The United Nations University

UNU/IIST

International Institute for Software Technology

Process Algebra Semantics of φSDL

Jan A. Bergstra and Cornelis A. Middelburg

April 1996

UNU/IIST Report No. 68

UNU/IIST

UNU/IIST enables developing countries to attain self-reliance in software technology by: (i) their own development of high integrity computing systems, (ii) highest level post-graduate university teaching, (iii) international level research, and, through the above, (iv) use of as sophisticated software as reasonable.

UNU/IIST contributes through: (a) advanced, joint industry-university advanced development projects in which rigorous techniques supported by semantics-based tools are applied in case studies to large scale software developments, (b) own and joint university and academy institute research in which new techniques for application domain and computing platform modeling, requirements capture, software engineering and programming are being investigated, (c) advanced, post-graduate and post-doctoral level courses which typically teach Design Calculi oriented software development techniques, (d) events [panels, task forces, workshops and symposia], and (e) dissemination.

Application-wise, the advanced development projects presently focus on software to support large-scale infrastructure systems such as railways, manufacturing industries, health care systems, etc., and are thus aligned with UN and International Aid System concerns. The research projects parallel and support the advanced development projects.

At present, the technical focus of UNU/IIST in all of the above is on applying, teaching, researching, and disseminating Design Calculi oriented techniques and tools for trustworthy software development. UNU/IIST currently emphasizes techniques that permit proper development steps and interfaces. UNU/IIST also endeavours to promulgate sound project and product management principles.

UNU/IIST's primary dissemination strategy is to act as a clearing house for reports from research and technology centres in industrial countries to industries and academic institutions in developing countries. At present more than 175 institutions worldwide contribute to UNU/IIST's report collection while UNU/IIST at the same time subscribes to more than 125 international scientific and technical journals. Information on reports received (and produced) and on journal articles is to be disseminated regularly to developing country centres — which are then free to order a reasonable number of report and article copies from UNU/IIST.

Dines Bjørner, Director

UNU/IIST Reports are either $\mathcal{R}$esearch, $\mathcal{T}$echnical, $\mathcal{C}$ompendia or $\mathcal{A}$dministrative reports:

$\boxed{\mathcal{R}}$ Research Report • $\boxed{\mathcal{T}}$ Technical Report • $\boxed{\mathcal{C}}$ Compendium • $\boxed{\mathcal{A}}$ Administrative Report

The United Nations University

UNU/IIST

International Institute for Software Technology

P.O. Box 3058
Macau

Process Algebra Semantics of φSDL

Jan A. Bergstra and Cornelis A. Middelburg

Abstract

A new semantics of an interesting subset of the specification language SDL is given by a translation to a discrete-time extension of process algebra in the form of ACP with data added as in μCRL. The strength of the chosen subset, called φSDL, 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.

Note: This paper is a major revision of [14].

Jan Bergstra is a Professor of Programming and Software Engineering at the University of Amsterdam and a Professor of Applied Logic at Utrecht University, both in the Netherlands. His research interest is in mathematical aspects of software and system development, in particular in the design of algebras that can contribute to a better understanding of the relevant issues at a conceptual level. He is perhaps best known for his contributions to the field of process algebra. E-mail: janb@fwi.uva.nl

Kees Middelburg is a Senior Research Fellow at UNU/IIST. He is on a two year leave (1996–1997) from KPN Research and Utrecht University, the Netherlands, where he is a Senior Computer Scientist and a Professor of Applied Logic, respectively. His research interest is in formal techniques for the development of software for reactive and distributed systems, including related subjects such as semantics of specification languages and concurrency theory. E-mail: cam@iist.unu.edu

Contents

1 Introduction

A process algebra semantics of φSDL is presented. φSDL 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, φSDL does not deal with the specification of abstract data types. An algebraic specification of all data types used in an φSDL 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 φSDL 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 inclusion 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 φSDL for other reasons: the semantics of services is hard to understand, ETSI forbids for this reason their use in European telecommunication standards (see [24]), and the SDL community currently discusses its usefulness (see [22]).

Apart from the data type definitions, SDL system definitions can be transformed to φSDL system definitions, provided that no use is made of facilities whose inclusion has been postponed. 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 [26] and defined in a fully precise manner in its Annex F.2 [28]. The second and final part is essentially the mapping *extract-dict* which is defined in its Annex F.3 [29]; φSDL system definitions can actually be viewed as textual presentations of the extracted *Entity-dict*s which are interpreted instead of the SDL system definitions proper.

[1] This subset is called φSDL, where φ 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 [26], 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 [24]); (2) ASN.1 [25] 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 [30]).

The semantics of φSDL agrees with the semantics of SDL as far as reasonably possible. This means in the first place that obvious errors in [29] 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 [29] – 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 [27]). 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 φSDL 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 generally had to make our own choices with respect to the time related aspects of SDL, because they 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 [23] 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 $\mathrm{ACP}_{\mathrm{drt}}$ (see [5, 6]), as the basis of the presented semantics.

In telecommunications, systems are frequently specified using SDL. This is usually done with the intention to be able to analyse the behavioural properties of these systems. Systems are increasingly described at different levels of abstraction. This gives rise to an growing need to verify that the properties represented by an abstract specification are preserved in a more concrete one. The current situation is that there are only means for limited analysis and no means at all for formal verification. The intrinsic highly reactive and distributed nature of the systems developed in telecommunications demands more advanced analysis than currently possible, e.g. analysis giving considerations to timing properties as well. Besides, the increasing complexity of systems will become a compelling reason to use, at least to a certain extent, formal verification to justify design steps. Prerequisites for advanced analysis and formal verification is a dramatically simplified version of SDL and an adequate semantics for it. Only after that possibilities for advanced analysis can be elaborated and proof rules for formal verification devised. The language φSDL and the presented semantics for it are primarily intended to come up to these prerequisites.

The structure of this paper is as follows. First of all, we give an overview of φSDL (Section 2). Next, we give a brief summary of the ingredients of process algebra which make up the basis for the semantics of φSDL 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 φSDL (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). There are appendices about notational conventions used (Appendix A), details concerning the contexts used to model scope in the presented semantics (Appendix B), and details concerning the environment of systems modelled using SDL (Appendix C).

2 Overview of φSDL

This section gives an overview of φSDL. 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 φSDL is presented.

2.1 System definition

First of all, the φSDL 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

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

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 will be instantaneously delivered to that process. Otherwise there may be an arbitrary transmission delay. A channel may be contained in more than one signal route.

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 ... 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 φSDL 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 signals of certain types may be consumed and in which, for each of these types, a certain transition is made on consumption of a signal of that type. 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 φSDL 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, φSDL 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 environment, 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 φSDL 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 ACP,[4] introduced in [13], extended with the silent step τ and the abstraction operator τ_I for abstraction. Semantically, we adopt the approach to abstraction, originally proposed for ACP in [18], which is based on branching bisimulation. ACP with this kind of abstraction is called ACP$^\tau$. In ACP with abstraction, processes can be composed by sequential composition, written $P \cdot Q$, alternative composition, written $P + Q$, parallel composition, written $P \parallel Q$, encapsulation, written $\partial_H(P)$, and abstraction, written $\tau_I(P)$. For a systematic introduction to ACP, the reader is referred to [7].

We will use the following abbreviations. Let $(P_i)_{i\in I}$ be an indexed set of process expressions where $I = \{i_1, \ldots, i_n\}$. Then, we write $\sum_{i\in I} P_i$ for $P_{i_1} + \ldots + P_{i_n}$ and $\|_{i\in I}\ P_i$ for $P_{i_1} \parallel \ldots \parallel P_{i_n}$ if $n > 0$ and δ if $n = 0$. Let P be a process expression and let $n \in \mathbb{N}$. Then, we write $\|^n\ P$ for $\underbrace{P \parallel \ldots \parallel P}_{n\times}$.

Further we will use the following extensions:

state operator We will use the state operator λ_S^m, added to ACP in [1]. This operator formalizes execution of a process in the state S of an object m. 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 $\lambda_S^m(a \cdot P) = a' \cdot \lambda_{S'}^m(P)$.

process creation We will also use the process creation mechanism, added to ACP in [9]. The process creation operator E_ϕ introduced there allows, given a mapping ϕ 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 $:\!\rightarrow$. The expression $b :\!\rightarrow P$ is to be read as "if b then P". The operator $:\!\rightarrow$ can best be defined in terms of the two-armed conditional operator $\lhd\bullet\rhd$, with the defining equations $P\lhd\mathsf{true}\rhd Q = P$ and $P\lhd\mathsf{false}\rhd Q = Q$, added to ACP in [3]. The one-armed conditional is then defined by $b :\!\rightarrow P = P\lhd b\rhd\delta$.

iteration We will also use the binary version of Kleene's star operator *, added to ACP in [10], 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.

propositional signals We will further use the root signal emission operator $\overset{\frown}{\rightarrow}$ as in [4]. The expression $\phi \overset{\frown}{\rightarrow} P$ is the process P where the proposition ϕ is made to hold at its start. The most crucial equations concerning the operator $\overset{\frown}{\rightarrow}$ are $(\phi \overset{\frown}{\rightarrow} P) + Q = \phi \overset{\frown}{\rightarrow} (P + Q)$,

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

$\phi \leadsto (\psi \leadsto P) = (\phi \wedge \psi) \leadsto P$, and $\mathsf{false} \leadsto P = \bot$ where $\bot$ stands for an inconsistent process. We refer to [4] for further details.[5]

discrete time We need a discrete time extension of ACP with relative timing. We will use the extension introduced in [5], called $\mathrm{ACP}_{\mathrm{drt}}$, with abstraction as added to it in [6]. Here we give a brief summary. We refer to [5] and [6] for further details on $\mathrm{ACP}_{\mathrm{drt}}$ and $\mathrm{ACP}^{\tau}_{\mathrm{drt}}$, respectively.

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. We will use the constants a, $\underline{\underline{a}}$ (for each a in some given set of actions), $\underline{\underline{\tau}}$ and $\underline{\underline{\delta}}$, as well as the delay operator σ_{rel}. The process a is a performed in any time slice and $\underline{\underline{a}}$ is a performed in the current time slice. Similarly, $\underline{\underline{\tau}}$ is a silent step performed in the current time slice and $\underline{\underline{\delta}}$ is a deadlock in the current time slice. The process $\sigma_{\mathsf{rel}}(P)$ is P delayed one time slice. In this paper, we use the notations from [2]. In [5], the notations $\mathsf{ats}(a)$, $\mathsf{cts}(a)$ and $\mathsf{cts}(\delta)$ are used instead of a, $\underline{\underline{a}}$ and $\underline{\underline{\delta}}$, respectively. Likewise, in [6], the notation $\mathsf{cts}(\tau)$ is used instead of $\underline{\underline{\tau}}$. The process a is defined in terms $\underline{\underline{a}}$ and σ_{rel} by the equation $a = \underline{\underline{a}} + \sigma_{\mathsf{rel}}(a)$. 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_is. 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, $\underline{\underline{a}} \parallel \sigma_{\mathsf{rel}}(\underline{\underline{b}}) = \underline{\underline{a}} \cdot \sigma_{\mathsf{rel}}(\underline{\underline{b}})$.

We will also use the other extensions of ACP in the setting of $\mathrm{ACP}_{\mathrm{drt}}$. Their integration is generally straightforward. The most relevant axiom for the integration of propositional signals is $\phi \leadsto \sigma_{\mathsf{rel}}(x) = \sigma_{\mathsf{rel}}(\phi \leadsto x)$, with the intuition that the passage of time cannot change the propositions that hold in the current state of a process.

summation over data domains We will in addition use actions parametrized by data and summation over a data domain as in μCRL [20, 21]. The notation $a(d_1, \ldots, d_n)$, where the d_is 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 [7]. In ACP with abstraction, the operators λ^m_S and E_ϕ can be defined. Two objects are relevant to the semantics of SDL: the *s*ystem being defined and its *e*nvironment. These objects will be referred to by s and e, respectively.

In [3], the definition of the state operator is adapted for conditionals. This definition uses an evaluation function $eval^m_S$ for conditions. The additional equation in this case is $\lambda^m_S(b :\to P) = eval^m_S(b) :\to \lambda^m_S(P)$. Thus, execution of P is disabled in state S if b evaluates to false in S.

If propositional signals are used, the execution of a process in a state may change the proposition that holds. A signal function *sig*, associating a propositional signal with each state, is proposed in [4] to cover this. This is also used for the semantics of φSDL. In case of this extension, the

[5] In [4] these propositions are called propositional signals. In later sections of the current paper, they are simply termed propositions in order to prevent confusion with signals in the sense of SDL.

defining equations are of the form $\lambda_S^m(a \cdot P) = sig(S) \leadsto a' \cdot \lambda_{S'}^m(P)$. The above-mentioned treatment of conditionals differs from the one in [4] where the conditions are the propositions that may hold in the states. This allows to use *sig* instead of $eval_S^m$ – the crucial equation is $\lambda_S^m(\phi :\to P) = sig(S) \leadsto (\phi :\to \lambda_S^m(P))$. This approach is not adopted here.

The process creation operator used for the semantics of φSDL is a slight adaptation of the process creation operator described in [9], 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

SDL's input guards and actions constitute its mechanisms for storage, communication, timing and process creation. In the process algebra semantics of φSDL, 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 and SDL actions 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 φSDL 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 $\mathcal{A}$.[6] 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_is 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 φSDL 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 κ (the notation concerning contexts introduced in Appendix B is used):

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

First, we define the set Sig_κ of signals and the set $ExtSig_\kappa$ 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.

$$\begin{aligned} Sig_\kappa &= S_\kappa \times U^* \\ ExtSig_\kappa &= Sig_\kappa \times \mathbb{N} \times \mathbb{N} \end{aligned}$$

We write $snm(sig)$ and $vals(sig)$, where $sig = (s, vs) \in Sig_\kappa$, for s and vs, respectively. We write $sig(xsig)$, where $xsig = (sig, i, i') \in ExtSig_\kappa$, for sig.

[6] We have that $\mathbb{B} \subset U$ and $\mathbb{N} \subset U$ because of the assumption made in Section 1 that **Boolean** $\in Sort_{\mathcal{A}}$ and **Natural** $\in Sort_{\mathcal{A}}$.

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 Stg_κ of storages, the set $InpQ_\kappa$ of input queues and the set $Timers_\kappa$ of timers as follows:

$$\begin{aligned} Stg_\kappa &= \bigcup_{V \subseteq V_\kappa} (V \xrightarrow{fin} U) \\ InpQ_\kappa &= ExtSig_\kappa{}^* \\ Timers_\kappa &= \bigcup_{T \subseteq Sig_\kappa} (T \xrightarrow{fin} \mathbb{N} \cup \{\mathsf{nil}\}) \end{aligned}$$

We will follow the convention that the domain of a function from Stg_κ 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_κ 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

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

defined below. The value of $getnxt(\sigma, ss)$ is the first (extended) signal in σ that is of a type different from the ones in ss. The value of $rmvfirst(\sigma, sig)$ is the input queue σ 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

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

and the function $rmvfirst$ is recursively defined by

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

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

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

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 ρ, σ, θ 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_κ 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:

$$\begin{aligned} &unitdelay\colon ChQ_\kappa \to ChQ_\kappa,\\ &arriving \;\colon ChQ_\kappa \times \mathbb{N} \to InpQ_\kappa,\\ &coming \;\;\colon ChQ_\kappa \to ChQ_\kappa \end{aligned}$$

The value of $unitdelay(\gamma)$ is the channel queue γ in which the delay duration of the first signal is decreased by one time unit. The value of $arriving(\gamma, i)$ is the longest prefix of γ that consists of signals with delay duration zero, weeded of signals with other receivers than i and stripped of delay durations. The value of $coming(\gamma)$ is the longest suffix of γ 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:

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

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

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

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

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.

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

We write $\mathit{cnt}(G)$, $\mathit{now}(G)$, $\mathit{chs}(G)$ and $\mathit{lsts}(G)$, where $G = (c, n, \Gamma, \Sigma) \in \mathcal{M}_\kappa$, for c, n, Γ and Σ, 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 $\mathcal{G}_\kappa$:

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

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

4.3 Actions and expressions

In this subsection, we will introduce the actions that are used for the semantics of φSDL. Most of the actions used are parametrized. The arguments of the instances of these actions are often values that depend on the state in which the instances are executed, or they have such values as constituents. The conditional operator $:\!\to$ is used to supply such instances with the right values (the required adaptation of the state operator is described in Section 3). The syntax of the expressions used in the conditions concerned is described in this subsection as well.

Actions:

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. $\lambda^s_G(a \cdot P) = sig(G) \frown a \cdot \lambda^s_G(P)$. These actions are called *inert* actions.

The state transforming actions are parametrized by various domains. In addition to the sets $\mathbb{N}_1$, $\mathbb{N}$, V_κ, C_κ, P_κ, Sig_κ and $ExtSig_\kappa$, the set $SigP_\kappa$ of signal patterns and the set $SaveSet_\kappa$ of save sets are used. We define $SigP_\kappa$ and $SaveSet_\kappa$ as follows:

$$\begin{aligned} SigP_\kappa &= S_\kappa \times {V_\kappa}^* \\ SaveSet_\kappa &= \mathcal{P}_{fin}(S_\kappa) \end{aligned}$$

The domain $SigP_\kappa$ differs slightly from $ExtSig_\kappa$ because it represents signal patterns, with variables used for the unknown values.

The following state transforming actions are used:

$$\begin{aligned} &input &&: SigP_\kappa \times SaveSet_\kappa \times \mathbb{N}_1 \\ &output &&: ExtSig_\kappa \times (C_\kappa \cup \{\mathsf{nil}\}) \times \mathbb{N} \\ &set &&: \mathbb{N} \times Sig_\kappa \times \mathbb{N}_1 \\ &reset &&: Sig_\kappa \times \mathbb{N}_1 \\ &ass &&: V_\kappa \times U \times \mathbb{N}_1 \\ &\overline{cr} &&: P_\kappa \times {V_\kappa}^* \times U^* \times (\mathbb{N}_1 \cup \{\mathsf{nil}\}) \\ &stop &&: \mathbb{N}_1 \\ &inispont &&: \mathbb{N}_1 \end{aligned}$$

These are the ACP actions that correspond to input guards, SDL actions, the terminator **stop** and the void guard **input none**. The second argument of an *input* action is the save set being in force. The third argument of an *output* action is the delay that the signal experiences if it must pass through a channel. The last argument of all actions is the pid value of the process from which the action originates, except for the *output* actions where the pid value concerned is available as the pid value of the sender in the first argument. Recall that the second and third argument of a $\overline{cr}$ 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 argument of an *output* action is a channel if the signal to be sent must pass through a channel, and nil otherwise. The last argument of a $\overline{cr}$ action is 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:

$$
\begin{array}{ll}
cr & : P_\kappa \times {V_\kappa}^* \times U^* \times (\mathbb{N}_1 \cup \{\mathsf{nil}\}) \\
input' & : ExtSig_\kappa \\
output' & : ExtSig_\kappa \\
set' & : \mathbb{N} \times Sig_\kappa \times \mathbb{N}_1 \\
reset' & : Sig_\kappa \times \mathbb{N}_1 \\
\not{t} & :
\end{array}
$$

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 $\not{t}$ is a special action with no observable effect whatsoever. It appears as the result of executing an instance of *ass*, $\overline{cr}$, *stop* or *inispont* as well as during system start-up as explained in Section 5.2.

The second argument of instances of cr and $\overline{cr}$ 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 of formal parameters.

Expressions:

As explained above, we also need expressions that stand for values that generally depend on the state in which they are evaluated. 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>}*)] , <pid vexpr>)
| *now*
| <value nm>
| <vexpr> = <vexpr>
| *cnt*
| *waiting* (<signal nm> {, <signal nm>}* , <pid vexpr>)
| *type* (<pid vexpr>)
| *hasinst* (<process nm>)

We assume that the terminal productions of <operator nm>, <variable nm>, <signal nm> and <process nm> yield the sets $Op_{\mathcal{A}}$, V_κ, S_κ and P_κ, respectively. We also assume that the

terminal productions of <value nm> yield a fixed set of variables in the sense of μCRL and that this set includes the special value name *self*.

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 SDL construct exactly. Expressions of the form $x = t$, where x is a value name, are used, together with the conditional operator $:\rightarrow$, to supply instances of parametrized actions with state dependent values. Expressions of the form $self = cnt$, are used to supply processes with their pid values. Expressions of the form $t_1 = t_2$ are, as a matter of course, also used to give meaning to SDL's decisions. Expressions of the form $waiting(s_1, \ldots, s_n, t)$ 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 $type(t)$ 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.

4.4 State transformers, observers and propositions

In the process algebra semantics of φSDL, 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 σ_{rel}), transform states. For the sake of comprehensibility, we will first define matching state transforming operations, and also some state observing operations.

Two of the state observing operations are used directly to define the state operator; the others are used to define the evaluation function for the value expressions introduced in Section 4.3 – such expressions stand for values that generally depend on the state in which they are evaluated. In the next subsection, we will define, in addition to the state operator, the above-mentioned evaluation function.

Every state from $\mathcal{G}_\kappa$ produces a proposition which is considered to hold in the state concerned. In this way, the state of a process is made partly visible. In this subsection, we will also define a function that gives for each state the proposition produced by that state. This function will be defined such that a state makes visible exactly what may be modified by SDL actions as well as be interrogated by SDL expressions. That is, the current value of all local variables and the

current set of active timers are made visible for all existing processes. It is obvious that some of the state observing operations used to define the evaluation function are also used to define this 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.

$$rcvsig((sig, i, i'), \langle v_1, \ldots, v_n \rangle, G) = \begin{cases} (cnt(G), now(G), chs(G), lsts(G) \oplus \{i' \mapsto (\rho, \sigma, \theta, X)\}) & \textbf{if } exists(i', G) \\ G & \textbf{otherwise} \end{cases}$$

$$\begin{aligned} \text{where } \rho &= stg(lsts(G)_{i'}) \oplus \{v_1 \mapsto vals(sig)_1, \ldots, v_n \mapsto vals(sig)_n, \textsf{sender} \mapsto i\}, \\ \sigma &= rmvfirst(inpq(lsts(G)_{i'}), sig), \\ \theta &= \{sig\} \lhd timers(lsts(G)_{i'}), \\ X &= ptype(lsts(G)_{i'}) \end{aligned}$$

$rcvsig((sig, i, i'), \langle v_1, \ldots, v_n \rangle, G)$ 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 $sndsig : ExtSig_\kappa \times (C_\kappa \cup \{\textsf{nil}\}) \times \mathbb{N} \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's output actions transform states.

$$sndsig((sig, i, i'), c, d, G) =$$

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

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

$sndsig((sig, i, i'), c, d, G)$ deals with passing signal 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:

- 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:

- sig is put into the queue of the delaying channel.

Everything else is left unchanged.

The function $settimer : \mathbb{N} \times Sig_\kappa \times \mathbb{N}_1 \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's set actions transform states.

$$
\begin{array}{ll}
settimer(t, sig, i, G) = & \\
\quad (cnt(G), now(G), chs(G), lsts(G) \oplus \{i \mapsto (\rho, \sigma, \theta, X)\}) & \textbf{if } exists(i, G) \\
\quad G & \textbf{otherwise}
\end{array}
$$

$$
\begin{array}{rcll}
\text{where } \rho & = & stg(lsts(G)_i), & \\
\sigma & = & rmvfirst(inpq(lsts(G)_i), sig) & \textbf{if } t > now(G) \\
 & & rmvfirst(inpq(lsts(G)_i), sig) \frown \langle (sig, i, i) \rangle & \textbf{otherwise}, \\
\theta & = & timers(lsts(G)_i) \oplus \{sig \mapsto t\} & \textbf{if } t > now(G) \\
 & & timers(lsts(G)_i) \oplus \{sig \mapsto \mathsf{nil}\} & \textbf{otherwise}, \\
X & = & ptype(lsts(G)_i) &
\end{array}
$$

$settimer(t, sig, i, G)$ deals with setting a timer, identified with signal 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 sig in the input queue originating from an earlier setting, if any, is removed;

- *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:

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

Everything else is left unchanged.

The function $resettimer : Sig_\kappa \times \mathbb{N}_1 \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's reset actions transform states.

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

$resettimer(sig, i, G)$ deals with resetting a timer, identified with signal *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 *sig* in the input queue originating from an earlier setting, if any, is removed;
- if *sig* is an active timer, it is removed from the active timers.

Everything else is left unchanged.

Notice that $settimer(t, sig, i, G)$ and $settimer(t, sig, i, resettimer(sig, i, G))$ 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 $assignvar : V_\kappa \times U \times \mathbb{N}_1 \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's assignment task actions transform states.

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

$$
\begin{array}{lcl}
\text{where } \rho & = & stg(lsts(G)_i) \oplus \{v \mapsto u\}, \\
\sigma & = & inpq(lsts(G)_i), \\
\theta & = & timers(lsts(G)_i), \\
X & = & ptype(lsts(G)_i)
\end{array}
$$

$assignvar(v,u,i,G)$ deals with assigning value u 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:

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

Everything else is left unchanged.

The function $createproc : P_\kappa \times {V_\kappa}^* \times U^* \times (\mathbb{N}_1 \cup \{\mathsf{nil}\}) \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's create actions transform states.

$$
\begin{array}{l}
createproc(X, \langle v_1,\ldots,v_n\rangle, \langle u_1,\ldots,u_n\rangle, i, G) \;=\; \\
\quad (cnt(G)+1, now(G), chs(G), \\
\quad\; lsts(G) \oplus \{cnt(G)+1 \mapsto (\rho,\sigma,\theta,X), i \mapsto (\rho',\sigma',\theta',X')\}) \;\textbf{if}\; exists(i,G) \\
\quad (cnt(G)+1, now(G), chs(G), \\
\quad\; lsts(G) \oplus \{cnt(G)+1 \mapsto (\rho,\sigma,\theta,X)\}) \qquad\qquad \textbf{if}\; i = \mathsf{nil} \\
\quad G \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{otherwise}
\end{array}
$$

$$
\begin{array}{lcl}
\text{where } \rho & = & \{v_1 \mapsto u_1, \ldots, v_n \mapsto u_n, \textbf{parent} \mapsto i\}, \\
\sigma & = & \langle\,\rangle, \\
\theta & = & \{\,\}, \\
\rho' & = & stg(lsts(G)_i) \oplus \{\textbf{offspring} \mapsto cnt(G)+1\}, \\
\sigma' & = & inpq(lsts(G)_i), \\
\theta' & = & timers(lsts(G)_i), \\
X' & = & ptype(lsts(G)_i)
\end{array}
$$

$createproc(X, \langle v_1,\ldots,v_n\rangle, \langle u_1,\ldots,u_n\rangle, i, G)$ 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 $u_1,\ldots,u_n$ 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 $stoppproc : \mathbb{N}_1 \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how ACP actions corresponding to SDL's **stop** transform states.

$$stoppproc(i, G) = (cnt(G), now(G), chs(G), \{i\} \lhd lsts(G))$$

$stoppproc(i, G)$ 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 $inispont : \mathbb{N}_1 \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how ACP actions used to initiate spontaneous transitions transform states.

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

$inispont(i, G)$ 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.

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

$$ptype(lsts(G')_i) = ptype(lsts(G)_i)$$

$unitdelay(G)$ 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 some 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 $nxtsig : SaveSet_\kappa \times \mathbb{N}_1 \times \mathcal{G}_\kappa \to ExtSig_\kappa \cup \{\mathsf{nil}\}$ and $nxtsignm : SaveSet_\kappa \times \mathbb{N}_1 \times \mathcal{G}_\kappa \to S_\kappa \cup \{\mathsf{nil}\}$ are used to define the result of executing ACP actions corresponding to SDL's input guards in a state.

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

$nxtsig(ss, i, G)$ 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.

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

$nxtsignm(ss, i, G)$ 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 $contents : V_\kappa \times \mathbb{N}_1 \times \mathcal{G}_\kappa \to U \cup \{\mathsf{nil}\}$ is used to describe the value of expressions of the form $value(v, t)$ which correspond to SDL's variable accesses and view expressions.

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

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

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

The function $\mathit{is\text{-}active} : \mathit{Sig}_\kappa \times \mathbb{N}_1 \times \mathcal{G}_\kappa \to \mathbb{B}$ is used to describe the value of expressions of the form $\mathit{active}(\mathit{sig},t)$ which correspond to SDL's active expressions.

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

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

The function $\mathit{is\text{-}waiting} : \mathit{SaveSet}_\kappa \times \mathbb{N}_1 \times \mathcal{G}_\kappa \to \mathbb{B}$ is used to describe the value of expressions of the form $\mathit{waiting}(s_1,\ldots,s_n,t)$ which are used to give meaning to SDL's state definitions.

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

$\mathit{is\text{-}waiting}(ss,i,G)$ yields true iff there is no 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 $\mathit{type} : \mathbb{N} \times \mathcal{G}_\kappa \to P_\kappa \cup \{\textbf{env}, \mathsf{nil}\}$ is used to describe the value of expressions of the form $\mathit{type}(t)$ which are used to give meaning to SDL's output actions with explicit addressing.

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

$\mathit{type}(i,G)$ 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. **env**.

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

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

$\mathit{has\text{-}instance}(X,G)$ yields true iff there exists a process of type X.

State propositions:

The propositions produced by states from $\mathcal{G}_\kappa$ can be built from a set $Atom_\kappa$ of atomic propositions, true, false, and the connectives $\neg$ and $\rightarrow$. We consider conjunctions and disjunctions abbreviations as usual. We define the set $Atom_\kappa$ as follows:

$$Atom_\kappa = \{value(v,u,i) \mid (v,u,i) \in V_\kappa \times U \times \mathbb{N}_1\} \cup \{active(sig,i) \mid (sig,i) \in Sig_\kappa \times \mathbb{N}_1\}$$

We write $Prop_\kappa$ for the set of all propositions that can be built as described above, and Lit_κ for $Atom_\kappa \cup \{\neg\phi \mid \phi \in Atom_\kappa\}$. An atomic proposition of the form $value(v,u,i)$ is intended to indicate that u is the value of the local variable v of the process with pid value i. An atomic proposition of the form $active(sig,i)$ is intended to indicate that the timer of the process with pid value i identified with signal sig is active. By using only atomic propositions of these forms, the state of a process can not be made fully visible via the proposition produced. The proposition produced by each state, given by the function sig defined below, makes only visible the value of all local variables and the set of active timers for all existing processes.

First, we define the function $lits : \mathcal{G}_\kappa \rightarrow \mathcal{P}(Lit_\kappa)$ giving for each state the set of literals, i.e. atomic propositions and negated atomic propositions, that hold in that state. It is inductively defined by

$$contents(v,i,G) = u \Rightarrow value(v,u,i) \in lits(G)$$
$$contents(v,i,G) \neq u \Rightarrow (\neg value(v,u,i)) \in lits(G)$$
$$is\text{-}active(sig,i,G) = \mathsf{true} \Rightarrow active(sig,i) \in lits(G)$$
$$is\text{-}active(sig,i,G) \neq \mathsf{true} \Rightarrow (\neg active(sig,i)) \in lits(G)$$

We define now the function $sig : \mathcal{G}_\kappa \rightarrow Prop_\kappa$ as follows:

$$sig(G) = \bigwedge_{\phi \in lits(G)} \phi$$

So $sig(G)$ is the conjunction of all atomic propositions and negated atomic propositions that hold in state G.

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 $\mathcal{G}_\kappa$.[7] Because progress of time transforms states as well, we will also define the result of

[7] We follow the convention that, for each equation $\lambda^s_G(a \cdot P) = sig(G) \frown a' \cdot \lambda^s_{G'}(P)$, the equation $\lambda^s_G(a) = sig(G) \frown a'$ is implicit.

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 t 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. The action a' that appears as the result of executing a state transforming action a in a state G is reminiscent to a, provided the action is concerned with communication or timing. In case of an input action, the connection is most loose. An input action a has a signal pattern, a save set and a pid value as its arguments and the corresponding action a' has an extended signal matching this pattern as its sole argument. If the action is not concerned with communication or timing, the special action $\not t$ appears as the result of executing it.

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 $\underline{\underline{inispont}}(u)$ 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. Different from the other cases, the execution of an action $\underline{\underline{input}}((s, \langle v_1, \ldots, v_n \rangle), ss, X)$ may fail in certain states. It fails if the type of the first signal in the input queue of the process with pid value i 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.

$$\begin{aligned}
&\lambda^s_G(\underline{\underline{input}}((s, \langle v_1, \ldots, v_n \rangle), ss, i) \cdot P) \;=\\
&\quad sig(G) \frown \underline{\underline{input'}}(isig) \cdot \lambda^s_{rcvsig(isig, \langle v_1, \ldots, v_n \rangle, G)}(P) \;\textbf{if } nxtsignm(ss, i, G) = s\\
&\quad sig(G) \frown \delta \qquad\qquad\qquad\qquad\qquad\qquad \textbf{otherwise}\\
&\quad \text{where } isig = nxtsig(ss, i, G)
\end{aligned}$$

$$\lambda^s_G(\underline{\underline{output}}(osig, c, d) \cdot P) = sig(G) \frown \underline{\underline{output'}}(osig) \cdot \lambda^s_{sndsig(osig, c, d, G)}(P)$$

$$\lambda^s_G(\underline{\underline{set}}(t, tsig, i) \cdot P) = sig(G) \frown \underline{\underline{set'}}(t, tsig, i) \cdot \lambda^s_{settimer(t, tsig, i, G)}(P)$$

$$\lambda^s_G(\underline{\underline{reset}}(tsig, i) \cdot P) = sig(G) \frown \underline{\underline{reset'}}(tsig, i) \cdot \lambda^s_{resettimer(tsig, i, G)}(P)$$

$$\lambda^s_G(\underline{\underline{ass}}(v, u, i) \cdot P) = sig(G) \frown \underline{\underline{\not t}} \cdot \lambda^s_{assignvar(v, u, i, G)}(P)$$

$$\lambda^s_G(\underline{\underline{cr}}(X, fpar, apar, i) \cdot P) = sig(G) \frown \underline{\underline{\not t}} \cdot \lambda^s_{createproc(X, fpar, apar, i, G)}(P)$$

$$\lambda^s_G(\underline{\underline{stop}}(i) \cdot P) \;=\; sig(G) \frown \underline{\underline{tt}} \cdot \lambda^s_{stopproc(i,G)}(P)$$

$$\lambda^s_G(\underline{\underline{inispont}}(i) \cdot P) \;=\; sig(G) \frown \underline{\underline{tt}} \cdot \lambda^s_{inispont(i,G)}(P)$$

Recall that for each inert action a, we simply have

$$\lambda^s_G(a \cdot P) = sig(G) \frown a \cdot \lambda^s_G(P)$$

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:

$$\lambda^s_G(\sigma_{\mathsf{rel}}(P)) \;=\; sig(G) \frown \sigma_{\mathsf{rel}}(\textstyle\sum_{G' \in unitdelay(G)} \lambda^s_{G'}(P))$$

Evaluation function:

We will end this section with defining the evaluation function that is used to describe the value of an expression t 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 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 fixed set of value names ranged over by the meta-variable x is also used as a set of variables in the sense of μCRL.

$$\begin{array}{l} eval^s_G(op(t_1,\ldots,t_n)) \;= \\ \quad op(eval^s_G(t_1),\ldots,eval^s_G(t_n)) \;\textbf{if}\; eval^s_G(t_1) \neq \mathsf{nil} \wedge \ldots \wedge eval^s_G(t_n) \neq \mathsf{nil} \\ \quad \mathsf{nil} \qquad\qquad\qquad\qquad\qquad\quad \textbf{otherwise} \end{array}$$

$$\begin{array}{l} eval^s_G(cond(t_1,t_2,t_3)) \;= \\ \quad eval^s_G(t_2) \;\textbf{if}\; eval^s_G(t_1) = \mathsf{true} \\ \quad eval^s_G(t_3) \;\textbf{if}\; eval^s_G(t_1) = \mathsf{false} \\ \quad \mathsf{nil} \qquad \textbf{otherwise} \end{array}$$

$$\begin{array}{l} eval^s_G(value(v,t)) \;= \\ \quad contents(v, eval^s_G(t), G) \;\textbf{if}\; eval^s_G(t) \in \mathbb{N}_1 \\ \quad \mathsf{nil} \qquad\qquad\qquad\quad \textbf{otherwise} \end{array}$$

$$\begin{array}{l} eval^s_G(active(s(t_1,\ldots,t_n),t)) \;= \\ \quad is\text{-}active(sig, eval^s_G(t), G) \;\textbf{if}\; eval^s_G(t_1) \neq \mathsf{nil} \wedge \ldots \wedge eval^s_G(t_n) \neq \mathsf{nil} \wedge eval^s_G(t) \in \mathbb{N}_1 \\ \quad \mathsf{nil} \qquad\qquad\qquad\qquad\quad \textbf{otherwise} \end{array}$$

where $sig = (s, \langle eval^s_G(t_1), \ldots, eval^s_G(t_n)\rangle)$

$$eval^s_G(now) = now(G)$$

$$eval^s_G(x) = x$$

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

$$\begin{array}{l} eval^s_G(t_1 = t_2) = \\ \quad \mathsf{true} \ \textbf{if} \ eval^s_G(t_1) = eval^s_G(t_2) \wedge eval^s_G(t_1) \neq \mathsf{nil} \wedge eval^s_G(t_2) \neq \mathsf{nil} \\ \quad \mathsf{false} \ \textbf{if} \ eval^s_G(t_1) \neq eval^s_G(t_2) \wedge eval^s_G(t_1) \neq \mathsf{nil} \wedge eval^s_G(t_2) \neq \mathsf{nil} \\ \quad \mathsf{nil} \quad \textbf{otherwise} \end{array}$$

$$eval^s_G(cnt) = cnt(G)$$

$$\begin{array}{l} eval^s_G(waiting(s_1, \ldots, s_n, t)) = \\ \quad is\text{-}waiting(\{s_1, \ldots, s_n\}, eval^s_G(t), G) \ \textbf{if} \ eval^s_G(t) \in \mathbb{N}_1 \\ \quad \mathsf{nil} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{otherwise} \end{array}$$

$$\begin{array}{l} eval^s_G(type(t)) = \\ \quad type(eval^s_G(t), G) \ \textbf{if} \ eval^s_G(t) \in \mathbb{N} \\ \quad \mathsf{nil} \qquad\qquad\qquad\qquad \textbf{otherwise} \end{array}$$

$$eval^s_G(hasinst(X)) = has\text{-}instance(X, G)$$

5 Process algebra semantics

In this section, we will present a process algebra semantics of φSDL. 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 φSDL is defined by interpretation functions, one for each syntactic category, which are all written in the form $[\![\bullet]\!]^\kappa$. The superscript κ 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. If an optional clause represents a sequence, its absence is always taken to stand for an empty sequence. Otherwise, it is treated as a separate case.

In the presented semantics, we will use the following abbreviation. Let $a(u_1, \ldots, u_n)$ be an instance of a parametrized action $a : D_1 \times \ldots \times D_n$ where $D_i \subseteq U$ $(1 \leq i \leq n)$ and let

t_i be a value expression. Then we write $a(u_1, \ldots, u_{i-1}, t_i, u_{i+1}, \ldots, u_n)$ for $\sum_{x_i:D_i} x_i = t_i :\rightarrow a(u_1, \ldots, u_{i-1}, x_i, u_{i+1}, \ldots, u_n)$. We will also use the obvious extensions of this notation to the cases where D_i is somehow composed of subsets of U and other domains by cartesian product. Note that this abbreviation covers exactly the use of the conditional operator $:\rightarrow$ mentioned in Section 4.3: to supply instances of parametrized actions with state dependent values.

In Section 5.1, we will use the term system input stream for a stream of signals that a system can receive via signal routes from the environment. Details concerning system input streams are given in Appendix C.

5.1 System definition

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

- ζ is a mapping from system input streams to process expressions describing the behaviour of the system from its start-up for all possible system input streams;
- ϕ is the mapping from process names to process expressions that is to be associated with the process creation operator used in the process expressions in the range of ζ;
- 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 ϕ;
- $\mathcal{G}$ is the state space that is to be associated with the state operator used in the process expressions in the range of ζ.

The first component depends on the definitions of signal types, channels and signal routes, on the names introduced by the process definitions, 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 (ϕ, E) where:

- ϕ is a 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 ϕ 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 ϕ 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.

$$
\begin{aligned}
&[\![\textsf{system}\ S; D_1 \ldots D_n\ \textsf{endsystem};]\!] := \\
&\quad (\{\alpha \mapsto \tau_{I\cup\{t\}}(\lambda^s_{G_0}(E_\phi(P) \parallel Env(\alpha))) \mid \alpha \in \mathcal{I}_\kappa\}, \\
&\qquad\qquad [\![D_1]\!]^\kappa_\phi \cup \ldots \cup [\![D_n]\!]^\kappa_\phi, [\![D_1]\!]^\kappa_E \cup \ldots \cup [\![D_n]\!]^\kappa_E, \mathcal{G}_\kappa) \\
&\quad \text{where } P = \|_{X\in procs(\kappa)}\ (\|^{init(\kappa,X)}\ \underline{\underline{cr}}(X, \langle\,\rangle, \langle\,\rangle, \mathsf{nil})), \\
&\qquad\quad G_0 = (0, 0, \{c \mapsto \langle\,\rangle \mid c \in chans(\kappa)\}, \{\,\}), \\
&\qquad\quad \kappa = \{\!|\textsf{system}\ S; D_1 \ldots D_n\ \textsf{endsystem};|\!\}
\end{aligned}
$$

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

$[\![D]\!]^\kappa := (\{\,\}, \{\,\})$ 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;**

The set $\mathcal{I}_\kappa$ of possible system input streams and the notation $Env(\alpha)$ for the process generating the system input stream α are defined in Appendix C. In the case of a system definition, the process expression $\tau_{I\cup\{t\}}(\lambda^s_{G_0}(E_\phi(P) \parallel Env(\alpha)))$ 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 while it receives a stream α of signals via signal routes from the environment. Additionally, the internal action t as well as the actions in I are hidden. The set 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 related to communication and timing 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 $\{X \mapsto \sum_{self:\mathbb{N}} cnt = self :\to [\![tr]\!]^{\kappa'}\}$ expresses that, for each process of the type X, its behaviour is the behaviour determined by the given start transition tr in case *self* stands for the last issued pid value. The process equations $[\![d_1]\!]^{\kappa'}, \ldots, [\![d_n]\!]^{\kappa'}$ describe how the process X behaves from each of the n states in which it may come while it proceeds.

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.

$$\begin{aligned}
&[\![\textbf{state } st\textbf{; save } s_1,\ldots,s_m\textbf{;}alt_1\ \ldots\ alt_n]\!]^{\kappa} := \\
&\quad X_{st} = \neg waiting(s_1,\ldots,s_m,self) :\rightarrow ([\![alt_1]\!]^{\kappa'} + \ldots + [\![alt_n]\!]^{\kappa'}) + \\
&\qquad\quad waiting(s_1,\ldots,s_m,self) :\rightarrow \sigma_{\mathsf{rel}}(X_{st}) \\
&\quad \text{where } X = scopeunit(\kappa), \\
&\qquad\quad \kappa' = updsaveset(\kappa,\{s_1,\ldots,s_m\})
\end{aligned}$$

$$\begin{aligned}
&[\![\textbf{input } s\textbf{(}v_1,\ldots,v_n\textbf{);}tr]\!]^{\kappa} := \\
&\quad (lt(cnt,n_0) :\rightarrow \underline{\underline{tt}})^{*} \cdot (\neg lt(cnt,n_0) :\rightarrow \underline{\underline{input}}((s,\langle v_1,\ldots,v_n\rangle),ss,self)\cdot [\![tr]\!]^{\kappa}) \\
&\quad \text{where } n_0 = \textstyle\sum_{X\in procs(\kappa)} init(\kappa,X),^{8} \\
&\qquad\quad ss = saveset(\kappa)
\end{aligned}$$

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

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 $\underline{\underline{input}}((s,\langle v_1,\ldots,v_n\rangle),ss,self)\cdot[\![tr]\!]^{\kappa}$ 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.

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

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

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

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

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.

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

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

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).

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

$$[\![\textbf{output } s\textbf{(}e_1, \ldots, e_n\textbf{) via } r_1, \ldots, r_m;]\!]^\kappa :=$$
$$(lt(cnt, n_0) :\to \underline{\underline{tt}})^*$$
$$(\neg lt(cnt, n_0) :\to (\textstyle\sum_{i:\mathbb{N}}(type(i) = X_1 :\to P_1 + \ldots + type(i) = X_m :\to P_m) +$$

$$\neg(hasinst(X_1) \wedge \ldots \wedge hasinst(X_m)) :\rightarrow \underline{\underline{\#}})$$

where $n_0 = \sum_{X \in procs(\kappa)} init(\kappa, X)$,

for $1 \leq j \leq m$:

$$P_j = \begin{cases} \underline{\underline{output}}(((s, \langle [\![e_1]\!], \ldots, [\![e_n]\!] \rangle), self, i), c_j, 0) & \textbf{if } c_j = \mathsf{nil} \\ \sum_{d:\mathbb{N}} \underline{\underline{output}}(((s, \langle [\![e_1]\!], \ldots, [\![e_n]\!] \rangle), self, i), c_j, d) & \textbf{otherwise,} \end{cases}$$

$$X_j = rcv(\kappa, r_j),$$

$$c_j = ch(\kappa, r_j)$$

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

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

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

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

All cases except the ones for output actions are straightforward. The cases of output actions need 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 $type(t) = X_j$ 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 $\neg(type([\![e]\!]) = X_1 \vee \ldots \vee type([\![e]\!]) = X_m) :\rightarrow \underline{\underline{\#}}$. 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 $\neg(hasinst(X_1) \wedge \ldots \wedge hasinst(X_m)) :\rightarrow \underline{\underline{\#}}$. 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 $\sum_{d:\mathbb{N}} \underline{\underline{output}}(sig, c, d)$ 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.

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

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

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

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

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

$$[\![\textbf{now}]\!] := now$$

$$[\![\textbf{self}]\!] := self$$

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

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

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

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

In [11], timed frames, which are closely related to the kind of transition systems used for the operational semantics of $\mathrm{ACP}_{\mathrm{drt}}$, are studied in a general algebraic setting and results concerning the connection between timed frames and discrete time processes are given. In [12], a general first-order logic of timed frames, called TFL, is proposed and results concerning its strong distinguishing power and its connections with the logics underlying two model checkers are given. The results of the work reported in [11] and [12], together with the semantics of φSDL given in this paper, are meant to be used to devise a general logic of discrete time processes, and to adapt an existing model checker to φSDL and a fragment of this logic where model checking is feasible.

In a forthcoming paper, process creation is left out from φSDL. In this way we confine ourselves there to the most distinctive features of SDL and allow a more intelligible presentation of their semantics. Besides, an example is used in that paper to illustrate how time related behavioural aspects of systems specified in this restricted version of φSDL can be analysed using the process algebra semantics.

Presently, we are also starting to elaborate a more abstract semantics for SDL, based on dataflow networks. The intended result is expected to provide convincing mathematical arguments in favor of the choice of concepts concerning storage, communication, timing and process creation around which SDL has been set up. By way of preparation, we have studied dataflow networks in a general algebraic setting [15].

In [16] 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 [17] and [19] 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.

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 W.R. Cleaveland, editor, *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. Process algebra with propositional signals. Logic Group Preprint Series 123, Utrecht University, Department of Philosophy, November 1994.

[5] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra. Report P9208c, University of Amsterdam, Programming Research Group, March 1995. To appear in Formal Aspects of Computing.

[6] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra with abstraction. In H. Reichel, editor, *Fundamentals of Computation Theory*, pages 1–15. LNCS 965, Springer-Verlag, 1995.

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

[8] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification*. Prentice-Hall, 1991.

[9] 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.

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

[11] J.A. Bergstra, W.J. Fokkink, and C.A. Middelburg. Algebra of timed frames. Logic Group Preprint Series 148, Utrecht University, Department of Philosophy, November 1995.

[12] J.A. Bergstra, W.J. Fokkink, and C.A. Middelburg. A logic for signal inserted timed frames. Logic Group Preprint Series 155, Utrecht University, Department of Philosophy, January 1996.

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

[14] J.A. Bergstra and C.A. Middelburg. Process algebra semantics of φSDL. Logic Group Preprint Series 129, Utrecht University, Department of Philosophy, March 1995.

[15] J.A. Bergstra, C.A. Middelburg, and Gh. Ştefănescu. Network algebra for synchronous and asynchronous dataflow. Report P9508, University of Amsterdam, Programming Research Group, October 1995.

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

[17] 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.

[18] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics (extended abstract). In G.X. Ritter, editor, *Information Processing 89*, pages 613–618. North-Holland, 1989. Full version: Report CS-9120, CWI.

[19] 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.

[20] J.F. Groote and A. Ponse. Proof theory for μCRL: 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.

[21] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. 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.

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

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

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

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

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

[27] SDL predefined data. ITU-T Recommendation Z.100 D, Revision 1, 1994. Annex D to Recommendation Z.100.

[28] 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.

[29] 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.

[30] 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 φSDL 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 $\mathcal{P}_{fin}(A)$ 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 $f : A \xrightarrow{fin} B$ instead. We write $dom(f)$, 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

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

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 unary operators hd and tl stand for selection of head and tail, respectively, of sequences. 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 φSDL 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*.

These names are in addition connected with their static attributes. For example, a name of a variable is connected with the name of the sort of the values that may be assigned to it; and a name of a process type 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.

$$\begin{array}{l} \mathit{Context} = \\ \quad \mathcal{P}_{\mathit{fin}}(\mathit{VarD}) \times \mathcal{P}_{\mathit{fin}}(\mathit{SigD}) \times \mathcal{P}_{\mathit{fin}}(\mathit{ChanD}) \times \mathcal{P}_{\mathit{fin}}(\mathit{RouteD}) \times \mathcal{P}_{\mathit{fin}}(\mathit{ProcD}) \times \\ \quad (\mathit{ProcId} \cup \{\mathsf{nil}\}) \times \mathcal{P}_{\mathit{fin}}(\mathit{SigId}) \end{array}$$

where

$$\begin{array}{lcl} \mathit{VarD} & = & \mathit{VarId} \times \mathit{SortId} \\ \mathit{SigD} & = & \mathit{SigId} \times \mathit{SortId}^* \\ \mathit{ChanD} & = & \mathit{ChanId} \\ \mathit{RouteD} & = & \mathit{RouteId} \times (\mathit{ProcId} \cup \{\mathbf{env}\}) \times (\mathit{ProcId} \cup \{\mathbf{env}\}) \times \mathcal{P}_{\mathit{fin}}(\mathit{SigId}) \times \\ & & (\mathit{ChanId} \cup \{\mathsf{nil}\}) \\ \mathit{ProcD} & = & \mathit{ProcId} \times \mathit{VarId}^* \times \mathbb{N} \end{array}$$

We write $\mathit{vards}(\kappa)$, $\mathit{sigds}(\kappa)$, $\mathit{chands}(\kappa)$, $\mathit{routeds}(\kappa)$, $\mathit{procds}(\kappa)$, $\mathit{scopeunit}(\kappa)$ and $\mathit{saveset}(\kappa)$, where $\kappa = (V, S, C, R, P, X, ss) \in \mathit{Context}$, for V, S, C, R, P, X and ss, respectively. We write $\mathit{vars}(\kappa)$ for $\{v \mid \exists T \cdot (v, T) \in \mathit{vards}(\kappa)\}$. The abbreviations $\mathit{sigs}(\kappa)$, $\mathit{chans}(\kappa)$ and $\mathit{procs}(\kappa)$ are used analogously. 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

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

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

$$
\begin{array}{l}
(r, X_1, X_2, ss, c) \in \mathit{routeds}(\kappa) \Rightarrow \mathit{rcv}(\kappa, r) = X_2,\\
(r, X_1, X_2, ss, c) \in \mathit{routeds}(\kappa) \Rightarrow \mathit{ch}(\kappa, r) = c
\end{array}
$$

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

$$
\begin{array}{l}
(X, vs, k) \in \mathit{procds}(\kappa) \Rightarrow \mathit{fpars}(\kappa, X) = vs,\\
(X, vs, k) \in \mathit{procds}(\kappa) \Rightarrow \mathit{init}(\kappa, X) = k
\end{array}
$$

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

$$
\begin{array}{l}
\kappa = (V, S, C, R, P, X, ss) \Rightarrow \mathit{updscopeunit}(\kappa, X') = (V, S, C, R, P, X', ss),\\
\kappa = (V, S, C, R, P, X, ss) \Rightarrow \mathit{updsaveset}(\kappa, ss') = (V, S, C, R, P, X, ss')
\end{array}
$$

The function *envd* is used to determine the possible system input streams, i.e. streams of signals that the system can receive via signal routes from the environment. It is inductively defined by

$$
\begin{array}{l}
s \in ss \wedge (s, \langle T_1, \ldots, T_n\rangle) \in \mathit{sigds}(\kappa) \wedge (r, \textbf{env}, X_2, ss, c) \in \mathit{routeds}(\kappa) \Rightarrow\\
\quad (s, \langle T_1, \ldots, T_n\rangle, c) \in \mathit{envd}(\kappa)
\end{array}
$$

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 save set which are permanently the same – nil and { }, respectively.

$$
\begin{array}{l}
\{\!|\textbf{system}\ S;D_1 \ldots D_n\ \textbf{endsystem;}|\!\} :=\\
\quad (\mathit{vards}(\{\!|D_1|\!\}) \cup \ldots \cup \mathit{vards}(\{\!|D_n|\!\}),\\
\quad \mathit{sigds}(\{\!|D_1|\!\}) \cup \ldots \cup \mathit{sigds}(\{\!|D_n|\!\}),\\
\quad \mathit{chands}(\{\!|D_1|\!\}) \cup \ldots \cup \mathit{chands}(\{\!|D_n|\!\}),\\
\quad \mathit{routeds}(\{\!|D_1|\!\}) \cup \ldots \cup \mathit{routeds}(\{\!|D_n|\!\}),
\end{array}
$$

$procds(\{\![D_1]\!\}) \cup \ldots \cup procds(\{\![D_n]\!\}),$
$\mathsf{nil}, \{\,\})$

$\{\![\textbf{dcl}\ v\ T;]\!\} := (\{(v, T)\}, \{\,\}, \{\,\}, \{\,\}, \{\,\}, \mathsf{nil}, \{\,\})$

$\{\![\textbf{signal}\ s\textbf{(}\ T_1, \ldots, T_n\textbf{);}]\!\} := (\{\,\}, \{(s, \langle T_1, \ldots, T_m\rangle)\}, \{\,\}, \{\,\}, \{\,\}, \mathsf{nil}, \{\,\})$

$\{\![\textbf{channel}\ c;]\!\} := (\{\,\}, \{\,\}, \{c\}, \{\,\}, \{\,\}, \mathsf{nil}, \{\,\})$

$\{\![\textbf{signalroute}\ r\ \textbf{from}\ X_1\ \textbf{to}\ X_2\ \textbf{with}\ s_1, \ldots, s_n\ \textbf{delayed by}\ c;]\!\} :=$
$(\{\,\}, \{\,\}, \{\,\}, \{(r, X_1, X_2, \{s_1, \ldots, s_n\}, c)\}, \{\,\}, \mathsf{nil}, \{\,\})$

$\{\![\textbf{signalroute}\ r\ \textbf{from}\ X_1\ \textbf{to}\ X_2\ \textbf{with}\ s_1, \ldots, s_n;]\!\} :=$
$(\{\,\}, \{\,\}, \{\,\}, \{(r, X_1, X_2, \{s_1, \ldots, s_n\}, \mathsf{nil})\}, \{\,\}, \mathsf{nil}, \{\,\})$

$\{\![\textbf{process}\ X\textbf{(}k\textbf{);}\ \textbf{fpar}\ v_1, \ldots, v_m\textbf{;}\ \textbf{start;}\ tr\ \ d_1 \ldots d_n\ \textbf{endprocess;}]\!\} :=$
$(\{\,\}, \{\,\}, \{\,\}, \{\,\}, \{(X, \langle v_1, \ldots, v_m\rangle, k)\}, \mathsf{nil}, \{\,\})$

C System environment

The set of possible system input streams depends upon the specific types of signals and signal routes introduced in the system definition concerned. The function *envd*, defined in Appendix B, is meant for the extraction of the relevant information: applying *envd* to the context ascribed to a system definition yields an environment description for the system concerned. From this environment description, the set of signals that the system can receive via signal routes from the environment can be determined. In order to put such a signal into the right channel queues, it is needed that the delaying channel in its route, if it is present, is attached to it. That is, they are elements of $EnvSig_\kappa \subseteq ExtSig_\kappa \times (C_\kappa \cup \{\mathsf{nil}\})$. We define the set of such environment signals for arbitrary contexts κ:

$$EnvSig_\kappa = \bigcup_{((s,\langle T_1,\ldots,T_n\rangle),c)\in envd(\kappa)} \{(((s, \langle u_1, \ldots, u_n\rangle), 0, i), c) \,|\, u_1 \in T_1, \ldots, u_n \in T_n, i \in \mathbb{N}_1\}$$

We write $xsig(esig)$ and $chan(esig)$, where $esig = (xsig, c)$, for $xsig$ and c, respectively.

The set $\mathcal{I}_\kappa$ of possible system input streams is now defined by

$$\mathcal{I}_\kappa = (EnvSig_\kappa{}^*)^\infty$$

That is, a possible system input stream α is an infinite sequence of finite sequences of environment signals; $(\alpha_i)_j$ $(i > 0,\ 0 < j \leq len(\alpha_i))$ is the jth signal arriving during the ith time slice from the start-up of the system.

We use the notation $Env(\alpha)$, where $\alpha \in \mathcal{I}_\kappa$, for the process generating the system input stream α.

$$Env(\alpha) \;=\; \lambda^e_\alpha(X_{env}) \text{ where } X_{env} \;=\; \underline{\underline{envinp}} \cdot X_{env} + \sigma_{\mathsf{rel}}(X_{env})$$

The state operator for the environment is defined below.

$$\begin{array}{l}
\lambda^e_\alpha(\underline{\underline{envinp}} \cdot P) \;= \\
\quad P' \textbf{ if } hd(\alpha) \neq \langle\,\rangle \\
\quad \delta \quad \textbf{otherwise} \\
\quad \text{where } P' \;=\; \underline{\underline{output}}(osig, c, 0) \cdot \lambda^e_{tl(hd(\alpha))^\frown tl(\alpha)}(P) \qquad \textbf{if } c = \mathsf{nil} \\
\qquad\qquad\qquad\quad \sum_{d:\mathbb{N}}(\underline{\underline{output}}(osig, c, d) \cdot \lambda^e_{tl(hd(\alpha))^\frown tl(\alpha)}(P)) \;\; \textbf{otherwise}, \\
\qquad\qquad osig \;=\; xsig(hd(hd(\alpha))), \\
\qquad\qquad c \;=\; chan(hd(hd(\alpha)))
\end{array}$$

$$\begin{array}{l}
\lambda^e_\alpha(\sigma_{\mathsf{rel}}(P)) \;= \\
\quad \sigma_{\mathsf{rel}}(\lambda^e_{tl(\alpha)}(P)) \textbf{ if } hd(\alpha) = \langle\,\rangle \\
\quad \delta \qquad\qquad\qquad \textbf{otherwise}
\end{array}$$

We see immediately that the jth signal generated by $Env(\alpha)$ during the ith time slice from its start-up is $(\alpha_i)_j$ (for $i > 0$, $0 < j \leq len(\alpha_i)$).