where $[\![D]\!]^\kappa = (\phi, E)$, for $\phi$ and $E$, respectively. Thus, we have $[\![D]\!]^\kappa = ([\![D]\!]^\kappa_\phi, [\![D]\!]^\kappa_E)$

The second and third component of the meaning of a system definition are obtained by taking the union of the first components and second components, respectively, of the meaning of all definitions occurring in it.

$$
\begin{array}{l}
[\![\mathsf{system}\ S;D_1 \ldots D_n\ \mathsf{endsystem};]\!] \ := \\
\quad (\tau_{I\cup\{\boldsymbol{t}\}} \circ \lambda_{G_0} \circ E_\phi(P), [\![D_1]\!]^\kappa_\phi \cup \ldots \cup [\![D_n]\!]^\kappa_\phi, [\![D_1]\!]^\kappa_E \cup \ldots \cup [\![D_n]\!]^\kappa_E, \mathcal{G}_\kappa) \\
\text{where } P \ = \ \|_{X\in procs(\kappa)}\ (\|^{init(\kappa,X)}\ \underline{\underline{cr}}(X, \langle\,\rangle, \langle\,\rangle, \mathsf{nil})), \\
\qquad G_0 \ = \ (0, 0, \{c \mapsto \langle\,\rangle | c \in chans(\kappa)\}, \{\,\}), \\
\qquad \kappa \ = \ \{\!|\mathsf{system}\ S;D_1 \ldots D_n\ \mathsf{endsystem};|\!\}
\end{array}
$$

$$
\begin{array}{l}
[\![\mathsf{process}\ X(k);\ \mathsf{fpar}\ v_1, \ldots, v_m;\ \mathsf{start};\ tr\ d_1 \ldots d_n\ \mathsf{endprocess};]\!]^\kappa \ := \\
\quad (\{X \mapsto \textstyle\sum_{self:\mathbb{N}} cnt = self :\rightarrow [\![tr]\!]^{\kappa'}\}, \{[\![d_1]\!]^{\kappa'}, \ldots, [\![d_n]\!]^{\kappa'}\}) \\
\text{where } \ \kappa' \ = \ updscopeunit(\kappa, X)
\end{array}
$$

$[\![D]\!]^\kappa := (\{\},\{\})$ if the definition $D$ is not of the form
process $X(k)$; fpar $v_1,\ldots,v_m$; start; $tr$ $d_1 \ldots d_n$ endprocess;

In the case of a system definition, the process expression $\tau_{I\cup\{\boldsymbol{t}\}} \circ \lambda_{G_0} \circ E_\phi(P)$ expresses that, for each process type defined, the given initial number of processes are created and the result is executed in the state $G_0$. Additionally, the internal action $\boldsymbol{t}$ as well as the actions in $I$ are hidden. $I$ is to be regarded as a parameter of the semantics. If one takes the empty set for $I$, one gets an extreme semantics, viz. a concrete one corresponding to the viewpoint that all internal actions of a system are observable. By taking appropriate non-empty sets, one can get a range of more abstract semantics, including the interesting one that corresponds to the viewpoint that only the communication with the environment is observable. $G_0$ is the state in which the last issued pid value and the system time are zero, there is an empty queue for each channel defined, and there are no local states. Recall that the pid value zero is reserved for the environment and that a newly created process gets its pid value and local state only when its execution starts. In the case of a process definition, the process expression in the singleton mapping $\{X \mapsto \sum_{self:\mathbb{N}} cnt = self :\rightarrow [\![tr]\!]^{\kappa'}\}$ expresses that, for each process of the type $X$, its behaviour is the behaviour determined by the given start transition $tr$ if *self* stands for the last issued pid value.

## 5.2 Process behaviours

The meaning of a state definition, occurring in the scope of a process definition, is a process-equation defining, for the process type defined, the common behaviour of its instances from the state being defined (using parametrization by the identifying pid value *self*). It is expressed in terms of the meaning of its transition alternatives, which are process expressions describing the behaviour from the state being defined for the individual signal types of which instances may be consumed and, in addition, possibly for some spontaneous transitions. The meaning of each transition alternative is in turn expressed in terms of the meaning of its input guard, if the alternative is not a spontaneous transition, and its transition.

$$[\![\text{state } st; \text{save } s_1,\ldots,s_m; alt_1 \ \ldots \ alt_n]\!]^\kappa :=$$
$$X_{st} = \neg waiting(s_1,\ldots,s_m,self) :\rightarrow ([\![alt_1]\!]^{\kappa'} + \ldots + [\![alt_n]\!]^{\kappa'}) +$$
$$waiting(s_1,\ldots,s_m,self) :\rightarrow \sigma_{\mathsf{rel}}(X_{st})$$

where $X = scopeunit(\kappa)$,
$\kappa' = updsaveset(\kappa,\{s_1,\ldots,s_m\})$

$$[\![\text{input } s(v_1,\ldots,v_n); tr]\!]^\kappa :=$$
$$(lt(cnt,n_0) :\rightarrow \underline{\boldsymbol{t}})^* (\neg lt(cnt,n_0) :\rightarrow \underline{\underline{input}}(((s,\langle v_1,\ldots,v_n\rangle),\mathsf{nil},self),ss) \cdot [\![tr]\!]^\kappa)$$

where $n_0 = \sum_{X\in procs(\kappa)} init(\kappa,X)$,[7]
$ss = saveset(\kappa)$

[7]Here, we use $\sum$ for summation of a set of natural numbers.

$$[\![\textsf{input none; } tr]\!]^\kappa \;:=\; \underline{\underline{inispont}}(self) \cdot [\![tr]\!]^\kappa$$

In the case of a state definition, the process-equation describes that the processes of type $X$ behave from the state $st$ as one of the given transition alternatives, and that this behaviour is possibly delayed till the first future time slice in which there is a signal to consume if there are no more signals to consume in the current time slice. In process-equations, we use names of process types with state name subscripts, such as $X_{st}$ above, as variables; in process expressions elsewhere, we use them to refer to the processes defined thus. Note that, in the absence of spontaneous transitions, a delay becomes inescapable if there are no more signals to consume in the current time slice. In the case of a guarded transition alternative, the process expression $\underline{\underline{input}}(((s, \langle v_1, \ldots, v_n\rangle), \textsf{nil}, self), ss) \cdot [\![tr]\!]^\kappa$ expresses that the transition $tr$ is initiated on consumption of a signal of type $s$; iteration is used to guarantee that no communication takes place till the start-up of the system has come to an end. In the case of an unguarded transition alternative, the process expression expresses that the transition $tr$ is initiated spontaneously, i.e. without a preceding signal consumption, with sender set to the value of self.

The meaning of a transition, occurring in the scope of a process definition, is a process expression describing the behaviour of the transition. It is expressed in terms of the meaning of its actions and its transition terminator.

$$[\![a_1 \ldots a_n \textsf{ nextstate } st;]\!]^\kappa \;:=\; [\![a_1]\!]^\kappa \cdot \ldots \cdot [\![a_n]\!]^\kappa \cdot X_{st}$$
$$\text{where } X \;=\; scopeunit(\kappa)$$

$$[\![a_1 \ldots a_n \textsf{ stop;}]\!]^\kappa \;:=\; [\![a_1]\!]^\kappa \cdot \ldots \cdot [\![a_n]\!]^\kappa \cdot \underline{\underline{stop}}(self)$$

$$[\![a_1 \ldots a_n \; dec;]\!]^\kappa \;:=\; [\![a_1]\!]^\kappa \cdot \ldots \cdot [\![a_n]\!]^\kappa \cdot [\![dec]\!]^\kappa$$

In the case of a transition terminated by nextstate $st$, the process expression expresses that the transition performs the actions $a_1, \ldots, a_n$ in sequential order and ends with entering state $st$ – i.e. goes on behaving as defined for state $st$ of the processes of the type defined. In case of termination by stop, it ends with ceasing to exist; and in case of termination by a decision $dec$, it goes on behaving as described by $dec$.

Of course, the meaning of a decision is a process expression as well. It is expressed in terms of the meaning of its expressions and transitions.

$$[\![\textsf{decision } e;(e_1){:}tr_1 \ldots (e_n){:}tr_n \textsf{ enddecision}]\!]^\kappa \;:=$$
$$\quad [\![e]\!] = [\![e_1]\!] :\rightarrow [\![tr_1]\!]^\kappa + \ldots + [\![e]\!] = [\![e_n]\!] :\rightarrow [\![tr_n]\!]^\kappa$$

$$[\![\textsf{decision any; }(){:}tr_1 \ldots (){:}tr_n \textsf{ enddecision}]\!]^\kappa \;:=\; [\![tr_1]\!]^\kappa + \ldots + [\![tr_n]\!]^\kappa$$

In the case of a decision with a question expression $e$, the process expression expresses that the decision transfers control to the transition $tr_i$ for which the value of $e$ equals the value of $e_i$. In the case of a decision with any instead,

the process expression expresses that the decision transfers non-deterministically control to one of the transitions $tr_1, \ldots, tr_n$.

The meaning of an SDL action is also a process expression. It is expressed in terms of the meaning of the expressions occurring in it. It also depends on the occurring names (names of variables, signal types, signal routes and process types – dependent on the kind of action).

$$
\begin{array}{l}
[\![\mathsf{output}\ s(e_1,\ldots,e_n)\ \mathsf{to}\ e\ \mathsf{via}\ r_1,\ldots,r_m;]\!]^\kappa\ :=\\
\quad (lt(cnt,n_0) :\rightarrow \underline{\underline{\boldsymbol{t}}})^*\\
\quad (\neg lt(cnt,n_0) :\rightarrow (type([\![e]\!]) = X_1 :\rightarrow P_1 + \ldots + type([\![e]\!]) = X_m :\rightarrow P_m +\\
\qquad\qquad \neg(type([\![e]\!]) = X_1 \vee \ldots \vee type([\![e]\!]) = X_m) :\rightarrow \underline{\underline{\boldsymbol{t}}}))\\
\text{where } n_0 \ = \ \sum_{X\in procs(\kappa)} init(\kappa, X),\\
\qquad \text{for } 1 \le j \le m\text{:}\\
\qquad P_j \ = \ \underline{\underline{output}}(((s,\langle [\![e_1]\!],\ldots,[\![e_n]\!]\rangle), self, [\![e]\!]), c_j, 0) \qquad \textbf{if } c_j = \mathsf{nil}\\
\qquad\qquad \sum_{d:\mathbb{N}} \underline{\underline{output}}(((s,\langle [\![e_1]\!],\ldots,[\![e_n]\!]\rangle), self, [\![e]\!]), c_j, d) \ \textbf{otherwise},\\
\qquad X_j \ = \ rcv(\kappa, r_j),\\
\qquad c_j \ = \ ch(\kappa, r_j)
\end{array}
$$

$$
\begin{array}{l}
[\![\mathsf{output}\ s(e_1,\ldots,e_n)\ \mathsf{via}\ r_1,\ldots,r_m;]\!]^\kappa\ :=\\
\quad (lt(cnt,n_0) :\rightarrow \underline{\underline{\boldsymbol{t}}})^*\\
\quad (\neg lt(cnt,n_0) :\rightarrow (\textstyle\sum_{i:\mathbb{N}}(type(i) = X_1 :\rightarrow P_1 + \ldots + type(i) = X_m :\rightarrow P_m) +\\
\qquad\qquad \neg(hasinst(X_1) \wedge \ldots \wedge hasinst(X_m)) :\rightarrow \underline{\underline{\boldsymbol{t}}}))\\
\text{where } n_0 \ = \ \sum_{X\in procs(\kappa)} init(\kappa, X),\\
\qquad \text{for } 1 \le j \le m\text{:}\\
\qquad P_j \ = \ \underline{\underline{output}}(((s,\langle [\![e_1]\!],\ldots,[\![e_n]\!]\rangle), self, i), c_j, 0) \qquad \textbf{if } c_j = \mathsf{nil}\\
\qquad\qquad \sum_{d:\mathbb{N}} \underline{\underline{output}}(((s,\langle [\![e_1]\!],\ldots,[\![e_n]\!]\rangle), self, i), c_j, d) \ \textbf{otherwise},\\
\qquad X_j \ = \ rcv(\kappa, r_j),\\
\qquad c_j \ = \ ch(\kappa, r_j)
\end{array}
$$

$$[\![\mathsf{set}\,(e,s(e_1,\ldots,e_n));]\!]^\kappa \ := \ \underline{\underline{set}}([\![e]\!], (s,\langle [\![e_1]\!],\ldots,[\![e_n]\!]\rangle), self)$$

$$[\![\mathsf{reset}\,(s(e_1,\ldots,e_n));]\!]^\kappa \ := \ \underline{\underline{reset}}((s,\langle [\![e_1]\!],\ldots,[\![e_n]\!]\rangle), self)$$

$$[\![\mathsf{task}\ v := e;]\!]^\kappa \ := \ \underline{\underline{ass}}(v, [\![e]\!], self)$$

$$[\![\mathsf{create}\ X(e_1,\ldots,e_n);]\!]^\kappa \ := \ \underline{\underline{cr}}(X, fpars(\kappa, X), \langle [\![e_1]\!],\ldots,[\![e_n]\!]\rangle, self)$$

All cases except the ones for output actions are straightforward. The cases of output actions needs further explanation. The receiver of a signal sent via a certain signal route must be of the receiver type associated with that signal route. Therefore, the conditions of the form $type(u) = X_j$ are used. In the case of an output action with a receiver expression $e$, if none of the signal routes $r_1, \ldots, r_m$ has the type of the process with pid value $e$ as its receiver type, or a process with that pid value does not exist, the signal is simply discarded and no error occurs. This is expressed by the summand $\neg(type([\![e]\!]) = X_1 \vee \ldots \vee type([\![e]\!]) = X_m) :\rightarrow \underline{\underline{\boldsymbol{t}}}$.

In the case of an output action without a receiver expression, first an arbitrary choice from the signal routes $r_1, \ldots, r_m$ is made and thereafter an arbitrary choice from the existing processes of the receiver type for the chosen signal route is made. However, there may be no existing process of the receiver type for that signal route. Should this occasion arise, the signal is simply discarded. This is expressed by the summand $\neg(\mathit{hasinst}(X_1) \wedge \ldots \wedge \mathit{hasinst}(X_m)) :\rightarrow \underline{\underline{t}}$. Note that this occasion may already arise if there is one signal route for which there exists no process of its receiver type. Note further that a process expression of the form $\sum_{d:\mathbb{N}} \underline{\underline{\mathit{output}}}(\mathit{sig}, c, d)$ is used for each signal route containing a delaying channel $c$. Thus, the arbitrary delay is modelled by an arbitrary choice between all possible delay durations $d$ as already mentioned in Section 4.2. As for input guards, iteration is used to guarantee that no communication takes place till the start-up of the system has come to an end.

## 5.3 Values

The meaning of an SDL expression is given by a translation to a value expression of the same kind. There is a close correspondence between the SDL expressions and their translations. Essential of the translation is that *self* is added where the local states of different processes need to be distinguished. Consequently, a variable access $v$ is just treated as a view expression view ($v$, self ). For convenience, the expressions parent, offspring and sender are also regarded as variable accesses.

$$[\![\mathit{op}(e_1, \ldots, e_n)]\!] := \mathit{op}([\![e_1]\!], \ldots, [\![e_n]\!])$$

$$[\![\textsf{if } e_1 \textsf{ then } e_2 \textsf{ else } e_3 \textsf{ fi}]\!] := \mathit{cond}([\![e_1]\!], [\![e_2]\!], [\![e_3]\!])$$

$$[\![v]\!] := \mathit{value}(v, \mathit{self})$$

$$[\![\textsf{view}\,(v, e)]\!] := \mathit{value}(v, [\![e]\!])$$

$$[\![\textsf{active}\,(s(e_1, \ldots, e_n))]\!] := \mathit{active}((s, \langle [\![e_1]\!], \ldots, [\![e_n]\!] \rangle), \mathit{self})$$

$$[\![\textsf{now}]\!] := \mathit{now}$$

$$[\![\textsf{self}]\!] := \mathit{self}$$

$$[\![\textsf{parent}]\!] := \mathit{value}(\textsf{parent}, \mathit{self})$$

$$[\![\textsf{offspring}]\!] := \mathit{value}(\textsf{offspring}, \mathit{self})$$

$$[\![\textsf{sender}]\!] := \mathit{value}(\textsf{sender}, \mathit{self})$$

All cases are very straightforward and need no further explanation. This is due to the choice of value expressions and the evaluation function defined on them in Section 4.5.

# 6 Closing remarks

Models of highly reactive and distributed systems, in particular telecommunications systems, are frequently made using SDL. This is done with, among other things, the intention to allow for the analysis of their behaviour. Largely due to their intrinsic reactive and distributed nature, giving considerations to time is inherent to the analysis of the behaviour of such systems. The semantics of SDL according to the ITU/TS recommendation is at some points insufficiently precise, and at other points too complex, to allow for interesting analysis; in particular the time related features of SDL, such as timers and channels with delay, miss an adequate semantics. Besides, the existing tools for analysis of models described in SDL are very limited; at best a limited kind of model checking, closely related to simulation of the described behaviour, is provided, and no time related features are supported. In a joint project of KPN Research – the research institute of the telecommunications operator PTT Telecom and the industrial affiliation of the second author – and Utrecht University, a state-of-the-art model checker is adapted to the common needs for analysis of systems modelled using $\varphi$SDL. The intention of that work is to do some first steps in the improvement of the possibilities for analysis of models described in SDL. The work on $\varphi$SDL reported in this paper was initiated by that project.

In [10] a foundation for the semantics of SDL, based on streams and stream processing functions, has been proposed. This proposal indicates that the SDL view of systems gives an interesting type of dynamic dataflow networks, but the treatment of time in the proposal is however too sketchy to be used as a starting point for the semantics of the time related features of SDL. In [11] and [12] attempts have been made to give a structured operational semantics of SDL, the latter including the time related features. However, not all relevant details were worked out, and the results will probably have to be turned inside out in order to deal with full SDL. At the outset, we also tried shortly to give a structured operational semantics of SDL, but we found that it is very difficult, especially if time aspects have to be taken into account. Of course, a structured operational semantics can be derived from the process algebra semantics, and most probably, we will have to do so for the above-mentioned project.

# References


[1] J.C.M. Baeten and J.A. Bergstra. Global renaming operators in concrete process algebra. *Information and Control*, 78:205–245, 1988.

[2] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra (extended abstract). In *CONCUR'92*, pages 401–420. LNCS 630, Springer-Verlag, 1992. Full version: Report PRG 9208b, Programming Research Group, University of Amsterdam.

[3] J.C.M. Baeten and J.A. Bergstra. Process algebra with signals and conditions. In M. Broy, editor, *Programming and Mathematical Methods*, pages 273–323. NATO ASI Series F88, Springer-Verlag, 1992.

[4] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra, 1995. To appear in Formal Aspects of Computing.

[5] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.

[6] J.A. Bergstra. A process creation mechanism in process algebra. In J.C.M. Baeten, editor, *Applications of Process Algebra*, pages 81–88. Cambridge Tracts in Theoretical Computer Science 17, Cambridge University Press, 1990.

[7] J.A. Bergstra, I. Bethke, and A. Ponse. Process algebra with iteration. *The Computer Journal*, 37:243–258, 1994.

[8] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

[9] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[10] M. Broy. Towards a formal foundation of the specification and description language SDL. *Formal Aspects of Computing*, 3:21–57, 1991.

[11] A. Gammelgaard and J.E. Kristensen. A correctness proof of a translation from SDL to CRL. In O. Færgemand and A. Sarma, editors, *SDL '93: Using Objects*, pages 205–219. Elsevier (North-Holland), 1991. Full version: Report TFL RR 1992-4, Tele Danmark Research.

[12] J.C. Godskesen. An operational semantics model for Basic SDL (extended abstract). In O. Færgemand and R. Reed, editors, *SDL '91: Evolving Methods*, pages 15–22. Elsevier (North-Holland), 1991. Full version: Report TFL RR 1991-2, Tele Danmark Research.

[13] J.F. Groote and A. Ponse. Proof theory for $\mu$CRL: A language for processes with data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Semantics of Specification Languages*, pages 232–251. Workshop in Computing Series, Springer-Verlag, 1994.

[14] J.F. Groote and A. Ponse. The syntax and semantics of $\mu$CRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes 1994*, pages 26–62. Workshop in Computing Series, Springer-Verlag, 1995.

[15] R. Milner. *A Calculus of Communicating Systems*. LNCS 92, Springer-Verlag, 1980.

[16] B. Møller-Pedersen. On the simplification of SDL. *SDL Newsletter*, 17:4–6, 1994.

[17] X. Nicollin and J. Sifakis. The algebra of timed processes ATP: Theory and application. *Information and Computation*, 114:131–178, 1994.

[18] L. Pruitt, 1994. Personal Communications.

[19] Rules for the use of SDL. ETSI Document MTS (93) 10, 1993.

[20] Specification of abstract syntax notation one (ASN.1). Blue Book Fasc. VIII.4, Recommendation X.208, 1989.

[21] Specification and description language (SDL). ITU-T Recommendation Z.100, Revision 1, 1994.

[22] SDL predefined data. ITU-T Recommendation Z.100 D, Revision 1, 1994. Annex D to Recommendation Z.100.
[23] Specification and description language (SDL) – SDL formal definition: Static semantics. ITU-T Recommendation Z.100 F2, Revision 1, 1994. Annex F.2 to Recommendation Z.100.
[24] Specification and description language (SDL) – SDL formal definition: Dynamic semantics. ITU-T Recommendation Z.100 F3, Revision 1, 1994. Annex F.3 to Recommendation Z.100.
[25] SDL combined with ASN.1 (SDL/ASN.1). Proposed New ITU-T Recommendation Z.105, 1994.

# A Notational conventions

**Meta-language for syntax:**

The syntax of $\varphi$SDL is described by means of production rules in the form of an *extended* BNF grammar. The curly brackets "{" and "}" are used for grouping. The asterisk "*" and the plus sign "+" are used for zero or more repetitions and one or more repetitions, respectively, of curly bracketed groups. The square brackets "[" and "]" are also used for grouping, but indicate that the group is optional. An underlined part included in a nonterminal symbol does not belong to the context free syntax; it describes a semantic condition.

**Special set, function and sequence notation:**

We write $\mathcal{P}(A)$ for the set of all subsets of $A$, and we write $\mathcal{P}_{\mathit{fin}}(A)$ for the set of all finite subsets of $A$.

We write $f : A \to B$ to indicate that $f$ is a total function from $A$ to $B$, that is $f \subseteq A \times B \wedge \forall x \in A \cdot \exists_1 y \in B \cdot (x, y) \in f$. If $A$ is finite, we emphasize this by writing $f : A \xrightarrow{\mathit{fin}} B$ instead. We write $\mathit{dom}(f)$, where $f : A \to B$, for $A$. For an (ordered) pair $(x, y)$, where $x$ and $y$ are intended for argument and value of some function, we use the notation $x \mapsto y$ to emphasize this intention. The binary operators $\lhd$ (domain subtraction) and $\oplus$ (overriding) on functions are defined by

$$\begin{aligned} A \lhd f &= \{x \mapsto y \mid x \in \mathit{dom}(f) \wedge x \notin A \wedge f(x) = y\} \\ f \oplus g &= (\mathit{dom}(g) \lhd f) \cup g \end{aligned}$$

For a function $f : A \to B$, presenting a family $B$ indexed by $A$, we use the notation $f_i$ (for $i \in A$) instead of $f(i)$.

Functions are also used to present sequences; as usual we write $\langle x_1, \ldots, x_n \rangle$ for the sequence presented by the function $\{1 \mapsto x_1, \ldots, n \mapsto x_n\}$. The binary operator $\frown$ stands for concatenation of sequences. We write $x \,\&\, t$ for $\langle x \rangle \frown t$.

# B Contextual information

The meaning of a $\varphi$SDL construct generally depends on the definitions in the scope in which it occurs. Contexts are primarily intended for modeling the scope. The context that is ascribed to a complete system definition is also used to define the state space used to describe its meaning. The context of a construct contains all names introduced by the definitions of variables, signal types, channels, signal routes and process types occurring in the system definition on hand and additionally:

- if the construct occurs in the scope of a process definition, the name introduced by that process definition, called the *scope unit*;
- if the construct occurs in the scope of a state definition, the set of names occurring in the save part of that state definition, called the *save set*.

In case of a signal route, the name is in addition connected with the names of its receiver type and its delaying channel, if present; and in case of a process type, the name is connected with the names of the variables that are its formal parameters and the number of processes of this type that have to be created during the start-up of the system.

$$
\begin{aligned}
&\mathit{Context} = \\
&\quad \mathcal{P}_{\mathit{fin}}(\mathit{VarId}) \times \mathcal{P}_{\mathit{fin}}(\mathit{SigId}) \times \mathcal{P}_{\mathit{fin}}(\mathit{ChanId}) \times \mathcal{P}_{\mathit{fin}}(\mathit{RouteDes}) \times \mathcal{P}_{\mathit{fin}}(\mathit{ProcDes}) \times \\
&\quad (\mathit{ProcId} \cup \{\mathsf{nil}\}) \times \mathcal{P}_{\mathit{fin}}(\mathit{SigId})
\end{aligned}
$$

$$
\begin{aligned}
\text{where } \mathit{RouteDes} &= \mathit{RouteId} \times (\mathit{ProcId} \cup \{\mathsf{env}\}) \times (\mathit{ChanId} \cup \{\mathsf{nil}\}) \\
\mathit{ProcDes} &= \mathit{ProcId} \times \mathit{VarId}^* \times \mathbb{N}
\end{aligned}
$$

We write $\mathit{vars}(\kappa)$, $\mathit{sigs}(\kappa)$, $\mathit{chans}(\kappa)$, $\mathit{routeds}(\kappa)$, $\mathit{procds}(\kappa)$, $\mathit{scopeunit}(\kappa)$ and $\mathit{saveset}(\kappa)$, where $\kappa = (V, S, C, Rd, Pd, X, ss) \in \mathit{Context}$, for $V$, $S$, $Ch$, $Rd$, $Pd$, $X$ and $ss$, respectively. We write $\mathit{procs}(\kappa)$ for $\{X \mid \exists vs, k \cdot (X, vs, k) \in \mathit{procds}(\kappa)\}$. For constructs that do not occur in a process definition, the absence of a scope unit will be represented by nil and, for constructs that do not occur in a state definition, the absence of a save set will be represented by $\{\,\}$.

Useful operations on *Context* are the functions

$$
\begin{aligned}
&\mathit{rcv} && : \mathit{Context} \times \mathit{RouteId} \to \mathit{ProcId} \cup \{\mathsf{env}\}, \\
&\mathit{ch} && : \mathit{Context} \times \mathit{RouteId} \to \mathit{ChanId} \cup \{\mathsf{nil}\}, \\
&\mathit{fpars} && : \mathit{Context} \times \mathit{ProcId} \to \mathit{VarId}^*, \\
&\mathit{init} && : \mathit{Context} \times \mathit{ProcId} \to \mathbb{N}, \\
&\mathit{updscopeunit} && : \mathit{Context} \times \mathit{ProcId} \to \mathit{Context}, \\
&\mathit{updsaveset} && : \mathit{Context} \times \mathcal{P}_{\mathit{fin}}(\mathit{SigId}) \to \mathit{Context}
\end{aligned}
$$

defined below. The functions *rcv* and *ch* are used to extract the receiver type and the delaying channel, respectively, of a given signal route from the context. These functions are inductively defined by

$$
\begin{aligned}
&(r, X, c) \in \mathit{routeds}(\kappa) \Rightarrow \mathit{rcv}(\kappa, r) = X, \\
&(r, X, c) \in \mathit{routeds}(\kappa) \Rightarrow \mathit{ch}(\kappa, r) = c
\end{aligned}
$$

The functions *fpars* and *init* are used to extract the formal parameters and the initial number of processes, respectively, of a given process type from the context. These functions are inductively defined by

$$(X, vs, k) \in procds(\kappa) \Rightarrow fpars(\kappa, X) = vs,$$
$$(X, vs, k) \in procds(\kappa) \Rightarrow init(\kappa, X) = k$$

The functions *updscopeunit* and *updsaveset* are used to update the scope unit and the save set, respectively, of the context. These functions are inductively defined by

$$\kappa = (V, S, C, Rd, Pd, X, ss) \Rightarrow updscopeunit(\kappa, X') = (V, S, C, Rd, Pd, X', ss),$$
$$\kappa = (V, S, C, Rd, Pd, X, ss) \Rightarrow updsaveset(\kappa, ss') = (V, S, C, Rd, Pd, X, ss')$$

The context ascribed to a system definition is a minimal context in the sense that the contextual information available in it is common to all contexts on which constructs occurring in it depend. The additional information that may be available applies to the scope unit for constructs occurring in a process definition and the save set for constructs occurring in a state definition. The context ascribed to a system definition is obtained by taking the union of the corresponding components of the (partial) contexts contributed by all definitions occurring in it, except for the scope unit and the saveset which are permanently the same – nil and $\{\,\}$, respectively.

$$\begin{aligned}
&\{\!|\textsf{system } S;D_1 \ldots D_n \textsf{ endsystem;}|\!\} \;:= \\
&\quad (vars(\{\!|D_1|\!\}) \cup \ldots \cup vars(\{\!|D_n|\!\}), \\
&\quad sigs(\{\!|D_1|\!\}) \cup \ldots \cup sigs(\{\!|D_n|\!\}), \\
&\quad chans(\{\!|D_1|\!\}) \cup \ldots \cup chans(\{\!|D_n|\!\}), \\
&\quad routeds(\{\!|D_1|\!\}) \cup \ldots \cup routeds(\{\!|D_n|\!\}), \\
&\quad procds(\{\!|D_1|\!\}) \cup \ldots \cup procds(\{\!|D_n|\!\}), \\
&\quad \textsf{nil}, \{\,\})
\end{aligned}$$

$$\{\!|\textsf{dcl } v\ T;|\!\} \;:= (\{v\}, \{\,\}, \{\,\}, \{\,\}, \{\,\}, \textsf{nil}, \{\,\})$$

$$\{\!|\textsf{signal } s(T_1, \ldots, T_n);|\!\} \;:= (\{\,\}, \{s\}, \{\,\}, \{\,\}, \{\,\}, \textsf{nil}, \{\,\})$$

$$\{\!|\textsf{channel } c;|\!\} \;:= (\{\,\}, \{\,\}, \{c\}, \{\,\}, \{\,\}, \textsf{nil}, \{\,\})$$

$$\begin{aligned}
&\{\!|\textsf{signalroute } r \textsf{ from } X_1 \textsf{ to } X_2 \textsf{ with } s_1, \ldots, s_n \textsf{ delayed by } c;|\!\} \;:= \\
&\quad (\{\,\}, \{\,\}, \{\,\}, \{(r, X_2, c)\}, \{\,\}, \textsf{nil}, \{\,\})
\end{aligned}$$

$$\begin{aligned}
&\{\!|\textsf{process } X(k); \textsf{ fpar } v_1, \ldots, v_m; \textsf{ start; } tr\ d_1 \ldots d_n \textsf{ endprocess;}|\!\} \;:= \\
&\quad (\{\,\}, \{\,\}, \{\,\}, \{\,\}, \{(X, \langle v_1, \ldots, v_m \rangle, k)\}, \textsf{nil}, \{\,\})
\end{aligned}$$