LPS definitions and an ATerm representation format for μCRL with multiactions and time

Jan Friso Groote   Yaroslav S. Usenko

March 20, 2023

Contents

1 LPS definitions

The equation below represents a Linear Process Equation for μCRL with multiactions and time (MTLPS).

$$
\begin{aligned}
\mathsf{X}(\overrightarrow{d{:}D}) = & \sum_{i\in I}\sum_{\overrightarrow{e_i:E_i}} c_i(\overrightarrow{d,e_i}) \to \mathsf{a}_i^0(\overrightarrow{f_{i,0}}(\overrightarrow{d,e_i})) \mid \cdots \mid \mathsf{a}_i^{n(i)}(\overrightarrow{f_{i,n(i)}}(\overrightarrow{d,e_i})) \,{}^{\varsigma}\, t_i(\overrightarrow{d,e_i}) \cdot \mathsf{X}_i(\overrightarrow{g_i}(\overrightarrow{d,e_i})) \\
& + \sum_{j\in J}\sum_{\overrightarrow{e_j:E_j}} c_j(\overrightarrow{d,e_j}) \to \mathsf{a}_j^0(\overrightarrow{f_{j,0}}(\overrightarrow{d,e_j})) \mid \cdots \mid \mathsf{a}_j^{n(j)}(\overrightarrow{f_{j,n(j)}}(\overrightarrow{d,e_j})) \,{}^{\varsigma}\, t_j(\overrightarrow{d,e_j}) \\
& + \sum_{\overrightarrow{e_\delta:E_\delta}} c_\delta(\overrightarrow{d,e_\delta}) \to \delta \,{}^{\varsigma}\, t_\delta(\overrightarrow{d,e_\delta})
\end{aligned}
$$

where I and J are disjoint.

It is possible to translate multiactions to regular μCRL actions (with longer parameter lists). In this way a MTLPS can be translated to a TLPS, preserving equivalence. The TLPS that corresponds to the above MTLPS is defined in the following way.

$$\begin{aligned}
\mathsf{X}(\overrightarrow{d{:}D}) = &\sum_{i\in I}\sum_{\overrightarrow{e_i:E_i}} c_i(\overrightarrow{d,e_i}) \to \mathsf{a}_i^0_\mathsf{a}_i^1_\ldots_\mathsf{a}_i^{n(i)}(\overrightarrow{f_{i,0}}(\overrightarrow{d,e_i}),\ldots,\overrightarrow{f_{i,n(i)}}(\overrightarrow{d,e_i})) \mathbin{^c} t_i(\overrightarrow{d,e_i}) \cdot \mathsf{X}_i(\overrightarrow{g_i}(\overrightarrow{d,e_i}))\\
&+\sum_{j\in J}\sum_{\overrightarrow{e_j:E_j}} c_j(\overrightarrow{d,e_j}) \to \mathsf{a}_j^0_\mathsf{a}_j^1_\ldots_\mathsf{a}_j^{n(j)}(\overrightarrow{f_{j,0}}(\overrightarrow{d,e_j}),\ldots,\overrightarrow{f_{j,n(j)}}(\overrightarrow{d,e_j})) \mathbin{^c} t_j(\overrightarrow{d,e_j})\\
&+\sum_{\overrightarrow{e_\delta:E_\delta}} c_\delta(\overrightarrow{d,e_\delta}) \to \delta \mathbin{^c} t_\delta(\overrightarrow{d,e_\delta})
\end{aligned}$$

where I and J are disjoint, and $\mathsf{a}_i^0\_\mathsf{a}_i^1\_\ldots\_\mathsf{a}_i^{n(i)}$ and $\mathsf{a}_j^0\_\mathsf{a}_j^1\_\ldots\_\mathsf{a}_j^{n(j)}$ are new actions (for each i and j), parameterized by the concatenation of the parameter lists of the contained actions.

`YSU: TODO :USY` formalize below !!!

Theorem 1.1. *Given* MTLPS1 = (mCRL2) = MTLPS2*,*
TLPS(MTLPS1) = (timed mcrl) = TLPS(MTLPS2)*.*

Time can be eliminated from TLPSs in a similar way (see page 106 of the thesis).

A Aterm format for mCRL2 after parsing

B Static Semantics and Well-formedness

In this section it is defined when a specification is correctly defined. We use the syntactical categories from the previous section (in teletype font) to refer to items in a specification. If we denote a concrete part of a specification, we prefer using the latex symbols, to increase readability. The definitions below are an adapted copy from those in [**?**].

In essence the static semantics says that functions and terms are well typed, and some sorts and functions are present in the specification. The validity of all static semantic requirements can efficiently be decided for any specification.

A specification is well formed, if it satisfies the static semantic requirements, there are no empty sorts and the sort *Time* is appropriately defined. We only give an operational semantics to well-formed specifications.

B.1 Static semantics

A `Specification` must be internally consistent. This means that all objects that are used must be declared exactly once and are used such that the sorts are correct. It also means that action, process, constant and variable names cannot be confused. Furthermore, it means that communications are specified in a functional way and that it is guaranteed that the terms used in an equation are well-typed. Because all these properties can be statically decided, a specification that is internally consistent is called SSC (*Statically Semantically Correct*). All next definitions culminate in Definition **??**.

B.1.1 SSC of `Specification`

We assume that the specification has the form $\mathsf{spec}(\mathit{sortspec}?, \mathit{opspec}?, \mathit{eqnspec}?, \mathit{actspec}?, \mathit{proscpec}?, \mathit{init})$ (an easy transformation of the input aterm brings it to this form). All of the parameters are optional except the last one (the minimal specification is $\mathsf{spec}(\mathsf{init}(\mathsf{tau}())))$. Sometimes part of the specification is not used. For example, any sort specification is useless unless some functions are defined for them. And also functions specifications are useless if they do not occur in expressions. Such specifications are still considered SSC, although an implementation of the checker may issue a warning in such cases.

Let Sig be a signature and $\mathcal{V}$ be a set of variables over Sig. We define the predicate 'is SSC wrt. Sig' inductively over the syntax of a `Specification`.

Sorts Sort declarations:

- A `SortSpec` $\mathsf{SortSpec}([\mathit{sspec}_1, \ldots, \mathit{sspec}_m])$ with $m \geq 1$ is SSC wrt. Sig iff
  - all $\mathit{sspec}_1, \ldots, \mathit{sspec}_m$ are SSC wrt. Sig.
  - Defined sort names are different: for all $i < j$, $\mathit{defined\_sorts}(\mathit{sspec}_i) \neq \mathit{defined\_sorts}(\mathit{sspec}_j)$.
- A `SortDecl` $\mathsf{SortDeclStandard}([n_1 \cdots n_m])$ with $m \geq 1$ is SSC wrt. Sig iff all $n_1, \ldots, n_m$ are pairwise different.
- A `SortDecl` $\mathsf{SortDeclRef}(n, s)$ with $m \geq 1$ is SSC wrt. Sig iff the *sort expression* s is SSC w.r.t $Sig \setminus [n_1 \cdots n_m]$. Here we note that no recursive sort references are allowed.

- A `SortDecl` $\mathsf{SortDeclStruct}(n, [cons_1 \ldots, cons_m])$ with $m \geq 1$ is SSC wrt. *Sig* iff all *constructor expressions cons* are SSC w.r.t *Sig*.
- A `ConstDecl` $\mathsf{StructDeclCons}(n, [proj_1 \ldots, proj_m]), k)$ with $m \geq 0$ is SSC wrt. *Sig* iff
  - both n and k are not declared as function or map (or $k == nil()$)
  - all *projection expressions proj* are SSC w.r.t *Sig*.
- A `ProjDecl` $\mathsf{StructDeclProj}(n, \mathsf{Dom}([s_1 \cdots s_m]))$ with $m \geq 1$ is SSC wrt. *Sig* iff
  - both n is not declared as function or map (or $n == nil()$)
  - *sort expressions s* are SSC w.r.t *Sig*

Data types

- A `OpSpec`
  $\mathsf{ConsSpec}([IdDecls([n_{11}, \ldots, n_{1l_1}], s_1), \ldots, IdDecls([n_{m1}, \ldots, n_{ml_m}], s_m)])$ or
  $\mathsf{MapSpec}([IdDecls([n_{11}, \ldots, n_{1l_1}], s_1), \ldots, IdDecls([n_{m1}, \ldots, n_{ml_m}], s_m)])$ with $m \geq 1$, $l_i \geq 1$, $k_i \geq 0$ for $1 \leq i \leq m$ is SSC wrt. *Sig* iff
  - for all $1 \leq i \leq m$ $n_{i1}, \ldots, n_{il_i}$ are pairwise different,
  - for all $1 \leq i \leq m$ it holds that s_i is SSC wrt. *Sig*.
  - for all $1 \leq i < j \leq m$ it holds that if $n_{ik} \equiv n_{jk'}$ for some $1 \leq k \leq l_i$ and $1 \leq k' \leq l_j$, then $type_{Sig}(s_i) \neq type_{Sig}(s_j)$,
- A `EqnSpec` of the form:

$$\mathsf{EqnSpec}([IdDecls([n_{11}, \ldots, n_{1l_1}], s_1), \ldots, IdDecls([n_{m1}, \ldots, n_{ml_m}], s_m)], \tag{1}$$
$$[\mathsf{EqnSpec}(d_1, d_1') \ldots \mathsf{EqnSpec}(d_k, d_k')]) \tag{2}$$

  with $m \geq 1$, $l_i \geq 1$, $k_i \geq 0$ for $1 \leq i \leq m$ is SSC wrt. *Sig* iff
  - for all $1 \leq i, j \leq m$ n_{ij} are pairwise different,
  - for all $1 \leq i \leq m$ it holds that s_i is SSC wrt. *Sig*.
  - for all $1 \leq j \leq k$ it holds that d_i and d_i' is SSC wrt. $Sig + ns$.
  - for all $1 \leq j \leq k$ it holds that the types of d_i and d_i' are uniquly compatible (wrt. *Sig*).

Actions A `ActSpec` of the form:
$\mathsf{ActSpec}([\mathsf{ActDecl}([n_{11}, \ldots, n_{1l_1}], d_1) \ldots \mathsf{ActDecl}([n_{m1}, \ldots, n_{ml_m}], d_m)])$ with $m \geq 1$ is SSC wrt. *Sig* iff

- for all $1 \leq i \leq m$ all n_{ij} are pairwise different.
- none of them are in $Sig.Fun \cup Sig.Map$
- d_i is $Nil()$ or d_i is $Dom([s_1, \ldots, s_n])$, and all s_j are SSC wrt. *Sig*.

Processes

- A `ProcSpec` $\mathsf{ProcSpec}([\mathsf{ProcDecl}(n_1, vars_1, p_1), \ldots, \mathsf{ProcDecl}(n_m, vars_m, p_m)])$ with $m \geq 1$ is SSC wrt. Sig iff
  - for each $1 \leq i < j \leq m$: if $type(vars_i) = type(vars_j)$, then $n_i \neq n_j$,
  - none of them are in $Sig.Fun \cup Sig.Map \cup Sig.Act$
  - for each `Name` S' it holds that $n{:}S_1 \times \cdots \times S_k \to S' \notin Sig.Fun \cup Sig.Map$,
  - the `Names` $x_1, \ldots, x_k$ are pairwise different and $\{\langle x_j{:}S_j\rangle \mid 1 \leq j \leq k\}$ is a set of variables over Sig,
  - p is SSC wrt. Sig and $\{\langle x_j{:}S_j\rangle \mid 1 \leq j \leq k\}$.
- A `Init` of the form $\mathsf{Init}(p)$ is SSC wrt. Sig iff p SSC wrt. to Sig and $\emptyset$.

Definition B.1. Let E be a `Specification`. We say that E is SSC iff E is SSC wrt. $Sig(E)$.

B.1.2 Process and Data Terms. (Sub-)Typing

Process terms Let Sig be a signature and $\mathcal{V}$ be a set of variables over Sig. We say that a `Process-term` p is *SSC wrt. to Sig and* $\mathcal{V}$ iff one of the following hold:

- $p \equiv p_1 + p_2$, $p \equiv p_1 \parallel p_2$, $p \equiv p_1 \,⌊\!⌊\, p_2$, $p \equiv p_1 \mid p_2$, $p \equiv p_1 {\cdot} p_2$ or $p \equiv p_1 \ll p_2$ and both p_1 and p_2 are SSC wrt. Sig and $\mathcal{V}$,
- $p \equiv p_1 \triangleleft t \triangleright p_2$ and
  - p_1 and p_2 are SSC wrt. Sig and $\mathcal{V}$,
  - t is SSC wrt. Sig and $\mathcal{V}$ and $sort_{Sig,\mathcal{V}}(t) = \langle Bool\rangle$.
- $p \equiv p_1 ^{\varsigma} t$ and
  - p_1 is SSC wrt. Sig and $\mathcal{V}$
  - t is SSC wrt. Sig and $\mathcal{V}$ and $sort_{Sig,\mathcal{V}}(t) = Time$.
- $p \equiv \delta$ or $p \equiv \tau$.
- $p \equiv \partial_{\{n_1,\ldots,n_m\}} p_1$ or $p \equiv \tau_{\{n_1,\ldots,n_m\}} p_1$ with $m \geq 1$ and
  - for all $1 \leq i \neq j \leq m$ $n_i \not\sqsubseteq n_j$,
  - for $1 \leq i \leq m$, if $n_i = n_{i,1} \mid \cdots \mid n_{i,k}$, then $n_{i,j} \in Sig.ActNames$.
  - p_1 is SSC wrt. Sig and $\mathcal{V}$.
- $p \equiv \nabla_{\{n_1,\ldots,n_m\}} p_1$ with $m \geq 1$ and
  - for all $1 \leq i < j \leq m$ $n_i \not\equiv n_j$,
  - for $1 \leq i \leq m$, if $n_i = n_{i,1} \mid \cdots \mid n_{i,k}$, then $n_{i,j} \in Sig.ActNames$.
  - p_1 is SSC wrt. Sig and $\mathcal{V}$.
- $p \equiv \rho_{\{n_1 \to n'_1,\ldots,n_m \to n'_m\}} p_1$ and
  - for $1 \leq i \leq m$ both $n_i, n'_i \in Sig.ActNames$.
  - for each $1 \leq i < j \leq m$ it holds that $n_i \not\equiv n_j$,

- for $1 \leq i \leq m$, the types of n_i and n'_i are the same in Sig.
- p_1 is SSC wrt. Sig and $\mathcal{V}$.

$p \equiv \Gamma_{\{n_1 \to n'_1, \ldots, n_m \to n'_m\}} p_1$ and

- for $1 \leq i \leq m$, if $n_i = n_{i,1} \mid \cdots \mid n_{i,k}$, then $n_{i,j} \in Sig.ActNames$.
- for $1 \leq i \leq m$ either $n'_i \in Sig.ActNames$ or $n'_i = \tau$.
- for each $1 \leq i \neq j \leq m$ it holds that $n_i \not\sqsubseteq n_j$,
- for $1 \leq i \leq m$ it holds that, if $n_i = n_{i,1} \mid \cdots \mid n_{i,k}$, then the types of all $n_{i,j}$ and n'_i are the same in Sig.
- p_1 is SSC wrt. Sig and $\mathcal{V}$.

- $p \equiv \Sigma_{x:S} p_1$ and iff
  - $(\mathcal{V} \backslash \{\langle x{:}S' \rangle \mid S' \text{ is a } \texttt{Name}\}) \cup \{\langle x{:}S \rangle\}$ is a set of variables over Sig,
  - p_1 is SSC wrt. Sig and $(\mathcal{V} \backslash \{\langle x{:}S' \rangle \mid S' \text{ is a } \texttt{Name}\}) \cup \{\langle x{:}S \rangle\}$.
- $p \equiv n$ and $n = p' \in Sig.Proc$ for some `Process-term` p', or $n \in Sig.Act$.
- $p \equiv n(t_1, \ldots, t_m)$ with $m \geq 1$ and
  - $n(x_1{:}sort_{Sig,\mathcal{V}}(t_1), \ldots, x_m{:}sort_{Sig,\mathcal{V}}(t_m)) = p' \in Sig.Proc$ for `Names` $x_1, \ldots, x_m$ and `Process-term` p', or $n{:}sort_{Sig,\mathcal{V}}(t_1) \times \cdots \times sort_{Sig,\mathcal{V}}(t_m) \in Sig.Act$,
  - for $1 \leq i \leq m$ the `Data-term` t_i is SSC wrt. Sig and $\mathcal{V}$.

Sort expressions

- A `SortExpr` $\mathsf{SortBool}(), \mathsf{SortPos}(), \mathsf{SortNat}(), \mathsf{SortInt}()$ are SSC.
- A `SortExpr` $\mathsf{SortList}(s), \mathsf{SortSet}(s), \mathsf{SortBag}(s)$ are SSC wrt. Sig iff sort expression s is SSC wrt. Sig.
- A `SortExpr` $\mathsf{SortRef}(n)$ is SSC wrt. Sig iff $n \in Sorts(Sig)$.
- A `SortExpr` $\mathsf{SortArrow}(\mathsf{Dom}([n_1, \ldots, n_m]), n)$ with $m \geq 1$ is SSC wrt. Sig iff all *sort expressions* n are SSC w.r.t Sig.

Any sort expression that is SCC is also well-typed. I.e. it is impossible to specify an incorrectly typed sort.

C Context Information

The context consists of two parts. The static part corresponds to the global information in the specification. The dynamic part contains the definitions of the variables, and can change depending on the scope. Given a context of a specification κ, we denote the static context as $Sig(\kappa)$ and the dynamic part as $Vars(\kappa)$. The static context is a tupple

$$(BasicSorts, DefinedSorts, Operations, Actions, Processes)$$

which represents the names and types of the sorts, operations, actions and processes defined in the specification. The types of the context operatinos are defined below:

$$\begin{aligned}
\mathit{BasicSorts} &= \{\mathit{String}\}\\
\mathit{DefinedSorts} &: \mathit{String} \to \mathit{Type}\\
\mathit{Operations} &\in \mathit{String} \times \mathit{Type}\\
\mathit{Actions} &\in \mathit{String} \times \mathit{Type}\\
\mathit{Processes} &: \mathit{String} \to \mathit{Type}
\end{aligned}$$

The sort *Type* is a sort expression containing defined sorts, a list of such expressions, or the empty type (unit type). It can be also unknown. The function $\mathit{basicType} : \mathit{Type} \to \mathit{Type}$ unfolds all occurrences of derived sort names in a type expression.

The variables are defined as a function from Variable name to a variable type $\mathit{Variables} : \mathit{String} \to \mathit{Type}$.

Data Terms Let *Sig* be a signature, and let $\mathcal{V}$ be a set of variables over *Sig*. A `Data-term` t is called SSC wrt. *Sig* and $\mathcal{V}$ iff one of the following holds

- $t \equiv n$ with n a `Name` and $\langle n{:}S \rangle \in \mathcal{V}$ for some S, or $n{:} \to \mathit{sort}_{\mathit{Sig},\mathcal{V}}(n) \in \mathit{Sig.Fun} \cup \mathit{Sig.Map}$.
- $t \equiv n(t_1, \ldots, t_m)$ $(m \geq 1)$ and $n{:}\mathit{sort}_{\mathit{Sig},\mathcal{V}}(t_1) \times \cdots \times \mathit{sort}_{\mathit{Sig},\mathcal{V}}(t_m) \to \mathit{sort}_{\mathit{Sig},\mathcal{V}}(n(t_1, \ldots, t_m)) \in \mathit{Sig.Fun} \cup \mathit{Sig.Map}$ and all t_i $(1 \leq i \leq m)$ are SSC wrt. *Sig* and $\mathcal{V}$.

The typing rules of the built-in data types can be defined as follows: As for the sort and process expressions we introduce the following functions for data expressions: *is_well_named* – all ids are defined, *id_vars* to identify the variables, *types* – all possible types the term can be, *is_well_typed* – is the term well-typed?

The function $\mathbb{T}_\kappa$ is defined defined as (well-namedness of the arguments is assumed):

$\mathsf{DataVar}(\mathit{String})$		$\mathit{type}(\kappa, \mathit{String})$
$\mathsf{OpId}(\mathit{String})$		$\mathit{type}(\kappa, \mathit{String})$
$\mathsf{Number}(\mathit{NumberString})$		$\mathit{PNI}(\mathit{NumberString})$
$\mathsf{ListEnum}(d_0, \ldots, d_n)$	$\forall i \in \overline{0,n}\ \mathbb{T}(d_i) \equiv_t \mathbb{T}(d_0)$	$\mathit{List}(\mathit{minC}(\mathbb{T}(d_0), \ldots, \mathbb{T}(d_n)))$
$\mathsf{SetEnum}(d_0, \ldots, d_n)$	$\forall i \in \overline{0,n}\ \mathbb{T}(d_i) \equiv_t \mathbb{T}(d_0)$	$\mathit{Set}(\mathit{minC}(\mathbb{T}(d_0), \ldots, \mathbb{T}(d_n)))$
$\mathsf{BagEnum}(\mathsf{BagEnumElt}(d_0, d'_0),$ $\ldots, \mathsf{BagEnumElt}(d_n, d'_n))$	$\forall i \in \overline{0,n}\ (\mathbb{T}(d_i) \equiv_t \mathbb{T}(d_0) \wedge \mathbb{T}(d'_i) \equiv_t \mathit{PN})$	$\mathit{Bag}(\mathit{minC}(\mathbb{T}(d_0), \ldots, \mathbb{T}(d_n)))$
$\mathsf{SetBagComp}(\mathsf{IdDecl}(v, s), d)$	$\mathit{wt}(\kappa + (v, s), d) \wedge$ $(\mathbb{T}_{\kappa'}(d) = \mathit{Bool} \vee \mathbb{T}_{\kappa'}(d) \equiv_t \mathit{PN})$	$\mathit{Set}(s)$ if $\mathbb{T}_{\kappa'}(d) = \mathit{Bool}$ $\mathit{Bag}(s)$ if $\mathbb{T}_{\kappa'}(d) \equiv_t \mathit{PN}$
$\mathsf{DataApp}(d, d_0, \ldots, d_n)$	$\mathbb{T}(d) = A_0 \ldots A_n \to B \wedge$ $\mathbb{T}(d_0) \equiv_t A_0 \wedge \cdots \wedge \mathbb{T}(d_n) \equiv_t A_n$	B
$\mathsf{Forall}([\mathsf{IdsDecl}(\overrightarrow{v_0}, s_0),$ $\ldots, \mathsf{IdsDecl}(\overrightarrow{v_n}, s_n)], d)$	$\mathit{wt}(\kappa + (\overrightarrow{v_0}, s_0, \ldots, \overrightarrow{v_n}, s_n), d)$ $\wedge\ \mathbb{T}_{\kappa'}(d) = \mathit{Bool}$	Bool
$\mathsf{Exists}([\mathsf{IdsDecl}(\overrightarrow{v_0}, s_0),$ $\ldots, \mathsf{IdsDecl}(\overrightarrow{v_n}, s_n)], d)$	$\mathit{wt}(\kappa + (\overrightarrow{v_0}, s_0, \ldots, \overrightarrow{v_n}, s_n), d)$ $\wedge\ \mathbb{T}_{\kappa'}(d) = \mathit{Bool}$ Bool	Bool
$\mathsf{Lambda}([\mathsf{IdsDecl}(\overrightarrow{v_0}, s_0),$ $\ldots, \mathsf{IdsDecl}(\overrightarrow{v_n}, s_n)], d)$	$\mathit{wt}(\kappa + (\overrightarrow{v_0}, s_0, \ldots, \overrightarrow{v_n}, s_n), d)$	$s_0^{\mathit{len}(\overrightarrow{v_0})}, \ldots, s_n^{\mathit{len}(\overrightarrow{v_n})} \to \mathbb{T}_{\kappa'}(d)$
$\mathsf{Whr}(d, [v_0, d_0, \ldots, v_n, d_n])$	$\mathit{wt}(\kappa + (v_0, \mathbb{T}(d_0), \ldots, v_n, \mathbb{T}(d_n)), d)$	$\mathbb{T}_{\kappa'}(d)$

The following internal, or system, identifiers have the corresponding (polymorphic) types:

EmptyList()		$List(TypeAny)$
EmptySetBag()		$SB(TypeAny)$
NotOrCompl(d)	$\mathbb{T}(d) = Bool \vee \mathbb{T}(d) \equiv_t SB(TypeAny)$	$\mathbb{T}(d)$
Neg(d)	$\mathbb{T}(d) \equiv_t PNI$	Int
Size(d)	$\mathbb{T}(d) \equiv_t LSB(TypeAny)$	Nat
ListAt(d, d')	$\mathbb{T}(d) \equiv_t List(TypeAny) \wedge \mathbb{T}(d') \equiv_t PN$	$ListArg(d)$
Div(d, d')	$\mathbb{T}(d) \equiv_t PNI \equiv_t \mathbb{T}(d')$	$div(PNI)$
Mod(d, d')	$\mathbb{T}(d) \equiv_t PNI \wedge \mathbb{T}(d') \equiv_t Pos$	$mod(PNI)$
MultOrIntersect(d, d')	$(\mathbb{T}(d) \equiv_t PNI \vee \mathbb{T}(d) \equiv_t SB(TypeAny))$	$maxMoI(\mathbb{T}(d), \mathbb{T}(d'))$
AddOrUnion(d, d')	$\wedge\ \mathbb{T}(d) \equiv_t \mathbb{T}(d')$	
SubtOrDiff(d, d')		
LTOrPropSubset(d, d')		$Bool$
GTOrPropSupset(d, d')		$Bool$
LTEOrSubset(d, d')		$Bool$
GTEOrSupSet(d, d')		$Bool$
In(d, d')	$LSB(\mathbb{T}(d)) \equiv_t \mathbb{T}(d')$	$Bool$
Cons(d, d')	$List(\mathbb{T}(d)) \equiv_t \mathbb{T}(d')$	$max(List(\mathbb{T}(d)), \mathbb{T}(d'))$
Snoc(d, d')	$List(\mathbb{T}(d')) \equiv_t \mathbb{T}(d)$	$max(List(\mathbb{T}(d')), \mathbb{T}(d))$
Concat(d, d')	$\mathbb{T}(d) \equiv_t List(TypeAny) \equiv_t \mathbb{T}(d')$	$List(max(\mathbb{T}(d), \mathbb{T}(d')))$
EqNeq(d, d')	$\mathbb{T}(d) \equiv_t \mathbb{T}(d')$	$Bool$

True()		$Bool$
False()		$Bool$
Imp(d, d')	$\mathbb{T}(d) \equiv_t \mathbb{T}(d') = Bool$	
And(d, d')	$\mathbb{T}(d) \equiv_t \mathbb{T}(d') = Bool$	

C.1 The signature of a specification

Definition C.1. The signature $Sig(E)$ of a `Specification` E consists of a seven-tuple

$$(Sort, Fun, Map, Act, Comm, Proc, Init)$$

where each component is a set containing all elements of a main syntactical category declared in E. The signature $Sig(E)$ of E is inductively defined as follows:

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

- If $E \equiv\ : fd \rightarrow_1 \cdots fd_m$ with $m \geq 1$, then $Sig(E) \stackrel{\text{def}}{=} (\emptyset, Fun, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$, where

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

- If $E \equiv$ **map** $md_1 \cdots md_m$ with $m \geq 1$, then $Sig(E) \stackrel{\text{def}}{=} (\emptyset, \emptyset, Map, \emptyset, \emptyset, \emptyset, \emptyset)$, where

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

- If E is a `Equation-specification`, then $\mathit{Sig}(E) \stackrel{\text{def}}{=} (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$.
- If $E \equiv\ ad_1 \cdots ad_m$ with $m \geq 1$, then $\mathit{Sig}(E) \stackrel{\text{def}}{=} (\emptyset, \emptyset, \emptyset, \mathit{Act}, \emptyset, \emptyset, \emptyset)$, where

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

- If $E \equiv \textbf{comm}\ cd_1 \cdots cd_m$ with $m \geq 1$, then $\mathit{Sig}(E) \stackrel{\text{def}}{=} (\emptyset, \emptyset, \emptyset, \emptyset, \{cd_i \mid 1 \leq i \leq m\}, \emptyset, \emptyset)$.
- If $E \equiv \textbf{proc}\ pd_1 \cdots pd_m$ with $m \geq 1$, then $\mathit{Sig}(E) \stackrel{\text{def}}{=} (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \{pd_i \mid 1 \leq i \leq m\}, \emptyset)$.
- If $E \equiv \textbf{init}\ pe$ then $\mathit{Sig}(E) \stackrel{\text{def}}{=} (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \{pe\})$.
- If $E \equiv E_1\ E_2$ with $\mathit{Sig}(E_i) = (\mathit{Sort}_i, \mathit{Fun}_i, \mathit{Map}_i, \mathit{Act}_i, \mathit{Comm}_i, \mathit{Proc}_i, \mathit{Init}_i)$ for $i = 1, 2$, then

$$\begin{array}{l} \mathit{Sig}(E) \stackrel{\text{def}}{=} (\mathit{Sort}_1 \cup \mathit{Sort}_2, \mathit{Fun}_1 \cup \mathit{Fun}_2, \mathit{Map}_1 \cup \mathit{Map}_2, \\ \qquad\qquad \mathit{Act}_1 \cup \mathit{Act}_2, \mathit{Comm}_1 \cup \mathit{Comm}_2, \mathit{Proc}_1 \cup \mathit{Proc}_2, \mathit{Init}_1 \cup \mathit{Init}_2). \end{array}$$

Definition C.2. Let $\mathit{Sig} = (\mathit{Sort}, \mathit{Fun}, \mathit{Map}, \mathit{Act}, \mathit{Comm}, \mathit{Proc}, \mathit{Init})$ be a signature. We write

$$\begin{array}{llll} \mathit{Sig}.\mathit{Sort} \text{ for } \mathit{Sort}, & \mathit{Sig}.\mathit{Fun} \text{ for } \mathit{Fun}, & \mathit{Sig}.\mathit{Map} \text{ for } \mathit{Map}, & \mathit{Sig}.\mathit{Act} \text{ for } \mathit{Act}, \\ \mathit{Sig}.\mathit{Comm} \text{ for } \mathit{Comm}, & \mathit{Sig}.\mathit{Proc} \text{ for } \mathit{Proc}, & \mathit{Sig}.\mathit{Init} \text{ for } \mathit{Init}. & \end{array}$$

C.2 Variables

Variables play an important role in specifications. The next definition says given a specification E, which elements from `Name` can play the role of a variable without confusion with defined constants. Moreover, variables must have an unambiguous and declared sort.

Definition C.3. Let Sig be a signature. A set $\mathcal{V}$ containing pairs $\langle x{:}S\rangle$ with x and S from `Name`, is called a *set of variables* over Sig iff for each $\langle x{:}S\rangle \in \mathcal{V}$:

- for each `Name` S' and `Process-term` p it holds that $x{:} \to S' \notin \mathit{Sig}.\mathit{Fun} \cup \mathit{Sig}.\mathit{Map}$, $x \notin \mathit{Sig}.\mathit{Act}$ and $x = p \notin \mathit{Sig}.\mathit{Proc}$,
- $S \in \mathit{Sig}.\mathit{Sort}$,
- for each `Name` S' such that $S' \not\equiv S$ it holds that $\langle x{:}S'\rangle \notin \mathcal{V}$.

Definition C.4. Let vd be a `Variable-declaration`. The function Vars is defined by:

$$\mathit{Vars}(vd) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } vd \text{ is empty,} \\ \{\langle x_{ij}{:}S_i\rangle \mid 1 \leq i \leq m, & \\ \qquad\qquad 1 \leq j \leq l_i\} & \text{if } vd \equiv \textbf{var}\ x_{11}, \ldots, x_{1l_1}{:}S_1\ \ldots\ x_{m1}, \ldots, x_{ml_m}{:}S_m. \end{cases}$$

In the following definitions we give functions yielding the sort of and the variables in a `Data-term` t.

Definition C.5. Let t be a `data-term` and Sig a signature. Let $\mathcal{V}$ be a set of variables over Sig. We define:

$$sort_{Sig,\mathcal{V}}(t) \stackrel{\text{def}}{=} \begin{cases} \langle S \rangle & \text{if } t \equiv x \text{ and } \langle x{:}S \rangle \in \mathcal{V}, \\ \langle S \rangle & \text{if } t \equiv n,\ n{:} \to S \in Sig.Fun \cup Sig.Map \text{ or in constructors.} \\ \langle Pos, Nat, Int \rangle & \text{if } t \equiv \mathsf{Number}(n),\ n > 0 \\ \langle Nat, Int \rangle & \text{if } t \equiv \mathsf{Number}(0) \\ \langle Int \rangle & \text{if } t \equiv \mathsf{Number}(n),\ n < 0 \\ \langle Bool \rangle & \text{if } t \equiv \mathsf{True}() \text{ or } t \equiv \mathsf{False}() \\ & \quad \text{and for no } S' \not\equiv S\ n{:} \to S' \in Sig.Fun \cup Sig.Map, \\ S & \text{if } t \equiv n(t_1, \ldots, t_m), \\ & \quad n{:}sort_{Sig,\mathcal{V}}(t_1) \times \cdots \times sort_{Sig,\mathcal{V}}(t_m) \to S \in Sig.Fun \cup Sig.Map \\ & \quad \text{and for no } S' \not\equiv S\ n{:}sort_{Sig,\mathcal{V}}(t_1) \times \cdots \times sort_{Sig,\mathcal{V}}(t_m) \to \\ & \quad S' \in Sig.Fun \cup Sig.Map, \\ \perp & \text{otherwise.} \end{cases}$$

If a variable or a function is not or inappropriately declared no answer can be obtained. In this case $\perp$ results.

Definition C.6. Let Sig be a signature, $\mathcal{V}$ a set of variables over Sig and let t be a `Data-term`.

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

We call a `Data-term` t *closed* wrt. a signature Sig and a set of variables $\mathcal{V}$ iff $Var_{Sig,\mathcal{V}}(t) = \emptyset$. Note that $Var_{Sig,\mathcal{V}}(t) \subseteq \mathcal{V} \cup \{\perp\}$ for any `data-term` t. If $\perp \in Var_{Sig,\mathcal{V}}(t)$, then due to some missing or inappropriate declaration it can not be determined what the variables of t are on basis of Sig and $\mathcal{V}$.

C.3 Well-formed μCRL specifications

We define what well-formed specifications are. We only provide well-formed `Specifications` with a semantics. Well-formedness is a decidable property.

Definition C.7. Let Sig be a signature. We call a `Name` S a *constructor sort* iff $S \in Sig.Sort$ and there exists `Names` $S_1, \ldots, S_k, f$ $(k \geq 0)$ such that $f{:}S_1 \times \cdots \times S_k \to S \in Sig.Fun$.

Definition C.8. Let E be a `Specification` that is SSC. We inductively define which sorts are *non empty constructor sorts* in E. A constructor sort S is called *non empty* iff there is a function $f{:}S_1 \times \cdots \times S_k \to S \in Sig.Fun$ $(k \geq 0)$ such that for all $1 \leq i \leq k$ if S_i is a constructor sort, it is non empty. We say that E has *no empty constructor sorts* iff each constructor sort is non empty.

Definition C.9. Let E be a `Specification`. E is called *well-formed* iff

- E is SSC,
- E has no empty constructor sorts,
- There is no indirect set, bag, or list recursion. A=Set(B), B=Ref(A).
- There is no empty sort due to nonterminating struct recursion. C=struct(leaf(C),node(C,C))
- If $Time \in Sig(E).Sort$, then $\mathbf{0}{:} \to Time \in Sig(E).Fun \cup Sig(E).Map$ and $\leq{:}Time \times Time \to Bool \in Sig(E).Map$.

D ATerm representation format for MTLPSs

A MTLPS is stored as an ATerm with the following functions. The sort of stored MTLPS is *MTLPS*.

spec2gen : $DataTypes \times ActionSpec^* \times InitProcSpec \rightarrow MTLPS$

actspec : $String \times String^* \rightarrow ActionSpec$

initprocspec : $TermAppl \times Variable^* \times Summand^* \rightarrow InitProcSpec$

smd : $Variable^* \times Action^* \times Time \times IndexedTerm^* \times TermAppl \rightarrow Summand$

act : $String \times TermAppl \rightarrow Action$

time : $TermAppl \rightarrow Time$

notime :$\rightarrow Time$

it : $Nat \times TermAppl \rightarrow IndexedTerm$

dc : $Nat \rightarrow IndexedTerm$

d : $Signature \times Equation^* \rightarrow DataTypes$

e : $Variable^* \times TermAppl \times TermAppl \times TermAppl \rightarrow Equation$

v : $String \times String \rightarrow Variable$

s : $String^* \times Function^* \times Function^* \rightarrow Signature$

f : $String \times String^* \times String \rightarrow Function$

The sort *TermAppl* consists of ATerm terms of the form $TermAppl(f, t)$ or constant/variable symbols. The sort *String* consists of quoted constants, i.e. function symbols of arity 0. The sort *Nat* is the built-in sort of natural numbers in the ATerm library. The list of elements of sort D is denoted by D^*.

The constructor of sort *InitProcSpec* contains the actual LPS parameters (from `init`) as the first parameter, the formal LPS parameters as the second argument, and the list of summands as the third parameter. The third parameter of smd is the term of sort *Time* representing the time at which the multiaction happens, or notime, indicating that no time info is given. The last parameter of smd is the boolean term representing the condition.

The second parameter of e is the boolean condition used for conditional term rewriting.

The first parameter of v is the variable name, appended with '#'. The first parameter of f is the function name, appended with its parameter types list, separated by '#' (for constants only '#' is appended).

If the delta summand of the TLPS is present, δ has to be represented by the ATerm string `"Delta"`, and actions with this name should not be allowed. An alternative is in using a special summand construction.

E ATerm representation format for LPSs (for μCRL v1)

An LPS is stored as an ATerm with the following functions. The sort of stored LPS is LPS.

spec2gen : $\mathit{DataTypes} \times \mathit{InitProcSpec} \rightarrow \mathit{LPS}$

initprocspec : $\mathit{Term}^* \times \mathit{Variable}^* \times \mathit{Summand}^* \rightarrow \mathit{InitProcSpec}$

smd : $\mathit{Variable}^* \times \mathit{String} \times \mathit{Term}^* \times \mathit{NextState} \times \mathit{Term} \rightarrow \mathit{Summand}$

terminated :$\rightarrow \mathit{NextState}$

i : $\mathit{Term}^* \rightarrow \mathit{NextState}$

d : $\mathit{Signature} \times \mathit{Equation}^* \rightarrow \mathit{DataTypes}$

e : $\mathit{Variable}^* \times \mathit{Term} \times \mathit{Term} \rightarrow \mathit{Equation}$

v : $\mathit{String} \times \mathit{String} \rightarrow \mathit{Variable}$

s : $\mathit{String}^* \times \mathit{Function}^* \times \mathit{Function}^* \rightarrow \mathit{Signature}$

f : $\mathit{String} \times \mathit{String}^* \times \mathit{String} \rightarrow \mathit{Function}$

The sort *Term* consists of arbitrary ATerm terms where all function symbols must be quoted. The sort *String* consists of quoted constants, i.e. function symbols of arity 0. The list of elements of sort D is denoted by D^*.

The first parameter of v is the variable name, appended with '#'. The first parameter of f is the function name, appended with its parameter types list, separated by '#' (for constants only '#' is appended).

The constructor of sort *InitProcSpec* contains the actual LPS parameters (from `init`) as the first parameter, the formal LPS parameters as the second argument, and the list of summands as the third parameter. The last parameter of cmd is the boolean term representing the condition.

F ATerm representation format for input muCRL (for μCRL v1)

An LPS is stored as an ATerm with the following functions. The sort of stored LPS is *LPS*.

spec2gen : $\mathit{DataTypes} \times \mathit{InitProcSpec} \to \mathit{LPS}$

initprocspec : $\mathit{Term}^* \times \mathit{Variable}^* \times \mathit{Summand}^* \to \mathit{InitProcSpec}$

smd : $\mathit{Variable}^* \times \mathit{String} \times \mathit{Term}^* \times \mathit{NextState} \times \mathit{Term} \to \mathit{Summand}$

terminated :$\to \mathit{NextState}$

i : $\mathit{Term}^* \to \mathit{NextState}$

d : $\mathit{Signature} \times \mathit{Equation}^* \to \mathit{DataTypes}$

e : $\mathit{Variable}^* \times \mathit{Term} \times \mathit{Term} \to \mathit{Equation}$

v : $\mathit{String} \times \mathit{String} \to \mathit{Variable}$

s : $\mathit{String}^* \times \mathit{Function}^* \times \mathit{Function}^* \to \mathit{Signature}$

f : $\mathit{String} \times \mathit{String}^* \times \mathit{String} \to \mathit{Function}$

The sort *Term* consists of arbitrary ATerm terms where all function symbols must be quoted. The sort *String* consists of quoted constants, i.e. function symbols of arity 0. The list of elements of sort D is denoted by D^*.

The first parameter of v is the variable name, appended with '#'. The first parameter of f is the function name, appended with its parameter types list, separated by '#' (for constants only '#' is appended).

The constructor of sort *InitProcSpec* contains the actual LPS parameters (from `init`) as the first parameter, the formal LPS parameters as the second argument, and the list of summands as the third parameter. The last parameter of cmd is the boolean term representing the condition.