1992

W.J. Fokkink

A simple specification language combining processes, time and data

Computer Science/Department of Software Technology Report CS-R9209 February

A Simple Specification Language combining Processes, Time and Data

Willem Jan Fokkink
Department of Software Technology, CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

Groote and Ponse have proposed a simple specification language based on CRL (*Common Representation Language*) in [5]. This language, called μCRL, combines processes and data and contains only essential constructs. In this paper μCRL is extended with time, resulting in the language $r\mu$CRL.

Mathematics Subject Classification: 68N99.
CR Categories: D.2.1, D.3.1, D.4.1.
Key Words & Phrases: Specification Language, Real Time, Process Algebra.
Note: The author is partially supported by ESPRIT Basic Research Action 3006 (CONCUR).

1 Introduction

Groote and Ponse defined in [5] the language μCRL (*micro* CRL, where CRL stands for Common Representation Language [15]). The language contains only the core constructs present in languages such as CRL [15], PSF [13] and LOTOS [10].

In this paper μCRL is equipped with the feature real time and will be called $r\mu$CRL (*real-time* μCRL). It consists of data, time and processes. The data are defined by declaring sorts and functions working upon these sorts. The meaning of these functions is described by equations. The process part is given in the syntax of ACP [3, 2]. It consists of a set of actions that may be parameterised by data and are provided with a time stamp. There are sequential, alternative and parallel composition, a shift operator and recursive processes. Our time domain consists of the floating-point numbers, so infinite processes do not exist in $r\mu$CRL.

Several authors have given a real-time extension of a process algebra. We use the one for ACP that was given by Baeten and Bergstra in [2]. They have introduced the notion of integration, which expresses the possibility of an action occurring somewhere within a dense interval. Klusener has modified the semantics and proved completeness in [12].

The structure of this paper is on purpose very similar to the paper in which Groote and Ponse have introduced μCRL. Parts of the contents of [5] have simply been copied. Furthermore, many elements of [2] in the style of [12] have been added.

Report CS-R9209
ISSN 0169-118X
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Acknowledgements. Jan Friso Groote and Alban Ponse defined the language μCRL. The author would like to thank Jan Bergstra, the initiator of μCRL, Jos Baeten, Steven Klusener and Alban Ponse for useful comments on draft versions of this paper.

2 The syntax of real-time μCRL

In this section the syntax of $r\mu$CRL is presented. We use $\equiv$ to denote syntactic equivalence.

2.1 Names

We assume the existence of a set $\mathcal{N}$ of *names*, i.e. sequences over an alphabet not containing

$$\bot, +, \|, \,\underline{\|}\,, \mid, \gg, \triangleleft, \triangleright, \cdot, \delta, \tau, \partial, \rho, \mathcal{I}, \Sigma, \surd, \times, \rightarrow, :, =, -, \in, \leq, <, (,), \{, \}, `,',$$

a space and a newline.

Moreover, $\mathcal{N}$ does not contain the reserved keywords **sort**, **proc**, **var**, **act**, **func**, **comm**, **rew** and **from**.

2.2 Lists

In the sequel *X-nelist*, $\times$-*X-nelist* and *space-X-nelist* (where nelist denotes non-empty list) for any syntactic category X are defined by

$$\begin{aligned} X\text{-}nelist &::= X \mid X\text{-}nelist, X \\ \times\text{-}X\text{-}nelist &::= X \mid \times\text{-}X\text{-}nelist \times X \\ space\text{-}X\text{-}nelist &::= X \mid space\text{-}X\text{-}nelist\ X \end{aligned}$$

Let λ denote the empty list. A *list* is a non-empty list or empty:

$$\begin{aligned} X\text{-}list &::= X\text{-}nelist \mid \lambda \\ \times\text{-}X\text{-}list &::= \times\text{-}X\text{-}nelist \mid \lambda \\ space\text{-}X\text{-}list &::= space\text{-}X\text{-}nelist \mid \lambda \end{aligned}$$

Non-empty lists are often described by the (informal) use of dots, e.g. $b_1 \times ... \times b_m$ with $m \geq 1$ is a $\times$-*X-nelist* where $b_1, ..., b_m$ are expressions in the syntactic category X.

2.3 Sort specifications

A *sort-specification* consists of a list of *names* representing sorts, preceded by the keyword **sort**.

$$sort\text{-}specification \ ::= \ \textbf{sort}\ space\text{-}name\text{-}list$$

2.4 Function specifications

A *function-specification* consists of a list of function declarations. A *function-declaration* consists of a non-empty list of constant and function names, the list of sorts of their parameters and their target sort:

function-specification ::= **func** *space-function-declaration-list*

function-declaration ::= *name-nelist* : ×*-name-list* ⟶ *name*

2.5 The standard sorts Bool, Real and Time

In every sort specification the *sort-declarations* **Bool**, **Real** and **Time** must be included. **Bool** contains the booleans, while **Time** is used to declare time as an ordered set and **Real** is an auxiliary sort to define multiplication. Every *function-specification* has to contain the following *function-declarations*:

$$\begin{aligned} T, F &: \rightarrow \mathbf{Bool} \\ add, subt &: \mathbf{Time} \times \mathbf{Time} \rightarrow \mathbf{Time} \\ mult &: \mathbf{Real} \times \mathbf{Time} \rightarrow \mathbf{Time} \\ leq &: \mathbf{Time} \times \mathbf{Time} \rightarrow \mathbf{Bool} \end{aligned}$$

Definition 2.1

- *A real-number is of the form* $.a_1 ... a_k Eb$ *with* $1 \leq k \leq p$, $a_i \in \{0, 1, ..., 9\}$ *for* $1 \leq i \leq k$, $a_1, a_k \not\equiv 0$ *and* $b \in \{E_{min}, ..., E_{max}\}$ *(where* p, E_{min} *and* E_{max} *are integer parameters),*
- *A time-number is* 0, ∞ *or of the form* aT *with* a *a real-number*

The *real-numbers* and *time-numbers* are the constants belonging to the sorts **Real** and **Time** respectively. The *function-declarations* a :⟶ **Real** for a a *real-number* and c :⟶ **Time** for c a *time-number* are considered standard and do not have to be included in the *function-declaration-list*.

The functions *add*, *subt* and *mult* describe addition, subtraction and multiplication of real numbers, while *leq* denotes the 'less-or-equal' relation. We have $subt(s_0, s_1) = 0$ iff $leq(s_0, s_1) = T$. The rewrite rules needed to describe these functions are considered standard and do not have to be included in the *rewrite-rules-section*.

Let s_0, s_1, s_2 be *time-terms*. Note that in general equations like $subt(add(s_0, s_1), s_2) = add(subt(s_0, s_2), s_1)$ and even $add(add(s_0, s_1), s_2) = add(add(s_0, s_2), s_1)$ do not hold. Detailed information about the floating-point arithmetic can be found in [9].

2.6 Rewrite specifications

A *rewrite-specification* is given by a many-sorted term rewriting system.

rewrite-specification ::= *variable-declaration-section*
rewrite-rules-section

The sorts of the variables that are used in a *rewrite-rules-section* must be declared in a *variable-declaration-section*.

variable-declaration-section ::= **var** *space-variable-declaration-list*

In a *variable-declaration*, the *name-nelist* contains the declared variables and the *name* denotes their sort:

variable-declaration ::= *name-nelist* : *name*

Data-terms are defined in the standard way. The *name* without brackets represents a variable or a constant.

data-term ::= *name*
| *name*(*data-term-nelist*)

The notion of a *time-term* originates from [12], where it is called a bound. The *name* represents a time-variable or a constant.

time-term ::= *name*
| *add(time-term,time-term)*
| *subt(time-term,time-term)*
| *mult(real-number,time-term)*

The meaning of functions operating on data is defined by a *rewrite-rules-section*.

rewrite-rules-section ::= **rew** *space-rewrite-rule-list*
rewrite-rule ::= *name* = *data-term*
| *name*(*data-term-nelist*) = *data-term*

2.7 Process expressions and process specifications

We define what *process-expressions* look like, explicitly taking care of the precedence among operators.

process-expression ::= *parallel-expression*
| *parallel-expression* + *process-expression*

parallel-expression ::= *merge-parallel-expression*
| *comm-parallel-expression*
| *cond-expression*
| *cond-expression* $\|\!\lfloor$ *cond-expression*
| *time-term* $\gg$ *cond-expression*

merge-parallel-expression	::=	*cond-expression* $\|$ *merge-parallel-expression*
	$\mid$	*cond-expression* $\|$ *cond-expression*
comm-parallel-expression	::=	*cond-expression* $\mid$ *comm-parallel-expression*
	$\mid$	*cond-expression* $\mid$ *cond-expression*
cond-expression	::=	*dot-expression*
	$\mid$	*dot-expression* $\lhd$ *data-term* $\rhd$ *dot-expression*
dot-expression	::=	*basic-expression*
	$\mid$	*basic-expression* $\cdot$ *dot-expression*
basic-expression	::=	$\mathcal{I}$(*interval-declaration*, *process-expression*)
	$\mid$	∂({*name-nelist*}, *process-expression*)
	$\mid$	τ({*name-nelist*}, *process-expression*)
	$\mid$	ρ({*renaming-declaration-nelist*}, *process-expression*)
	$\mid$	Σ(*single-variable-declaration*, *process-expression*)
	$\mid$	*name*
	$\mid$	*name*(*data-term-nelist*)
	$\mid$	*name*(*data-term-nelist*)(*time-term*)
	$\mid$	δ(*time-term*)
	$\mid$	τ(*time-term*)
	$\mid$	(*process-expression*)

The $+$ stands for alternative and the $\cdot$ for sequential composition.

The merge $\|$ interleaves the behaviour of both arguments, except that some actions in the arguments may communicate. The left merge $\lfloor\!\lfloor$ and the communication merge $\mid$ behave exactly as the parallel operator, except that for the left merge its first step must originate from its left argument, while for the communication merge the first action must be a communication between both components.

The $\gg$ is the (absolute) time-shift, that has been introduced in [2]. A *parallel-expression* $s \gg p$ denotes the process p starting at time s. This means that all actions that have to be performed at or before time s turn into deadlocks, because their execution has been delayed too long.

The conditional construct *dot-expression* $\lhd$ *data-term* $\rhd$ *dot-expression* is an alternative way to write an **if - then - else**-expression and is introduced by HOARE cs. [8] (see also [1]). The *data-term* is supposed to be of the standard sort of the Booleans. The $\lhd$-part is executed if the *data-term* evaluates to true and the $\rhd$-part is executed if the *data-term* evaluates to false.

The integral $\mathcal{I}$ denotes the alternative composition over a (finite) set of *time-numbers*. It was introduced in [2]. We use the notion of a prefixed integral, which originates from [12].

An *interval-declaration* is defined by:

$$\textit{interval-declaration} ::= \textit{name} \in \lhd\textit{time-term},\textit{time-term}\rhd \qquad (\lhd \in \{(,[\},\ \rhd \in \{),]\})$$

The *name* in the *interval-declaration* denotes a variable and its scope is the process mentioned in the second part of the integral.

The constant δ describes a deadlock, while the constant τ deprives actions of their identity, but not of their visibility.

The encapsulation operator ∂ and the hiding operator τ rename actions of which the *name* is mentioned in the first part of the argument into δ and τ respectively. The renaming operator ρ is more general. It renames the *names* of actions according to the scheme in its first argument. A *renaming-declaration* is given by

$$\textit{renaming-declaration} \quad ::= \quad \textit{name} \rightarrow \textit{name}$$

The sum operator is used to declare a variable of a specific sort for use in a *process-expression.* A *single-variable-declaration* is defined by

$$\textit{single-variable-declaration} ::= \textit{name} : \textit{name}$$

The behaviour of this construct is a choice between the behaviours of the *process-expressions* that result from substituting a value of the sort of the variable for the variable.

The construct *name* refers to a declared process, *name*(*data-term-nelist*) refers to a declared process or denotes an action (in which case the *data-term-nelist* consists of a single *time-term*) and *name*(*data-term-nelist*)(*time-term*) denotes an action.

The syntax of *process-expressions* says that $\cdot$ binds strongest, the conditional construct binds stronger than the parallel and shift operators, which in turn bind stronger than $+$.

A *process-specification* consists of a list of (parameterised) names, which are used as process identifiers, that are declared together with their bodies.

process-specification	::=	**proc** *space-process-declaration-list*
process-declaration	::=	*name*(*name*) = *process-expression*
	|	*name*(*single-variable-declaration-nelist*)(*name*) = *process-expression*

2.8 Action specification

In an *action-specification* all actions that are used are declared. If an action is parameterised by data, then we must declare on which sorts an action depends.

action-specification	::=	**act** *space-action-declaration-list*
action-declaration	::=	*name-nelist*
	|	*name-nelist* : $\times$-*name-nelist*

2.9 Communication specification

A *communication-specification* prescribes how actions may communicate. If it is specified that $in \mid out = comm$, then each action $in(t_1, ..., t_k)(c)$ can communicate with $out(t'_1, ..., t'_m)(c')$ to $comm(t_1, ..., t_k)(c)$, provided $k = m$ and t_i and t'_i denote the same data-element for $i = 1, ..., k$ and c and c' denote the same *time-number*.

communication-specification	::=	**comm** *space-communication-declaration-list*
communication-declaration	::=	*name* | *name* = *name*

2.10 Specifications

specification	::=	*sort-specification*
	|	*function-specification*
	|	*rewrite-specification*
	|	*action-specification*
	|	*communication-specification*
	|	*process-specification*
	|	*specification specification*

2.11 Some notations

In our syntax we allow a *real-number* $.a_1...a_k El$ to be abbreviated to $a_1...a_l.a_{l+1}...a_k$ if $0 \leq l < k$ and to $a_1...a_k$ if $k = l$.

We now give some definitions concerning *time-terms*. Let s_0, s_1 be *time-terms* and a a *real-number*. In this paper we use the following standard notations:

$$
\begin{array}{rll}
s_0 + s_1 & \text{denotes} & add(s_0, s_1), \\
s_0 \dot{-} s_1 & \text{denotes} & subt(s_0, s_1), \\
a \cdot s_0 & \text{denotes} & mult(a, s_0), \\
s_0 \leq s_1 & \text{denotes} & leq(s_0, s_1) = T, \\
s_1 < s_0 & \text{denotes} & leq(s_0, s_1) = F.
\end{array}
$$

Definition 2.2 *Let V be a set of time-numbers.*

- *$max(V)$ denotes the* maximum *of V, that is the time-number $c \in V$ such that $c' \leq c$ for all $c' \in V$.*

- *$min(V)$ denotes the* minimum *of V, that is the time-number $c \in V$ such that $c \leq c'$ for all $c' \in V$.*

We put $max(\emptyset) = min(\emptyset) = 0$.

Definition 2.3 *Let c, c_0, c_1 be time-numbers. We say that*

- *c is in the interval $\langle c_0, c_1 \rangle$ iff $c_0 < c$ and $c < c_1$,*

- *c is in the interval $\langle c_0, c_1]$ iff $c_0 < c$ and $c \leq c_1$,*
- *c is in the interval $[c_0, c_1\rangle$ iff $c_0 \leq c$ and $c < c_1$,*
- *c is in the interval $[c_0, c_1]$ iff $c_0 \leq c$ and $c \leq c_1$.*

2.12 The from construct

For a *process-expression* or a *data-term* t, we write t **from** E for a *specification* E where we mean the *process-expression* or *data-term* t as defined in E. Often, it is clear from the context to which *specification* E the item t belongs. In this case we generally write t without explicit reference to E.

3 Static semantics

Not every *specification* is necessarily correctly defined. In this section we define under which circumstances a *specification* has a correct *static semantics.* Furthermore, we define some functions that will be used in the definition of the semantics of $r\mu$CRL.

3.1 The signature of a specification

The *signature* of a specification consists of a five-tuple. Each component is a set, containing all elements of a main syntactic category declared in a *specification* E.

Definition 3.1 *Let E be a specification. The* signature $Sig(E) = (Sort, Fun, Act, Comm, Proc)$ *of E is defined as follows:*

- *If $E \equiv$ **sort** $n_1 \ldots n_m$ with $m \geq 1$, then $Sig(E) \stackrel{\text{def}}{=} (\{n_1, \ldots, n_m\}, \emptyset, \emptyset, \emptyset, \emptyset)$.*
- *If $E \equiv$ **func** $fd_1 \ldots fd_m$ with $m \geq 1$, then $Sig(E) \stackrel{\text{def}}{=} (\emptyset, Fun, \emptyset, \emptyset, \emptyset)$, where*

$$\begin{aligned} Fun \stackrel{\text{def}}{=} & \{n_{ij} :\rightarrow S_i \mid fd_i \equiv n_{i1}, \ldots, n_{il_i} :\rightarrow S_i, 1 \leq i \leq m, 1 \leq j \leq l_i\} \\ \cup & \{n_{ij} : S_{i1} \times \ldots \times S_{ik_i} \rightarrow S_i \mid \\ & \quad fd_i \equiv n_{i1}, \ldots, n_{il_i} : S_{i1} \times \ldots \times S_{ik_i} \rightarrow S_i, 1 \leq i \leq m, 1 \leq j \leq l_i\}. \end{aligned}$$

- *If E is a rewrite-specification, then $Sig(E) \stackrel{\text{def}}{=} (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$.*
- *If $E \equiv$ **act** $ad_1 \ldots ad_m$ with $m \geq 1$, then $Sig(E) \stackrel{\text{def}}{=} (\emptyset, \emptyset, Act, \emptyset, \emptyset)$, where*

$$\begin{aligned} Act \stackrel{\text{def}}{=} & \{n_{ij} \mid ad_i \equiv n_{i1}, \ldots, n_{il_i}, 1 \leq i \leq m, 1 \leq j \leq l_i\} \\ \cup & \{n_{ij} : S_{i1} \times \ldots \times S_{ik_i} \mid \\ & \quad ad_i \equiv n_{i1}, \ldots, n_{il_i} : S_{i1} \times \ldots \times S_{ik_i}, 1 \leq i \leq m, 1 \leq j \leq l_i\}. \end{aligned}$$

- *If $E \equiv$ **comm** $cd_1 \ldots cd_m$ with $m \geq 1$, then $Sig(E) \stackrel{\text{def}}{=} (\emptyset, \emptyset, \emptyset, \{cd_i \mid 1 \leq i \leq m\}, \emptyset)$.*
- *If $E \equiv$ **proc** $pd_1 \ldots pd_m$ with $m \geq 1$, then $Sig(E) \stackrel{\text{def}}{=} (\emptyset, \emptyset, \emptyset, \emptyset, \{pd_i \mid 1 \leq i \leq m\})$.*

- *If* $E \equiv E_1\ E_2$ *with* $Sig(E_i) = (Sort_i, Fun_i, Act_i, Comm_i, Proc_i)$ *for* $i = 1, 2$, *then*

  $Sig(E) \stackrel{\text{def}}{=} (Sort_1 \cup Sort_2, Fun_1 \cup Fun_2, Act_1 \cup Act_2, Comm_1 \cup Comm_2, Proc_1 \cup Proc_2)$.

Definition 3.2 *Let* $Sig = (Sort, Fun, Act, Comm, Proc)$ *be a signature. We write*

Sig.Sort for Sort,
Sig.Fun for Fun,
Sig.Act for Act,
Sig.Comm for Comm,
Sig.Proc for Proc.

3.2 Variables

The next definition says which *names* can play the role of a variable without confusion with defined constants. Moreover, variables must have an unambiguous and declared sort.

Definition 3.3 *Let Sig be a signature. A set* $\mathcal{V}$ *containing elements* $\langle x : S\rangle$ *with* x *and* S *names, is called a set of variables over Sig iff for each* $\langle x : S\rangle \in \mathcal{V}$:

- *for each name* S' *it holds that* $x :\rightarrow S' \notin Sig.Fun$,
- $S \in Sig.Sort$,
- *for each name* S' *such that* $S' \not\equiv S$ *it holds that* $\langle x : S'\rangle \notin \mathcal{V}$.

Definition 3.4 *Let var-dec be a variable-declaration-section. The function Vars is defined by:*

$$Vars(var\text{-}dec) \stackrel{\text{def}}{=} \{\langle x_{ij} : S_i\rangle \mid 1 \le i \le m, 1 \le j \le l_i\}$$

if for some $m \ge 0$ $var\text{-}dec \equiv \textbf{var}\ x_{11}, \ldots, x_{1l_1} : S_1\ \ldots\ x_{m1}, \ldots, x_{ml_m} : S_m$.

In the following definitions we give functions yielding the sort and the variables in a *data-term* t. If for some reason no answer can be obtained, a $\perp$ results.

Definition 3.5 *Let* t *be a data-term and Sig a signature. Let* $\mathcal{V}$ *be a set of variables over Sig. We define:*

$$sort_{Sig,\mathcal{V}}(t) \stackrel{\text{def}}{=} \begin{cases} S & \text{if } t \equiv x \text{ and } \langle x : S\rangle \in \mathcal{V}, \\ S & \text{if } t \equiv n,\ n :\rightarrow S \in Sig.Fun \text{ and for no } S' \not\equiv S \\ & n :\rightarrow S' \in Sig.Fun, \\ S & \text{if } t \equiv n(t_1, \ldots, t_m), \\ & n : sort_{Sig,\mathcal{V}}(t_1) \times \ldots \times sort_{Sig,\mathcal{V}}(t_m) \rightarrow S \in Sig.Fun \text{ and for no} \\ & S' \not\equiv S\ n : sort_{Sig,\mathcal{V}}(t_1) \times \ldots \times sort_{Sig,\mathcal{V}}(t_m) \rightarrow S' \in Sig.Fun, \\ \perp & \text{otherwise.} \end{cases}$$

Definition 3.6 *Let Sig be a signature,* $\mathcal{V}$ *a set of variables over Sig and let* t *be a data-term.*

$$Var_{Sig,\mathcal{V}}(t) \stackrel{\text{def}}{=} \begin{cases} \{\langle x : S\rangle\} & \text{if } t \equiv x \text{ and } \langle x : S\rangle \in \mathcal{V}, \\ \emptyset & \text{if } t \equiv n \text{ and } n :\rightarrow S \in Sig.Fun, \\ \bigcup_{1 \le i \le m} Var_{Sig,\mathcal{V}}(t_i) & \text{if } t \equiv n(t_1, \ldots, t_m), \\ \{\perp\} & \text{otherwise.} \end{cases}$$

We call a *data-term* t *closed* w.r.t. a signature *Sig* iff $\text{Var}_{Sig,\emptyset}(t) = \emptyset$.

3.3 Static semantics

A *specification* must be internally consistent. In this section we define what are the conditions for a *specification* to be SSC (*Statically Semantically Correct*).

Definition 3.7 *Let Sig be a signature and $\mathcal{V}$ be a set of variables over Sig. We define the predicate 'is SSC w.r.t. Sig' inductively over the syntax of a specification.*

- *A specification* **sort** $n_1 \ldots n_m$ *with $m \geq 0$ is SSC w.r.t. Sig iff all names $n_1, \ldots, n_m$ are pairwise different.*

- *A specification* **func** $n_{11}, \ldots, n_{1l_1} : S_{11} \times \ldots \times S_{1k_1} \rightarrow S_1$

  $\vdots$

  $n_{m1}, \ldots, n_{ml_m} : S_{m1} \times \ldots \times S_{mk_m} \rightarrow S_m$

  with $m \geq 0$, $l_i \geq 1$, $k_i \geq 0$ for $1 \leq i \leq m$ is SSC w.r.t. Sig iff

  - *for all $1 \leq i \leq m$ the names $n_{i1}, \ldots, n_{il_i}$ are pairwise different,*
  - *for all $1 \leq i < j \leq m$ it holds that if $n_{ik} \equiv n_{jk'}$ for some $1 \leq k \leq il_i$ and $1 \leq k' \leq jl_j$, then either $k_i \neq k_j$, or $S_{il} \not\equiv S_{jl}$ for some $1 \leq l \leq k_i$,*
  - *for all $1 \leq i \leq m$ and $1 \leq j \leq k_i$ it holds that $S_{ij} \in Sig.Sort$ and $S_i \in Sig.Sort$.*

- *A specification of the form: var-dec*

  rew-rul

  where var-dec is a variable-declaration-section and rew-rul is a rewrite-rules-section is SSC w.r.t. Sig iff

  - *var-dec is SSC w.r.t. Sig,*
  - *rew-rul is SSC w.r.t. Sig and Vars(var-dec).*

⋆ *A variable-declaration-section* **var** $n_{11}, \ldots, n_{1k_1} : S_1$

  $\vdots$

  $n_{m1}, \ldots, n_{mk_m} : S_m$

  with $m \geq 0$, $k_i \geq 1$ for $1 \leq i \leq m$ is SSC w.r.t. Sig iff

  - $n_{ij} \not\equiv n_{i'j'}$ *whenever $i \neq i'$ or $j \neq j'$ for $1 \leq i \leq m$, $1 \leq i' \leq m$, $1 \leq j \leq k_i$ and $1 \leq j' \leq k_{i'}$,*
  - *the set Vars(*var $n_{11}, \ldots, n_{1k_1} : S_1 \ldots n_{m1}, \ldots, n_{mk_m} : S_m$*) is a set of variables over Sig.*

⋆ *A rewrite-rules-section* **rew** $rw_1 \ldots rw_m$ *with $m \geq 0$ is SSC w.r.t. Sig and $\mathcal{V}$ iff*

  - *if $rw_i \equiv n = t$ for some $1 \leq i \leq m$, then*
    - $n :\rightarrow sort_{Sig,\emptyset}(t) \in Sig.Fun$,
    - *t is SSC w.r.t. Sig and $\emptyset$,*
  - *if $rw_i \equiv n(t_1, \ldots, t_k) = t$ for some $1 \leq i \leq m$ and $k \geq 1$, then*

* $n : sort_{Sig,\mathcal{V}}(t_1) \times ... \times sort_{Sig,\mathcal{V}}(t_k) \rightarrow sort_{Sig,\mathcal{V}}(t) \in Sig.Fun$,
* t, t_j $(1 \leq j \leq k)$ are SSC w.r.t. Sig and $\mathcal{V}$,
* $Var_{Sig,\mathcal{V}}(t) \subseteq \bigcup_{1 \leq j \leq k_i} Var_{Sig,\mathcal{V}}(t_j)$.

⋆ *A data-term n with n a name is SSC w.r.t. Sig and $\mathcal{V}$ iff $\langle n : S\rangle \in \mathcal{V}$ for some S, or $n :\rightarrow sort_{Sig,\mathcal{V}}(n) \in Sig.Fun$,*

A data-term $n(t_1, ..., t_m)$ $(m \geq 1)$ is SSC w.r.t. Sig and $\mathcal{V}$ iff $n : sort_{Sig,\mathcal{V}}(t_1) \times ... \times sort_{Sig,\mathcal{V}}(t_m) \rightarrow sort_{Sig,\mathcal{V}}(n(t_1, ..., t_m)) \in Sig.Fun$ and $t_1, ..., t_m$ are SSC w.r.t. Sig and $\mathcal{V}$.

A time-term n with n a name is SSC w.r.t. Sig and $\mathcal{V}$ iff $\langle n : \textbf{Time}\rangle \in \mathcal{V}$ or n is a time-number,

The time-terms $add(s_0, s_1)$ and $subt(s_0, s_1)$ are SSC w.r.t. Sig and $\mathcal{V}$ iff s_0, s_1 are time-terms that are SSC w.r.t. Sig and $\mathcal{V}$,

A time-term $mult(a, s)$ is SSC w.r.t. Sig and $\mathcal{V}$ iff s is a time-term that is SSC w.r.t. Sig and $\mathcal{V}$ and a is a real-number,

• *A specification* **act** $ad_1 ... ad_m$ *with $m \geq 0$ is SSC w.r.t. Sig iff*

  – *for all $1 \leq i \leq m$ the action-declaration ad_i is SSC w.r.t. Sig,*
  – *for all $1 \leq i < j \leq m$ it holds that $Sig(\textbf{act}\ ad_i).Act \cap Sig(\textbf{act}\ ad_j).Act = \emptyset$.*

⋆ *An action-declaration $n_1, ..., n_m$ with $m \geq 1$ is SSC w.r.t. Sig iff*

  – *for all $1 \leq i < j \leq m$ it holds that $n_i \not\equiv n_j$,*
  – *for all $1 \leq i \leq m$ and for all process-expressions p it holds that $n_i = p \notin Sig.Proc$.*

An action-declaration $n_1, ..., n_m : S_1 \times ... \times S_k$ with $k, m \geq 1$ is SSC w.r.t. Sig iff

  – *for all $1 \leq i < j \leq m$ it holds that $n_i \not\equiv n_j$,*
  – *for all $1 \leq i \leq k$ it holds that $S_i \in Sig.Sort$,*
  – *for all $1 \leq i \leq m$ and for all names $x_1, ..., x_k$ and process-expressions p it holds that $n_i(x_1 : S_1, ..., x_k : S_k) = p \notin Sig.Proc$.*

• *A specification* **comm** $n_{11} | n_{12} = n_{13} ... n_{m1} | n_{m2} = n_{m3}$ *with $m \geq 0$ is SSC w.r.t. Sig iff*

  – *for each $1 \leq i < j \leq m$ it is not the case that $n_{i1} \equiv n_{j1}$ and $n_{i2} \equiv n_{j2}$, or $n_{i1} \equiv n_{j2}$ and $n_{i2} \equiv n_{j1}$,*
  – *for each $1 \leq i \leq m$ either $n_{i1} \in Sig.Act$ or there is a $k \geq 1$ such that $n_{i1} : S_1 \times ... \times S_k \in Sig.Act$,*
  – *for each $1 \leq i \leq m$, $k \geq 1$ and names $S_1, ..., S_k$ it holds that if $n_{i1} : S_1 \times ... \times S_k \in Sig.Act$ then $n_{i2} : S_1 \times ... \times S_k \in Sig.Act$ and $n_{i3} : S_1 \times ... \times S_k \in Sig.Act$,*
  – *for each $1 \leq i \leq m$ it holds that if $n_{i1} \in Sig.Act$ then $n_{i2} \in Sig.Act$ and $n_{i3} \in Sig.Act$.*

- *A specification* **proc** $pd_1 \dots pd_m$ *with* $m \geq 0$ *is SSC w.r.t. Sig iff*
  - *for each* $1 \leq i < j \leq m$:
    - * *if* $pd_i \equiv n_i = p_i$ *and* $pd_j \equiv n_j = p_j$ *then* $n_i \not\equiv n_j$,
    - * *if for some* $k \geq 1$ *it holds that* $pd_i \equiv n_i(x_1 : S_1, \dots, x_k : S_k) = p_i$ *and* $pd_j \equiv n_j(x'_1 : S_1, \dots, x'_k : S_k) = p_j$ *then* $n_i \not\equiv n_j$,
  - *if* $pd_i \equiv n_i = p_i$, *then* p_i *is SSC w.r.t. Sig and* $\emptyset$,
  - *if* $pd_i \equiv n_i(x_1 : S_1, \dots, x_k : S_k) = p_i$ $(k \geq 1)$, *then*
    - * $k > 1$ *or* $S_1 \not\equiv$ **Time** *or* $n_i \notin Sig.Act$,
    - * *the names* $x_1, \dots, x_k$ *are pairwise different and* $\{\langle x_j : S_j \rangle \mid 1 \leq j \leq k\}$ *is a set of variables over Sig*,
    - * p_i *is SSC w.r.t. Sig and* $\{\langle x_j : S_j \rangle \mid 1 \leq j \leq k\}$.

⋆ *A process-expression* $p_1 + p_2$, *a dot-expression* $p_1 \cdot p_2$ *and parallel-expressions* $p_1 \parallel p_2$, $p_1 ⌊⌊ p_2$, $p_1 \mid p_2$ *are SSC w.r.t. Sig and* $\mathcal{V}$ *iff* p_1 *and* p_2 *are SSC w.r.t. Sig and* $\mathcal{V}$.

A parallel-expression $s \gg p$ *is SSC w.r.t. Sig and* $\mathcal{V}$ *iff* p *and* s *are SSC w.r.t. Sig and* $\mathcal{V}$.

A cond-expression $p_1 \triangleleft t \triangleright p_2$ *is SSC w.r.t. Sig and* $\mathcal{V}$ *iff*

- p_1 *and* p_2 *are SSC w.r.t. Sig and* $\mathcal{V}$,
- t *is SSC w.r.t. Sig and* $\mathcal{V}$ *and* $sort_{Sig,\mathcal{V}}(t) =$ **Bool**.

The basic-expressions $\partial(\{n_1, \dots, n_m\}, p)$ *and* $\tau(\{n_1, \dots, n_m\}, p)$ *with* $m \geq 1$ *are SSC w.r.t. Sig and* $\mathcal{V}$ *iff*

- *for all* $1 \leq i < j \leq m$ $n_i \not\equiv n_j$,
- *for* $1 \leq i \leq m$ *either* $n_i \in Sig.Act$ *or* $n_i : S_1 \times \dots \times S_k \in Sig.Act$ *for some* $k \geq 1$ *and names* $S_1, \dots, S_k$,
- p *is SSC w.r.t. Sig and* $\mathcal{V}$.

The basic-expression $\rho(\{n_1 \to n'_1, \dots, n_m \to n'_m\}, p)$ *with* $m \geq 1$ *is SSC w.r.t. Sig and* $\mathcal{V}$ *iff*

- *for* $1 \leq i \leq m$ *either* $n_i \in Sig.Act$ *or* $n_i : S_1 \times \dots \times S_k \in Sig.Act$ *for some* $k \geq 1$ *and names* $S_1, \dots, S_k$,
- *for each* $1 \leq i < j \leq m$ *it holds that* $n_i \not\equiv n_j$,
- *for* $1 \leq i \leq m$, $k \geq 1$ *and names* $S_1, \dots, S_k$ *it holds that if* $n_i : S_1 \times \dots \times S_k \in Sig.Act$, *then also* $n'_i : S_1 \times \dots \times S_k \in Sig.Act$,
- *for* $1 \leq i \leq m$ *it holds that if* $n_i \in Sig.Act$, *then also* $n'_i \in Sig.Act$,
- p *is SSC w.r.t. Sig and* $\mathcal{V}$.

A basic-expression $\mathcal{I}(x \in V, p)$ *is SSC w.r.t. Sig and* $\mathcal{V}$ *iff*

- *for each name* S' *it holds that* $x :\to S' \notin Sig.Fun$,

- *the process-expression p is SSC w.r.t. Sig and* $(\mathcal{V}\backslash\{\langle x : S'\rangle \mid S' \text{ a name}\}) \cup \{\langle x : \textbf{Time}\rangle\}$,
- $V = \langle\!\langle s_0, s_1 \rangle\!\rangle$, *where the time-terms s_0 and s_1 are SSC w.r.t. Sig and* $\mathcal{V}$.

A basic-expression $\Sigma(x : S, p)$ *is SSC w.r.t. Sig and* $\mathcal{V}$ *iff*

- $S \in Sig.Sort$, $S \not\equiv \textbf{Time}$,
- p *is SSC w.r.t. Sig and* $\mathcal{V}\backslash\{\langle x : S'\rangle \mid S' \text{ a name}\} \cup \{\langle x : S\rangle\}$,
- *for each name S' it holds that* $x :\rightarrow S' \notin Sig.Fun$.

A basic-expression n is SSC w.r.t. Sig and $\mathcal{V}$ iff $n = p \in Sig.Proc$ *for some process-expression p.*

A basic-expression $n(t_1, ..., t_m)$ *with* $m \geq 1$ *is SSC w.r.t. Sig and* $\mathcal{V}$ *iff*

- $n(x_1 : sort_{Sig,\mathcal{V}}(t_1), ..., x_m : sort_{Sig,\mathcal{V}}(t_m)) = p \in Sig.Proc$ *for some names* $x_1, ..., x_m$ *and process-expression p and the data-terms $t_1, ..., t_m$ are SSC w.r.t. Sig and $\mathcal{V}$, or*
- $n \in Sig.Act$, $m = 1$ *and t_1 is a time-term that is SSC w.r.t. Sig and* $\mathcal{V}$.

A basic-expression $n(t_1, ..., t_m)(s)$ *is SSC w.r.t. Sig and* $\mathcal{V}$ *iff*

- $n : sort_{Sig,\mathcal{V}}(t_1) \times ... \times sort_{Sig,\mathcal{V}}(t_m) \in Sig.Act$,
- *the data-terms $t_1, ..., t_m$ and the time-term s are SSC w.r.t. Sig and* $\mathcal{V}$.

A basic-expression (p) is SSC w.r.t. Sig and $\mathcal{V}$ iff p is SSC w.r.t. Sig and $\mathcal{V}$.

- *A specification* $E_1\ E_2$ *is SSC w.r.t. Sig iff*
  - E_1 *and* E_2 *are SSC w.r.t. Sig,*
  - $Sig(E_1).Sort \cap Sig(E_2).Sort = \emptyset$,
  - *if* $n : S_1 \times ... \times S_m \rightarrow S \in Sig(E_1).Fun$ *for some* $m \geq 0$ *then* $n : S_1 \times ... \times S_m \rightarrow S' \notin Sig(E_2).Fun$ *for any name* S',
  - $Sig(E_1).Act \cap Sig(E_2).Act = \emptyset$,
  - *if* $n_1 \,|\, n_2 = n_3 \in Sig(E_1).Comm$ *then for any names* n_3' *and* n_3'' $n_1 \,|\, n_2 = n_3' \notin Sig(E_2).Comm$ *and* $n_2 \,|\, n_1 = n_3'' \notin Sig(E_2).Comm$,
  - *if* $pd_1 \in Sig(E_1).Proc$ *and* $pd_2 \in Sig(E_2).Proc$, *then*
    * *if* $pd_1 \equiv n_1 = p_1$ *and* $pd_2 \equiv n_2 = p_2$, *then* $n_1 \not\equiv n_2$,
    * *if* $pd_1 \equiv n_1(x_1 : S_1, ..., x_m : S_m) = p_1$ *and* $pd_2 \equiv n_2(x_1' : S_1, ..., x_m' : S_m) = p_2$ *for some* $m \geq 1$, *then* $n_1 \not\equiv n_2$.

Definition 3.8 *Let E be a specification. We say that E is SSC iff E is SSC w.r.t. $Sig(E)$.*

Lemma 3.9 *Let Sig be a signature and $\mathcal{V}$ be a set of variables over Sig. Let t be a data-term that is SSC w.r.t. Sig and $\mathcal{V}$. Then* $sort_{Sig,\mathcal{V}}(t) \neq \perp$ *and* $\perp \notin Var_{Sig,\mathcal{V}}(t)$.

3.4 The communication function

The following definition guarantees that the communication function is commutative and associative. This implies that the merge is also commutative and associative, which allows us to write parallel processes without brackets.

Definition 3.10 *Let Sig be a signature. The set Sig.Comm* is defined by:*

$$Sig.Comm^* \stackrel{\text{def}}{=} \{n_1 \mid n_2 = n_3,\ n_2 \mid n_1 = n_3 \mid n_1 \mid n_2 = n_3 \in Sig.Comm\}.$$

In Sig.Comm communication is always commutative. We say that a specification E is* communication-associative *iff*

$$n_1 \mid n_2 = n,\ n \mid n_3 = n' \in Sig(E).Comm^* \Rightarrow \\ \exists n'' :\ n_2 \mid n_3 = n'',\ n_1 \mid n'' = n' \in Sig(E).Comm^*.$$

With the condition that E is SSC this exactly implies that communication is associative.

4 Well-formed $r\mu$CRL specifications

We define what well-formed specifications are. Only well-formed *specifications* are provided with a semantics. Well-formedness is a decidable property.

Definition 4.1 *Let E be a specification that is SSC. We say that E has* no empty sorts *iff for all $S \in Sig(E).Sort$ there is a data-term t that is SSC w.r.t. $Sig(E)$ and $\emptyset$ such that $sort_{Sig(E),\emptyset}(t) \equiv S$.*

Definition 4.2 *Let E be a specification. E is called well-formed iff*

- *E is SSC,*
- *E is communication-associative,*
- *E has no empty sorts,*
- **Bool**, **Real**, **Time** $\in Sig(E).Sort$,
- $T, F :\to$ **Bool** $\in Sig(E).Fun$,
- *add*, *subt* : **Time** $\times$ **Time** $\to$ **Time** $\in Sig(E).Fun$,
- *mult* : **Real** $\times$ **Time** $\to$ **Time** $\in Sig(E).Fun$,
- *leq* : **Time** $\times$ **Time** $\to$ **Bool** $\in Sig(E).Fun$.

5 Algebraic semantics

In this section we present the semantics of well-formed $r\mu$CRL specifications.

5.1 Algebras

First we adapt the standard definitions of algebras etc. to $r\mu$CRL (see e.g. [4] for these definitions).

Definition 5.1 *Let E be a well-formed specification. A $Sig(E)$-*algebra *$\boldsymbol{A}$ is a structure containing*

- *for each $S \in Sig(E).Sort$ a non-empty domain $D(\boldsymbol{A}, S)$,*
- *for each $n :\rightarrow S \in Sig(E).Fun$ a constant $C(\boldsymbol{A}, n) \in D(\boldsymbol{A}, S)$,*
- *for each $n : S_1 \times ... \times S_m \rightarrow S \in Sig(E).Fun$ a function $F(\boldsymbol{A}, n : S_1 \times ... \times S_m)$ from $D(\boldsymbol{A}, S_1) \times ... \times D(\boldsymbol{A}, S_m)$ to $D(\boldsymbol{A}, S)$.*

For two elements $a_1 \in D(\boldsymbol{A}, S_1)$ and $a_2 \in D(\boldsymbol{A}, S_2)$, we write $a_1 = a_2$ iff $S_1 \equiv S_2$ and a_1 and a_2 represent exactly the same element.

Definition 5.2 *Let E be a well-formed specification and let $\boldsymbol{A}$ be a $Sig(E)$-algebra. We define the* interpretation *$[\![\cdot]\!]_{\boldsymbol{A}}$ from data-terms that are SSC w.r.t. $Sig(E)$ and $\emptyset$ into the domains of $\boldsymbol{A}$ as follows:*

- *if $t \equiv n$, then $[\![t]\!]_{\boldsymbol{A}} \stackrel{\text{def}}{=} C(\boldsymbol{A}, n)$,*
- *if $t \equiv n(t_1, ..., t_m)$ for some $m \geq 1$, then $[\![t]\!]_{\boldsymbol{A}} \stackrel{\text{def}}{=} F(\boldsymbol{A}, n : sort_{Sig(E),\emptyset}(t_1) \times ... \times sort_{Sig(E),\emptyset}(t_m))([\![t_1]\!]_{\boldsymbol{A}}, ..., [\![t_m]\!]_{\boldsymbol{A}})$.*

We say that a $Sig(E)$-algebra $\boldsymbol{A}$ is minimal *iff for each $a \in D(\boldsymbol{A}, S)$ and $S \in Sig(E).Sort$, there is some data-term t that is SSC w.r.t. $Sig(E)$ and $\emptyset$ such that $[\![t]\!]_{\boldsymbol{A}} = a$. For data-terms t_1, t_2 that are SSC w.r.t. $Sig(E)$ and $\emptyset$ we write $\boldsymbol{A} \models t_1 = t_2$ iff $[\![t_1]\!]_{\boldsymbol{A}} = [\![t_2]\!]_{\boldsymbol{A}}$.*

Definition 5.3 *Let E be a well-formed specification and let $\boldsymbol{A}$ be a minimal $Sig(E)$-algebra. A function r mapping pairs of a sort S and an element from $D(\boldsymbol{A}, S)$ to data-terms that are SSC w.r.t. to $Sig(E)$ and $\emptyset$ is called a* representation function *of E and $\boldsymbol{A}$ iff $\boldsymbol{A} \models t = r(sort_{Sig(E),\emptyset}(t), [\![t]\!]_{\boldsymbol{A}})$ for each data-term t that is SSC w.r.t. $Sig(E)$ and $\emptyset$.*

5.2 Substitutions

We define substitutions on *data-terms* and on *process-expressions*.

Definition 5.4 *Let E be a well-formed specification and $\mathcal{V}$ a set of variables over $Sig(E)$. Let Term be the set of data-terms that are SSC w.r.t. $Sig(E)$ and $\mathcal{V}$. A* substitution *σ over $Sig(E)$ and $\mathcal{V}$ is a mapping*

$$\sigma : \mathcal{V} \rightarrow Term$$

such that

- *for each $\langle x : S \rangle \in \mathcal{V}$ it holds that $sort_{Sig(E),\mathcal{V}}(\sigma(\langle x : S \rangle)) = S$,*
- *for each $\langle y : \textbf{Time} \rangle \in \mathcal{V}$ it holds that $\sigma(\langle y : \textbf{Time} \rangle)$ is a time-term.*

A substitution σ is extended to *data-terms* by:

$$\sigma(x) \stackrel{\text{def}}{=} \sigma(\langle x : S\rangle) \quad \text{if } \langle x : S\rangle \in \mathcal{V} \text{ for some } \textit{name } S,$$
$$\sigma(n) \stackrel{\text{def}}{=} n \quad \text{if } n :\rightarrow S \in \textit{Sig}(E).\textit{Fun},$$
$$\sigma(n(t_1, ..., t_m)) \stackrel{\text{def}}{=} n(\sigma(t_1), ..., \sigma(t_m)).$$

Let $\sigma_{\langle x:S\rangle}$ be defined by

$$\sigma_{\langle x:S\rangle}(\langle x' : S'\rangle) \stackrel{\text{def}}{=} \begin{cases} \langle x' : S'\rangle & \text{if } x' \equiv x \text{ and } S' \equiv S, \\ \sigma(\langle x' : S'\rangle) & \text{if } \langle x' : S'\rangle \in \mathcal{V} \setminus \langle x : S\rangle. \end{cases}$$

We extend σ to *process-expressions* that are SSC w.r.t. $\textit{Sig}(E)$ and $\mathcal{V}$ as follows:

$$\begin{array}{lcl}
\sigma(p_1 \Box p_2) & \stackrel{\text{def}}{=} & \sigma(p_1)\Box\sigma(p_2) \text{ for } \Box \in \{+, \|, ⌊⌊, |, \cdot\} \\
\sigma(s \gg p) & \stackrel{\text{def}}{=} & \sigma(s) \gg \sigma(p) \\
\sigma(p_1 \lhd t \rhd p_2) & \stackrel{\text{def}}{=} & \sigma(p_1) \lhd \sigma(t) \rhd \sigma(p_2) \\
\sigma(\mathcal{I}(x \in \prec s_0, s_1 \succ, p)) & \stackrel{\text{def}}{=} & \mathcal{I}(x \in \prec \sigma(s_0), \sigma(s_1) \succ, \sigma_{\langle x:\text{Time}\rangle}(p)) \\
\sigma(\Sigma(x : S, p)) & \stackrel{\text{def}}{=} & \Sigma(x : S, \sigma_{\langle x:S\rangle}(p)) \\
\sigma(\Box(gl, p)) & \stackrel{\text{def}}{=} & \Box(gl, \sigma(p)) \text{ for } \Box \in \{\partial, \tau, \rho\} \\
\sigma(n) & \stackrel{\text{def}}{=} & n \\
\sigma(n(t_1, ..., t_m)) & \stackrel{\text{def}}{=} & n(\sigma(t_1), ..., \sigma(t_m)) \\
\sigma(n(t_1, ..., t_m)(s)) & \stackrel{\text{def}}{=} & n(\sigma(t_1), ..., \sigma(t_m))(\sigma(s)) \\
\sigma(\delta(s)) & \stackrel{\text{def}}{=} & \delta(\sigma(s)) \\
\sigma(\tau(s)) & \stackrel{\text{def}}{=} & \tau(\sigma(s)) \\
\sigma((p)) & \stackrel{\text{def}}{=} & (\sigma(p)).
\end{array}$$

Let t be a *data-term* that is SSC w.r.t. $\textit{Sig}(E)$ and $\emptyset$ with $\textit{sort}_{\textit{Sig}(E),\emptyset}(t) = S$, σ the substitution over $\textit{Sig}(E)$ and $\{\langle x : S\rangle\}$ with $\sigma(\langle x : S\rangle) \equiv t$ and p a *process-expression* resp. t' a *data-term* that is SSC w.r.t. $\textit{Sig}(E)$ and $\{\langle x : S\rangle\}$. Then $\sigma(p)$ resp. $\sigma(t')$ is denoted by $p[t/x]$ resp. $t'[t/x]$.

Lemma 5.5 *Let E be a well-formed specification and $\mathcal{V}$ a set of variables over $\textit{Sig}(E)$. Let σ be a substitution over $\textit{Sig}(E)$ and $\mathcal{V}$.*

- *For any data-term t that is SSC w.r.t. $\textit{Sig}(E)$ and $\mathcal{V}$, $\sigma(t)$ is also a data-term that is SSC w.r.t. $\textit{Sig}(E)$ and $\mathcal{V}$. Moreover, $\textit{sort}_{\textit{Sig}(E),\mathcal{V}}(t) \equiv \textit{sort}_{\textit{Sig}(E),\mathcal{V}}(\sigma(t))$.*

- *For any time-term s that is SSC w.r.t. $\textit{Sig}(E)$ and $\mathcal{V}$, $\sigma(s)$ is also a time-term that is SSC w.r.t. $\textit{Sig}(E)$ and $\mathcal{V}$.*

- *For any process-expression p that is SSC w.r.t. $\textit{Sig}(E)$ and $\mathcal{V}$, $\sigma(p)$ is also a process-expression that is SSC w.r.t. $\textit{Sig}(E)$ and $\mathcal{V}$.*

5.3 Boolean and time preserving models

The function *rewrites* extracts the rewrite clauses together with declared variables from a *specification*.

Definition 5.6 *We define the function* rewrites *on a specification E inductively as follows:*

- *If $E \equiv$ sort-spec with sort-spec a sort-specification, then $rewrites(E) \stackrel{\text{def}}{=} \emptyset$.*
- *If $E \equiv$ func-spec with func-spec a function-specification, then $rewrites(E) \stackrel{\text{def}}{=} \emptyset$.*
- *If $E \equiv V\ R$ with V a variable-declaration-section and R a* rewrite-rules-section *with $R \equiv$ **rew** $rd_1 \ldots rd_m$ for some $m \geq 1$, then*

$$rewrites(E) \stackrel{\text{def}}{=} \{\langle\{rd_i \mid 1 \leq i \leq m\}, Vars(V)\rangle\}.$$

- *If $E \equiv$ act-spec with act-spec an action-specification, then $rewrites(E) \stackrel{\text{def}}{=} \emptyset$.*
- *If $E \equiv$ comm-spec with comm-spec a communication-specification, then $rewrites(E) \stackrel{\text{def}}{=} \emptyset$.*
- *If $E \equiv$ proc-spec with proc-spec a process-specification, then $rewrites(E) \stackrel{\text{def}}{=} \emptyset$.*
- *If $E \equiv E_1\ E_2$ where E_1 and E_2 are specifications, then $rewrites(E) \stackrel{\text{def}}{=} rewrites(E_1) \cup rewrites(E_2)$.*

Definition 5.7 *Let E be a well-formed specification. A $Sig(E)$-algebra $\boldsymbol{A}$ is a* model *of E, notation $\boldsymbol{A} \models_D E$, iff whenever $t = t' \in R$ with $\langle R, \mathcal{V}\rangle \in rewrites(E)$, then for any substitution σ over $Sig(E)$ and $\mathcal{V}$ such that $\sigma(t)$ and $\sigma(t')$ are closed it holds that $\boldsymbol{A} \models \sigma(t) = \sigma(t')$.*

We write $\boldsymbol{A} \models_D E$ with a subscript D because the model only concerns the data in E.

Definition 5.8 *Let E be a well-formed specification. A $Sig(E)$-algebra $\boldsymbol{A}$ is called* boolean and time preserving *w.r.t. E iff*

- $D(\boldsymbol{A}, \textbf{Bool}) = \{T, F\}$,
- *it is not the case that $\boldsymbol{A} \models T = F$,*
- *$D(\boldsymbol{A}, \textbf{Real})$ is the collection of real-numbers,*
- *for all real-numbers a, a' with $a \not\equiv a'$ it is not the case that $\boldsymbol{A} \models a = a'$,*
- *$D(\boldsymbol{A}, \textbf{Time})$ is the collection of time-numbers,*
- *for all time-numbers c, c' with $c \not\equiv c'$ it is not the case that $\boldsymbol{A} \models c = c'$.*

5.4 The ultimate delay operator

Let E be a well-formed *specification* and $\boldsymbol{A}$ a minimal model for E. For each *process-expression* p that is SSC w.r.t. $Sig(E)$ and $\emptyset$ we define its *ultimate delay* $U(p)$, which is the smallest time-stamp that is not reachable by p without performing an action. The ultimate delay originates from [2] and is used to formulate the action rules for the parallel operators and the deadlock.

If s_0 and s_1 are *time-terms* that are SSC w.r.t. $Sig(E)$ and $\emptyset$, then $s_0 = c_0$ and $s_1 = c_1$ hold for unique *time-numbers* c_0 and c_1. In the sequel we assume that a statement like 's_0 is greater than s_1' is well defined and holds iff $c_1 < c_0$.

Definition 5.9 *Let the process-expressions below all be SSC w.r.t. Sig(E) and* $\emptyset$.

- $U(\mathcal{I}(x \in V, P)) \stackrel{\text{def}}{=} max(V)$,
- $U(n_{\delta\tau}(s)) \stackrel{\text{def}}{=} s$ *resp.* $U(n(t_1, ..., t_m)(s)) \stackrel{\text{def}}{=} s$ *if* $n_{\delta\tau}$ *denotes* δ, τ *or* $n_{\delta\tau} \in Sig.Act$ *resp.* $n : S_1 \times ... \times S_m \in Sig.Act$,
- $U(p + q) \stackrel{\text{def}}{=} max\{U(p), U(q)\}$,
- $U(p \cdot q) \stackrel{\text{def}}{=} U(p)$,
- $U(s \gg p) \stackrel{\text{def}}{=} max\{U(p), s\}$,
- $U(p \parallel q) = U(p \mathbin{||\!\_} q) = U(p \mid q) \stackrel{\text{def}}{=} min\{U(p), U(q)\}$,
- $U(p \lhd t \rhd q) \stackrel{\text{def}}{=} \begin{cases} U(p) \text{ if } \boldsymbol{A} \models t = T, \\ U(q) \text{ if } \boldsymbol{A} \models t = F, \end{cases}$
- $U(n) \stackrel{\text{def}}{=} U(p)$ *resp.* $U(n(t_1, ..., t_m)) \stackrel{\text{def}}{=} U(\sigma(p))$ *if* $n = p \in Sig(E).Proc$ *resp.* $n(x_1 : S_1, ..., x_m : S_m) = p \in Sig(E).Proc$, *where* σ *is the substitution over* $Sig(E)$ *and* $\{\langle x_j : S_j\rangle \mid 1 \le j \le m\}$ *with* $\sigma(\langle x_j : S_j\rangle) \equiv t_j$,
- $U(\partial(\{n_1, ..., n_k\}, p)) = U(\tau(\{n_1, ..., n_k\}, p)) = U(\rho(\{n_1 \to n_1', ..., n_k \to n_k'\}, p)) \stackrel{\text{def}}{=} U(p)$,
- $U(\Sigma(x : S, p)) \stackrel{\text{def}}{=} max\{U(p[t/x]) \mid t$ *is a data-term that is SSC w.r.t.* $Sig(E)$ *and* $\emptyset$ *with* $sort_{Sig(E),\emptyset}(t) = S\}$.

5.5 The process part

In this section we define for each *process-expression* p that is SSC w.r.t. $Sig(E)$ and $\emptyset$ and each minimal model $\boldsymbol{A}$ of E that preserves the booleans and time, where E is some well-formed *specification*, a meaning in terms of a referential transition system (cf. the operational semantics in [3, 14, 15]). The action rules for the parallel operators and the deadlock were taken from [12].

Definition 5.10 *A transition system* $\mathcal{A}$ *is a quadruple* $(S, L, \longrightarrow, p_0)$ *where*

- S *is a set of* states,
- L *is a set of* labels,
- $\longrightarrow \subseteq S \times L \times S$ *is a* transition relation,
- $p_0 \in S$ *is the* initial state.

Elements $(p', l, p'') \in \longrightarrow$ *are generally written as* $p' \stackrel{l}{\longrightarrow} p''$.

Definition 5.11 *Let E be a well-formed specification, $\boldsymbol{A}$ a minimal model of E that is boolean and time preserving and r a representation function of E and $\boldsymbol{A}$. Let p be a process-expression that is SSC w.r.t. $Sig(E)$ and $\emptyset$. The* meaning *of p* **from** *E in $\boldsymbol{A}$ with representation function r is the* referential *transition system $\mathcal{A}(\boldsymbol{A}, r, p$* **from** *$E)$ defined by*

$$(S, L, \longrightarrow, p_0)$$

where

- *S is the collection containing $\surd$ and all process-expressions that are SSC w.r.t. $Sig(E)$ and $\emptyset$,*
- *L contains all elements of the form $\delta(c)$, $\tau(c)$ and $n(t_1, ..., t_m)(c)$ $(m \geq 0)$, where*
  - *$n \in Sig(E).Act$ if $m = 0$, or* *$n : sort_{Sig(E),\emptyset}(t_1) \times ... \times sort_{Sig(E),\emptyset}(t_m) \in Sig(E).Act$ if $m \geq 1$,*
  - *c is a time-number, $c \not\equiv 0$,*
  - *$t_i \equiv r(S_i, a)$ for some $a \in D(\boldsymbol{A}, S_i)$ where $S_i \equiv sort_{Sig(E),\emptyset}(t_i)$,*
- *$p_0 \stackrel{\text{def}}{=} p$,*
- *$\longrightarrow$ is the transition relation that contains exactly all transitions provable using the rules below (see for provability e.g. [6]).*

Let all the process-expressions below be SSC w.r.t. $Sig(E)$ and $\emptyset$. Let l and $l(c)$ range over $L \backslash \{\delta(c) \mid c$ is a time-number$\}$, where $l(c)$ has the time-number c as time-stamp. Let $t_1, ..., t_m$ be data-terms that are SSC w.r.t. $Sig(E)$ and $\emptyset$, and s a time-term that is SSC w.r.t. $Sig(E)$ and $\emptyset$.

- $$\frac{p \stackrel{l}{\longrightarrow} p'}{n \stackrel{l}{\longrightarrow} p'} \qquad \frac{p \stackrel{l}{\longrightarrow} \surd}{n \stackrel{l}{\longrightarrow} \surd}$$

  - *$n = p \in Sig(E).Proc.$*

- $$\frac{\sigma(p) \stackrel{l}{\longrightarrow} p'}{n(t_1, ..., t_m) \stackrel{l}{\longrightarrow} p'} \qquad \frac{\sigma(p) \stackrel{l}{\longrightarrow} \surd}{n(t_1, ..., t_m) \stackrel{l}{\longrightarrow} \surd}$$

  - *$n(x_1 : sort_{Sig(E),\emptyset}(t_1), ..., x_m : sort_{Sig(E),\emptyset}(t_m)) = p \in Sig(E).Proc \quad (m \geq 1)$,*

- σ *a substitution over* $Sig(E)$ *and* $\{\langle x_1 : sort_{Sig(E),\emptyset}(t_1)\rangle, ..., \langle x_m : sort_{Sig(E),\emptyset}(t_m)\rangle\}$ *with* $\sigma(\langle x_i : sort_{Sig(E),\emptyset}(t_i)\rangle) \equiv t_i$ *for* $1 \leq i \leq m$.

- $$\frac{p \xrightarrow{l} p'}{p+q \xrightarrow{l} p', \quad q+p \xrightarrow{l} p'} \qquad \frac{p \xrightarrow{l} \surd}{p+q \xrightarrow{l} \surd, \quad q+p \xrightarrow{l} \surd}$$

- $$\frac{p \xrightarrow{l} p'}{p \cdot q \xrightarrow{l} p' \cdot q} \qquad \frac{p \xrightarrow{l(c)} \surd}{p \cdot q \xrightarrow{l(c)} c \gg q}$$

- $$\frac{p \xrightarrow{l(c)} p' \quad s < c}{s \gg p \xrightarrow{l(c)} p'} \qquad \frac{p \xrightarrow{l(c)} \surd \quad s < c}{s \gg p \xrightarrow{l(c)} \surd}$$

- $$\frac{p \xrightarrow{l} p'}{p \lhd t \rhd q \xrightarrow{l} p'} \qquad \frac{p \xrightarrow{l} \surd}{p \lhd t \rhd q \xrightarrow{l} \surd}$$

  - $\boldsymbol{A} \models t = T$,

  - $$\frac{q \xrightarrow{l} q'}{p \lhd t \rhd q \xrightarrow{l} q'} \qquad \frac{q \xrightarrow{l} \surd}{p \lhd t \rhd q \xrightarrow{l} \surd}$$

  - $\boldsymbol{A} \models t = F$.

- $$\frac{p \xrightarrow{l(c)} p' \quad c < U(q)}{p \mathbin{\lfloor\!\lfloor} q \xrightarrow{l(c)} p' \parallel (c \gg q)} \qquad \frac{p \xrightarrow{l(c)} \surd \quad c < U(q)}{p \mathbin{\lfloor\!\lfloor} q \xrightarrow{l(c)} c \gg q}$$

- $$\frac{p \xrightarrow{n_1(t_1,\ldots,t_m)(c)} p' \quad q \xrightarrow{n_2(t_1,\ldots,t_m)(c)} q'}{p \mid q \xrightarrow{n(t_1,\ldots,t_m)(c)} p' \parallel q'} \qquad \frac{p \xrightarrow{n_1(t_1,\ldots,t_m)(c)} \surd \quad q \xrightarrow{n_2(t_1,\ldots,t_m)(c)} \surd}{p \mid q \xrightarrow{n(t_1,\ldots,t_m)(c)} \surd}$$

  $$\frac{p \xrightarrow{n_1(t_1,\ldots,t_m)(c)} p' \quad q \xrightarrow{n_2(t_1,\ldots,t_m)(c)} \surd}{p \mid q \xrightarrow{n(t_1,\ldots,t_m)(c)} p', \quad q \mid p \xrightarrow{n(t_1,\ldots,t_m)(c)} p'}$$

  - $n_1 \mid n_2 = n \in Sig(E).Comm^*$ *and* $m \geq 0$.

- $$\frac{p \mathbin{\lfloor\!\lfloor} q \xrightarrow{l} p'}{p \parallel q \xrightarrow{l} p', \quad q \parallel p \xrightarrow{l} p'} \qquad \frac{p \mathbin{\lfloor\!\lfloor} q \xrightarrow{l} \surd}{p \parallel q \xrightarrow{l} \surd, \quad q \parallel p \xrightarrow{l} \surd}$$

  - $$\frac{p \mid q \xrightarrow{l} p'}{p \parallel q \xrightarrow{l} p'} \qquad \frac{p \mid q \xrightarrow{l} \surd}{p \parallel q \xrightarrow{l} \surd}$$

- $$\frac{p \xrightarrow{l} p'}{\tau(\{n_1,...,n_k\},p) \xrightarrow{l} \tau(\{n_1,...,n_k\},p')} \qquad \frac{p \xrightarrow{l} \surd}{\tau(\{n_1,...,n_k\},p) \xrightarrow{l} \surd}$$

  – $l \equiv n(t_1,...,t_m)(c)$ $(m \geq 0)$ *and* $n \not\equiv n_i$ *for all* $1 \leq i \leq k$, *or* $l \equiv \tau(c)$,

  $$\frac{p \overset{n(t_1,...,t_m)(c)}{\longrightarrow} p'}{\tau(\{n_1,...,n_k\},p) \xrightarrow{\tau(c)} \tau(\{n_1,...,n_k\},p')} \qquad \frac{p \overset{n(t_1,...,t_m)(c)}{\longrightarrow} \surd}{\tau(\{n_1,...,n_k\},p) \xrightarrow{\tau(c)} \surd}$$

  – $n \equiv n_i$ *for some* $1 \leq i \leq k$.

- $$\frac{p \xrightarrow{l} p'}{\rho(\{n_1 \to n'_1,...,n_k \to n'_k\},p) \xrightarrow{l} \rho(\{n_1 \to n'_1,...,n_k \to n'_k\},p')}$$

  $$\frac{p \xrightarrow{l} \surd}{\rho(\{n_1 \to n'_1,...,n_k \to n'_k\},p) \xrightarrow{l} \surd}$$

  – $l \equiv n(t_1,...,t_m)(c)$ *and* $n \not\equiv n_i$ *for all* $1 \leq i \leq k$, *or* $l \equiv \tau(c)$,

  $$\frac{p \overset{n(t_1,...,t_m)(c)}{\longrightarrow} p'}{\rho(\{n_1 \to n'_1,...,n_k \to n'_k\},p) \overset{n'(t_1,...,t_m)(c)}{\longrightarrow} \rho(\{n_1 \to n'_1,...,n_k \to n'_k\},p')}$$

  $$\frac{p \overset{n(t_1,...,t_m)(c)}{\longrightarrow} \surd}{\rho(\{n_1 \to n'_1,...,n_k \to n'_k\},p) \overset{n'(t_1,...,t_m)(c)}{\longrightarrow} \surd}$$

  – $n \equiv n_i$ *and* $n' \equiv n'_i$ *for some* $1 \leq i \leq k$.

- $$\frac{p \xrightarrow{l} p'}{\partial(\{n_1,...,n_k\},p) \xrightarrow{l} \partial(\{n_1,...,n_k\},p')} \qquad \frac{p \xrightarrow{l} \surd}{\partial(\{n_1,...,n_k\},p) \xrightarrow{l} \surd}$$

  – $l \equiv n(t_1,...,t_m)(c)$ *and* $n \not\equiv n_i$ *for all* $1 \leq i \leq k$, *or* $l \equiv \tau(c)$.

- $$\mathcal{I}(y \in V, \tau(y) \cdot p) \xrightarrow{\tau(c)} c \gg p[c/y] \qquad \mathcal{I}(y \in V, \tau(y)) \xrightarrow{\tau(c)} \surd$$

  $$\mathcal{I}(y \in V, n(y) \cdot p) \overset{n()(c)}{\longrightarrow} c \gg p[c/y] \qquad \mathcal{I}(y \in V, n(y)) \overset{n()(c)}{\longrightarrow} \surd$$

  $$\mathcal{I}(y \in V, n(t_1,...,t_m)(y) \cdot p) \overset{n(u_1,...u_m)(c)}{\longrightarrow} c \gg p[c/y]$$

  $$\mathcal{I}(y \in V, n(t_1,...,t_m)(y)) \overset{n(u_1,...u_m)(c)}{\longrightarrow} \surd$$

  – $n \in Sig(E).Act$ *or*
  $n : sort_{Sig(E),\{(y:\mathbf{Time})\}}(t_1) \times ... \times sort_{Sig(E),\{(y:\mathbf{Time})\}}(t_m) \in Sig(E).Act \quad (m \geq 1)$,

- *c is in the interval* V, $c \not\equiv \infty$,
- $u_i \equiv r(sort_{Sig(E),\emptyset}(t_i), [\![t_i[c/y]]\!]_{\mathbf{A}})$ $(1 \leq i \leq m)$.

- $$n(s) \xrightarrow{n()(c)} \surd \qquad\qquad n(t_1, ..., t_m)(s) \xrightarrow{n(u_1,...,u_m)(c)} \surd$$

  - $n \in Sig(E).Act$ *or* $n : sort_{Sig(E),\emptyset}(t_1) \times ... \times sort_{Sig(E),\emptyset}(t_m) \in Sig(E).Act$ $(m \geq 1)$,
  - $s = c \not\equiv \infty$,
  - $u_i \equiv r(sort_{Sig(E),\emptyset}(t_i), [\![t_i]\!]_{\mathbf{A}})$ $(1 \leq i \leq m)$.

- $$\frac{p[t/x] \xrightarrow{l} p'}{\Sigma(x : S, p) \xrightarrow{l} p'} \qquad\qquad \frac{p[t/x] \xrightarrow{l} \surd}{\Sigma(x : S, p) \xrightarrow{l} \surd}.$$

  - *t is a data-term that is SSC w.r.t.* $Sig(E)$ *and* $\emptyset$ *with* $sort_{Sig(E),\emptyset}(t) = S$.

- $$\frac{p \xrightarrow{l} p'}{(p) \xrightarrow{l} p'} \qquad\qquad \frac{p \xrightarrow{l} \surd}{(p) \xrightarrow{l} \surd}$$

We want to add one extra transition rule to our term rewriting system, concerning δ. For that purpose we define for every *process-expression* p a set *action*(p) of *time-numbers*.

Definition 5.12 *A time-number c is in action(p) iff there is a label* $l(c)$ *in* L *such that* $p \xrightarrow{l(c)} p'$ *for some process-expression* p' *or* $p \xrightarrow{l(c)} \surd$.

Now the extra action rule is:

- $$\frac{max(action(p)) < U(p)}{p \xrightarrow{\delta(c)} \surd}$$

  - $U(p) = c$

Lemma 5.13 *Let E be a well-formed specification,* $\mathbf{A}$ *be a minimal model of E that is boolean and time preserving and r a representation function of E and* $\mathbf{A}$. *Consider a process-expression p that is SSC w.r.t.* $Sig(E)$ *and* $\emptyset$ *and let* $(S, L, \longrightarrow, p_0) \stackrel{\text{def}}{=} \mathcal{A}(\mathbf{A}, r, p)$. *If for some sequence of labels* $l_1, ..., l_m$ *it holds that* $p \xrightarrow{l_1} \dots \xrightarrow{l_m} p'$, *then either* $p' \equiv \surd$ *or* p' *is SSC w.r.t.* $Sig(E)$ *and* $\emptyset$.

Note that 'Achilles and the tortoise' situations, like described in [2], do not appear in $r\mu$CRL, since our time domain is discrete. Furthermore our time domain contains only a finite number of elements, so the solution of a recursive specification consists of finite processes.

We generally consider transition systems modulo strong bisimulation equivalence.

Definition 5.14 *Let E be a well-formed specification, $\boldsymbol{A}$ a minimal, boolean and time preserving model of E, r a representation function of E and $\boldsymbol{A}$ and p and q two process-expressions that are SSC w.r.t. $Sig(E)$ and $\emptyset$. We say that $\mathcal{A}(\boldsymbol{A}, r, p$ **from** $E)$ and $\mathcal{A}(\boldsymbol{A}, r, q$ **from** $E)$, defined by $(S, L, \rightarrow, p_0)$ and $(S, L, \rightarrow, q_0)$ respectively, are* bisimilar, *notation*

$$p \textbf{ from } E \leftrightarrows_{\boldsymbol{A},r} q \textbf{ from } E$$

iff there is a relation $R \subseteq S \times S$ such that

- $(p_0, p_1) \in R$,
- *for each pair $(t_1, t_2) \in R$:*
  - $t_1 \xrightarrow{l} t_1' \Rightarrow \exists t_2'\ t_2 \xrightarrow{l} t_2'$ *and* $(t_1', t_2') \in R$,
  - $t_2 \xrightarrow{l} t_2' \Rightarrow \exists t_1'\ t_1 \xrightarrow{l} t_1'$ *and* $(t_1', t_2') \in R$,
  - t_1 *and* t_2 *have the same ultimate delay.*

The following lemma allows us to write $\leftrightarrows_{\boldsymbol{A}}$ instead of $\leftrightarrows_{\boldsymbol{A},r}$. Note that according to our own convention we do not explicitly say where p and q stem from, as they can only come from E.

Lemma 5.15 *Let E be a well-formed specification, $\boldsymbol{A}$ a minimal, boolean and time preserving model of E and p, q process-expressions that are SSC w.r.t. $Sig(E)$ and $\emptyset$. If $p \leftrightarrows_{\boldsymbol{A},r} q$ for some representation function r of E and $\boldsymbol{A}$, then $p \leftrightarrows_{\boldsymbol{A},r'} q$ for each representation function r' of E and $\boldsymbol{A}$.*

6 An SDF-syntax for real-time μCRL

We present an SDF-syntax for $r\mu$CRL [7]. According to the convention in SDF we write syntactic categories with a capital and keywords with small letters. The first LAYOUT rule says that spaces (' '), tabs (`\t`) and newlines (`\n`) are not part of the $r\mu$CRL specification itself. The second LAYOUT rule says that lines starting with a `%`-sign followed by zero or more non-newline characters (`{~\n}*`) followed by a newline must be taken as comments and are therefore also not a part of the $r\mu$CRL syntax.

Names are arbitrary strings over `a-z`, `A-Z` and `0-9`, except that keywords are not *names*. The symbol `+` stands for one or more and `*` for zero or more occurrences. For instance, a list of one or more *names* separated by commas is denoted by `{ Name "," }+`.

`Obracket` denotes the collection $\{[, \langle\}$ and `Cbracket` the collection $\{], \rangle\}$. We denote $\in$ by `in` and ∞ by `infinity`. The phrase `right` means that an operator is right-associative and `assoc` means that an operator is associative. The phrase `bracket` says that the defined construct is not an operator, but just a way to disambiguate the construction of a syntax tree. Instead of $\mathcal{I}, \delta, \partial, \tau, \rho$ and Σ we write `integral`, `delta`, `encap`, `tau`, `hide`, `rename` and `sum`.

The priorities say that '`.`' has highest and `+` has lowest priority on *process-expressions*.

```
exports
  sorts Name
```

```
         Name-nelist
         X-name-nelist
         Space-name-nelist
         Name-list
         X-name-list
         Space-name-list
         Sort-specification
         Function-specification
         Function-declaration
         Rewrite-specification
         Variable-declaration-section
         Variable-declaration
         Data-term
         Time-term
         Rewrite-rules-section
         Rewrite-rule
         Process-expression
         Renaming-declaration
         Single-variable-declaration
         Process-specification
         Process-declaration
         Action-specification
         Action-declaration
         Communication-specification
         Communication-declaration
         Specification

lexical  syntax
         {" ", "\t", "\n"}                        -> LAYOUT
         "%" {~\n}* "\n"                          -> LAYOUT
         {a-zA-Z0-9}*                             -> Name
         {"[", "<"}                               -> Obracket
         {"]", ">"}                               -> Cbracket
context-free syntax
         { Name ","}+                             -> Name-nelist
         { Name "#"}+                             -> X-name-nelist
         { Name " "}+                             -> Space-name-nelist
         { Name ","}*                             -> Name-list
         { Name "#"}*                             -> X-name-list
         { Name " "}*                             -> Space-name-list
         sort Space-name-list                     -> Sort-specification
         func Function-declaration*               -> Function-specification
         Name-nelist ":" X-name-list "->" Name    -> Function-declaration

         Variable-declaration-section
                 Rewrite-rules-section            -> Rewrite-specification
         var Variable-declaration*                -> Variable-declaration-section
         Name-nelist ":" Name                     -> Variable-declaration
         Name                                     -> Data-term
         Name "(" { Data-term "," }+ ")"          -> Data-term
         Name                                     -> Time-term
```

```
add "(" Time-term "," Time-term ")"              -> Time-term
subt "(" Time-term "," Time-term ")"             -> Time-term
mult "(" Name "," Time-term ")"                  -> Time-term
rew Rewrite-rule+                                -> Rewrite-rules-section
Name "(" { Data-term "," }+ ")" "="
                           Data-term             -> Rewrite-rule
Name "=" Data-term                               -> Rewrite-rule

Process-expression "+" Process-expression        -> Process-expression right
Process-expression "||" Process-expression       -> Process-expression right
Process-expression "||_" Process-expression      -> Process-expression
Process-expression "|"   Process-expression      -> Process-expression right
Time-term ">>" Process-expression                -> Process-expression
Process-expression "<|" Data-term "|>"
           Process-expression                    -> Process-expression
Process-expression "." Process-expression        -> Process-expression right
integral "(" Interval-declaration "," delta
                     "(" Name ")" ")"            -> Process-expression
integral "(" Interval-declaration "," tau
                     "(" Name ")" ")"            -> Process-expression
integral "(" Interval-declaration "," Name
                     "(" Name ")" ")"            -> Process-expression
integral "(" Interval-declaration "," Name
 "(" { Data-term "," }+ ")" "(" Name ")" ")"     -> Process-expression
integral "(" Interval-declaration "," delta
  "(" Name ")" "." Process-expression ")"        -> Process-expression
integral "(" Interval-declaration "," tau
  "(" Name ")" "." Process-expression ")"        -> Process-expression
integral "(" Interval-declaration "," Name
  "(" Name ")" "." Process-expression ")"        -> Process-expression
integral "(" Interval-declaration "," Name
 "(" { Data-term "," }+ ")" "(" Name ")"
 "." Process-expression ")"                      -> Process-expression
delta "(" Time-term ")"                          -> Process-expression
tau "(" Time-term ")"                            -> Process-expression
Name "(" { Data-term "," }+ ")"
           "(" Time-term ")"                     -> Process-expression
Name "(" { Data-term "," }+ ")"                  -> Process-expression
Name                                             -> Process-expression
encap "(" "{" Name-nelist "}" ","
            Process-expression ")"               -> Process-expression
hide "(" "{" Name-nelist "}" ","
            Process-expression ")"               -> Process-expression
rename "(" "{" { Renaming-declaration "," }+
          "}" "," Process-expression ")"         -> Process-expression
sum "(" Single-variable-declaration ","
               Process-expression ")"            -> Process-expression
"(" Process-expression ")"                       -> Process-expression bracket

Name \in Obracket Time-term "," Time-term
                                Cbracket         -> Interval-declaration
```

```
Name "->" Name                                          -> Renaming-declaration
Name ":" Name                                           -> Single-variable-declaration
proc Process-declaration*                               -> Process-specification
Name "(" { Single-variable-declaration "," }+
        ")" "=" Process-expression                      -> Process-declaration
Name "=" Process-expression                             -> Process-declaration

act Action-declaration*                                 -> Action-specification
Name-nelist ":" X-name-nelist                           -> Action-declaration
Name-nelist                                             -> Action-declaration

comm Communication-declaration*                         -> Communication-specification
Name "|" Name "=" Name                                  -> Communication-declaration

Sort-specification                                      -> Specification
Function-specification                                  -> Specification
Rewrite-specification                                   -> Specification
Action-specification                                    -> Specification
Communication-specification                             -> Specification
Process-specification                                   -> Specification
Specification Specification                             -> Specification  assoc

priorities
      "+" < { "||", "|", "||_", ">>"} < "<|" "|>" < "."
```

As an example we give a $r\mu$CRL specification of a timed alternating bit protocol.

```
sort   Bool,Real,Time
func   T,F: -> Bool
       add,subt: Time#Time -> Time
       mult: Real#Time -> Time
       leq: Time#Time -> Time

sort   D
func   d1,d2,d3 -> D

sort   acknowledge
func   ack: -> acknowledge

sort   bit
func   b1,b2: -> bit
       invert: bit -> bit

rew    invert(b1)=b2
       invert(b2)=b1

act    r1,s4: D
       s2,r2,c2: D#bit
       s3,r3,c3: D#bit
       s5,r5,c5: acknowledge
       s6,r6,c6: acknowledge
```

```
       i,j

comm   s2|r2 = c2
       s3|r3 = c3
       s5|r5 = c5
       s6|r6 = c6

proc   S = S(b1)
       S(b:bit) = sum(d:D,integral(x in <0,infinity>,r1(d)(x).
                          S(d,b,mult(1.001,x)))
       S(d:D,b:bit,y:Time) = s2(d,b)(y).(integral(x in <y,mult(1.004,y)>,
                             r6(ack)(x).S(invert(b))) + S(d,b,mult(1.004,y)))

       R = R(b1)
       R(b:bit) = sum(d:D,integral(x in <0,infinity>,r3(d,b)(x).
                  s5(ack)(mult(1.001,x)).s4(d)(mult(1.0015,x)).R(invert(b)))+
                  integral(x in <0,infinity>,r3(d,invert(b))(x).
                  s5(ack)(mult(1.001,x)).R(b)))

       K = sum(d:D,sum(b:bit,integral(x in <0,infinity>,r2(d,b)(x).
           (s3(d,b)(mult(1.001,x))+i(mult(1.001,x))).K)))

       L = integral(x in <0,infinity>,r5(ack)(x).
           (s6(ack)(mult(1.001,x))+j(mult(1.001,x))).L)

       TABP = hide({c2,c3,c5,c6},encap({s2,r2,s3,r3,s5,r5,s6,r6},S||R||K||L))
```

Since this specification is quite hard to read, we also give it in the style of [2].

$$S(b) \ = \ \Sigma_{d\in D} \int_{x\in\langle 0,\infty\rangle} r1(d)(x) \cdot S(d,b,x+1))$$

$$S(d,b,t) \ = \ s2(d,b)(t) \cdot \int_{x\in\langle t,t+4\rangle} r6(ack)(x) \cdot S(1-b) \ + \ S(d,b,t+4)$$

$$R(b) = \Sigma_{d\in D}\{\int_{x\in\langle 0,\infty\rangle} r3(d,b)(x) \cdot s5(ack)(x+1) \cdot s4(d)(x+1.5) \cdot R(1-b) + \int_{x\in\langle 0,\infty\rangle} r3(d,1-b)(x) \cdot s5(ack)(x+1) \cdot R(b)\}$$

$$K \ = \ \Sigma_{(d,b)\in D\times B} \int_{x\in\langle 0,\infty\rangle} r2(d,b)(x) \cdot (s3(d,b)(x+1) + i(x+1)) \cdot K$$

$$L \ = \ \int_{x\in\langle 0,\infty\rangle} r5(ack)(x) \cdot (s6(ack)(x+1) + j(x+1)) \cdot L$$

$$TABP \ = \ \tau_{\{c2,c3,c5,c6\}} \circ \partial_{\{s2,r2,s3,r3,s5,r5,s6,r6\}}(S(0) \parallel R(0) \parallel K \parallel L)$$

References

[1] J.C.M. Baeten and J.A. Bergstra. Process algebra with signals and conditions. To appear in M. Broy, editor, *Proceedings NATO Summer School*, Marktoberdorf. Springer-Verlag, 1990.

[2] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Journal of Formal Aspects of Computing*, 3:142-188, 1990.

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

[4] H. Ehrig and B. Mahr. *Fundamentals of algebraic specifications I*, volume 6 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1985.

[5] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. Report CS-R9076, Centre for Mathematics and Computer Science, 1991.

[6] J.F. Groote and F.W. Vaandrager. Structured operational semantics and bisimulation as a congruence. Extended abstract in G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Proceedings* 16^{th} *ICALP,* Stresa, volume 372 of *Lecture Notes in Computer Science*, pages 423–438. Springer-Verlag, 1989. Full version to appear in *Information and Computation.*

[7] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF – reference manual –. *ACM SIGPLAN Notices*, 24(11):43–75, 1989.

[8] C.A.R. Hoare, I.J. Hayes, He Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, and B.A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, August 1987.

[9] IEEE. IEEE standard for binary floating-point arithmetic. *SIGPLAN-notices*, 22(2):9–25, 1987.

[10] ISO. *Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour* ISO/TC97/SC21/N DIS8807, 1987.

[11] J.W. Klop. Term rewriting systems. To appear in *Handbook of Logic in Computer Science*, volume 1. Oxford University Press, 1990.

[12] S. Klusener. Completeness in real time process algebra. To appear in J. Baeten and J.F. Groote, editors, *Proceedings CONCUR '91*, Amsterdam. Springer-Verlag, 1991.

[13] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.

[14] G.D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Proceedings IFIP TC2 Working Conference on Formal Description of Programming Concepts – II*, Garmisch, pages 199–225, Amsterdam. North-Holland, 1983.

[15] SPECS-semantics. *Definition of MR and CRL Version 2.1*, 1990.