# Discrete Time Process Algebra and the Semantics of SDL

J.A. Bergstra[1,3], C.A. Middelburg[2,3], Y.S. Usenko[4,*]

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

[2] Computing Science Department, Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, the Netherlands

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

[4] Cluster of Software Engineering, CWI
P.O. Box 94079, 1090 GB Amsterdam, the Netherlands

janb@wins.uva.nl, keesm@win.tue.nl, ysu@cwi.nl

**Abstract**

We present an extension of discrete time process algebra with relative timing where recursion, propositional signals and conditions, a counting process creation operator, and the state operator are combined. Except the counting process creation operator, which subsumes the original process creation operator, these features have been developed earlier as largely separate extensions of time free process algebra. The change to the discrete time case and the combination of the features turn out to be far from trivial. We also propose a semantics for a simplified version of SDL, using this extension of discrete time process algebra to describe the meaning of the language constructs. This version covers all behavioural aspects of SDL, except for communication via delaying channels – which can easily be modelled. The semantics presented here facilitates the generation of finitely branching transition systems for SDL specifications and thus it enables validation.

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

*1994 CR Categories:* D.2.1, D.3.1, F.3.1, F.3.2

# 1 Introduction

In this chapter, we present an extension of discrete time process algebra with relative timing where recursion, propositional signals and conditions, a counting process creation operator, and the state operator are combined. We also propose a semantics for

*The work presented in this chapter has been partly carried out while the second and third author were at UNU/IIST (United Nations University, International Institute for Software Technology) in Macau.

a simplified version of SDL, called $\varphi^-$SDL, using this extension of discrete time process algebra with relative timing to describe the meaning of the language constructs. The choice of a process algebra in the style of ACP [8, 9] as the basis of the presented semantics is obvious. This algebraic approach to concurrency represents a large body of relevant theory. In particular, many features of $\varphi^-$SDL are related to topics that have been studied extensively in the framework of ACP. Besides, the axiom system and operational semantics of an ACP-style process algebra facilitate advances in the areas of validation and verification.

We take the remainder of this introductory section to introduce $\varphi^-$SDL, to motivate the choices made in the selection of this dramatically simplified version of SDL, and to describe its close connection with full SDL. We also explain the need for a semantics that deals properly with the time related aspects of SDL in case one intends to validate SDL specifications, or to justify design steps made using SDL by formal verification.

## 1.1 Background

At present, SDL [10, 18] is widely used in telecommunications for describing structure and behaviour of generally complex systems at different levels of abstraction. It originated from an informal graphical description technique already commonly used in the telecommunications field at the time of the first computer controlled telephone switches. Our starting-point is the version of SDL defined in [33], the ITU-T Recommendation Z.100 published in 1994. There, a subset of SDL, called Basic SDL, is identified and used to describe the meaning of the language constructs of SDL that are not in Basic SDL. This subset is still fairly complicated.

$\varphi^-$SDL is a simplified version of Basic SDL.[1] The following simplifications have been made:

- blocks and channels are removed;
- all variables are revealed and they can be viewed freely;
- timer setting is regarded as just a special use of signals;
- timer setting is based on discrete time.

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

We decided to focus in $\varphi^-$SDL on the behavioural aspects of SDL. We did so for the following two reasons. Firstly, the structural aspects of SDL are mostly of a static nature and therefore not very relevant from a semantic point of view. Secondly, the part of SDL that deals with the specification of abstract data types is well understood – besides, it can easily be isolated and treated as a parameter.[2] For practical reasons,

[1] This subset is called $\varphi^-$SDL, where $\varphi$ stands for flat, as it does not cover the structural aspects of SDL, and $^-$ indicates that delaying channels are left out.

[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 [31]); (2) ASN.1 [32] is widely used for data type specification in the telecommunications field, and there is an ITU-T Recommendation, Z.105, for combining SDL and ASN.1 (see [36]).

we also chose not to include initially procedures, syntypes with a range condition and process types with a bound on the number of instances that may exist simultaneously. Similarly, the **any** expression is omitted as well. Services are not supported by $\varphi^-$SDL for the following reasons: the semantics of services is hard to understand, ETSI forbids for this reason their use in European telecommunication standards (see [31]), and the SDL community discusses its usefulness.

In [14], we introduced a simplified version of SDL, called $\varphi$SDL, which covers all behavioural aspects of SDL, including communication via delaying channels. $\varphi^-$SDL is $\varphi$SDL without communication via delaying channels. The process algebra semantics of $\varphi$SDL proposed in [14] made clear that $\varphi$SDL specifications can always be transformed to semantically equivalent ones in $\varphi^-$SDL. Apart from the data type definitions, SDL specifications can be transformed to $\varphi$SDL specifications, and hence to $\varphi^-$SDL specifications, provided that no use is made of facilities that are not included initially. The transformation from SDL to $\varphi$SDL 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-T Recommendation Z.100 [33] and defined in a fully precise manner in its Annex F.2 [34]. The second and final part is essentially the mapping *extract-dict* defined in its Annex F.3 [35].

The semantics of $\varphi^-$SDL agrees with the semantics of SDL as far as reasonably possible. This means in the first place that obvious errors in [35] 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 [35] – allowing amongst other things that a process ceases to exist while a signal is sent to it instantaneously. 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, real numbers are used for times and durations. So when a timer is set, its expiration time is given by a real number. However, the time considered is the system time which proceeds actually in a discrete manner: the system receives ticks from the environment which increase the system time with a certain amount (how much real time they represent is left open). Therefore, the timer is considered to expire when the system receives the first tick that indicates that its expiration time has passed. So nothing is lost by adopting in $\varphi^-$SDL a discrete time approach, using natural numbers for times and durations, 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.

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-T 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 [26]. They provided convincing practical justification for the premise of our current choices: communication with the environment takes a good deal of time, whereas internal processing takes a negligible deal of time. Ease of adaptation to other viewpoints on time in SDL is guaranteed relatively well by using a discrete time process algebra, $\mathrm{PA}^-_{\mathrm{drt}}$ (see [5]) without immediate deadlock, as the basis of the presented semantics.

In the telecommunications field, SDL is increasingly used for describing generally

complex telecommunications systems, including switching systems, services and protocols, at different levels of abstraction – from initial specification till implementation. Initial specification of systems is done with the intention to analyse the behavioural properties of these systems and thus to validate the specification. There is also a growing need to verify whether the properties represented by one specification are preserved in another, more concrete, specification and thus to justify design steps. However, neither SDL nor the tools and techniques that are used in conjunction with SDL provide appropriate support for validation of SDL specifications and verification of design steps made using SDL. The main reason is that the semantics of SDL according to the ITU-T Recommendation Z.100 is at some points inadequate for advanced validation and formal verification. In particular, the semantics of time related features, such as timers and delaying channels, is insufficiently precise. Moreover, the semantics is at some other points unnecessarily complex. Consequently, rules of logical reasoning, indispensable for formal verification, have not yet been developed and most existing analysis tools, e.g. GEODE [2] and SDT [37], offer at best a limited kind of model checking for validation.

Prerequisites for advanced validation 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 $\varphi^-$SDL and the presented semantics for it are primarily intended to come up to these prerequisites.

## 1.2 Organization of this chapter

The structure of this chapter is as follows. In Section 2, we present the extension of discrete time process algebra with relative timing that is used for the process algebra semantics of $\varphi^-$SDL proposed in Section 4. An overview of $\varphi^-$SDL is given in Section 3. Following the overview, in Section 4, we present the proposed semantics of $\varphi^-$SDL in two steps. First, a semantics of $\varphi^-$SDL process definitions, which are the main elements of $\varphi^-$SDL specifications, is given. This semantics abstracts from dynamic aspects of process behaviour such as process creation and process execution in a state. A semantics of $\varphi^-$SDL system definitions, i.e. complete $\varphi^-$SDL specifications, is then given in terms of the semantics of $\varphi^-$SDL process definitions using the counting process creation operator and the state operator. In Section 5, we give an overview of related work and we explain how the semantics presented in Section 4 can be used to transform $\varphi^-$SDL specifications to transition systems that can be used for advanced validation. There are appendices about notational conventions used (Appendix A) and details concerning the contexts used to model scope in the presented semantics (Appendix B). Small examples of specification in $\varphi^-$SDL and the meaning of the process definitions being found in these examples are also presented (in Section 3 and Section 4, respectively).

## Acknowledgements

For one month all three authors were at UNU/IIST (United Nations University, International Institute for Software Technology) in Macau. During that period, some of the more important corrections and technical changes of the material presented here

were made. We thank Dines Bjørner, the former director of UNU/IIST, for bringing us together in Macau. The third author thanks Radu Şoricuţ and Bogdan Warinschi for their helpful comments and discussions.

# 2 Process algebra

## 2.1 Introduction

In this section, we present an extension of discrete time process algebra with relative timing where recursion, propositional signals and conditions, a counting process creation operator, and the state operator are combined. Its signature, axioms and a structural operational semantics are given and it is shown that strong bisimulation equivalence is a congruence for all operations. Except the counting process creation operator, which subsumes the original process creation operator, these features have been developed earlier as largely separate extensions of time free process algebra. However, both the change to the discrete time case and the combination of the features turn out to be far from trivial. Besides, some of the features are slightly adapted versions of the original ones in order to meet the needs of the semantics of $\varphi^-$SDL.

In Section 2.2, we present discrete relative time process algebra without immediate deadlock and delayable actions ($\mathrm{PA}^-_{\mathrm{drt}}$-ID). In Section 2.3, we add propositional signals and conditions to $\mathrm{PA}^-_{\mathrm{drt}}$-ID. In the discrete relative time case, this addition requires some axioms to be refined. We introduce a new guarded command operator that yields a deadlock in the current time slice if the condition does not hold at the start, i.e. waiting is no option if the condition does not hold. In Section 2.4, we add recursion to the extension presented in Section 2.3. The main definitions related to recursion, such as the definitions of recursive specification, solution and guardedness, are given here for the case with relative timing in discrete time as well as propositional signals and conditions.

In Section 2.5 and 2.6, we describe the counting process creation operator and the state operator, respectively, for the discrete relative time case in the presence of propositional signals and conditions. The counting process creation operator is a straighforward extension of the original process creation operator that allows to assign a unique "process identification value" to each process created. The state operator presented here allows to deal with conditions whose truth depends on the state and with state changes due to progress of time to the next time slice.

The main reference to discrete time process algebra in the style of ACP is [5]. The features with which it is combined here are discussed as separate extensions of time free process algebra in [6] (propositional signals and conditions, state operator), [8] (recursion) and [11] (process creation). Our discussion of axioms is concentrated on the crucial axioms for the discrete time case and each of these features, and on the alterations and additions needed if all this is combined. For a systematic introduction to process algebra in the style of ACP, the reader is referred to [8] and [9].

## 2.2 Discrete relative time process algebra

In this subsection, we present discrete relative time process algebra without immediate deadlock and delayable actions. The term discrete time is used here to indicate that

time is divided into time slices and timing of actions is done with respect to the time slices in which they are performed – within a time slice there is only the order in which actions are performed. Additionally, performance of actions and passage to the next time slice are separated here. This corresponds to the two-phase functioning scheme for modeling timed processes [25]. Note that it means that processes are supposed to be capable of performing certain actions, like in time free process algebra, as well as passing to the next time slice. A coherent collection of versions of ACP with timing where performance of actions and passage of time are separated, is presented in [7].

First we treat the basic discrete relative time process algebra $\mathrm{BPA}^-_{\mathrm{drt}}$-ID. Then we treat $\mathrm{PA}^-_{\mathrm{drt}}$-ID, the extension of $\mathrm{BPA}^-_{\mathrm{drt}}$-ID with parallel composition in which no communication between processes is involved. $\mathrm{ACP}^-_{\mathrm{drt}}$-ID, the extension of $\mathrm{BPA}^-_{\mathrm{drt}}$-ID with parallel composition in which synchronous communication between processes is involved, will not be treated. $\mathrm{BPA}^-_{\mathrm{drt}}$-ID, $\mathrm{PA}^-_{\mathrm{drt}}$-ID and $\mathrm{ACP}^-_{\mathrm{drt}}$-ID are presented in detail in [27]. We also present the extension of $\mathrm{PA}^-_{\mathrm{drt}}$-ID with encapsulation, described before for the discrete relative time case without immediate deadlock in [4].

### 2.2.1 Basic process algebra

In $\mathrm{BPA}^-_{\mathrm{drt}}$-ID, we have the sort $\mathsf{P}$ of processes, the constants $\underline{\underline{a}}$ (one for each action $a$) and $\underline{\underline{\delta}}$, the unary operator $\sigma_{\mathsf{rel}}$ (time unit delay), and the binary operators $\cdot$ (sequential composition) and $+$ (alternative composition). The constants $\underline{\underline{a}}$ stand for $a$ in the current time slice. Similarly, the constant $\underline{\underline{\delta}}$ stands for a deadlock in the current time slice. The process $\sigma_{\mathsf{rel}}(x)$ is the process $x$ delayed till the next time slice. The process $x \cdot y$ is the process $x$ followed after successful termination by the process $y$. The process $x + y$ is the process that proceeds with either the process $x$ or the process $y$, but not both. We also have the auxiliary unary operator $\nu_{\mathsf{rel}}$ (now) in $\mathrm{BPA}^-_{\mathrm{drt}}$-ID. This operator makes an elegant axiomatization of $\mathrm{PA}^-_{\mathrm{drt}}$-ID possible. The process $\nu_{\mathsf{rel}}(x)$ is the part of $x$ that is not delayed till the next time slice.

It is assumed that a fixed but arbitrary set $A$ of *actions* has been given.

**Signature of $\mathrm{BPA}^-_{\mathrm{drt}}$-ID** The signature of $\mathrm{BPA}^-_{\mathrm{drt}}$-ID consists of the *undelayable action* constants $\underline{\underline{a}} : \to \mathsf{P}$ (for each $a \in A$), the *undelayable deadlock* constant $\underline{\underline{\delta}} : \to \mathsf{P}$, the *alternative composition* operator $+ : \mathsf{P} \times \mathsf{P} \to \mathsf{P}$, the *sequential composition* operator $\cdot : \mathsf{P} \times \mathsf{P} \to \mathsf{P}$, the *time unit delay* operator $\sigma_{\mathsf{rel}} : \mathsf{P} \to \mathsf{P}$, and the *now* operator $\nu_{\mathsf{rel}} : \mathsf{P} \to \mathsf{P}$.

We assume that an infinite set of variables (of sort $\mathsf{P}$) has been given. Given the signature of $\mathrm{BPA}^-_{\mathrm{drt}}$-ID, terms of $\mathrm{BPA}^-_{\mathrm{drt}}$-ID, often referred to as process expressions, are constructed in the usual way. The need to use parentheses is reduced by ranking the precedence of the operators. Throughout this chapter we adhere to the following precedence rules: (i) all unary operators have the same precedence, (ii) unary operators have a higher precedence than binary operators, (iii) the operator $\cdot$ has the highest precedence amongst the binary operators, (iv) the operator $+$ has the lowest precedence amongst the binary operators, and (v) all other binary operators have the same precedence. We will also use the following abbreviation. Let $(p_i)_{i \in \mathcal{I}}$ be an indexed set of terms of $\mathrm{BPA}^-_{\mathrm{drt}}$-ID where $\mathcal{I} = \{i_1, \ldots, i_n\}$. Then we write $\sum_{i \in \mathcal{I}} p_i$ for $p_{i_1} + \ldots + p_{i_n}$. We further use the convention that $\sum_{i \in \mathcal{I}} p_i$ stands for $\underline{\underline{\delta}}$ if $\mathcal{I} = \emptyset$.

We denote variables by $x, x', y, y', \ldots$. We use the convention that $a, a', b, b', \ldots$ denote elements of $A \cup \{\delta\}$ in the context of an equation, but elements of $A$ in the

context of an operational semantics rule. Furthermore, $H$ denotes a subset of $A$. We write $A_\delta$ for $A \cup \{\delta\}$.

**Axioms of $\mathrm{BPA}^-_{\mathrm{drt}}$-ID** The axiom system of $\mathrm{BPA}^-_{\mathrm{drt}}$-ID consists of the equations A1-A5, DRT1-DRT4A and DCS1-DCS4 given in Table 1.

| | | | |
|---|---|---|---|
| $x + y = y + x$ | A1 | $\sigma_{\mathsf{rel}}(x) + \sigma_{\mathsf{rel}}(y) = \sigma_{\mathsf{rel}}(x + y)$ | DRT1 |
| $(x + y) + z = x + (y + z)$ | A2 | $\sigma_{\mathsf{rel}}(x) \cdot y = \sigma_{\mathsf{rel}}(x \cdot y)$ | DRT2 |
| $x + x = x$ | A3 | $\underline{\underline{\delta}} \cdot x = \underline{\underline{\delta}}$ | DRT3 |
| $(x + y) \cdot z = x \cdot z + y \cdot z$ | A4 | $x + \underline{\underline{\delta}} = x$ | DRT4A |
| $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ | A5 | $\nu_{\mathsf{rel}}(\underline{a}) = \underline{a}$ | DCS1 |
| | | $\nu_{\mathsf{rel}}(x + y) = \nu_{\mathsf{rel}}(x) + \nu_{\mathsf{rel}}(y)$ | DCS2 |
| | | $\nu_{\mathsf{rel}}(x \cdot y) = \nu_{\mathsf{rel}}(x) \cdot y$ | DCS3 |
| | | $\nu_{\mathsf{rel}}(\sigma_{\mathsf{rel}}(x)) = \underline{\underline{\delta}}$ | DCS4 |

Table 1: Axioms of $\mathrm{BPA}^-_{\mathrm{drt}}$-ID ($a \in A_\delta$)

Axioms DRT1 and DRT2 represent the interaction of time unit delay with alternative composition and sequential composition, respectively. Axiom DRT1, called the time factorization axiom, expresses that passage to the next time slice by itself can not determine a choice. Axiom DRT2 expresses that timing is relative to the performance of the previous action.

In [27], a structural operational semantics of $\mathrm{BPA}^-_{\mathrm{drt}}$-ID is presented and proofs are given of the soundness and completeness of the axiom system of $\mathrm{BPA}^-_{\mathrm{drt}}$-ID for the set of closed $\mathrm{BPA}^-_{\mathrm{drt}}$-ID terms modulo (strong) *bisimulation* equivalence. This notion is precisely defined in [8]. Roughly, bisimilarity of two processes means that if one process is capable of doing a certain step, i.e. performing some action or passing to the next time slice, and next going on as a certain follow-up process then the other process is capable of doing the same step and next going on as a process bisimilar to the follow-up process.

### 2.2.2 Parallel composition

In $\mathrm{PA}^-_{\mathrm{drt}}$-ID, we have, in addition to sequential and alternative composition, parallel composition of processes. In $\mathrm{PA}^-_{\mathrm{drt}}$-ID, unlike in $\mathrm{ACP}^-_{\mathrm{drt}}$-ID, parallel composition does not involve communication between processes. The parallel composition operator $\parallel$ of $\mathrm{PA}^-_{\mathrm{drt}}$-ID is called free merge to indicate that no communication is involved. The process $x \parallel y$ is the process that proceeds simultaneously with the processes $x$ and $y$. In order to get a finite axiomatization, we also have the auxiliary operator $\between$ (left merge) in $\mathrm{PA}^-_{\mathrm{drt}}$-ID. The processes $x \between y$ and $x \parallel y$ are the same except that $x \between y$ must start with a step of $x$.

**Signature of $\mathrm{PA}^-_{\mathrm{drt}}$-ID** The signature of $\mathrm{PA}^-_{\mathrm{drt}}$-ID is the signature of $\mathrm{BPA}^-_{\mathrm{drt}}$-ID extended with the *free merge* operator $\parallel\colon \mathsf{P} \times \mathsf{P} \to \mathsf{P}$ and the *left merge* operator $\between\colon \mathsf{P} \times \mathsf{P} \to \mathsf{P}$.

We will use the following abbreviation. Let $(p_i)_{i \in \mathcal{I}}$ be an indexed set of terms of $\mathrm{PA}^-_{\mathrm{drt}}$-ID where $\mathcal{I} = \{i_1, \ldots, i_n\}$. Then, we write $\parallel_{i \in \mathcal{I}} p_i$ for $p_{i_1} \parallel \ldots \parallel p_{i_n}$.

| | |
|---|---|
| $x \parallel y = x \between y + y \between x$ | DRTM1 |
| $\underline{\underline{a}} \between x = \underline{\underline{a}} \cdot x$ | DRTM2 |
| $\underline{\underline{a}} \cdot x \between y = \underline{\underline{a}} \cdot (x \parallel y)$ | DRTM3 |
| $(x + y) \between z = x \between z + y \between z$ | DRTM4 |
| $\sigma_{\mathsf{rel}}(x) \between \nu_{\mathsf{rel}}(y) = \underline{\underline{\delta}}$ | DRTM5 |
| $\sigma_{\mathsf{rel}}(x) \between (\nu_{\mathsf{rel}}(y) + \sigma_{\mathsf{rel}}(z)) = \sigma_{\mathsf{rel}}(x \between z)$ | DRTM6 |

Table 2: Additional axioms for $\mathrm{PA}^{-}_{\mathrm{drt}}$-ID ($a \in A_\delta$)

**Axioms of $\mathrm{PA}^{-}_{\mathrm{drt}}$-ID** The axiom system of $\mathrm{PA}^{-}_{\mathrm{drt}}$-ID consists of the axioms of $\mathrm{BPA}^{-}_{\mathrm{drt}}$-ID and the equations DRTM1-DRTM6 given in Table 2.

Axioms DRTM5 and DRTM6 represent the interaction between time unit delay and left merge. These axioms express that passage to the next time slice of parallel processes must synchronize.

In [27], a structural operational semantics of $\mathrm{PA}^{-}_{\mathrm{drt}}$-ID is presented and proofs are given of the soundness and completeness of the axiom system of $\mathrm{PA}^{-}_{\mathrm{drt}}$-ID for the set of closed $\mathrm{PA}^{-}_{\mathrm{drt}}$-ID terms modulo (strong) bisimulation equivalence.

### 2.2.3 Encapsulation

We extend the signature of $\mathrm{PA}^{-}_{\mathrm{drt}}$-ID with the encapsulation operator $\partial_H : \mathsf{P} \to \mathsf{P}$. This operator turns all undelayable actions $\underline{\underline{a}}$, where $a$ in $H \subseteq A$, into undelayable deadlock. The encapsulation operator is defined by the equations DRTD1-DRTD5 given in Table 3.

| | |
|---|---|
| $\partial_H(\underline{\underline{a}}) = \underline{\underline{a}}$ if $a \notin H$ | DRTD1 |
| $\partial_H(\underline{\underline{a}}) = \underline{\underline{\delta}}$ if $a \in H$ | DRTD2 |
| $\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$ | DRTD3 |
| $\partial_H(x + y) = \partial_H(x) + \partial_H(y)$ | DRTD4 |
| $\partial_H(\sigma_{\mathsf{rel}}(x)) = \sigma_{\mathsf{rel}}(\partial_H(x))$ | DRTD5 |

Table 3: Axioms for encapsulation ($a \in A_\delta$)

An operational semantics of encapsulation is presented in [4].

## 2.3 Propositional signals and conditions

In [6], process algebra with propositional signals and conditions is introduced for the time free case. In this subsection, we adapt it for discrete relative time. The result is referred to by $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$. In later sections, we will call propositional signals "propositions" in order to avoid ambiguity with signals of $\varphi^{-}$SDL.

In process algebra with propositional signals and conditions, propositions are used both as signals that are emitted by processes and as conditions that are imposed on processes to proceed. Condition testing is looked upon as signal inspection. The intuition is that the signal emitted by a process, as well as each of its logical consequences, holds at the start of the process. The signal emitted by a process is called its root signal.

Like in the time free case we have $\bot$ (non-existence) as additional constant and $\mathbin{\frown\!\!\!\rightarrow}$ (root signal emission) and $:\!\!\rightarrow$ (guarded command) as additional operators. Like the constant $\underline{\underline{\delta}}$, the constant $\bot$ stands for a process that is incapable of doing any step and incapable of terminating successfully. In addition, going on as $\bot$ after performing an action is impossible. The process $\phi \mathbin{\frown\!\!\!\rightarrow} x$ is the process $x$ where the proposition $\phi$ holds at its start. Broadly speaking, the process $\phi :\!\!\rightarrow x$ is the process that may proceed as the process $\nu_{\mathsf{rel}}(x)$ if the proposition $\phi$ holds at its start, but may also proceed as the process $\sigma_{\mathsf{rel}}(\phi :\!\!\rightarrow y)$ in case $x = \nu_{\mathsf{rel}}(x) + \sigma_{\mathsf{rel}}(y)$. In other words, with the guarded command operator $:\!\!\rightarrow$, it is possible to wait till a proposition holds. This agrees with the original intention of the operator to make actions conditional.

We also have a non-waiting version of the guarded command operator, namely the operator $\mathbin{:\!\!:\!\!\rightarrow}$ (strict guarded command). The process $\phi \mathbin{:\!\!:\!\!\rightarrow} x$ is the process that proceeds as the process $x$ if the proposition $\phi$ holds at its start, and otherwise yields a deadlock in the current time slice. Both guarded commands agree with the one in the time free case for processes that are not capable of passing to the next time slice.

Lifting propositional signals and conditions to the discrete time case requires the axioms for left merge to be adapted. If these axioms are not adapted, the equation $\sigma_{\mathsf{rel}}(x \between y) = \underline{\underline{\delta}}$ becomes derivable – unless axiom NE1 (Table 5) is not adopted. Without adaptations, the root signal of $x \between y$ would be the root signal of the process $x$. With the chosen adaptations, the root signal of $x \between y$ is the conjunction of the root signals of the processes $x$ and $y$. Thus, different from the time free case, the process $x \between y$ is neither capable of performing an action nor capable of passing to the next time slice if the root signal of $y$ is equal to $\mathsf{f}$. This difference is not relevant to the free merge operator because the root signal of $x \parallel y$ is the conjunction of the root signals of $x$ and $y$ anyhow.

It is assumed that a fixed but arbitrary set $B_{\mathrm{at}}$ of *atomic propositions* has been given. From now on we have, in addition to the sort $\mathsf{P}$ of processes, the sort $\mathsf{B}$ of propositions over $B_{\mathrm{at}}$; with constants $\mathsf{t}$, $\mathsf{f}$ (true, false) and operators $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$ (negation, disjunction, conjunction, implication, bi-implication). In case $B_{\mathrm{at}}$ is empty, $\mathsf{B}$ represents the boolean algebra over the set $\mathbb{B} = \{\mathsf{t}, \mathsf{f}\}$.

We denote propositions by $\phi, \psi, \ldots$. In derivations we may always use logical equivalences of (classical) propositional logic. So we are actually using equivalence classes of formulas, with respect to logical equivalence, instead of the formulas themselves.

A *valuation* $v$ of atomic formulas is a function $v : B_{\mathrm{at}} \to \mathbb{B}$. Any valuation $v$ can be extended to $\mathsf{B}$ in the usual homomorphic way, i.e.:

$v(\kappa) = \kappa$ for the constants $\kappa \in \{\mathsf{t}, \mathsf{f}\}$,
$v(\neg\phi) = \neg v(\phi)$,
$v(\phi \mathbin{o} \psi) = v(\phi) \mathbin{o} v(\psi)$ for the binary operators $o \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$.

We will use the same name for a valuation $v$ and its extension to $\mathsf{B}$. If a proposition $\phi$ is satisfied by a valuation $v$ ($v(\phi) = \mathsf{t}$), we write $v \models \phi$ to indicate this.

**Signature of $\mathbf{PA}^{\mathrm{psc}}_{\mathrm{drt}}$** The signature of $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$ is the signature of the $\mathrm{PA}^{-}_{\mathrm{drt}}$-ID extended with the encapsulation operator, the *non-existence* constant $\bot : \to \mathsf{P}$, the *strict guarded command* operator $\mathbin{:\!\!:\!\!\rightarrow}\colon \mathsf{B} \times \mathsf{P} \to \mathsf{P}$, the *root signal emission* operator $\mathbin{\frown\!\!\!\rightarrow} : \mathsf{B} \times \mathsf{P} \to \mathsf{P}$ and the *weak guarded command* operator $:\!\!\rightarrow\colon \mathsf{B} \times \mathsf{P} \to \mathsf{P}$.

**Axioms of $\mathbf{PA}^{\mathrm{psc}}_{\mathrm{drt}}$** The axiom system of $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$ consists of the axioms of $\mathrm{BPA}^{-}_{\mathrm{drt}}$-ID and the equations DRTM1, DRTM4 from Table 2, the equations DRTM2′, DRTM3′,

DRTM5′, DRTM6′ from Table 4, the equations NE1-NE3 from Table 5, the equations SGC1-SGC6, MSGC from Table 6, the equations SRSE1-SRSE7, MSRSE from Table 7, the equations DCS5, DCS6, DRTM7 from Table 8, the equations DGC1-DGC8 from Table 9, the equations DRTD1-DRTD5 from Table 3, and the equations PD1-PD3 from Table 10.

| | |
|---|---|
| $\underline{\underline{a}} \between x = \underline{\underline{a}} \cdot x + \partial_A(\nu_{\mathsf{rel}}(x))$ | DRTM2′ |
| $\underline{\underline{a}} \cdot x \between y = \underline{\underline{a}} \cdot (x \parallel y) + \partial_A(\nu_{\mathsf{rel}}(y))$ | DRTM3′ |
| $\sigma_{\mathsf{rel}}(x) \between \nu_{\mathsf{rel}}(y) = \partial_A(\nu_{\mathsf{rel}}(y))$ | DRTM5′ |
| $\sigma_{\mathsf{rel}}(x) \between (\nu_{\mathsf{rel}}(y) + \sigma_{\mathsf{rel}}(z)) = \sigma_{\mathsf{rel}}(x \between z) + \partial_A(\nu_{\mathsf{rel}}(y))$ | DRTM6′ |

Table 4: Adapted axioms for left merge ($a \in A_\delta$)

| | |
|---|---|
| $x + \bot = \bot$ | NE1 |
| $\bot \cdot x = \bot$ | NE2 |
| $\underline{\underline{a}} \cdot \bot = \underline{\underline{\delta}}$ | NE3 |

Table 5: Axioms for non-existence ($a \in A_\delta$)

| | |
|---|---|
| $\mathsf{t} \mathrel{:}\!\!\rightarrow x = x$ | SGC1 |
| $\mathsf{f} \mathrel{:}\!\!\rightarrow x = \underline{\underline{\delta}}$ | SGC2 |
| $\phi \mathrel{:}\!\!\rightarrow (x + y) = (\phi \mathrel{:}\!\!\rightarrow x) + (\phi \mathrel{:}\!\!\rightarrow y)$ | SGC3 |
| $(\phi \vee \psi) \mathrel{:}\!\!\rightarrow x = (\phi \mathrel{:}\!\!\rightarrow x) + (\psi \mathrel{:}\!\!\rightarrow x)$ | SGC4 |
| $\phi \mathrel{:}\!\!\rightarrow (\psi \mathrel{:}\!\!\rightarrow x) = (\phi \wedge \psi) \mathrel{:}\!\!\rightarrow x$ | SGC5 |
| $\phi \mathrel{:}\!\!\rightarrow (x \cdot y) = (\phi \mathrel{:}\!\!\rightarrow x) \cdot y$ | SGC6 |
| $(\phi \mathrel{:}\!\!\rightarrow x) \between y = \phi \mathrel{:}\!\!\rightarrow (x \between y)$ | MSGC |

Table 6: Axioms for strict guarded command

| | |
|---|---|
| $(\phi \mathrel{\frown\!\!\rightarrow} x) \cdot y = \phi \mathrel{\frown\!\!\rightarrow} (x \cdot y)$ | SRSE1 |
| $(\phi \mathrel{\frown\!\!\rightarrow} x) + y = \phi \mathrel{\frown\!\!\rightarrow} (x + y)$ | SRSE2 |
| $\phi \mathrel{\frown\!\!\rightarrow} (\psi \mathrel{\frown\!\!\rightarrow} x) = (\phi \wedge \psi) \mathrel{\frown\!\!\rightarrow} x$ | SRSE3 |
| $\mathsf{t} \mathrel{\frown\!\!\rightarrow} x = x$ | SRSE4 |
| $\mathsf{f} \mathrel{\frown\!\!\rightarrow} x = \bot$ | SRSE5 |
| $\phi \mathrel{:}\!\!\rightarrow (\psi \mathrel{\frown\!\!\rightarrow} x) = (\phi \to \psi) \mathrel{\frown\!\!\rightarrow} (\phi \mathrel{:}\!\!\rightarrow x)$ | SRSE6 |
| $\phi \mathrel{\frown\!\!\rightarrow} (\phi \mathrel{:}\!\!\rightarrow x) = \phi \mathrel{\frown\!\!\rightarrow} x$ | SRSE7 |
| $(\phi \mathrel{\frown\!\!\rightarrow} x) \between y = \phi \mathrel{\frown\!\!\rightarrow} (x \between y)$ | MSRSE |

Table 7: Axioms for root signal emission

| | |
|---|---|
| $\nu_{\mathsf{rel}}(\phi \mathrel{:}\!\!\rightarrow x) = \phi \mathrel{:}\!\!\rightarrow \nu_{\mathsf{rel}}(x)$ | DCS5 |
| $\nu_{\mathsf{rel}}(\phi \mathrel{\frown\!\!\rightarrow} x) = \phi \mathrel{\frown\!\!\rightarrow} \nu_{\mathsf{rel}}(x)$ | DCS6 |
| $\sigma_{\mathsf{rel}}(x) \between ((\phi \mathrel{:}\!\!\rightarrow y) + z) = (\phi \mathrel{:}\!\!\rightarrow (\sigma_{\mathsf{rel}}(x) \between (y + z))) + (\neg\phi \mathrel{:}\!\!\rightarrow (\sigma_{\mathsf{rel}}(x) \between z))$ | DRTM7 |

Table 8: Additional axioms for time unit delay and now operator

| | |
|---|---|
| $\phi :\to \underline{\underline{\delta}} = \underline{\underline{\delta}}$ | DGC1 |
| $\phi :\to \underline{\underline{a}} = \phi :\dot{\to} \underline{\underline{a}}$ | DGC2 |
| $\phi :\to (\underline{\underline{a}} \cdot x) = (\phi :\to \underline{\underline{a}}) \cdot x$ | DGC3 |
| $\phi :\to (\psi :\dot{\to} x) = \psi :\dot{\to} (\phi :\to x)$ | DGC4 |
| $\phi :\to (x + y) = (\phi :\to x) + (\phi :\to y)$ | DGC5 |
| $\phi :\to \sigma_{\mathsf{rel}}(x) = \sigma_{\mathsf{rel}}(\phi :\to x)$ | DGC6 |
| $\phi :\to \nu_{\mathsf{rel}}(x) = \nu_{\mathsf{rel}}(\phi :\to x)$ | DGC7 |
| $\phi :\to (\psi \mathbin{\frown\!\!\!\blacktriangleright} x) = \psi \mathbin{\frown\!\!\!\blacktriangleright} (\phi :\to x)$ | DGC8 |

Table 9: Axioms for weak guarded command ($a \in A_\delta$)

| | |
|---|---|
| $\partial_H(\phi :\dot{\to} x) = \phi :\dot{\to} \partial_H(x)$ | PD1 |
| $\partial_H(\phi :\to x) = \phi :\to \partial_H(x)$ | PD2 |
| $\partial_H(\phi \mathbin{\frown\!\!\!\blacktriangleright} x) = \phi \mathbin{\frown\!\!\!\blacktriangleright} \partial_H(x)$ | PD3 |

Table 10: Additional axioms for encapsulation

The axioms A3, DRT3 and DRT4A of $\mathrm{BPA}^-_{\mathrm{drt}}$-ID (Table 1) are derivable from the axioms SGC1, SGC2, SGC4 and SGC6 (Table 6).

The axioms DRTM2′, DRTM3′, DRTM5′ and DRTM6′ (Table 4) are the axioms DRTM2, DRTM3, DRTM5 and DRTM6 of $\mathrm{PA}^-_{\mathrm{drt}}$-ID (Table 2) where a summand is added to the right hand side of the axioms, viz. $\partial_A(\nu_{\mathsf{rel}}(x))$ in case of DRTM2 and $\partial_A(\nu_{\mathsf{rel}}(y))$ in case of the other axioms. Thus is expressed that the root signal of the left merge of two processes is always the conjunction of the root signals of both processes.

The axioms NE1-NE3, SGC1-SGC6, SRSE1-SRSE7, MSGC and MSRSE (Tables 5-7) are straightforward reformulations of corresponding axioms for the time free case, i.e. axioms of $\mathrm{PA}_{\mathrm{ps}}$, given in [6]. The constants $a$ and the constant $\delta$ have been replaced by the constants $\underline{\underline{a}}$ and the constant $\underline{\underline{\delta}}$, respectively; and the operator $:\to$ has been replaced by the operator $:\dot{\to}$. The axioms NE1 and NE2 (Table 5) are derivable from the axiom A1 of $\mathrm{BPA}^-_{\mathrm{drt}}$-ID and the axioms SRSE1, SRSE2 and SRSE5 (Table 7). Axiom NE3 expresses that going on as $\bot$ after performing an action is impossible. We do not have $\sigma_{\mathsf{rel}}(\bot) = \underline{\underline{\delta}}$, an equation expressing that going on as $\bot$ after passing to the next time slice is impossible. The reason is that $\sigma_{\mathsf{rel}}(x) = \underline{\underline{\delta}}$ would become derivable. In view of this mismatch between $\underline{\underline{a}} \cdot \bot$ and $\sigma_{\mathsf{rel}}(\bot)$, in retrospect, axiom NE3 may be considered to be a wrong choice in [6]. Axiom SRSE5 expresses that a process where falsity holds at its start is non-existent. The crucial axioms are SRSE6 and SRSE7 which represent the interaction between the root signal emission operator and the strict guarded command. Axiom SRSE6 expresses that if a proposition holds at the start of a process and that process is guarded by another proposition then at the start of the whole the former proposition holds or the latter proposition does not hold. Axiom SRSE7 expresses that it is superfluous to guard a process by a proposition if the proposition holds at the start of the whole.

The additional axioms DCS5, DCS6 and DRTM7 (Table 8) are needed because propositional signals and conditions are lifted to the discrete time case. The strict guarded command is non-waiting, i.e. we do not have $\phi :\dot{\to} \sigma_{\mathsf{rel}}(x) = \sigma_{\mathsf{rel}}(\phi :\dot{\to} x)$. Root signal emission is non-persistent, i.e. we do not have $\phi \mathbin{\frown\!\!\!\blacktriangleright} \sigma_{\mathsf{rel}}(x) = \sigma_{\mathsf{rel}}(\phi \mathbin{\frown\!\!\!\blacktriangleright} x)$. Axiom DRTM7 is necessary for the elimination of parallel composition. This axiom

expresses that if processes are capable of passing to the next time slice conditionally then their parallel composition can do so if all conditions concerned hold. Note that from DRTM7 we can derive $\sigma_{\mathsf{rel}}(x) \between (\phi \mathrel{:}\!\!\rightarrow y) = \phi \mathrel{:}\!\!\rightarrow (\sigma_{\mathsf{rel}}(x) \between y)$.

The axioms DGC1-DGC8 (Table 9) define the weak guarded command with which, unlike with the strict guarded command, it is possible to wait till a proposition holds. From these axioms we can derive

$$x = \nu_{\mathsf{rel}}(x) \;\Rightarrow\; \phi :\!\rightarrow x = (\phi \mathrel{:}\!\!\rightarrow x) + \partial_A(\nu_{\mathsf{rel}}(x))$$
$$x = \nu_{\mathsf{rel}}(x) + \sigma_{\mathsf{rel}}(y) \;\Rightarrow\; \phi :\!\rightarrow x = (\phi \mathrel{:}\!\!\rightarrow \nu_{\mathsf{rel}}(x)) + \sigma_{\mathsf{rel}}(\phi :\!\rightarrow y) + \partial_A(\nu_{\mathsf{rel}}(x))$$

which gives a full picture of the differences between the two guarded commands.

**Semantics of $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$** We shall give a structural operational semantics for $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$ using rules in the style of Plotkin to define the following unary and binary relations on the closed terms of $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$:

| | | |
|---|---|---|
| a unary relation | $\_ \xrightarrow{v,a} \surd$ | for each valuation $v$ and $a \in A$, |
| a binary relation | $\_ \xrightarrow{v,a} \_$ | for each valuation $v$ and $a \in A$, |
| a binary relation | $\_ \xrightarrow{v,\sigma} \_$ | for each valuation $v$, |
| a unary relation | $v \in [\mathsf{s}_\rho(\_)]$ | for each valuation $v$. |

These relations can be explained as follows:

- $t \xrightarrow{v,a} \surd$: under valuation $v$, $t$ is capable of first performing $a$ in the current time slice and then terminating successfully;
- $t \xrightarrow{v,a} t'$: under valuation $v$, $t$ is capable of first performing $a$ in the current time slice and then proceeding as $t'$;
- $t \xrightarrow{v,\sigma} t'$: under valuation $v$, $t$ is capable of first passing to the next time slice and then proceeding as $t'$;
- $v \in [\mathsf{s}_\rho(t)]$: $v$ makes the root signal of $t$ true.

The rules have the form $\dfrac{p_1, \ldots, p_m}{c_1, \ldots, c_n}\ s$, where $s$ is optional. They are to be read as "if $p_1$ and $\ldots$ and $p_m$ then $c_1$ and $\ldots$ and $c_n$, provided $s$". As usual, $p_1, \ldots, p_m$ and $c_1, \ldots, c_n$ are called the premises and the conclusions, respectively. The conclusions are positive formulas of the form $t \xrightarrow{v,a} \surd$, $t \xrightarrow{v,a} t'$, $t \xrightarrow{v,\sigma} t'$ or $v \in [\mathsf{s}_\rho(t)]$, where $t$ and $t'$ are open terms of $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$. The premises are positive formulas of the above forms or negative formulas of the form $t \xrightarrow{v,\sigma}\!\!\!\!\!\!/\ \ $. A negative formula $t \xrightarrow{v,\sigma}\!\!\!\!\!\!/\ \ $ means that for all closed terms $t'$ of $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$ not $t \xrightarrow{v,\sigma} t'$. The rules are actually rule schemas. The optional $s$ is a side-condition restricting the valuations over which $v$ ranges, the actions over which $a$ ranges, the propositions over which $\phi$ ranges, and the sets of actions over which $H$ ranges. If $m = 0$ and there is no side-condition, the horizontal bar is left out.

The signature of $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$ together with the rules that will be given constitute a term deduction system in *panth* format as defined in [29]. It is known from [29] that if a term deduction system in panth format is *stratifiable*, (strong) bisimulation equivalence is a congruence for the operators in the signature concerned. For a comprehensive introduction to rule formats guaranteeing that bisimulation equivalence is a congruence, the reader is referred to [1].

Let $T$ be a term deduction system and $\mathrm{PF}(T)$ be the set of positive formulas occurring in the rules of $T$. Then a mapping $S : \mathrm{PF}(T) \to \alpha$ for an ordinal $\alpha$ is called a *stratification* for $T$ if for all rules $\dfrac{P}{C}$ of $T$, formulas $c$ in $C$, and closed substitutions $\sigma$ the following conditions hold:

for all positive formulas $p$ in $P$, $\quad S(\sigma(p)) \leq S(\sigma(c))$;
for all negative formulas $t \neg R$ in $P, S(\sigma(tRt')) < S(\sigma(c))$ for all closed terms $t'$;
for all negative formulas $\neg Pt$ in $P, S(\sigma(Pt)) < S(\sigma(c))$.

Recall that the rules that will be given are actually rule schemas. Within the framework of term deduction systems, the instances of the rule schemas that satisfy the stated side-conditions should be taken as the rules under consideration. For the rest, we continue to use the word rule in the broader sense.

A structural operational semantics of $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$ is described by the rules given in Tables 11, 12, 13 and 14. Note that we write $t \xrightarrow{v,\sigma}\!\!\!\!\!\!/\;$ instead of $t \neg \xrightarrow{v,\sigma}$.

$$\underline{a} \xrightarrow{v,a} \surd \qquad \sigma_{\mathsf{rel}}(x) \xrightarrow{v,\sigma} x$$

$$\frac{x \xrightarrow{v,a} \surd}{\phi \mathbin{:\!\!\rightarrow} x \xrightarrow{v,a} \surd}\ v \models \phi \qquad \frac{x \xrightarrow{v,a} x'}{\phi \mathbin{:\!\!\rightarrow} x \xrightarrow{v,a} x'}\ v \models \phi \qquad \frac{x \xrightarrow{v,\sigma} x'}{\phi \mathbin{:\!\!\rightarrow} x \xrightarrow{v,\sigma} x'}\ v \models \phi$$

$$\frac{x \xrightarrow{v,a} \surd}{\phi :\!\!\rightarrow x \xrightarrow{v,a} \surd}\ v \models \phi \qquad \frac{x \xrightarrow{v,a} x'}{\phi :\!\!\rightarrow x \xrightarrow{v,a} x'}\ v \models \phi \qquad \frac{x \xrightarrow{v,\sigma} x'}{\phi :\!\!\rightarrow x \xrightarrow{v,\sigma} \phi :\!\!\rightarrow x'}$$

$$\frac{x \xrightarrow{v,a} \surd}{\phi \frown x \xrightarrow{v,a} \surd}\ v \models \phi \qquad \frac{x \xrightarrow{v,a} x'}{\phi \frown x \xrightarrow{v,a} x'}\ v \models \phi \qquad \frac{x \xrightarrow{v,\sigma} x'}{\phi \frown x \xrightarrow{v,\sigma} x'}\ v \models \phi$$

$$\frac{x \xrightarrow{v,a} \surd,\ w \in [\mathsf{s}_\rho(y)]}{x \cdot y \xrightarrow{v,a} y} \qquad \frac{x \xrightarrow{v,a} x'}{x \cdot y \xrightarrow{v,a} x' \cdot y} \qquad \frac{x \xrightarrow{v,\sigma} x'}{x \cdot y \xrightarrow{v,\sigma} x' \cdot y}$$

$$\frac{x \xrightarrow{v,a} \surd,\ v \in [\mathsf{s}_\rho(y)]}{x + y \xrightarrow{v,a} \surd,\ y + x \xrightarrow{v,a} \surd} \qquad \frac{x \xrightarrow{v,a} x',\ v \in [\mathsf{s}_\rho(y)]}{x + y \xrightarrow{v,a} x',\ y + x \xrightarrow{v,a} x'}$$

$$\frac{x \xrightarrow{v,\sigma} x',\ y \xrightarrow{v,\sigma} y'}{x + y \xrightarrow{v,\sigma} x' + y'} \qquad \frac{x \xrightarrow{v,\sigma} x',\ y \xrightarrow{v,\sigma}\!\!\!\!\!\!/\;,\ v \in [\mathsf{s}_\rho(y)]}{x + y \xrightarrow{v,\sigma} x',\ y + x \xrightarrow{v,\sigma} x'}$$

Table 11: Rules for basic operators of $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$ $(a \in A)$

$$\frac{x \xrightarrow{v,a} \surd,\ v \in [\mathsf{s}_\rho(y)]}{x \parallel y \xrightarrow{v,a} y,\ y \parallel x \xrightarrow{v,a} y,\ x \between y \xrightarrow{v,a} y}$$

$$\frac{x \xrightarrow{v,a} x',\ v \in [\mathsf{s}_\rho(y)],\ w \in [\mathsf{s}_\rho(x')],\ w \in [\mathsf{s}_\rho(y)]}{x \parallel y \xrightarrow{v,a} x' \parallel y,\ y \parallel x \xrightarrow{v,a} y \parallel x',\ x \between y \xrightarrow{v,a} x' \parallel y}$$

$$\frac{x \xrightarrow{v,\sigma} x',\ y \xrightarrow{v,\sigma} y'}{x \parallel y \xrightarrow{v,\sigma} x' \parallel y',\ x \between y \xrightarrow{v,\sigma} x' \between y'}$$

Table 12: Rules for parallel composition $(a \in A)$

All rules are in panth format. In order to prove the fact that strong bisimulation is a congruence, we only have to find a stratification. We define a stratification $S$ as follows:

$$\frac{}{v \in [\mathsf{s}_\rho(\underline{a})]} \qquad \frac{}{v \in [\mathsf{s}_\rho(\underline{\delta})]} \qquad \frac{}{v \in [\mathsf{s}_\rho(\sigma_{\mathsf{rel}}(x))]}$$

$$\frac{v \in [\mathsf{s}_\rho(x)]}{v \in [\mathsf{s}_\rho(\phi :\!\!\rightarrow x)]} \qquad \frac{}{v \in [\mathsf{s}_\rho(\phi :\!\!\rightarrow x)]}\ v \not\models \phi$$

$$\frac{v \in [\mathsf{s}_\rho(x)]}{v \in [\mathsf{s}_\rho(\phi :\!\!\rightarrow x)]} \qquad \frac{v \in [\mathsf{s}_\rho(x)]}{v \in [\mathsf{s}_\rho(\phi \mathbin{\overset{\frown}{\rightarrow}} x)]}\ v \models \phi$$

$$\frac{v \in [\mathsf{s}_\rho(x)]}{v \in [\mathsf{s}_\rho(x \cdot y)]} \qquad \frac{v \in [\mathsf{s}_\rho(x)],\ v \in [\mathsf{s}_\rho(y)]}{v \in [\mathsf{s}_\rho(x + y)],\ v \in [\mathsf{s}_\rho(x \parallel y)],\ v \in [\mathsf{s}_\rho(x \between y)]}$$

Table 13: Rules for $v \in [\mathsf{s}_\rho(\_)]$

$$\frac{x \xrightarrow{v,a} \surd}{\nu_{\mathsf{rel}}(x) \xrightarrow{v,a} \surd} \qquad \frac{x \xrightarrow{v,a} x'}{\nu_{\mathsf{rel}}(x) \xrightarrow{v,a} x'} \qquad \frac{v \in [\mathsf{s}_\rho(x)]}{v \in [\mathsf{s}_\rho(\nu_{\mathsf{rel}}(x))]}$$

$$\frac{x \xrightarrow{v,a} \surd}{\partial_H(x) \xrightarrow{v,a} \surd}\ a \notin H \qquad \frac{x \xrightarrow{v,a} x'}{\partial_H(x) \xrightarrow{v,a} \partial_H(x')}\ a \notin H$$

$$\frac{x \xrightarrow{v,\sigma} x'}{\partial_H(x) \xrightarrow{v,\sigma} \partial_H(x')} \qquad \frac{v \in [\mathsf{s}_\rho(x)]}{v \in [\mathsf{s}_\rho(\partial_H(x))]}$$

Table 14: Rules for now operator and encapsulation ($a \in A$)

$$S(t \xrightarrow{v,\sigma} t') = n_+(t) \text{ and } S(t \xrightarrow{v,a} \surd) = S(t \xrightarrow{v,a} t') = S(v \in [\mathsf{s}_\rho(t)]) = 0,$$

where $n_+(t)$ stands for the number of occurrences of $+$ in $t$. So $S(F)$ is the number of occurrences of $+$ in the terms $t$ occurring as the left-hand side of the formulas in $F$ that have the form $t \xrightarrow{v,\sigma} t'$. It is straightforward to prove that the mapping $S$ is a stratification. We have to check all rules. This is trivial except for the only rule with a negative formula in its premises, viz. the last rule of Table 11. The number of occurrences of $+$ in the conclusion of that rule is strictly greater than the number of occurrences of $+$ in the negative formula in the premises.

Note that the two rules for alternative composition concerning passage to the next time slice (Table 11) have complementary conditions. Together they enforce that the choice between two processes that both can pass to the next time slice is postponed till after the passage to the next time slice. This corresponds to the property reflected by the axiom DRT1 of $\mathrm{BPA}^-_{\mathrm{drt}}$-ID (Table 1).

In order to rule out processes that are capable of performing an action and then going on as $\bot$, there are premises in the first rule for sequential composition (Table 11) and the second rule for parallel composition (Table 12) concerning the existence of valuations that makes the root signal of certain processes true. This corresponds to the property reflected by the axiom NE3 of $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$ (Table 5).

The rule for time unit delay (Table 11), shows that $\sigma_{\mathsf{rel}}(t)$ is capable of passing to the next time slice under all valuations instead of only the valuations under which $t$ is capable of doing things. This excludes persistency of root signal emission and waiting of the strict guarded command because all rules for these operators restrict the valuations under which the resulting processes are capable of doing things.

## 2.4 Recursion

In this subsection, we add recursion to $\mathrm{PA}_{\mathrm{drt}}^{\mathrm{psc}}$. Recursive specification, solution, guardedness, etc. are defined in a similar way as for BPA in [8].

Let $V$ be a set of variables (of sort $\mathsf{P}$). A *recursive specification* $E = E(V)$ in $\mathrm{PA}_{\mathrm{drt}}^{\mathrm{psc}}$ is a set of equations $E = \{X = s_X \mid X \in V\}$ where each $s_X$ is a $\mathrm{PA}_{\mathrm{drt}}^{\mathrm{psc}}$ term that only contains variables from $V$. We shall use $X, X', Y, Y', ...$ for variables bound in a recursive specification. A *solution* of a recursive specification $E(V)$ is a set of processes $\{\langle X|E\rangle \mid X \in V\}$ in some model of $\mathrm{PA}_{\mathrm{drt}}^{\mathrm{psc}}$ such that the equations of $E(V)$ hold if, for all $X \in V$, $X$ stands for $\langle X|E\rangle$. Mostly, we are interested in one particular variable $X \in V$.

We can now introduce the equational theory of $\mathrm{PA}_{\mathrm{drt}}^{\mathrm{psc}}\mathrm{rec}$.

**Signature of $\mathrm{PA}_{\mathrm{drt}}^{\mathrm{psc}}$rec** The signature of $\mathrm{PA}_{\mathrm{drt}}^{\mathrm{psc}}\mathrm{rec}$ consists of the signature of $\mathrm{PA}_{\mathrm{drt}}^{\mathrm{psc}}$ extended with a constant $\langle X|E\rangle : \to \mathsf{P}$ for each $X \in V$ and each recursive specification $E(V)$.

Let $t$ be an open term in $\mathrm{PA}_{\mathrm{drt}}^{\mathrm{psc}}$ and $E = E(V)$ be a recursive specification. Then we write $\langle t|E\rangle$ for $t$ with, for all $X \in V$, all occurrences of $X$ in $t$ replaced by $\langle X|E\rangle$.

**Axioms of $\mathrm{PA}_{\mathrm{drt}}^{\mathrm{psc}}$rec** The axiom system of $\mathrm{PA}_{\mathrm{drt}}^{\mathrm{psc}}\mathrm{rec}$ consists of the axioms of $\mathrm{PA}_{\mathrm{drt}}^{\mathrm{psc}}$ and an equation $\langle X|E\rangle = \langle s_X|E\rangle$ for each $X \in V$ and each recursive specification $E(V)$.

Let $t$ be a term of $\mathrm{PA}_{\mathrm{drt}}^{\mathrm{psc}}$ containing a variable $X$. We call an occurrence of $X$ in $t$ *guarded* if $t$ has a subterm of the form $\underline{\underline{a}} \cdot t'$ or $\sigma_{\mathsf{rel}}(t')$ with $t'$ a $\mathrm{PA}_{\mathrm{drt}}^{\mathrm{psc}}$ term containing this occurrence of $X$. We call a recursive specification *guarded* if all occurrences of all its variables in the right-hand sides of all its equations are guarded or it can be rewritten to such a recursive specification using the axioms of $\mathrm{PA}_{\mathrm{drt}}^{\mathrm{psc}}$ and its equations. An interesting form of guarded recursive specification is linear recursive specification. We call a recursive specification $E(V)$ *linear* if each equation in $E$ has the form

$$X = \sum_{i<n} \phi_i :\to \underline{\underline{a_i}} \cdot X_i + \sum_{i<m} \psi_i :\to \underline{\underline{b_i}} + \sum_{i<k} \chi_i :\to \sigma_{\mathsf{rel}}(X_i') + \xi \frown \underline{\underline{\delta}}$$

for certain actions $a_i$ and $b_i$, propositions $\phi_i$, $\psi_i$, $\chi_i$ and $\xi$, and variables $X, X_i, X_i' \in V$. Note that, without loss of generality we can assume that for all $i$ and $j$ such that $i \neq j$: $\underline{\underline{a_i}} \cdot X_i \not\equiv \underline{\underline{a_j}} \cdot X_j$, $\underline{\underline{b_i}} \not\equiv \underline{\underline{b_j}}$, $X_i' \not\equiv X_j'$ and $\chi_i \not\equiv \chi_j$. We can also assume that $\phi_i$, $\psi_i$ and $\chi_i$ are not $\underline{\underline{\mathsf{f}}}$.

**Principles of $\mathrm{PA}_{\mathrm{drt}}^{\mathrm{psc}}$rec** The *(restricted) recursive definition principle* ($\mathrm{RDP}^{(-)}$) is the assumption that every (guarded) recursive specification has a solution. The *recursive specification principle* (RSP) is the assumption that every guarded recursive specification has at most one solution.

Note that the axioms $\langle X|E\rangle = \langle s_X|E\rangle$ for a fixed $E$ express that the constants $\langle X|E\rangle$ make up a solution of $E$, i.e. RDP holds for any model of $\mathrm{PA}_{\mathrm{drt}}^{\mathrm{psc}}\mathrm{rec}$. The conditional equations $E \Rightarrow X = \langle X|E\rangle$ express that this solution is the only one. So RSP can be described by means of conditional equations – as already mentioned in [28].

$$\frac{\langle s_X|E\rangle \xrightarrow{v,a} \surd}{\langle X|E\rangle \xrightarrow{v,a} \surd} \qquad \frac{\langle s_X|E\rangle \xrightarrow{v,a} y}{\langle X|E\rangle \xrightarrow{v,a} y}$$

$$\frac{\langle s_X|E\rangle \xrightarrow{v,\sigma} y}{\langle X|E\rangle \xrightarrow{v,\sigma} y} \qquad \frac{v \in [\mathsf{s}_\rho(\langle s_X|E\rangle)]}{v \in [\mathsf{s}_\rho(\langle X|E\rangle)]}$$

Table 15: Rules for recursion ($a \in A$)

**Semantics of $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$rec** A structural operational semantics of $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$rec is described by the rules for $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$ and the rules given in Table 15.

The rules added for recursion are also in panth format. We define a stratification $S$ as follows:

$$S(t \xrightarrow{v,\sigma} t') = \omega \cdot n_{\mathrm{sol}}(t) + n_+(t) \text{ and } S(t \xrightarrow{v,a} \surd) = S(t \xrightarrow{v,a} t') = S(v \in [\mathsf{s}_\rho(t)]) = 0,$$

where $n_{\mathrm{sol}}(t)$ stands for the number of unguarded occurrences of constants $\langle X|E\rangle$ in $t$ and $n_+(t)$ stands for the number of occurrences of $+$ in $t$. The addition of the summand $\omega \cdot n_{\mathrm{sol}}(t)$ for formulas $t \xrightarrow{v,\sigma} t'$ solves the problem that the conclusion of the rule for recursion concerning passage to the next time slice does not contain occurrences of $+$.

Let $E = \{X = s_X \mid X \in V\}$ be a recursive specification. Then roughly, the rules for recursion come down to looking upon $\langle X|E\rangle$ as the process $s_X$ with, for all $X' \in V$, all occurrences of $X'$ in $s_X$ replaced by $\langle X'|E\rangle$.

## 2.5 Counting process creation

In this subsection, we introduce the counting process creation operator $\mathrm{E}^n_\Phi$ that is used for the semantics of $\varphi^-$SDL. This operator subsumes the original process creation operator introduced in [11]. The latter process creation operator was used in [14] for the semantics of $\varphi$SDL. But the approach used there does not guarantee that a unique process identification is assigned to each created process.

It is assumed that a fixed but arbitrary set $D$ of *data* has been given together with a function $\Phi : \mathbb{N} \times D \to \mathsf{P}$, and that there exist actions $cr(d)$ and $\overline{cr}(n,d)$ for all $d \in D$ and $n \in \mathbb{N}$. The process creation operator $\mathrm{E}^n_\Phi$ allows, given the function $\Phi$, the use of actions $cr(d)$ to create processes $\Phi(n,d)$.

The counting process creation operator $\mathrm{E}^n_\Phi : \mathsf{P} \to \mathsf{P}$ is defined by the equations PRCR1-PRCR8 given in Table 16. The crucial axiom is PRCR4. It expresses that counting process creation applied to a process, with the counter set to $n$, leaves the action $\underline{\overline{cr}}(n,d)$ as a trace and starts a process $\Phi(n,d)$ in parallel with the remaining process when it comes across an undelayable process creation action $\underline{\underline{cr}}(d)$. Besides, it increases the counter by one. The counting process creation operator $\mathrm{E}^n_\Phi$ is an extension of the process creation operator $\mathrm{E}_\phi$ from [11]. We can write $\mathrm{E}^n_\Phi = \mathrm{E}_\phi$ if $\Phi(n,d) = \phi(d)$ for all $n \in \mathbb{N}$ and $d \in D$.

A structural operational semantics for the counting process creation operator is described by the rules given in Table 17. The stratification introduced for $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$rec still works if we add the rules for the counting process creation operator.

| | |
|---|---|
| $\mathrm{E}^n_\Phi(\underline{a}) = \underline{a}$ if $a \neq cr(d)$ for $d \in D$ | PRCR1 |
| $\mathrm{E}^n_\Phi(\underline{\underline{cr}}(d)) = \underline{\underline{\overline{cr}}}(n,d) \cdot \mathrm{E}^{n+1}_\Phi(\Phi(n,d))$ | PRCR2 |
| $\mathrm{E}^n_\Phi(\underline{a} \cdot x) = \underline{a} \cdot \mathrm{E}^n_\Phi(x)$ if $a \neq cr(d)$ for $d \in D$ | PRCR3 |
| $\mathrm{E}^n_\Phi(\underline{\underline{cr}}(d) \cdot x) = \underline{\underline{\overline{cr}}}(n,d) \cdot \mathrm{E}^{n+1}_\Phi(\Phi(n,d) \parallel x)$ | PRCR4 |
| $\mathrm{E}^n_\Phi(x + y) = \mathrm{E}^n_\Phi(x) + \mathrm{E}^n_\Phi(y)$ | PRCR5 |
| $\mathrm{E}^n_\Phi(\sigma_{\mathsf{rel}}(x)) = \sigma_{\mathsf{rel}}(\mathrm{E}^n_\Phi(x))$ | PRCR6 |
| $\mathrm{E}^n_\Phi(\phi \mathbin{:\!\!\rightarrow} x) = \phi \mathbin{:\!\!\rightarrow} \mathrm{E}^n_\Phi(x)$ | PRCR7 |
| $\mathrm{E}^n_\Phi(\phi \mathbin{\frown\!\!\blacktriangleright} x) = \phi \mathbin{\frown\!\!\blacktriangleright} \mathrm{E}^n_\Phi(x)$ | PRCR8 |

Table 16: Axioms for counting process creation ($a \in A_\delta$)

$$\frac{x \xrightarrow{v,a} \surd}{\mathrm{E}^n_\Phi(x) \xrightarrow{v,a} \surd}\ a \neq cr(d) \qquad \frac{x \xrightarrow{v,cr(d)} \surd,\ w \in [\mathsf{s}_\rho(\Phi(n,d))]}{\mathrm{E}^n_\Phi(x) \xrightarrow{v,\overline{cr}(n,d)} \mathrm{E}^{n+1}_\Phi(\Phi(n,d))}$$

$$\frac{x \xrightarrow{v,a} x'}{\mathrm{E}^n_\Phi(x) \xrightarrow{v,a} \mathrm{E}^n_\Phi(x')}\ a \neq cr(d) \qquad \frac{x \xrightarrow{v,cr(d)} x',\ w \in [\mathsf{s}_\rho(\Phi(n,d))],\ w \in [\mathsf{s}_\rho(x')]}{\mathrm{E}^n_\Phi(x) \xrightarrow{v,\overline{cr}(n,d)} \mathrm{E}^{n+1}_\Phi(\Phi(n,d) \parallel x')}$$

$$\frac{x \xrightarrow{v,\sigma} x'}{\mathrm{E}^n_\Phi(x) \xrightarrow{v,\sigma} \mathrm{E}^n_\Phi(x')} \qquad \frac{v \in [\mathsf{s}_\rho(x)]}{v \in [\mathsf{s}_\rho(\mathrm{E}^n_\Phi(x))]}$$

Table 17: Rules for counting process creation ($a \in A$)

## 2.6 State operator

In this subsection, we introduce a state operator for $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$. It generalizes and extends the state operator for $\mathrm{ACP}_{\mathrm{ps}}$ proposed in [6]: the truth value of propositional signals and conditions may depend upon the state and passage to the next time slice may have an effect on the state. The possibility of non-deterministic behaviour as the result of applying the operator is included as well, like for the extended state operator $\Lambda$ (see e.g. [8]).

It is assumed that a fixed but arbitrary set $S$ of states has been given, together with functions:

$$\begin{aligned}
&\mathit{act} : A \times S \to \mathcal{P}_{\mathrm{fin}}(A)\\
&\mathit{eff} : A \times S \times A \to S\\
&\mathit{eff}_\sigma\colon S \to \mathcal{P}_{\mathrm{fin}}(S) \setminus \{\emptyset\}\\
&\mathit{sig} : S \to \mathsf{B}\\
&\mathit{val} : B_{\mathrm{at}} \times S \to \mathbb{B}
\end{aligned}$$

The state operator $\lambda_s$ ($s \in S$) allows, given these functions, processes to interact with a state. The process $\lambda_s(x)$ is the process $x$ executed in a state $s$. The function $\mathit{act}$ gives, for each action $a$ and state $s$, the set of actions that may be performed if $a$ is executed in state $s$. The function $\mathit{eff}$ gives, for each action $a$, state $s$ and action $a'$, the state that results when $a$ is executed in state $s$ and $a'$ is the action that is actually performed as the result of the execution. The function $\mathit{eff}_\sigma$ gives, for each state $s$, the set of states that may result when time passes to the next time slice in state $s$. The function $\mathit{sig}$ gives, for each state $s$, the propositional signal that holds at the start of any process executed in state $s$. The function $\mathit{val}$ gives, for each state $s$, the valuation $\mathit{val}(\_, s)$ of the atomic propositions in state $s$. The valuation $\mathit{val}(\_, s)$ can be extended to all propositions in the usual homomorphic way as any other valuation. We will use

the notation $val(\_, s)$ to refer to the extension as well.

The state operator $\lambda_s : \mathsf{P} \to \mathsf{P}$ is defined by the equations SO1-SO6 given in Table 18. The axioms SO1-SO3 are straightforward reformulations – in the same

| | |
|---|---|
| $\lambda_s(\underline{\underline{a}}) = sig(s) \frown \sum_{a' \in act(a,s)} \underline{\underline{a'}}$ | SO1 |
| $\lambda_s(\underline{\underline{a}} \cdot x) = sig(s) \frown \sum_{a' \in act(a,s)} (\underline{\underline{a'}} \cdot \lambda_{\mathit{eff}(a,s,a')}(x))$ | SO2 |
| $\lambda_s(x + y) = \lambda_s(x) + \lambda_s(y)$ | SO3 |
| $\lambda_s(\sigma_{\mathsf{rel}}(x)) = sig(s) \frown \sigma_{\mathsf{rel}}(\sum_{s' \in \mathit{eff}_\sigma(s)} \lambda_{s'}(x))$ | SO4 |
| $\lambda_s(\phi \mathrel{:}\!\!\rightarrow x) = sig(s) \frown (val(\phi, s) \mathrel{:}\!\!\rightarrow \lambda_s(x))$ | SO5 |
| $\lambda_s(\phi \frown x) = val(\phi, s) \frown \lambda_s(x)$ | SO6 |

Table 18: Axioms for state operator ($a \in A$)

way as for $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$ – of corresponding axioms given in [6] for the time free case. The additional axiom SO4 expresses how passage to the next time slice has influence on the execution of a process in a state. The axioms SO5 and SO6 are also reformulations of corresponding axioms given in [6]. In these axioms the proposition $\phi$ has been replaced by $val(\phi, s)$. Thus the case is covered where the truth value of propositional signals and conditions may depend upon the state. Note that from SO5 we can derive $\lambda_s(\underline{\delta}) = \underline{\delta}$.

A structural operational semantics for the state operator is described by the rules given in Table 19. Note that the rules added for the state operator have a common side-condition given at the bottom of the table. The stratification introduced for

$$\frac{x \xrightarrow{v,a} \surd}{\lambda_s(x) \xrightarrow{v',a'} \surd}\ a' \in act(a,s)$$

$$\frac{x \xrightarrow{v,a} x',\ w \in [\mathsf{s}_\rho(x')]}{\lambda_s(x) \xrightarrow{v',a'} \lambda_{\mathit{eff}(a,s,a')}(x')}\ a' \in act(a,s) \wedge w \models sig(\mathit{eff}(a,s,a'))$$

$$\frac{x \xrightarrow{v,\sigma} x'}{\lambda_s(x) \xrightarrow{v',\sigma} \lambda_{s'}(x')}\ s' \in \mathit{eff}_\sigma(s) \quad \frac{v \in [\mathsf{s}_\rho(x)]}{v' \in [\mathsf{s}_\rho(\lambda_s(x))]}$$

for all $v, v'$ and $s$ such that $v' \models sig(s) \wedge \forall \phi \bullet v \models \phi \leftrightarrow v' \models val(\phi, s)$

Table 19: Rules for state operator ($a \in A$)

$\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$rec still works if we add the rules for the state operator.

# 3 Overview of $\varphi^-$SDL

## 3.1 Introduction

In this section, we give an overview of $\varphi^-$SDL, i.e. $\varphi$SDL without delaying channels. $\varphi$SDL is a small subset of SDL, introduced in [14], which covers all behavioural aspects of SDL, including communication via delaying channels, timing and process creation. Leaving out delaying channels simplifies the presentation. Besides work on the process algebra semantics of $\varphi$SDL made clear that $\varphi$SDL specifications can always be transformed to semantically equivalent ones in $\varphi^-$SDL. At the end of Section 4 is shown how to model a delaying channel by means of a $\varphi^-$SDL process.

In Sections 3.2, 3.3 and 3.4, the syntax of $\varphi^-$SDL 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 rules is explained informally. Some peculiar details of the semantics, inherited from full SDL, are left out to improve the comprehensibility of the overview. These details will, however, be taken into account in Section 4, where the process algebra semantics of $\varphi^-$SDL is presented. In Section 3.5, some remarks are made about the context-sensitive conditions for syntactic correctness of $\varphi^-$SDL specifications. The syntactic differences with full SDL are summarized in this section as well. In Section 3.6 some examples of $\varphi^-$SDL specifications are given.

In line with full SDL, we can define a graphical representation for $\varphi^-$SDL specifications. We pay no attention to this practically important point because it is not relevant to the subject of this chapter.

## 3.2 System definition

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

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

A signal can be sent from the environment to a process, from a process to the environment or from a process to a process. A signal may carry values to be passed from the sender to the receiver; on consumption of the signal, these values are assigned to local variables of the receiver. A signal route is a unidirectional communication path for sending signals 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 it will be instantaneously delivered to that process.

Signals of $\varphi^-$SDL are not at all related to propositional signals of $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$. In Section 4, communication of $\varphi^-$SDL signals will be modeled by actions of $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$. Propositional signals of $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$ will be used to represent the visible part of the state of a system specified in $\varphi^-$SDL. Recall that, in the remainder of this chapter, we will call propositional signals of $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$ "propositions" in order to avoid ambiguity with signals of $\varphi^-$SDL.

**Syntax:**

<system definition> ::=
  **system** <system nm> **;** {<definition>}$^+$ **endsystem** **;**

<definition> ::=
  **dcl** <variable nm> <sort nm> **;**

| **signal** <signal nm> [ **(** <sort nm> {**,** <sort nm>}* **)** ] **;**
| **signalroute** <signalroute nm>
  **from** {<process nm> | **env**} **to** {<process nm> | **env**}
  **with** <signal nm> {**,** <signal 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 and the signal routes via which the signals are conveyed.

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 $T_1, \ldots, T_n$. If **(**$T_1$**,** $\ldots$**,**$T_n$**)** is absent, the signals of type $s$ do not carry any value.

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 delay signals sent by processes of type $X_1$ to processes of type $X_2$, for signals of the types $s_1, \ldots, s_n$. 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 process definition **process** $X$**(**$k$**);** **fpar** $v_1$**,** $\ldots$**,**$v_m$**;** **start;** $tr$ $d_1$ $\ldots$ $d_n$ **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 $v_1, \ldots, v_m$. If **fpar** $v_1$**,** $\ldots$**,**$v_m$**;** is absent, no values are passed on creation. The process body **start;** $tr$ $d_1$ $\ldots$ $d_n$ describes the behaviour of the processes of type $X$ in terms of states and transitions (see further Section 3.3). Each process will start by making the transition $tr$, called its start transition, to enter one of its states. The state definitions $d_1, \ldots, d_n$ define all the states in which the process may come while it proceeds.

## 3.3 Process behaviour

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

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

A transition from a state of a process is initiated by the consumption of a signal, unless it is a spontaneous transition. The start transition is not initiated by the

consumption of a signal either. A transition is made by performing certain actions: signals may be sent, variables may be assigned new values, new processes may be created and *timers* may be set and reset. A transition may at some stage also take one of a number of branches, but it will eventually come to an end and bring the process to a next state or to its termination.

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

**Syntax:**

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

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

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

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

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

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

A state definition **state** $st$**; save** $s_1$**,** $\ldots$**,**$s_m$**;**$alt_1$ $\ldots$ $alt_n$ defines a state $st$. The signals of the types $s_1, \ldots, s_m$ are saved for the state. The input guard of each of the transition alternatives $alt_1, \ldots, alt_n$ 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 alternatives 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 $v_1, \ldots, v_n$. 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 $a_1, \ldots, a_n$ 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 $e_1, \ldots, e_n$ to the process with the current (pid) value of the expression $e$ as its address, via one of the signal routes $r_1, \ldots, r_m$. 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 $r_1, \ldots, r_m$ 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 $e_1, \ldots, e_n$.

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 $e_1, \ldots, e_n$.

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 $e_1, \ldots, e_n$ 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 of such a branch results in an error. A non-deterministic choice can be obtained by replacing the expression $e$ by the keyword **any** and removing all the expressions $e_i$.

## 3.4 Values

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

**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 $e_1, \ldots, e_n$.

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*$_1$,...,*e*$_n$)) evaluates to the Boolean value true if the timer identified with the signal type $s$ and the current values of the expressions $e_1, \ldots, e_n$ 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. Natural numbers are used as pid values. The pid value 0 is a special pid value that never refers to any existing process – in full SDL this pid value is denoted by **null** – and the pid value 1 is reserved for the environment. The expressions **parent**, **offspring** and **sender** evaluate to 0 in case there exists no parent, offspring and sender, respectively.

## 3.5 Miscellaneous issues

### 3.5.1 Context-sensitive syntactic rules

We remain loose about the context-sensitive conditions for syntactic correctness of $\varphi^-$SDL specifications. For the most part, they are as usual: only defined names may be used, use of names must agree with their definitions, types of expressions must be in accordance with the type information in the definitions, etc. There is one condition that needs particular attention: signals of the same type may not be used for both signal sending and timer setting/resetting. All $\varphi^-$SDL specifications that are obtained by semantics preserving transformations of syntactically correct specifications in full SDL will be syntactically correct $\varphi^-$SDL specifications as well.

### 3.5.2 Syntactic differences with SDL

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

- variable definitions occur at the system level instead of inside process definitions;
- signal route definitions and process definitions occur at the system level instead of inside block definitions;
- 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.

## 3.6 Examples

We give three small examples to illustrate how systems are specified in $\varphi^-$SDL. The examples concern simple repeaters and routers.

### 3.6.1 Repeater

The first example concerns a simple repeater, i.e. a system that simply passes on what it receives. The system, called `Repeater`, consists of only one process, viz. `rep`, which communicates signals `s` with the environment via the signal routes `fromenv` and `toenv`. The process has only one state.

```
system Repeater
  signal s;

  signalroute fromenv from env to rep with s;
  signalroute toenv from rep to env with s;

  process rep(1);
    start;
      nextstate pass;
    state pass;
      input s;
        output s via toenv;
        nextstate pass;
  endprocess;
endsystem;
```

### 3.6.2 Address driven router

The second example concerns address driven routing of data. The system, called `AddrRouter`, consists of three processes, one instance of `rtr` and two instances of `rep`. The latter two processes are created by the former process. The process `rtr` consumes signals `s(a)`, delivered via signal route `fromenv`, and passes them to one of the instances of `rep` (via signal route `rs`) depending on the value `a`. The instances of `rep` then pass the signals received from `rtr` to the environment via the signal route `toenv`.

```
system AddrRouter
  signal s(Bool);

  signalroute fromenv from env to rtr with s;
  signalroute rs from rtr to rep with s;
  signalroute toenv from rep to env with s;

  dcl a Bool; dcl rep1 Nat; dcl rep2 Nat;

  process rtr(1);
    start;
      create rep; task rep1 := offspring;
      create rep; task rep2 := offspring;
      nextstate route;
    state route;
      input s(a);
        decision a;
          (False):
            output s(a) to rep1 via rs;
            nextstate route;
          (True):
            output s(a) to rep2 via rs;
```

```
          nextstate route;
        enddecision;
  endprocess;

  process rep(0);
    start;
        nextstate pass;
    state pass;
      input s(a);
        output s(a) via toenv;
        nextstate pass;
  endprocess;
endsystem;
```

### 3.6.3 Load driven router

The last example concerns load driven routing of data. The system, called `LoadRouter`, consists of three processes, one instance of `rtr` and two instances of `trep`. The latter two processes are created by the former process. The process `rtr` consumes signals `s`, delivered via signal route `fromenv`, and passes them to one of the instances of `trep` (via signal route `rs`) depending on their load. The instances of `trep` then pass the signals received from `rtr` to the environment via the signal routes `toenv`. Either delivers the data consumed after a fixed time `delay`. One repeater delivers twice as fast as the other one. Only two load factors are considered for each of the two repeaters: one indicating its idleness and one indicating the opposite.

```
system LoadRouter
  signal s; signal t;

  signalroute fromenv from env to rtr with s;
  signalroute rs from rtr to trep with s;
  signalroute toenv from trep to env with s;

  dcl idle Bool; dcl delay Nat;
  dcl rep1 Nat; dcl rep2 Nat;

  process rtr(1);
    start;
      create trep(10); task rep1 := offspring;
      create trep(20); task rep2 := offspring;
      nextstate route;
    state route;
      input s;
        decision view(idle,rep1) <-> view(idle,rep2);
          (True):
            output s via rs; nextstate route;
          (False):
            decision view(idle,rep1)
              (True):
                output s to rep1 via rs;
                nextstate route;
              (False):
                output s to rep2 via rs;
                nextstate route;
            enddecision
```

```
        enddecision
    endprocess;

    process trep(0) fpar delay;
      start;
        task idle := True;
        nextstate get;
      state get;
        input s;
          task idle := False;
          set(now + delay, t);
          nextstate put;
      state put;
        save s;
        input t;
          output s via toenv;
          task idle := True;
          nextstate get;
     endprocess;
  endsystem;
```

Most features of $\varphi^{-}$SDL are used in this example. Of the main features, only spontaneous transitions, i.e. transition alternatives with **input none;** instead of an input guard, are missing. In Section 4.6.3, use is made of this feature to define a process modeling a delaying channel.

# 4 Semantics of $\varphi^{-}$SDL

## 4.1 Introduction

In this section, we propose a process algebra semantics for $\varphi^{-}$SDL. This semantics is presented in two steps.

In Section 4.3, a semantics for $\varphi^{-}$SDL process definitions is given. This semantics abstracts from dynamic aspects of process behaviour such as process creation and process execution in a state. It describes the meaning of $\varphi^{-}$SDL process definitions by means of finite guarded recursive specifications in $\mathrm{BPA}_{\mathrm{drt}}^{\mathrm{psc}}$. The counting process creation operator and the state operator are not needed for this semantics. Preceeding, in Section 4.2, the actions and atomic conditions used are introduced. These actions and conditions are parametrized by expressions with values that depend on the state in which the action or condition concerned is executed.

In Section 4.6, a semantics of $\varphi^{-}$SDL system definitions is given in terms of the semantics for $\varphi^{-}$SDL process definitions using the counting process creation operator and the state operator. Preceeding, in Section 4.5, all the details of the instance of the state operator needed for this semantics are given. Included are the definitions of the state space and the functions that describe how the actions and conditions used for the semantics of $\varphi^{-}$SDL process definitions interact with the state when this instance of the state operator is applied. The interaction with the environment is another aspect covered by the semantics of $\varphi^{-}$SDL system definitions. For this purpose an environment process is introduced as well.

The semantics of $\varphi^{-}$SDL is described by means of a set of equations recursively defining interpretation functions for all syntactic categories. The special notation

used is explained in Appendix A. We will be lazy about specifying the range of each interpretation function, since this is usually clear from the context. Many of the interpretations are functions from natural numbers to process expressions or equations. These process expressions and equations will simply be written in their display form. 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. The semantics is defined using contextual information $\kappa$ extracted from the $\varphi^-$SDL specification concerned. This is further described in Appendix B.

The special notation used for parametrized actions, conditions and propositions is explained in Appendix A, and so is the uncustomary notation as regards sets, functions and sequences. The words action and process are used in connection with both ACP-style process algebras and versions of SDL, but with slightly different meanings. In case it is not clear from the context, these words will be preceded by ACP if they should be taken in the ACP sense, and by SDL otherwise.

### Data types

We mentioned before that $\varphi^-$SDL does not deal with the specification of abstract data types. We assume a fixed algebraic specification covering all data types used and an initial algebra semantics, denoted by $\mathcal{A}$, for it. The data types **Boolean** and **Natural**, with the obvious interpretation, must be included. 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 additionally assume a constant name, i.e. a nullary operation name, in $Op_{\mathcal{A}}$, referred to as $\underline{n}$, for each $n \in \mathbb{N}$. We will write $U$ for $\bigcup_{T \in Sort_{\mathcal{A}}} T^{\mathcal{A}}$, where $T^{\mathcal{A}}$ is the interpretation of the sort name $T$ in $\mathcal{A}$. We have that $\mathbb{B}, \mathbb{N} \subset U$ because of our earlier requirement that **Boolean**, **Natural** $\in Sort_{\mathcal{A}}$. We assume that $\mathsf{nil} \notin U$. In the sequel, we will use for each $op \in Op_{\mathcal{A}}$ an extension to $U \cup \{\mathsf{nil}\}$, also denoted by $op$, such that $op(t_1, \ldots, t_n) = \mathsf{nil}$ if not all of the $t_i$s are 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 the language constructs of $\varphi^-$SDL. We can do so without loss of generality, because it can (and should) be statically checked that only terms of appropriate sorts occur.

## 4.2 Actions and conditions

In the semantics of $\varphi^-$SDL process definitions, which will be presented in Section 4.3, ACP actions and conditions are used. Here, we introduce the actions and atomic conditions concerned.

The actions and atomic conditions, used to describe the meaning of $\varphi^-$SDL process definitions, are parametrized by various domains. These domains depend upon the specific variable names, signal names and process names introduced in the system definition concerned. These sets of names largely make up the context ascribed to the system definition by means of the function $\{\!|\_|\!\}$ defined in Appendix B. For convenience, we define these sets for arbitrary contexts $\kappa$ (the notation concerning contexts introduced in Appendix B is used):

$$\begin{aligned} V_\kappa &= vars(\kappa) \cup \{\textbf{parent}, \textbf{offspring}, \textbf{sender}\} \\ S_\kappa &= sigs(\kappa) \\ P_\kappa &= procs(\kappa) \end{aligned}$$

Most arguments of the parametrized actions and conditions introduced here are expressions that originate in $\varphi^-$SDL expressions or objects that are somehow composed of such expressions and variable, signal and process names. The reason of this is that the value of the original $\varphi^-$SDL expressions may vary according to the last values assigned to the variables referred to, the status of the timers referred to, etc. In other words, these expressions stand for values that are not known until the action or condition concerned is executed in the appropriate state.

The syntax of the expressions concerned, called value expressions, is as follows:

<value expr> ::=
  <operator nm> [ ( <value expr> {, <value expr>}* ) ]
 | *cond* ( <boolean value expr> , <value expr> , <value expr> )
 | *value* ( <variable nm> , <pid value expr> )
 | *active* ( <signal nm> [ ( <value expr> {, <value expr>}* ) ] , <pid value expr> )
 | *now*

where the terminal productions of <operator nm>, <variable nm> and <signal nm> are assumed to yield the sets $Op_{\mathcal{A}}$, $V_\kappa$ and $S_\kappa$, respectively. $ValE_\kappa$ denotes the set of all syntactically correct value expressions. The forms of value expressions distinguished above correspond to operator applications, conditional expressions, view expressions, active expressions, and the expression **now**, respectively, in $\varphi^-$SDL.

We define now the set $SigP_\kappa$ of signal patterns, the set $SigE_\kappa$ of signal expressions, the set $SaveSet_\kappa$ of save sets and the set $PrCrD_\kappa$ of process creation data. Further on, we will look at a signal as an object that consists of the name of the signal and the sequence of values that it carries. Signal patterns and signal expressions are like signals, but variables and value expressions, respectively, are used instead of values. A save set is just a finite set of signal names. A process creation datum consists of the name of the process to be created, its formal parameters, expressions denoting its actual parameters and the pid value of its creator.

$$\begin{array}{ll}
SigP_\kappa & = \{(s, \langle v_1, \ldots, v_n\rangle) \mid (s, \langle T_1, \ldots, T_n\rangle) \in sigds(\kappa), v_1, \ldots, v_n \in V_\kappa\} \\
SigE_\kappa & = \{(s, \langle e_1, \ldots, e_n\rangle) \mid (s, \langle T_1, \ldots, T_n\rangle) \in sigds(\kappa), e_1, \ldots, e_n \in ValE_\kappa\} \\
SaveSet_\kappa & = \mathcal{P}_{\mathrm{fin}}(S_\kappa) \\
PrCrD_\kappa & = \{(X, \langle v_1, \ldots, v_n\rangle, \langle e_1, \ldots, e_n\rangle, e) \mid \\
 & \qquad X \in P_\kappa, v_1, \ldots, v_n \in V_\kappa, e_1, \ldots, e_n, e \in ValE_\kappa\}
\end{array}$$

We write $pnm(d)$, where $d = (X, vs, es, e) \in PrCrD_\kappa$, for $X$. Each process creation datum contains the formal parameters for the process type concerned. The alternative would be to make the association between process types and their formal parameters itself a parameter of the state operator, which is very unattractive.

The following actions are used:

$$\begin{array}{ll}
input & : SigP_\kappa \times SaveSet_\kappa \times ValE_\kappa \\
outpute & : SigE_\kappa \times ValE_\kappa \times ValE_\kappa \\
outputi & : SigE_\kappa \times ValE_\kappa \times (P_\kappa \cup \{\mathbf{env}\}) \\
set & : ValE_\kappa \times SigE_\kappa \times ValE_\kappa \\
reset & : SigE_\kappa \times ValE_\kappa \\
ass & : V_\kappa \times ValE_\kappa \times ValE_\kappa \\
cr & : PrCrD_\kappa \\
stop & : ValE_\kappa \\
inispont & : ValE_\kappa \\
\mathbb{t} & :
\end{array}$$

$Cr_\kappa$ denotes the set of all *cr* actions, and $Act^-_\kappa$ denotes the set of all *input*, *outpute*, *outputi*, *set*, *reset*, *ass*, *stop* and *inispont* actions. The $\mathbb{t}$ action is a special action with no observable effect whatsoever. The other actions correspond to the input guards, the SDL actions, the terminator **stop** and the void guard **input none**. The last argument of each action is the pid value of the process from which the action originates, except for the *outpute* and *outputi* actions where the pid value concerned is available as the second argument. The *outpute* and *outputi* actions correspond to **output** actions with explicit addressing and implicit addressing, respectively, in $\varphi^-$SDL. The last argument of these actions refers to the receiver. With an *outpute* action, the receiver is fully determined by the pid value given as the last argument. With an *outputi* action, the receiver is not fully determined; it is an arbitrary process of the given type.

The conditions used are built from the following atomic conditions:

$$\begin{array}{ll} \mathit{waiting} & : \mathit{SaveSet}_\kappa \times \mathit{ValE}_\kappa \\ \mathit{type} & : \mathit{ValE}_\kappa \times (P_\kappa \cup \{\textsf{env}\}) \\ \mathit{hasinst} & : P_\kappa \cup \{\textsf{env}\} \\ \mathit{eq} & : \mathit{ValE}_\kappa \times \mathit{ValE}_\kappa \end{array}$$

$\mathit{AtCond}_\kappa$ denotes the set of all *waiting*, *type*, *hasinst* and *eq* conditions. A condition $\mathit{waiting}(ss, e)$ is used to test whether the process with the pid value denoted by $e$ has to wait for a signal if the signals with names in $ss$ are withhold from being consumed. A condition $\mathit{type}(e, X)$ is used to test whether $X$ is the type of the process with the pid value denoted by $e$. A condition $\mathit{hasinst}(X)$ is used to test whether there exists a process of type $X$. A condition $\mathit{eq}(e_1, e_2)$ is used to test whether the expressions $e_1$ and $e_2$ denote the same value. These conditions are used to give meaning to the state definitions, output actions and decisions of $\varphi^-$SDL.

## 4.3 Semantics of process definitions

We describe now the meaning of $\varphi^-$SDL process definitions and their constituents. The meaning of the process definitions occurring in the examples from Section 3.6 is presented in Section 4.4.

### 4.3.1 Process definitions

The meaning of each process definition occurring in a system definition consists of the process name introduced and a family of processes, one for each possible pid value, which are described by means of finite guarded recursive specifications in $\mathrm{BPA}^{\mathrm{psc}}_{\mathrm{drt}}$. The meaning of a process definition is expressed in terms of the meaning of its start transition and its state definitions.

$$\begin{array}{l} [\![\textsf{process}\ X\textsf{(}k\textsf{); fpar}\ v_1\textsf{,}\ldots\textsf{,}v_m\textsf{; start;}\ tr\ d_1 \ldots d_n\ \textsf{endprocess;}]\!]^\kappa\ := \\ \quad (X, \{i \mapsto \langle X | \{X\ =\ [\![tr]\!]^{\kappa'}_i, [\![d_1]\!]^{\kappa'}_i, \ldots, [\![d_n]\!]^{\kappa'}_i\}\rangle \mid i \in \mathbb{N}\}) \\ \quad \text{where}\ \ \kappa'\ =\ \mathit{updscopeunit}(\kappa, X) \end{array}$$

The recursive specification of the process of type $X$ with pid value $i$ describes how it behaves at its start (the equation $X = [\![tr]\!]^{\kappa'}_i$) and how it behaves from each of the $n$ states in which it may come while it proceeds (the equations $[\![d_1]\!]^{\kappa'}_i, \ldots, [\![d_n]\!]^{\kappa'}_i$).

In the remainder of this section, we will be loose in the explanation of the meaning of the constituents of process definitions about the fact that there is always a family of meanings, one for each possible pid value.

### 4.3.2 States and transitions

The meaning of a state definition, occurring in the scope of a process definition, is a process described by an equation of the form $Z = s_Z$ where $Z$ is a variable corresponding to the state and $s_Z$ is a $\mathrm{BPA}^{\mathrm{psc}}_{\mathrm{drt}}$ term that only contains variables corresponding to states introduced in the process definition concerned. The equation describes how a process of the type being defined behaves from the state. The meaning of the state definition 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.

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

$$\begin{aligned}
&[\![\textbf{input}\ s\textbf{(}v_1\textbf{,}\ldots\textbf{,}v_n\textbf{);}tr]\!]^\kappa_i\ :=\ \underline{\underline{\mathit{input}}}((s,\langle v_1,\ldots,v_n\rangle),ss,\underline{i}) \cdot [\![tr]\!]^\kappa_i\\
&\quad \text{where}\ \ ss = \mathit{saveset}(\kappa)
\end{aligned}$$

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

The equation that corresponds to a state definition describes that a process of type $X$ behaves 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. Entering a state is supposed to take place by way of some internal action – thus it is precluded that a process is in more than one state. We use process names with state name subscripts, such as $X_{st}$ above, as variables. Notice that, in the absence of spontaneous transitions, a delay becomes inescapable if there are no more signals to consume in the current time slice. The process expression that corresponds to a guarded transition alternative expresses that the transition $tr$ is initiated on consumption of a signal of type $s$. In 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** (see Section 4.5).

The meaning of a transition is described by a process expression – a $\mathrm{BPA}^{\mathrm{psc}}_{\mathrm{drt}}$ term to be precise. It is expressed in terms of the meaning of its actions and its transition terminator.

$$\begin{aligned}
&[\![a_1\ldots a_n\ \textbf{nextstate}\ st\textbf{;}]\!]^\kappa_i\ :=\ [\![a_1]\!]^\kappa_i \cdot \ldots \cdot [\![a_n]\!]^\kappa_i \cdot X_{st}\\
&\quad \text{where}\ \ X = \mathit{scopeunit}(\kappa)
\end{aligned}$$

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

$$[\![a_1\ldots a_n\ \mathit{dec}\textbf{;}]\!]^\kappa_i\ :=\ [\![a_1]\!]^\kappa_i \cdot \ldots \cdot [\![a_n]\!]^\kappa_i \cdot [\![\mathit{dec}]\!]^\kappa_i$$

The process expression that corresponds to a transition terminated by **nextstate** *st* 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*. In case of termination by **stop**, the process expression expresses that it ends with ceasing to exist; and in case of termination by a decision *dec*, that it goes on behaving as described by decision *dec*.

The meaning of a decision is described by a process expression as well. It is expressed in terms of the meaning of its expressions and transitions.

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

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

The process expression that corresponds to a decision with a question expression $e$ expresses that the decision transfers control to the transition $tr_i$ for which the value of $e$ equals the value of $e_i$. In case of a decision with **any** instead, the process expression expresses that the decision transfers non-deterministically control to one of the transitions $tr_1, \ldots, tr_n$.

### 4.3.3 Actions

The meaning of an SDL action is described by a process expression, of course. It is expressed in terms of the meaning of the expressions occurring in it. It also depends on the occurring names (variable names, signal names, signal route names and process names – dependent on the kind of action).

$$[\![\textbf{output } s\textbf{(}e_1\textbf{,}\ldots\textbf{,}e_n\textbf{) to } e \textbf{ via } r_1\textbf{,}\ldots\textbf{,}r_m\textbf{;}]\!]_i^\kappa \;:=\; type([\![e]\!]_i, X_1) \vee \ldots \vee type([\![e]\!]_i, X_m) \vdots\!\rightarrow \underline{\underline{output}}((s, \langle [\![e_1]\!]_i, \ldots, [\![e_n]\!]_i\rangle), \underline{i}, [\![e]\!]_i) + \neg(type([\![e]\!]_i, X_1) \vee \ldots \vee type([\![e]\!]_i, X_m)) \vdots\!\rightarrow \underline{\underline{tt}}$$

where for $1 \le j \le m$: $X_j \;=\; rcv(\kappa, r_j)$

$$[\![\textbf{output } s\textbf{(}e_1\textbf{,}\ldots\textbf{,}e_n\textbf{) via } r_1\textbf{,}\ldots\textbf{,}r_m\textbf{;}]\!]_i^\kappa \;:=\; \underline{\underline{outputi}}((s, \langle [\![e_1]\!]_i, \ldots, [\![e_n]\!]_i\rangle), \underline{i}, X_1)) + \ldots + \underline{\underline{outputi}}((s, \langle [\![e_1]\!]_i, \ldots, [\![e_n]\!]_i\rangle), \underline{i}, X_m)) + \neg(\overline{hasinst}(X_1) \wedge \ldots \wedge hasinst(X_m)) \vdots\!\rightarrow \underline{\underline{tt}}$$

where for $1 \le j \le m$: $X_j \;=\; rcv(\kappa, r_j)$

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

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

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

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

All cases except the ones for output actions are straightforward. The cases of output actions need further explanation. In the case of an output action with explicit addressing, the process with pid value $e$ must be of the receiver type associated with one of the signal routes $r_1, \ldots, r_m$. Therefore, the condition $type([\![e]\!]_i, X_1) \vee \ldots \vee type([\![e]\!]_i, X_m)$ is used. If the process with pid value $e$ is not of the receiver type associated with any of the signal routes $r_1, \ldots, r_m$, 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 $\neg(type([\![e]\!]_i, X_1) \vee \ldots \vee type([\![e]\!]_i, X_m)) \mathrel{:\!\!\rightarrow} \underline{\underline{\#}}$. In the case of an output action with implicit addressing, 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. Therefore, there is a summand for the receiver type of each signal route. If a process of the receiver type for the chosen signal route does not exist, the signal is simply discarded and no error occurs. This is expressed by the summand $\neg(hasinst(X_1) \wedge \ldots \wedge hasinst(X_m)) \mathrel{:\!\!\rightarrow} \underline{\underline{\#}}$. Note that the signal may already be discarded if there is one signal route for which there exists no process of its receiver type.

### 4.3.4 Values

The meaning of a $\varphi^-$SDL expression is given by a translation to a value expression of the same kind. There is a close correspondence between the $\varphi^-$SDL expressions and their translations. Essential of the translation is that $\underline{i}$ 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.

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

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

$[\![v]\!]_i := value(v, \underline{i})$

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

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

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

$[\![\textbf{self}]\!]_i := \underline{i}$

$[\![\textbf{parent}]\!]_i := value(\textbf{parent}, \underline{i})$

$[\![\textbf{offspring}]\!]_i := value(\textbf{offspring}, \underline{i})$

$[\![\textbf{sender}]\!]_i := value(\textbf{sender}, \underline{i})$

## 4.4 Examples

We present the meaning of the process definitions occurring in the examples from Section 3.6. To be more precise, we give for each process definition a constant of the form $\langle X|E \rangle$ that stands for the process of the type concerned with pid value $i$.

It is clear that there are many similarities with the original process definitions in $\varphi^-$SDL. There is an equation for each state, the right hand side of each equation has a summand for each transition alternative of the corresponding state, etc. In addition, there is always a summand in which the time unit delay operator appears; this summand allows a delay to a future time slice to occur if there is no input to be read from the input queue of the process concerned. The main difference between the $\varphi^-$SDL process definitions and the description of their meaning in $\mathrm{BPA}^{\mathrm{psc}}_{\mathrm{drt}}\mathrm{rec}$ is that the latter can be subjected to equational reasoning using the axioms of $\mathrm{BPA}^{\mathrm{psc}}_{\mathrm{drt}}\mathrm{rec}$.

### 4.4.1 Repeater

The rep process with pid value $i$ is

$$
\begin{array}{l}
\langle \mathtt{rep}| \\
\ \{\mathtt{rep} = \mathtt{rep}_{\mathtt{pass}}, \\
\quad \mathtt{rep}_{\mathtt{pass}} = \underline{\underline{tt}} \cdot (\underline{\underline{input}}((\mathtt{s}, \langle\,\rangle), \emptyset, \underline{i}) \cdot \\
\qquad\qquad (\underline{\underline{outputi}}((\mathtt{s}, \langle\,\rangle), \underline{i}, \textbf{env}) + \neg hasinst(\textbf{env}) \vdots\!\rightarrow \underline{\underline{tt}}) \cdot \mathtt{rep}_{\mathtt{pass}} + \\
\qquad\qquad \underline{\underline{waiting}}(\emptyset, \underline{i}) \vdots\!\rightarrow \sigma_{\mathsf{rel}}(\mathtt{rep}_{\mathtt{pass}})) \\
\ \} \\
\rangle
\end{array}
$$

### 4.4.2 Address driven router

The rtr process with pid value $i$ is

$$
\begin{array}{l}
\langle \mathtt{rtr}| \\
\ \{\mathtt{rtr} = \underline{\underline{cr}}((\mathtt{rep}, \langle\,\rangle, \langle\,\rangle, \underline{i})) \cdot \underline{\underline{ass}}(\mathtt{rep1}, value(\textbf{offspring}, \underline{i}), \underline{i}) \cdot \\
\qquad\quad \underline{\underline{cr}}((\mathtt{rep}, \langle\,\rangle, \langle\,\rangle, \underline{i})) \cdot \underline{\underline{ass}}(\mathtt{rep2}, value(\textbf{offspring}, \underline{i}), \underline{i}) \cdot \mathtt{rtr}_{\mathtt{route}}, \\
\quad \mathtt{rtr}_{\mathtt{route}} = \underline{\underline{tt}} \cdot (\underline{\underline{input}}((\mathtt{s}, \langle \mathtt{a} \rangle), \emptyset, \underline{i}) \cdot \\
\qquad\qquad (eq(value(\mathtt{a}, \underline{i}), \mathtt{False}) \vdots\!\rightarrow (type(value(\mathtt{rep1}, \underline{i}), \mathtt{rep}) \vdots\!\rightarrow \\
\qquad\qquad\ \underline{\underline{outpute}}((\mathtt{s}, \langle value(\mathtt{a}, \underline{i}) \rangle), \underline{i}, value(\mathtt{rep1}, \underline{i})) + \\
\qquad\qquad\ \neg type(value(\mathtt{rep1}, \underline{i}), \mathtt{rep}) \vdots\!\rightarrow \underline{\underline{tt}}) \cdot \mathtt{rtr}_{\mathtt{route}} + \\
\qquad\qquad\ eq(value(\mathtt{a}, \underline{i}), \mathtt{True}) \vdots\!\rightarrow (type(value(\mathtt{rep2}, \underline{i}), \mathtt{rep}) \vdots\!\rightarrow \\
\qquad\qquad\ \underline{\underline{outpute}}((\mathtt{s}, \langle value(\mathtt{a}, \underline{i}) \rangle), \underline{i}, value(\mathtt{rep2}, \underline{i})) + \\
\qquad\qquad\ \neg type(value(\mathtt{rep2}, \underline{i}), \mathtt{rep}) \vdots\!\rightarrow \underline{\underline{tt}}) \cdot \mathtt{rtr}_{\mathtt{route}}) + \\
\qquad\qquad waiting(\emptyset, \underline{i}) \vdots\!\rightarrow \sigma_{\mathsf{rel}}(\mathtt{rtr}_{\mathtt{route}})) \\
\ \} \\
\rangle
\end{array}
$$

The rep process with pid value $i$ is

$$
\begin{array}{l}
\langle \mathtt{rep}| \\
\ \{\mathtt{rep} = \mathtt{rep}_{\mathtt{pass}}, \\
\quad \mathtt{rep}_{\mathtt{pass}} = \underline{\underline{tt}} \cdot (\underline{\underline{input}}((\mathtt{s}, \langle \mathtt{a} \rangle), \emptyset, \underline{i}) \cdot \\
\qquad\qquad (\underline{\underline{outputi}}((\mathtt{s}, \langle \mathtt{a} \rangle), \underline{i}, \textbf{env}) + \neg hasinst(\textbf{env}) \vdots\!\rightarrow \underline{\underline{tt}}) \cdot \mathtt{rep}_{\mathtt{pass}} + \\
\qquad\qquad \underline{\underline{waiting}}(\emptyset, \underline{i}) \vdots\!\rightarrow \sigma_{\mathsf{rel}}(\mathtt{rep}_{\mathtt{pass}})) \\
\ \} \\
\rangle
\end{array}
$$

### 4.4.3 Load driven router

The rtr process with pid value $i$ is

$$
\begin{array}{l}
\langle \mathtt{rtr}| \\
\ \{\mathtt{rtr} = \underline{\underline{cr}}((\mathtt{trep}, \langle \mathtt{delay} \rangle, \langle 10 \rangle, \underline{i})) \cdot \underline{\underline{ass}}(\mathtt{rep1}, value(\textbf{offspring}, \underline{i}), \underline{i}) \cdot \\
\qquad\quad \underline{\underline{cr}}((\mathtt{trep}, \langle \mathtt{delay} \rangle, \langle 20 \rangle, \underline{i})) \cdot \underline{\underline{ass}}(\mathtt{rep2}, value(\textbf{offspring}, \underline{i}), \underline{i}) \cdot \mathtt{rtr}_{\mathtt{route}}, \\
\quad \mathtt{rtr}_{\mathtt{route}} = \underline{\underline{tt}} \cdot (\underline{\underline{input}}((\mathtt{s}, \langle\,\rangle), \emptyset, \underline{i}) \cdot \\
\qquad\qquad (eq(value(\mathtt{idle}, value(\mathtt{rep1}, \underline{i})) \leftrightarrow value(\mathtt{idle}, value(\mathtt{rep2}, \underline{i})), \mathtt{True}) \vdots\!\rightarrow \\
\qquad\qquad\ (\underline{\underline{outputi}}((\mathtt{s}, \langle\,\rangle), \underline{i}, \mathtt{trep}) + \neg hasinst(\mathtt{trep}) \vdots\!\rightarrow \underline{\underline{tt}}) \cdot \mathtt{rtr}_{\mathtt{route}} + \\
\qquad\qquad\ eq(value(\mathtt{idle}, value(\mathtt{rep1}, \underline{i})) \leftrightarrow value(\mathtt{idle}, value(\mathtt{rep2}, \underline{i})), \mathtt{False}) \vdots\!\rightarrow
\end{array}
$$

$$
\begin{array}{l}
\quad (eq(value(\mathtt{idle}, value(\mathtt{rep1}, \underline{i})), \mathtt{True}) \mathrel{:\!\!\rightarrow} \\
\quad\ (type(value(\mathtt{rep1}, \underline{i}), \mathtt{rep}) \mathrel{:\!\!\rightarrow} \underline{\underline{outpute}}((\mathtt{s}, \langle\,\rangle), \underline{i}, value(\mathtt{rep1}, \underline{i})) + \\
\quad\ \neg type(value(\mathtt{rep1}, \underline{i}), \mathtt{rep}) \mathrel{:\!\!\rightarrow} \underline{\underline{tt}}) \cdot \mathtt{rtr}_{\mathtt{route}} + \\
\quad\ eq(value(\mathtt{idle}, value(\mathtt{rep1}, \underline{i})), \mathtt{False}) \mathrel{:\!\!\rightarrow} \\
\quad\ (type(value(\mathtt{rep2}, \underline{i}), \mathtt{rep}) \mathrel{:\!\!\rightarrow} \underline{\underline{outpute}}((\mathtt{s}, \langle\,\rangle), \underline{i}, value(\mathtt{rep2}, \underline{i})) + \\
\quad\ \neg type(value(\mathtt{rep2}, \underline{i}), \mathtt{rep}) \mathrel{:\!\!\rightarrow} \underline{\underline{tt}}) \cdot \mathtt{rtr}_{\mathtt{route}}) + \\
\quad waiting(\emptyset, \underline{i}) \mathrel{:\!\!\rightarrow} \sigma_{\mathsf{rel}}(\mathtt{rtr}_{\mathtt{route}}))
\end{array}
$$

$\}$

$\rangle$

The `trep` process with pid value $i$ is

$$
\begin{array}{l}
\langle \mathtt{trep} | \\
\quad \{\mathtt{trep} = \underline{\underline{ass}}(\mathtt{idle}, \mathtt{True}) \cdot \mathtt{trep}_{\mathtt{get}}, \\
\quad\ \mathtt{trep}_{\mathtt{get}} = \underline{\underline{tt}} \cdot (\underline{\underline{input}}((\mathtt{s}, \langle\,\rangle), \emptyset, \underline{i}) \cdot \\
\qquad\qquad \underline{\underline{ass}}(\mathtt{idle}, \mathtt{False}) \cdot \underline{\underline{set}}(now + \mathtt{delay}, \mathtt{t}) \cdot \mathtt{trep}_{\mathtt{put}} + \\
\qquad\qquad waiting(\emptyset, \underline{i}) \mathrel{:\!\!\rightarrow} \sigma_{\mathsf{rel}}(\mathtt{trep}_{\mathtt{get}})), \\
\quad\ \mathtt{trep}_{\mathtt{put}} = \underline{\underline{tt}} \cdot (\underline{\underline{input}}((\mathtt{t}, \langle\,\rangle), \{\mathtt{s}\}, \underline{i}) \cdot \\
\qquad\qquad (\underline{\underline{outputi}}((\mathtt{s}, \langle\,\rangle), \underline{i}, \mathbf{env}) + \neg hasinst(\mathbf{env}) \mathrel{:\!\!\rightarrow} \underline{\underline{tt}}) \cdot \\
\qquad\qquad \underline{\underline{ass}}(\mathtt{idle}, \mathtt{True}) \cdot \mathtt{trep}_{\mathtt{get}} + \\
\qquad\qquad waiting(\{\mathtt{s}\}, \underline{i}) \mathrel{:\!\!\rightarrow} \sigma_{\mathsf{rel}}(\mathtt{trep}_{\mathtt{put}})) \\
\quad \}
\end{array}
$$

$\rangle$

## 4.5 Interaction with states

In the semantic of $\varphi^{-}$SDL process definitions, ACP actions are used to give meaning to input guards, SDL actions, the terminator **stop** and the void guard **input none**. Thus, the facilities for storage, communication, timing and process creation offered by these language constructs are not fully covered; the ACP actions are meant to interact with a system state. In the semantics of $\varphi^{-}$SDL system definitions, which will be presented in Section 4.6, the state operator mentioned in Section 2 is used to describe this. First, we will describe the state space, the actions that may appear as the result of executing the above-mentioned actions in a state, and the result of executing processes, built up from these actions, in a state from the state space.

### 4.5.1 State space, actions and propositional signals

The state space, used to describe the meaning of system definitions, depends upon the specific variable names, signal names and process names introduced in the system definition concerned. That is, the sets $V_\kappa$, $S_\kappa$ and $P_\kappa$ are used here as well.

First, we define the set $Sig_\kappa$ of signals and the set $ExtSig_\kappa$ of extended signals. A signal consist of the name of the signal 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.

$$
\begin{array}{lcl}
Sig_\kappa & = & \{(s, \langle u_1, \ldots, u_n\rangle) \mid (s, \langle T_1, \ldots, T_n\rangle) \in sigds(\kappa), u_1, \ldots, u_n \in U\} \\
ExtSig_\kappa & = & Sig_\kappa \times \mathbb{N}_1 \times \mathbb{N}_1
\end{array}
$$

We write $snm(sig)$ and $vals(sig)$, where $sig = (s, vs) \in Sig_\kappa$, for $s$ and $vs$, respectively. We write $sig(xsig)$, $snd(xsig)$ and $rcv(xsig)$, where $xsig = (sig, i, i') \in ExtSig_\kappa$, for $sig$, $i$ and $i'$, respectively. Note that 0 is excluded as pid value of the sender or receiver of a signal because it is a special pid value that never refers to any existing process.

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

$$\begin{array}{lcl} Stg_\kappa & = & \bigcup_{V \subseteq V_\kappa} (V \to U) \\ InpQ_\kappa & = & ExtSig_\kappa{}^* \\ Timers_\kappa & = & \bigcup_{T \in \mathcal{P}_{\mathrm{fin}}(Sig_\kappa)} (T \to (\mathbb{N} \cup \{\mathsf{nil}\})) \end{array}$$

We will follow the convention that the domain of a function from $Stg_\kappa$ does not contain variables with which no value is associated because a value has never been assigned to them. 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 $\mathsf{nil}$. The basic operations on $Stg_\kappa$ and $Timers_\kappa$ are general operations on functions: function application, overriding ($\oplus$) and domain subtraction ($\lhd$). Overriding and domain subtraction are defined in Appendix A. In so far as the facilities for communication are concerned, the basic operations on $InpQ_\kappa$ are the functions

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

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

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

and the function $rmvfirst$ is recursively defined by

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

For each process, 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. For a given set of extended signals it gives the set of all possible sequences of them. It is inductively defined by

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

We define now the set $\mathcal{L}_\kappa$ of local states. The local state of a process contains, in addition to the above-mentioned components, the name of the process. 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.

$$\mathcal{L}_\kappa \;=\; \mathit{Stg}_\kappa \times \mathit{InpQ}_\kappa \times \mathit{Timers}_\kappa \times P_\kappa$$

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

The global state of a system contains, besides a local state for each existing process, a component keeping track of the system time. To keep track of the system time, natural numbers suffice. We define the state space $\mathcal{G}_\kappa$ of global states as follows:

$$\mathcal{G}_\kappa \;=\; \mathbb{N} \times \bigcup_{I \in \mathcal{P}_{\mathrm{fin}}(\mathbb{N}_2)} (I \to \mathcal{L}_\kappa)$$

We write $\mathit{now}(G)$ and $\mathit{lsts}(G)$, where $G = (n, \Sigma) \in \mathcal{G}_\kappa$, for $n$ and $\Sigma$, respectively. We write $\mathit{exists}(i, G)$, where $i \in \mathbb{N}$ and $G \in \mathcal{G}_\kappa$, for $i \in \mathit{dom}(\mathit{lsts}(G))$. Note that 1 is excluded from being used as pid value of an existing process of the system because it is a special pid value that is reserved for the environment.

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. These propositions are built from the following atomic propositions:

$$\begin{array}{ll} \mathit{value} & : V_\kappa \times U \times \mathbb{N}_2 \\ \mathit{active} & : \mathit{Sig}_\kappa \times \mathbb{N}_2 \end{array}$$

$\mathit{AtProp}_\kappa$ denotes the set of all *value* and *active* propositions. We write $\mathit{Prop}_\kappa$ for the set of all propositions that can be built from $\mathit{AtProp}_\kappa$. An atomic proposition of the form $\mathit{value}(v, u, i)$ 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 $\mathit{active}(\mathit{sig}, i)$ 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 in Section 4.5.4, makes only visible the value of all local variables and the set of active timers for all existing processes.

Below, we introduce the additional actions that are used for the semantics of $\varphi^-$SDL system definitions. Like most of the actions used for the semantics of $\varphi^-$SDL process definitions, these actions are parametrized. The following additional actions are used:

$$\begin{array}{ll} \overline{cr} & : \mathbb{N} \times \mathit{PrCrD}_\kappa \\ \mathit{input}' & : \mathit{ExtSig}_\kappa \\ \mathit{output}' & : \mathit{ExtSig}_\kappa \\ \mathit{set}' & : \mathbb{N} \times \mathit{Sig}_\kappa \times \mathbb{N}_2 \\ \mathit{reset}' & : \mathit{Sig}_\kappa \times \mathbb{N}_2 \end{array}$$

$\overline{Cr}_\kappa$ denotes the set of all $\overline{cr}$ actions; and $\mathit{Act}'_\kappa$ denotes the set of all $\mathit{input}'$, $\mathit{output}'$, $\mathit{set}'$ and $\mathit{reset}'$ actions. The $\overline{cr}$ actions appears as the result of applying the process creation operator $\mathrm{E}^n_\Phi$ to *cr* actions. The $\mathit{input}'$, $\mathit{output}'$, $\mathit{set}'$ and $\mathit{reset}'$ actions appear as the result of applying the state operator $\lambda_G$ to *input*, *outpute*/*outputi*, *set* and *reset* actions, respectively (see Section 4.5.4).

### 4.5.2 State transformers and observers

In the process algebra semantics of $\varphi^-$SDL process definitions, presented in Section 4.3, ACP actions are used to describe the meaning of input guards, SDL actions, the terminator **stop** and the void guard **input none**. These ACP actions are meant to interact with a state from $\mathcal{G}_\kappa$. Later on, 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 partly boil down to describing how the actions, and the progress of time (modeled by the time unit delay operator $\sigma_{\mathsf{rel}}$), transform states. For the sake of comprehensibility, we will first define matching state transforming operations.

In addition, we will define some state observing operations. Two of the state observing operations are used directly to define the action and effect function of the state operator and three others are used to define the valuation function of the state operator. The remaining two are used to define the signal function of the state operator as well as an auxiliary evaluation function needed for the value expressions that occur as arguments, and as components of arguments, of the above-mentioned actions and conditions (see Section 4.2).

**State transformers:**

In general, the state transformers change one or two components of the local state of one process. The notable exception is *csmsig*, which is defined first. It may change all components except, of course, the process type. This is a consequence of the fact that the facilities for storage, communication and timing are rather intertwined on the consumption of signals in $\varphi^-$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 $csmsig : ExtSig_\kappa \times {V_\kappa}^* \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how the ACP actions corresponding to the input guards of $\varphi^-$SDL transform states.

$$
\begin{array}{l}
csmsig((sig, i, i'), \langle v_1, \ldots, v_n \rangle, G) \;= \\
\quad (now(G), lsts(G) \oplus \{i' \mapsto (\rho, \sigma, \theta, X)\}) \quad \textbf{if } exists(i', G) \\
\quad G \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{otherwise}
\end{array}
$$

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

$csmsig((sig, i, i'), \langle v_1, \ldots, v_n \rangle, G)$ deals with the consumption of signal *sig* by the process with pid value $i'$. It transforms the local state of the process with pid value $i'$, the process by which the signal is consumed, as follows:

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

Everything else is left unchanged.

The function $sndsig : ExtSig_\kappa \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how the ACP actions corresponding to the output actions of $\varphi^-$SDL transform states.

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

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

$sndsig((sig, i, i'), G)$ deals with passing signal $sig$ from the process with pid value $i$ to the process with pid value $i'$. It transforms the local state of the process with pid value $i'$, the process to which the signal is passed, as follows:

- $sig$ is put into the input queue of the process to which the signal is passed (unless $i' = 1$, indicating that the signal is passed to the environment).

Everything else is left unchanged.

The function $settimer : \mathbb{N} \times Sig_\kappa \times \mathbb{N}_2 \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how the ACP actions corresponding to the set actions of $\varphi^-$SDL transform states.

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

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

$settimer(t, sig, i, G)$ 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 $resettimer : Sig_\kappa \times \mathbb{N}_2 \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how the ACP actions corresponding to the reset actions of $\varphi^-$SDL transform states.

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

$$
\begin{array}{lll}
\text{where} & \rho & = \mathit{stg}(\mathit{lsts}(G)_i), \\
& \sigma & = \mathit{rmvfirst}(\mathit{inpq}(\mathit{lsts}(G)_i), \mathit{sig}), \\
& \theta & = \{\mathit{sig}\} \lhd \mathit{timers}(\mathit{lsts}(G)_i), \\
& X & = \mathit{ptype}(\mathit{lsts}(G)_i)
\end{array}
$$

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

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

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

The function $\mathit{assignvar} : V_\kappa \times U \times \mathbb{N}_2 \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how the ACP actions corresponding to the assignment task actions of $\varphi^-$SDL transform states.

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

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

$\mathit{assignvar}(v, u, i, G)$ 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 $\mathit{createproc} : \mathbb{N}_2 \times \mathit{PrCrD}'_\kappa \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how the ACP actions corresponding to the create actions of $\varphi^-$SDL transform states. The elements of $\mathit{PrCrD}'_\kappa$ are like process creation data, i.e. elements of $\mathit{PrCrD}_\kappa$, but values are used instead of value expressions: $\mathit{PrCrD}'_\kappa = \{(X, \langle v_1, \ldots, v_n \rangle, \langle u_1, \ldots, u_n \rangle, i) \mid X \in P_\kappa,\ v_1, \ldots, v_n \in V_\kappa,\ u_1, \ldots, u_n \in U,\ i \in \mathbb{N}_2\}$.

$$
\begin{array}{l}
\mathit{createproc}(i', (X', \langle v_1, \ldots, v_n \rangle, \langle u_1, \ldots, u_n \rangle, i), G) \; = \\
\quad \begin{array}{ll}
(\mathit{now}(G), \mathit{lsts}(G) \oplus \{i \mapsto (\rho, \sigma, \theta, X), i' \mapsto (\rho', \sigma', \theta', X')\}) & \textbf{if } \mathit{exists}(i, G) \\
G & \textbf{otherwise}
\end{array} \\
\\
\begin{array}{lll}
\text{where} \;\; \rho & = & \mathit{stg}(\mathit{lsts}(G)_i) \oplus \{\textsf{offspring} \mapsto i'\}, \\
\sigma & = & \mathit{inpq}(\mathit{lsts}(G)_i), \\
\theta & = & \mathit{timers}(\mathit{lsts}(G)_i), \\
X & = & \mathit{ptype}(\mathit{lsts}(G)_i), \\
\rho' & = & \{v_1 \mapsto u_1, \ldots, v_n \mapsto u_n, \textsf{parent} \mapsto i, \textsf{offspring} \mapsto 0, \textsf{sender} \mapsto 0\}, \\
\sigma' & = & \langle\,\rangle, \\
\theta' & = & \{\,\}
\end{array}
\end{array}
$$

$\mathit{createproc}(i', (X', \langle v_1, \ldots, v_n \rangle, \langle u_1, \ldots, u_n \rangle, i), G)$ deals with creating a process of type $X'$. It 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 ($i'$) 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 $\mathit{stopproc} : \mathbb{N}_2 \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how the ACP actions corresponding to **stop** in $\varphi^-$SDL transform states.

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

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

The function $\mathit{inispont} : \mathbb{N}_2 \times \mathcal{G}_\kappa \to \mathcal{G}_\kappa$ is used to describe how the ACP actions used to initiate spontaneous transitions transform states.

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

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

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

$$
\begin{array}{l}
G' \in \mathit{unitdelay}(G) \leftrightarrow \\
\quad \mathit{now}(G') = \mathit{now}(G) + 1 \wedge \\
\quad \forall i \in \mathit{dom}(\mathit{lsts}(G)) \bullet \\
\qquad \mathit{stg}(\mathit{lsts}(G')_i) = \mathit{stg}(\mathit{lsts}(G)_i) \wedge \\
\qquad (\exists \sigma \in \mathit{InpQ} \bullet \\
\qquad\quad \mathit{inpq}(\mathit{lsts}(G')_i) = \mathit{inpq}(\mathit{lsts}(G)_i) \frown \sigma \wedge \\
\qquad\quad \sigma \in \mathit{merge}(\{(\mathit{sig}, i, i) \mid \mathit{timers}(\mathit{lsts}(G)_i)(\mathit{sig}) \leq \mathit{now}(G)\})) \wedge \\
\qquad \mathit{timers}(\mathit{lsts}(G')_i) = \\
\qquad\quad \mathit{timers}(\mathit{lsts}(G)_i) \oplus \{\mathit{sig} \mapsto \mathsf{nil} \mid \mathit{timers}(\mathit{lsts}(G)_i)(\mathit{sig}) \leq \mathit{now}(G)\} \wedge \\
\qquad \mathit{ptype}(\mathit{lsts}(G')_i) = \mathit{ptype}(\mathit{lsts}(G)_i)
\end{array}
$$

$\mathit{unitdelay}(G)$ transforms the global state as follows:

- the system time is incremented by one unit;
- for the local state of each process:
  - its storage is left unchanged;
  - the signals that correspond to expiring timers are put into the input queue in a non-deterministic way;
  - for each of the expiring timers, 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 exceptions are *has_instance* and *instances*, 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 f. This will not be explicitly mentioned in the explanations given below.

The function $\mathit{contents} : V_\kappa \times \mathbb{N}_2 \times \mathcal{G}_\kappa \to U \cup \{\mathsf{nil}\}$ is used to describe the value of expressions of the form $\mathit{value}(v, e)$, which correspond to the variable accesses and view expressions of $\varphi^-$SDL.

$$
\begin{array}{lcll}
\mathit{contents}(v, i, G) & = & \rho(v) & \textbf{if } \mathit{exists}(i, G) \wedge v \in \mathit{dom}(\rho) \\
 & & \mathsf{nil} & \textbf{otherwise} \\
\text{where } \rho & = & \mathit{stg}(\mathit{lsts}(G)_i) & 
\end{array}
$$

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

The function $\mathit{is\_active} : \mathit{Sig}_\kappa \times \mathbb{N}_2 \times \mathcal{G}_\kappa \to \mathbb{B}$ is used to describe the value of expressions of the form $\mathit{active}(\mathit{sig}, e)$, which correspond to the active expressions of $\varphi^-$SDL.

$$
\begin{array}{lcll}
\mathit{is\_active}(\mathit{sig}, i, G) & = & \mathsf{t} & \textbf{if } \mathit{exists}(i, G) \wedge \mathit{sig} \in \mathit{dom}(\theta) \\
 & & \mathsf{f} & \textbf{otherwise} \\
\text{where } \theta & = & \mathit{timers}(\mathit{lsts}(G)_i) & 
\end{array}
$$

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

The function $is\_waiting : SaveSet_\kappa \times \mathbb{N}_2 \times \mathcal{G}_\kappa \to \mathbb{B}$ is used to describe the value of conditions of the form $waiting(\{s_1, \ldots, s_n\}, e)$, which are used to give meaning to the state definitions of $\varphi^-$SDL.

$$\begin{array}{lcll} is\_waiting(ss, i, G) & = & \mathsf{t} & \textbf{if } exists(i, G) \wedge getnxt(\sigma, ss) = \mathsf{nil} \\ & & \mathsf{f} & \textbf{otherwise} \end{array}$$

where $\sigma = inpq(lsts(G)_i)$

$is\_waiting(ss, i, G)$ yields true iff there is no signal in the input queue of the process with pid value $i$ that has a name different from the ones in $ss$.

The function $type : \mathbb{N}_1 \times \mathcal{G}_\kappa \to P_\kappa \cup \{\textbf{env}\} \cup \{\mathsf{nil}\}$ is used to describe the value of conditions of the form $type(e, X)$, which are used to give meaning to the output actions with explicit addressing of $\varphi^-$SDL.

$$\begin{array}{lcll} type(i, G) & = & X & \textbf{if } exists(i, G) \\ & & \textbf{env} & \textbf{if } i = 1 \\ & & \mathsf{nil} & \textbf{otherwise} \end{array}$$

where $X = ptype(lsts(G)_i)$

$type(i, G)$ yields the type of the process with pid value $i$. Different from the other state observers, it yields a result if $i = 1$ as well, viz. **env**.

The function $has\_instance : (P_\kappa \cup \{\textbf{env}\}) \times \mathcal{G}_\kappa \to \mathbb{B}$ is used to describe the value of conditions of the form $hasinst(X)$, which are used to give meaning to the output actions with implicit addressing of $\varphi^-$SDL.

$$\begin{array}{lcll} has\_instance(X, G) & = & \mathsf{t} & \textbf{if } \exists i \in \mathbb{N}_1 \bullet (i = 1 \vee exists(i, G)) \wedge type(i, G) = X \\ & & \mathsf{f} & \textbf{otherwise} \end{array}$$

$has\_instance(X, G)$ yields true iff there exists a process of type $X$.

The function $nxtsig : SaveSet_\kappa \times \mathbb{N}_2 \times \mathcal{G}_\kappa \to ExtSig_\kappa \cup \{\mathsf{nil}\}$ is used to describe the result of executing *input* actions, which are used to give meaning to the input guards of $\varphi^-$SDL.

$$\begin{array}{lcll} nxtsig(ss, i, G) & = & getnxt(\sigma, ss) & \textbf{if } exists(i, G) \\ & & \mathsf{nil} & \textbf{otherwise} \end{array}$$

where $\sigma = inpq(lsts(G)_i)$

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

The function $instances : (P_\kappa \cup \{\textbf{env}\}) \times \mathcal{G}_\kappa \to \mathcal{P}_{\mathrm{fin}}(\mathbb{N}_2) \cup \{\{1\}\}$ is used to describe the result of executing *outputi* actions, which correspond to the output actions with implicit addressing of $\varphi^-$SDL.

$$\begin{array}{lcll} instances(X, G) & = & \{i \in dom(lsts(G)) \mid type(i, G) = X\} & \textbf{if } X \neq \textbf{env} \\ & & \{1\} & \textbf{otherwise} \end{array}$$

$instances(X, G)$ yields the set of pid values of all existing processes of type $X$ if $X \neq \textbf{env}$ and the singleton set $\{1\}$ otherwise.

### 4.5.3 Evaluation of value expressions

The function $eval_G$ is used to evaluate value expressions in a state $G$. The state observers *contents* and *is_active* defined in Section 4.5.2 are used to define the evaluation function.

$$
\begin{array}{l}
eval_G(op(e_1,\ldots,e_n)) \;=\; \\
\quad \begin{array}{ll}
op(eval_G(e_1),\ldots,eval_G(e_n)) & \textbf{if } eval_G(e_1) \neq \mathsf{nil} \wedge \ldots \wedge eval_G(e_n) \neq \mathsf{nil} \\
\mathsf{nil} & \textbf{otherwise}
\end{array}
\end{array}
$$

$$
\begin{array}{lll}
eval_G(cond(e_1,e_2,e_3)) \;=\; & eval_G(e_2) & \textbf{if } eval_G(e_1) = \mathsf{t} \\
& eval_G(e_3) & \textbf{if } eval_G(e_1) = \mathsf{f} \\
& \mathsf{nil} & \textbf{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
eval_G(value(v,e)) \;=\; & contents(v, eval_G(e), G) & \textbf{if } \; eval_G(e) \in \mathbb{N}_2 \\
& \mathsf{nil} & \textbf{otherwise}
\end{array}
$$

$$
\begin{array}{l}
eval_G(active(s(e_1,\ldots,e_n),e)) \;=\; \\
\quad \begin{array}{ll}
is\_active(sig, eval_G(e), G) & \textbf{if } \; eval_G(e_1) \neq \mathsf{nil} \wedge \ldots \wedge eval_G(e_n) \neq \mathsf{nil} \wedge \\
& \quad eval_G(e) \in \mathbb{N}_2 \\
\mathsf{nil} & \textbf{otherwise}
\end{array} \\
\quad \text{where } sig \;=\; (s, \langle eval_G(e_1),\ldots,eval_G(e_n)\rangle)
\end{array}
$$

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

In all of these cases, 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.

We extend $eval_G$ to sequences of value expressions and signal expressions as follows:

$$
\begin{array}{l}
eval_G(\langle e_1,\ldots,e_n\rangle) \;=\; \\
\quad \begin{array}{ll}
\langle eval_G(e_1),\ldots,eval_G(e_n)\rangle & \textbf{if } eval_G(e_1) \neq \mathsf{nil} \wedge \ldots \wedge eval_G(e_n) \neq \mathsf{nil} \\
\mathsf{nil} & \textbf{otherwise}
\end{array}
\end{array}
$$

$$
\begin{array}{lll}
eval_G((s,es)) \;=\; & (s, eval_G(es)) & \textbf{if } eval_G(es) \neq \mathsf{nil} \\
& \mathsf{nil} & \textbf{otherwise}
\end{array}
$$

### 4.5.4 Definition of the state operator

In this subsection, we define the functions *act*, *eff*, $eff_\sigma$, *sig* and *val* in completion of the definition of the state operator used to describe the meaning of $\varphi^-$SDL system definitions. Recall that for this state operator $S = \mathcal{G}_\kappa$.

#### Action and effect functions:

In the current application of $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$, $A = Cr_\kappa \cup Act_\kappa \cup Act'_\kappa \cup \{\not t\}$, where $Act_\kappa = Act^-_\kappa \cup \overline{Cr}_\kappa$. The actions in $Act_\kappa$ are actions that may change the state in which they are executed. The actions in $Act'_\kappa$ are actions that are performed as the result of the execution of actions in $Act_\kappa$ in a state. The actions in $Cr_\kappa$ and $\overline{Cr}_\kappa$ are the process creation actions and the actions that are left as a trace of the process creations that

occur, respectively – note that we use the data set $D = PrCrD_\kappa$ for process creation. The action and effect functions are trivial for actions that are not in $Act_\kappa$.

In order to keep the definitions comprehensible, we will use the following abbreviations. Let $e$ be a value expression, let $es$ be a sequence of value expressions, and let $se$ be a signal expression. Then we write $e'$ for $eval_G(e)$, $es'$ for $eval_G(es)$ and $se'$ for $eval_G(se)$.

$$act(input((s, \langle v_1, \ldots, v_n\rangle), ss, e), G) = \{input'((sig, i, j)) \mid snm(sig) = s, nxtsig(ss, e', G) = (sig, i, j), e' \in \mathbb{N}_2\}$$

$$eff(input((s, \langle v_1, \ldots, v_n\rangle), ss, e), G, a) = \begin{array}{ll} csmsig(nxtsig(ss, e', G), \langle v_1, \ldots, v_n\rangle, G) & \textbf{if } a \in act(input((s, \langle v_1, \ldots, v_n\rangle), ss, e), G) \\ G & \textbf{otherwise} \end{array}$$

$$act(outpute(se, e_1, e_2), G) = \{output'((se', e_1', e_2')) \mid se' \neq \mathsf{nil}, e_1' \in \mathbb{N}_2, e_2' \in \mathbb{N}_1\}$$

$$eff(outpute(se, e_1, e_2), G, a) = \begin{array}{ll} sndsig((se', e_1', e_2'), G) & \textbf{if } a \in act(outpute(se, e_1, e_2), G) \\ G & \textbf{otherwise} \end{array}$$

$$act(outputi(se, e, X), G) = \{output'((se', e', i)) \mid se' \neq \mathsf{nil}, e' \in \mathbb{N}_2, i \in instances(X, G)\}$$

$$eff(outputi(se, e, X), G, a) = \begin{array}{ll} sndsig((se', e', i), G) & \textbf{if } a \in act(outputi(se, e, X), G) \wedge a = output'((se', e', i)) \\ G & \textbf{otherwise} \end{array}$$

$$act(set(e_1, se, e_2), G) = \{set'(e_1', se', e_2') \mid e_1' \in \mathbb{N}, se' \neq \mathsf{nil}, e_2' \in \mathbb{N}_2\}$$

$$eff(set(e_1, se, e_2), G, a) = \begin{array}{ll} settimer(e_1', se', e_2', G) & \textbf{if } a \in act(set(e_1, se, e_2), G) \\ G & \textbf{otherwise} \end{array}$$

$$act(reset(se, e), G) = \{reset'(se', e') \mid se' \neq \mathsf{nil}, e' \in \mathbb{N}_2\}$$

$$eff(reset(se, e), G, a) = \begin{array}{ll} resettimer(se', e', G) & \textbf{if } a \in act(reset(se, e), G) \\ G & \textbf{otherwise} \end{array}$$

$$act(ass(v, e_1, e_2), G) = \{\mathbb{t} \mid e_1' \neq \mathsf{nil}, e_2' \in \mathbb{N}_2\}$$

$$eff(ass(v, e_1, e_2), G, a) = \begin{array}{ll} assignvar(v, e_1', e_2', G) & \textbf{if } a \in act(ass(v, e_1, e_2), G) \\ G & \textbf{otherwise} \end{array}$$

$$act(\overline{cr}(i, (X, vs, es, e)), G) = \{\mathbb{t} \mid i \in \mathbb{N}_2, es' \neq \mathsf{nil}, e' \in \mathbb{N}_2\}$$

$$eff(\overline{cr}(i, (X, vs, es, e)), G, a) = \begin{array}{ll} createproc(i, (X, vs, es', e'), G) & \textbf{if } a \in act(\overline{cr}(i, (X, vs, es, e)), G) \\ G & \textbf{otherwise} \end{array}$$

$$act(stop(e), G) = \{\mathbb{t} \mid e' \in \mathbb{N}_2\}$$

$$eff(stop(e), G, a) = \begin{array}{ll} stopproc(e', G) & \textbf{if } a \in act(stop(e), G) \\ G & \textbf{otherwise} \end{array}$$

$$act(inispont(e), G) = \{\mathbb{t} \mid e' \in \mathbb{N}_2\}$$

$$eff(inispont(e), G, a) = \begin{array}{ll} inispont(e', G) & \textbf{if } a \in act(inispont(e), G) \\ G & \textbf{otherwise} \end{array}$$

For all actions $a \in Cr_\kappa \cup Act'_\kappa \cup \{\lightning\}$:

$$act(a, G) = a$$
$$\mathit{eff}(a, G, a') = G$$

The effect of applying the state operator to a process of the form $\sigma_{\mathsf{rel}}(x)$ is described by mean of the function $\mathit{eff}_\sigma$.

$$\mathit{eff}_\sigma(G) = unitdelay(G)$$

**Signal function:**

First, we define the function $atoms : \mathcal{G}_\kappa \to \mathcal{P}_{\mathrm{fin}}(AtProp_\kappa)$ giving for each state the set of atomic propositions that hold in that state. It is inductively defined by

$$contents(v, i, G) = u \;\Rightarrow\; value(v, u, i) \in atoms(G)$$
$$is\_active(sig, i, G) = \mathsf{t} \;\Rightarrow\; active(sig, i) \in atoms(G)$$

We now define the function $sig : \mathcal{G}_\kappa \to Prop_\kappa$, giving the propositions produced by the states. as follows:

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

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

**Valuation function:**

In the current application of $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$, $B_{\mathrm{at}} = AtCond_\kappa \cup AtProp_\kappa$.

The function *val* is defined in the following way:

$$val(eq(e_1, e_2), G) = \begin{array}{ll} \mathsf{t} & \textbf{if } e'_1 = e'_2 \wedge e'_1 \neq \mathsf{nil} \wedge e'_2 \neq \mathsf{nil} \\ \mathsf{f} & \textbf{otherwise} \end{array}$$

$$val(waiting(ss, e), G) = \begin{array}{ll} \mathsf{t} & \textbf{if } is\_waiting(ss, e', G) = \mathsf{t} \wedge e' \in \mathbb{N}_2 \\ \mathsf{f} & \textbf{otherwise} \end{array}$$

$$val(type(e, X), G) = \begin{array}{ll} \mathsf{t} & \textbf{if } type(e', G) = X \wedge e' \in \mathbb{N}_1 \\ \mathsf{f} & \textbf{otherwise} \end{array}$$

$$val(hasinst(X), G) = has\_instance(X, G)$$

$$val(value(v, u, i), G) = \begin{array}{ll} \mathsf{t} & \textbf{if } value(v, u, i) \in atoms(G) \\ \mathsf{f} & \textbf{otherwise} \end{array}$$

$$val(active(sig, i), G) = \begin{array}{ll} \mathsf{t} & \textbf{if } active(sig, i) \in atoms(G) \\ \mathsf{f} & \textbf{otherwise} \end{array}$$

Note that the sets $AtCond_\kappa$ and $AtProp_\kappa$ are disjoint. The elements of $AtCond_\kappa$ are used as conditions with a truth value that depends upon the state. The elements of $AtProp_\kappa$ are not used as conditions and their valuation in a state is not needed for the semantics of $\varphi^-$SDL.

## 4.6 Semantics of system definitions

In this subsection, we present a semantics for $\varphi^-$SDL system definitions. It relies heavily upon the specifics of the state operator defined in Section 4.5.

According to the semantics presented here, the meaning of a $\varphi^-$SDL system definition is a process described by a process expression – a term of $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$ extended with the counting process creation operator and the state operator to be precise. It is given in terms of the semantics of $\varphi^-$SDL process definitions presented in Section 4.3 and the contextual information extracted by means of the function $\{\!|\_|\!\}$ defined in Appendix B.

### 4.6.1 System definition

The semantics of a $\varphi^-$SDL system definition depends on a parameter *Env* representing the environment of the system. This environment *Env* has to be described by a $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$ term.

$$[\![\textsf{system}\ S;D_1 \ldots D_n\ \textsf{endsystem;}]\!](Env) \;:=\; \lambda_{G_0}(\mathrm{E}^{n_0+2}_{\Phi}(P) \parallel Env)$$

$$\begin{aligned}
\text{where } P &= \|_{i\in\{1,\ldots,n_0\}}\, \Phi(i+1,(pt(i+1),\langle\,\rangle,\langle\,\rangle,0)),\\
\Phi &= \{(i,d)\mapsto \Psi(i) \mid \exists D\in\{D_1,\ldots,D_n\}\bullet [\![D]\!]^{\kappa} = (pnm(d),\Psi)\},\\
G_0 &= (0,\{i+1\mapsto L_0(i+1)\mid i\in\{1,\ldots,n_0\}\}),\\
L_0(i) &= (\{\textsf{parent}\mapsto 0,\textsf{offspring}\mapsto 0,\textsf{sender}\mapsto 0\},\langle\,\rangle,\{\,\},pt(i)),\\
n_0 &= \textstyle\sum_{X\in procs(\kappa)} init(\kappa,X),\\
\kappa &= \{\!|[\![\textsf{system}\ S;D_1\ldots D_n\ \textsf{endsystem;}]\!]|\!\}
\end{aligned}$$

$$\begin{aligned}
\text{and}\quad & pt:\{1+1,\ldots,n_0+1\}\to procs(\kappa) \text{ is such that}\\
& \forall X\in procs(\kappa)\bullet \mathrm{card}(pt^{-1}(X)) = init(\kappa,X).
\end{aligned}$$

The function *pt* is used to assign pid values for the processes created during system start-up. It maps a set of pid values to the types of the processes with these pid values.

The process expression that corresponds to a system definition expresses that, for each of the process types defined, the given initial number of processes are created and these processes are executed in parallel, starting in the state $G_0$, while they receive signals via signal routes from the environment *Env*. $G_0$ is the state in which the system time is zero and there is a local state for each of the processes that is created during system start-up. Recall that the pid value 1 is reserved for the environment. The mapping $\Phi$ from pid values and process creation data to process expressions is derived from the meaning of the process definitions occurring in the system definition. This mapping is used by the counting process creation operator, which is needed for process creation after system start-up.

### 4.6.2 Environments

The semantics of $\varphi^-$SDL system definitions describes the meaning of a system definition for an arbitrary process *Env* that represents the behaviour of the environment. Here we explain how the environment's behaviour is described by a $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$ process.

Some general assumptions have to be made about the behaviour of the environment of any system described using $\varphi^-$SDL. Further assumptions may be made about the behaviour of the environment of a specific system described using $\varphi^-$SDL, including

ones that facilitate analysis of the system concerned. The general assumptions made about environments are:

- the environment can only send signals that are defined in the system definition concerned;
- the environment can only send signals to processes to which it is connected by signal routes;
- the environment can send only a finite number of signals during a time slice.

Besides, the viewpoint is taken that the processes that constitute a system are not observable to its environment. This leads to the use of output actions with implicit addressing in representing the environment's behaviour.

The set $EnvSig_\kappa$ of possible environment signals is determined by the specific types of signals and signal routes introduced in the system definition concerned. It can be obtained from the environment signal description yielded by applying the function *envsigd* (defined in Appendix B) to the context ascribed to the system definition. For an arbitrary context $\kappa$, the set of environment signals is obtained as follows:

$$EnvSig_\kappa \;=\; \bigcup_{((s,\langle T_1,\ldots,T_n\rangle),X)\in envsigd(\kappa)} \{((s,\langle t_1,\ldots,t_n\rangle),1,X) \mid t_1 \in T_1^{\mathcal{A}},\ldots,t_n \in T_n^{\mathcal{A}}\}$$

It is clear that in general the set $EnvSig_\kappa$ is infinite because signals can carry values from infinite domains. Besides, the environment can send an arbitrary signal from $EnvSig_\kappa$. To represent this we need the alternative composition of an infinite number of alternatives. This can be done using the operator $\sum_{d:D}$ of $\mu$CRL [22]. However, the combination of this operator with the extension of discrete time process algebra we are using has not been investigated thoroughly. Besides, the potential unbounded non-determinism introduced by this operator does not allow to use conventional validation techniques.

A process that satisfies the three above-mentioned assumptions can be defined in the following way:

$$\begin{aligned} Env_\kappa &= \sum_{n:\mathbb{N}} Env_n \\ Env_0 &= \sigma_{\mathsf{rel}}(Env_\kappa) \\ Env_{n+1} &= Env_n + \sum_{osig:EnvSig_\kappa} \underline{\underline{outputi}}(osig)\cdot Env_n \end{aligned}$$

In order to use an environment process as a parameter of the presented semantics of $\varphi^-$SDL, it has to be given as a process in $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$. Below we define such an environment process. We call it a standard environment process for the semantics of $\varphi^-$SDL. It is determined by two restrictions:

- the set of signals which the environment can send to the system is restricted to a finite subset $ES \subseteq EnvSig_\kappa$;
- the maximal number of signals which can be sent in one time slice is bounded by a natural number $N_s$.

A standard environment is defined as follows:

$$
\begin{aligned}
Env_{\kappa}^{\mathrm{st}} &= \sum_{n\in\{0,\ldots,N_s\}} Env'_n \\
Env'_0 &= \sigma_{\mathsf{rel}}(Env_{\kappa}^{\mathrm{st}}) \\
Env'_{n+1} &= Env'_n + \sum_{osig\in ES} \underline{\underline{outputi}}(osig)\cdot Env'_n
\end{aligned}
$$

One may also define another environment process for a specific system. In any case the process representing the system's environment must satisfy the above-mentioned assumptions.

### 4.6.3 Delaying channels

The process algebra semantics of $\varphi^{-}$SDL makes clear how to model a delaying channel by means of a $\varphi^{-}$SDL process. Below a $\varphi^{-}$SDL process definition of such a process, called `ch`, is given. It is assumed that the process can only receive signals of type `es`.

The process consumes signals `es(rcv,v1,...,vn)` and passes them on after an arbitrary delay, possibly zero, with `rcv` replaced by `snd`. Each signal consumed carries the pid value of the ultimate receiver, and this pid value is replaced by the one of the original sender when the signal is passed on. This is needed because the original sender and ultimate receiver have now an intermediate receiver and intermediate sender, respectively. The `decision` construct is used to find out whether the original sender used implicit or explicit addressing. Note that we write `Null` for `0`, i.e. the special pid value that never refers to any existing process.

```
process ch(1);
  start;
    nextstate receive;
  state receive;
    input es(rcv,v1,...,vn);
      task snd := sender;
      nextstate deliver;
  state deliver;
    save es;
    input none;
      decision rcv = Null;
        (True):
          output es(snd,v1,...,vn) via sr_out;
          nextstate receive;
        (False):
          output es(snd,v1,...,vn) to rcv via sr_out;
          nextstate receive;
      enddecision;
endprocess;
```

In state `deliver`, there will never be signals to consume because all signals are withhold from being consumed by means of `save es`. This means that the behaviour from this state may be delayed till any future time slice. The total lack of signals to consume does not preclude the process to proceed, because the only transition alternative is a spontaneous transition, i.e. it is not initiated by the consumption of a signal.

# 5 Closing remarks

In this section, we sum up what has been achieved. We also describe in broad outline what is anticipated to be achieved more, thus hinting at topics for future research. But, to begin with, we present an overview of related work.

## 5.1 Related work

In [17], 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 a system gives an interesting type of asynchronous dataflow networks, but the treatment of time in the proposal is however too sketchy to be used as a starting point for a semantics of the time related features of SDL. Besides, process creation is not covered. In [15], we present a process algebra model of asynchronous dataflow networks as a semantic foundation for SDL. The model is close to the concepts around which SDL has been set up. However, we are not able to cover process creation too.

An operational semantics for a subset of SDL, which covers timing, has been given in [21]. Many relevant details are worked out, but it is not quite clear whether time is treated satisfactory. This is largely due to the intricacy of the operational semantics. At the outset, we have also tried shortly to give an operational semantics for $\varphi^-$SDL, but we found that it is very difficult. Our experience is that the main motivations for the rules describing an operational semantics are unavoidably intuitive. This may already lead to an undesirable semantics in relatively simple cases. For example, working on $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$, we have seriously considered to have the premise $w \in [\mathsf{s}_\rho(x)]$ in the rule for time unit delay (see Table 11). However, this plausible option would render all delayed processes bisimilar to deadlock in the current time slice, i.e. $\sigma_{\mathsf{rel}}(x) = \underline{\delta}$ would hold.

It is likely that, if we had taken parallel composition with communication, we would have been able to use special processes, put in parallel with the other ones, instead of the counting process creation operator and the state operator. The approach to use special processes is followed in [35]. There, it is largely responsible for the exceptional intricacy of the resulting semantics, which, by the way, has kept various obvious errors unnoticed for a long time. Amongst other things for this reason, we have chosen to use the counting process creation operator and the state operator instead. Besides, the approach to use special processes brings along a lot of internal communication that is irrelevant from a semantic point of view. Of course, in case an ACP-style process algebra is used as the basis of the presented semantics, there is the option to use an abstraction operator to abstract from the internal communication (for abstraction in process algebra, see e.g. [9]). However, it seems far from easy to elaborate the addition of abstraction to $\mathrm{PA}^{\mathrm{psc}}_{\mathrm{drt}}$ or its adaptation to parallel composition with communication.

For a subset of SDL, called $\mu$SDL, both an operational semantics and a process algebra semantics has been given in [20]. The operational semantics of $\mu$SDL is related to the one presented in [21], but time is not treated. The basis of the corresponding process algebra semantics is a time free process algebra, essentially $\mu$CRL [22] extended with the state operator. Like in [35], special processes are used for channels and input queues although there is no need for that with the state operator at one's

disposal. Interesting is that first the intended meaning of the language constructs is laid down in an operational semantics so that it can be reasonably checked later whether the process algebra semantics reflects the intentions. However, the SDL facilities for storage, timing and process creation are not available at all in $\mu$SDL; and the facilities for communication are only partially available.

In [19], BSDL is proposed as a basis for the semantics of SDL. BSDL is developed from scratch, using Object-Z, but it does not seem to fit in very well with SDL. In [24], it is proposed to use Duration Calculus [30] to describe the meaning of the language constructs of SDL. Thus an interesting semantic link is made between SDL and Duration Calculus, but it seems a little bit odd to view the main semantic objects used, viz. traces, as phases of system behaviour, called state variables in Duration Calculus, of which the duration is the principal attribute.

## 5.2 Conclusions and future work

We have presented an extension of discrete time process algebra with relative timing and we have proposed a semantics for the core of SDL, using this extension to describe the meaning of the language constructs. The operational semantics and axiom system of this ACP-style process algebra facilitates advances in the areas of validation of SDL specifications and verification of design steps made using SDL, respectively. At present, we focus on validation. We do so because practically useful advanced tools and techniques are within reach now while there is a tremendous need for them.

For validation purposes, we have to transform $\varphi^-$SDL specifications to transition systems in accordance with the process algebra semantics. Generating transition systems from finite linear recursive specifications in $\mathrm{BPA}_{\mathrm{drt}}^{\mathrm{psc}}$ (see Section 2.4) is straightforward. In Section 4, the meaning of a $\varphi^-$SDL process definition is described by a finite guarded recursive specification in $\mathrm{BPA}_{\mathrm{drt}}^{\mathrm{psc}}$ that can definitely be brought into linear form. The meaning of a $\varphi^-$SDL system definition is given in terms of the meaning of the occurring process definitions using the parallel composition operator, the counting process creation operator and the state operator. An obvious direction is to devise, for each of these operators on processes, a corresponding syntactic operator on finite linear recursive specifications that, under certain conditions, yields a linear recursive specification of the process that results from applying the operator on processes to the process(es) defined by the recursive specification(s) to be operated on. Of course, we look for non-restrictive conditions, but finiteness of the state space, a finite upper bound on the total number of process creations and a finite upper bound on the number of signals per time slice originating from the environment are inescapable. It goes without saying that we have to show the correctness of these syntactic operators. For that purpose, we have available the axioms presented in Section 2 and RSP (see Section 2.4).

In [12], timed frames, which are closely related to the kind of transition systems presented by the operational semantics of, for example, $\mathrm{BPA}_{\mathrm{drt}}^-$-ID and $\mathrm{PA}_{\mathrm{drt}}^-$-ID, are studied in a general algebraic setting and results concerning the connection between timed frames and discrete time processes with relative timing are given. In [16], a model for $\mathrm{BPA}_{\mathrm{drt}}^-$-IDlin ($\mathrm{BPA}_{\mathrm{drt}}^-$-ID with finite linear recursive specifications) is presented that gives an interpretation of its constants and operators on timed frames; and it is shown that the bisimulation model induced by the original structured operational

semantics is isomorphic to the timed frame model. It is plausible that these results can be extended to $\mathrm{BPA}^{\mathrm{psc}}_{\mathrm{drt}}\mathrm{lin}$ – timed frames support propositional signals. This would mean that we can transform $\varphi^-$SDL specifications to timed frames. In that case, we can basically check whether a system described in $\varphi^-$SDL satisfies a property expressed in TFL [13], an expressive first-order logic proposed for timed frames. We are considering to look for a fragment of TFL that is suitable to serve as a logic for $\varphi^-$SDL and to adapt an existing model checker to $\varphi^-$SDL and this logic – and thus to automate the checking. In particular, the model checker MEC [3] seems suited for this purpose – at least for small-scale systems. A fragment of Duration Calculus may be considered as well, since in [23] validity for Duration Calculus formulas in timed frames is defined.

The extension of discrete time process algebra with relative timing, used to describe the meaning of the language constructs of $\varphi^-$SDL, is fairly large and rather intricate. Theoretically interesting general properties, such as elimination, conservativity, completeness, etc. have yet to be established. We think that we are near the limit of what can be made provably free from defect. Still, owing to the nontrivial state space taken for the state operator, the presented semantics for $\varphi^-$SDL uses an excursion outside the realm of process algebra that is not negligible. We wish to have abstraction added to the process algebra used in order to provide a more abstract semantics for $\varphi^-$SDL, but right now we consider the consequences of this addition too difficult to grasp. All this suggests the option to develop a special process algebra that is closer to the concepts around which SDL has been set up. Of course, there is also the alternative to simplify SDL by removing SDL features that introduce semantic complexities but do not serve any practical purpose. The presented semantics of $\varphi^-$SDL may assist in identifying such cases.

# References


[1] L. Aceto, W.J. Fokkink, and C. Verhoef. Structural operational semantics. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*. Elsevier, 2000. Chapter 2.1 in this issue.

[2] B. Algayres, Y. Lejeune, F. Hugonnet, and F. Hantz. The AVALON project: A validation enviroment for SDL/MSC descriptions. In O. Færgemand and A. Sarma, editors, *SDL '93: Using Objects*, pages 221–235. North-Holland, 1993.

[3] A. Arnold. MEC, a system for constructing and analysing transition systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, pages 117–132. LNCS 407, Springer-Verlag, 1990.

[4] 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 P9208b, Programming Research Group, University of Amsterdam.

[5] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra. *Formal Aspects of Computing*, 8:188–208, 1996.

[6] J.C.M. Baeten and J.A. Bergstra. Process algebra with propositional signals. *Theoretical Computer Science*, 177:381–405, 1997.

[7] J.C.M. Baeten and C.A. Middelburg. Process algebra with timing: Real time and discrete time. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*. Elsevier, 2000. Chapter 4.1 in this issue.

[8] J.C.M. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume IV*, pages 149–268. Oxford University Press, 1995.

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

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

[11] 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.

[12] J.A. Bergstra, W.J. Fokkink, and C.A. Middelburg. Algebra of timed frames. *International Journal of Computer Mathematics*, 61:227–255, 1996.

[13] 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.

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

[15] J.A. Bergstra, C.A. Middelburg, and R. Şoricuţ. Discrete time network algebra for a semantic foundation of SDL. Research Report 98, United Nations University, International Institute for Software Technology, October 1997.

[16] J.A. Bergstra, C.A. Middelburg, and B. Warinschi. Timed frame models for discrete time process algebras. Research Report 100, United Nations University, International Institute for Software Technology, October 1997.

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

[18] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL, Formal Object-oriented Language for Communicating Systems*. Prentice-Hall, 1997.

[19] J. Fischer, S. Lau, and A. Prinz. A short note about BSDL. *SDL Newsletter*, 18:15–21, 1995.

[20] 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), 1993. Full version: Report TFL RR 1992-4, Tele Danmark Research.

[21] 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.

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

[23] C.A. Middelburg. Truth of duration calculus formulae in timed frames. *Fundamenta Informaticae*, 36:235–263, 1998.

[24] S. Mørk, J.C. Godskesen, M.R. Hansen, and R. Sharp. A timed semantics for SDL. In R. Gotzhein and J. Bredereke, editors, *FORTE IX: Theory, Application and Tools*, pages 295–309. Chapman & Hall, 1997.

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

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

[27] M.A. Reniers and J.J. Vereijken. Completeness in discrete-time process algebra. Computing Science Report 96-15, Eindhoven University of Technology, Department of Mathematics and Computing Science, 1996.

[28] F.W. Vaandrager. *Algebraic Techniques for Concurrency and their Applications*. PhD thesis, Centre for Mathematics and Computer Science, Amsterdam, 1989.

[29] C. Verhoef. A congruence theorem for structured operational semantics with predicates and negative premises. *Nordic Journal of Computing*, 2:274–302, 1995.

[30] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A calculus of durations. *Information Processing Letters*, 40:269–276, 1991.

[31] Methods for Testing and Specification (MTS); use of SDL in European telecommunications standards rules for testability and facilitating validation. ETSI Document ETS 300 414, 1995.

[32] Specification of Abstract Syntax Notation One (ASN.1). ITU-T Recommendation X.208, 1988.

[33] Specification and Description Language (SDL). ITU-T Recommendation Z.100, 1994.

[34] Specification and Description Language (SDL) – SDL formal definition: Static semantics. ITU-T Recommendation Z.100 F2, 1994. Annex F.2 to Recommendation Z.100.

[35] Specification and Description Language (SDL) – SDL formal definition: Dynamic semantics. ITU-T Recommendation Z.100 F3, 1994. Annex F.3 to Recommendation Z.100.

[36] SDL combined with ASN.1 (SDL/ASN.1). ITU-T Recommendation Z.105, 1995.

[37] SDT user's guide. TeleLOGIC AB, Sweden, 1992.

# A Notational conventions

**Meta-language for syntax:**

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

**Meta-language for semantics:**

The semantics of $\varphi^{-}$SDL is described by means of a set of equations recursively defining interpretation functions for all syntactic categories. For each syntactic category, the corresponding interpretation function gives a meaning to each language construct $c$ belonging to the category. We use the notation $[\![c]\!]$ or $[\![c]\!]^{\kappa}$ for applications of all interpretation functions. The exact interpretation function is always clear from the context. If contextual information $\kappa$ is needed for the interpretation, it is provided by an additional argument and the notation $[\![c]\!]^{\kappa}$ is used.

**Special action, condition and proposition notation:**

We write $a : D_1 \times \ldots \times D_n$ to indicate that $a$ is an action parametrized by $D_1 \times \ldots \times D_n$. This means that there is an action, referred to as $a(d_1, \ldots, d_n)$, for each $(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$. For atomic conditions and propositions, we use analogous notations.

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

We write $\mathcal{P}(A)$ for the set of all subsets of $A$, and we write $\mathcal{P}_{\text{fin}}(A)$ for the set of all finite subsets of $A$. We use abbreviations $\mathbb{N}_1$ and $\mathbb{N}_2$ for the sets $\mathbb{N} \setminus \{0\}$ and $\mathbb{N} \setminus \{0, 1\}$, respectively.

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

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

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

Functions are also used to present sequences; as usual we write $\langle x_1, \ldots, x_n \rangle$ for the sequence presented by the function $\{1 \mapsto x_1, \ldots, n \mapsto x_n\}$. The 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 language construct of $\varphi^-$SDL 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 $\varphi^-$SDL specification is also used to define the state space used to describe its meaning. The context of a language construct contains all names introduced by the definitions of variables, signal types, signal routes and process types occurring in the specification on hand and additionally:

- if the language construct occurs in the scope of a process definition, the name introduced by that process definition, called the *scope unit*;
- if the language 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*.

The names introduced by the definitions 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.

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

where

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

For language constructs that do not occur in a process definition, the absence of a scope unit will be represented by nil and, for language constructs that do not occur in a state definition, the absence of a save set will be represented by $\emptyset$. We write $\mathit{vards}(\kappa)$, $\mathit{sigds}(\kappa)$, $\mathit{routeds}(\kappa)$, $\mathit{procds}(\kappa)$, $\mathit{scopeunit}(\kappa)$ and $\mathit{saveset}(\kappa)$, where $\kappa = (V, S, R, P, X, ss) \in \mathit{Context}$, for $V$, $S$, $R$, $P$, $X$ and $ss$, respectively. We write $\mathit{vars}(\kappa)$ for $\{v \mid \exists T \bullet (v, T) \in \mathit{vards}(\kappa)\}$. The abbreviations $\mathit{sigs}(\kappa)$ and $\mathit{procs}(\kappa)$ are used analogously.

We make use of the following functions on *Context*:

$$\begin{array}{ll} \mathit{rcv} & : \mathit{Context} \times \mathit{RouteId} \to \mathit{ProcId} \cup \{\mathbf{env}\}, \\ \mathit{fpars} & : \mathit{Context} \times \mathit{ProcId} \to \mathit{VarId}^*, \\ \mathit{init} & : \mathit{Context} \times \mathit{ProcId} \to \mathbb{N}, \\ \mathit{updscopeunit} & : \mathit{Context} \times \mathit{ProcId} \to \mathit{Context}, \\ \mathit{updsaveset} & : \mathit{Context} \times \mathcal{P}_{\mathrm{fin}}(\mathit{SigId}) \to \mathit{Context}, \\ \mathit{envsigd} & : \mathit{Context} \to \mathcal{P}_{\mathrm{fin}}(\mathit{SigD} \times \mathit{ProcId}) \end{array}$$

The first three functions are partial functions, but they will only be applied in cases where the result is defined. The function *rcv* is used to extract the receiver type of a given signal route from the context. This function is inductively defined by

$$(r, X_1, X_2, ss) \in routeds(\kappa) \;\Rightarrow\; rcv(\kappa, r) = X_2$$

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

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

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

$$\kappa = (V, S, R, P, X, ss) \;\Rightarrow\; updscopeunit(\kappa, X') = (V, S, R, P, X', ss),$$
$$\kappa = (V, S, R, P, X, ss) \;\Rightarrow\; updsaveset(\kappa, ss') = (V, S, R, P, X, ss')$$

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

$$s \in ss \wedge (s, \langle T_1, \ldots, T_n \rangle) \in sigds(\kappa) \wedge (r, \textbf{env}, X_2, ss) \in routeds(\kappa) \;\Rightarrow$$
$$((s, \langle T_1, \ldots, T_n \rangle), X_2) \in envsigd(\kappa)$$

The context ascribed to a complete $\varphi^-$SDL specification is a minimal context in the sense that the contextual information available in it is common to all contexts on which language constructs occurring in it depend. The additional information that may be available applies to the scope unit for language constructs occurring in a process definition and the save set for language constructs occurring in a state definition. The context ascribed to a complete specification 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 $\emptyset$, respectively.

$$\{\![\textbf{system}\ S\textbf{;}D_1 \ldots D_n\ \textbf{endsystem;}]\!\} :=$$
$$(vards(\{\![D_1]\!\}) \cup \ldots \cup vards(\{\![D_n]\!\}),$$
$$sigds(\{\![D_1]\!\}) \cup \ldots \cup sigds(\{\![D_n]\!\}),$$
$$routeds(\{\![D_1]\!\}) \cup \ldots \cup routeds(\{\![D_n]\!\}),$$
$$procds(\{\![D_1]\!\}) \cup \ldots \cup procds(\{\![D_n]\!\}),$$
$$\mathsf{nil}, \emptyset)$$

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

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

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

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