1 LPS definitions

The equation below represents a Linear Process Equation for \( \mu \)CRL with multi-actions and time (MTLPS).

\[
X(d; D) = \sum_{i \in I} \sum_{e_i \in E_i} c_i(d, e_i) \rightarrow a_i^0(f_i,0(d, e_i)) \cdot \cdots \cdot a_i^{n(i)}(f_i,n(i)(d, e_i)) \cdot t_i(d, e_i) \cdot X_i(g_i(d, e_i))
\]

\[
+ \sum_{j \in J} \sum_{e_j \in E_j} c_j(d, e_j) \rightarrow a_j^0(f_j,0(d, e_j)) \cdot \cdots \cdot a_j^{n(j)}(f_j,n(j)(d, e_j)) \cdot t_j(d, e_j)
\]

\[
+ \sum_{e_d \in E_d} c_d(d, e_d) \rightarrow \delta \cdot t_d(d, e_d)
\]

where \( I \) and \( J \) are disjoint.
It is possible to translate multiactions to regular \( \mu \)CRL 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.

\[
X(d; D) = \sum_{i \in I} \sum_{c_i \rightarrow e_i} c_i(d, e_i) \rightarrow a_i^0 \cdot a_i^1 \cdot \ldots \cdot a_i^{n(i)} \cdot (f_{i,0}(d, e_i), \ldots, f_{i,n(i)}(d, e_i)) \cdot t_i(d, e_i) \cdot X_i(\overrightarrow{g_i}(d, e_i)) \\
+ \sum_{j \in J} \sum_{c_j \rightarrow e_j} c_j(d, e_j) \rightarrow a_j^0 \cdot a_j^1 \cdot \ldots \cdot a_j^{n(j)} \cdot (f_{j,0}(d, e_j), \ldots, f_{j,n(j)}(d, e_j)) \cdot t_j(d, e_j) \\
+ \sum_{c_\delta \rightarrow e_\delta} c_\delta(d, e_\delta) \rightarrow \delta \cdot t_\delta(d, e_\delta)
\]

where \( I \) and \( J \) are disjoint, and \( a_i^0, a_i^1, \ldots, a_i^{n(i)} \) and \( a_j^0, a_j^1, \ldots, a_j^{n(j)} \) 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,
\[ \text{TLPS(MTLPS1)} = (\text{timed mcrl}) = \text{TLPS(MTLPS2)} \].

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 \( \text{Time} \) 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 \( \text{spec}(\text{sortspec}?, \text{opspec}?, \text{eqlspec}?, \text{actspec}?, \text{prosprec}?, \text{init}) \) (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 \( \text{spec}(\text{init}(\tau())) \)). 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 \( \text{Sig} \) be a signature and \( \mathbf{V} \) be a set of variables over \( \text{Sig} \). We define the predicate ‘is SSC wrt. \( \text{Sig} \)’ inductively over the syntax of a Specification.

Sorts Sort declarations:

- A SortDecl \( \text{SortDecl}[[n_1 \cdots n_m]] \) with \( m \geq 1 \) is SSC wrt. \( \text{Sig} \) iff
  - all \( sspec_1, \ldots, sspec_m \) are SSC wrt. \( \text{Sig} \).
  - Defined sort names are different: for all \( i < j \), \( \text{defined}_\text{sorts}(sspec_i) \neq \text{defined}_\text{sorts}(sspec_j) \).

- A SortDecl \( \text{SortDeclRef}(n, s) \) with \( m \geq 1 \) is SSC wrt. \( \text{Sig} \) iff the sort expression \( s \) is SSC w.r.t \( \text{Sig} \setminus [n_1 \cdots n_m] \). Here we note that no recursive sort references are allowed.
• A `SortDecl` `SortDeclStruct(n, [cons₁,..., consₘ])` with \(m \geq 1\) is SSC wrt. `Sig` iff all constructor expressions `cons` are SSC w.r.t `Sig`.

• A `ConstDecl` `StructDeclCons(n, [proj₁,..., projₘ], 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` `StructDeclProj(n, Dom([s₁,..., sₘ]))` 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 wrt `Sig`.

Data types

• A `OpSpec` `ConsSpec([IdDecls([n₁₁,..., n₁ₙ₁], s₁),..., IdDecls([nₘ₁,..., nₘₙₘ], sₘ])` or `MapSpec([IdDecls([n₁₁,..., n₁ₙ₁], s₁),..., IdDecls([nₘ₁,..., nₘₙₘ], sₘ])]` 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 \(1 \leq i \leq m\) \(n_i₁,..., n_iₙ_i\) 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:

\[
\text{EqnSpec}([\text{IdDecls([n₁₁,..., n₁ₙ₁], s₁),..., IdDecls([nₘ₁,..., nₘₙₘ], sₘ])}, \text{EqnSpec}(d₁,d₁'),..., \text{EqnSpec}(dₖ,dₖ')])
\]

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 uniquely compatible (wrt. `Sig`).

Actions
A `ActSpec` of the form:

\[
\text{ActSpec}([\text{ActDecl([n₁₁,..., n₁ₙ₁], d₁),..., ActDecl([nₘ₁,..., nₘₙₘ], dₘ])})
\]

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([s₁,..., sₙ])`, and all `s_j` are SSC wrt. `Sig`.
Processes

- A ProcSpec ProcSpec([ProcDecl(n_1, vars_1, p_1), ..., ProcDecl(n_m, vars_m, p_m)]) with m ≥ 1 is SSC wrt. Sig iff
  - for each 1 ≤ i < j ≤ m: if type(vars_i) = type(vars_j), then n_i ≠ n_j,
  - none of them are in Sig.Fun ∪ Sig.Map ∪ Sig.Act.
- for each Name S' it holds that n:S_1 × ⋯ × S_k → S' ∉ Sig.Fun ∪ Sig.Map,
- the Names x_1, ..., x_k are pairwise different and \{⟨x_j:S_j⟩ | 1 ≤ j ≤ k\} is a set of variables over Sig,
- p is SSC wrt. Sig and \{⟨x_j:S_j⟩ | 1 ≤ j ≤ k\}.

- A Init of the form Init(p) is SSC wrt. Sig iff p SSC wrt. to Sig and ∅.

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 V be a set of variables over Sig. We say that a Process-term p is SSC wrt. to Sig and V iff one of the following hold:

- p ≡ p_1 + p_2, p ≡ p_1 \parallel p_2, p ≡ p_1 \parallel p_2, p ≡ p_1 | p_2, p ≡ p_1 \cdot p_2 or p ≡ p_1 ≪ p_2 and both p_1 and p_2 are SSC wrt. Sig and V,
- p ≡ p_1 \cdot t and
  - p_1 is SSC wrt. Sig and V,
  - t is SSC wrt. Sig and V and sort_{Sig,V}(t) = ⟨Bool⟩.
- p ≡ p_1 \cdot t and
  - p_1 is SSC wrt. Sig and V,
  - t is SSC wrt. Sig and V and sort_{Sig,V}(t) = Time.
- p ≡ δ or p ≡ τ.
- p ≡ \partial(n_1, ..., n_m) p_1 or p ≡ τ(n_1, ..., n_m) p_1 with m ≥ 1 and
  - for all 1 ≤ i ≠ j ≤ m n_i ∉ n_j,
  - for 1 ≤ i ≤ m, if n_i = n_{i,1} | ⋯ | n_{i,k}, then n_{i,j} ∈ Sig.ActNames.
  - p_1 is SSC wrt. Sig and V.
- p ≡ \nabla(n_1, ..., n_m) p_1 with m ≥ 1 and
  - for all 1 ≤ i < j ≤ m n_i ∉ n_j,
  - for 1 ≤ i ≤ m, if n_i = n_{i,1} | ⋯ | n_{i,k}, then n_{i,j} ∈ Sig.ActNames.
  - p_1 is SSC wrt. Sig and V.
- p ≡ ρ(n_1 → n'_1, ..., n_m → n'_m) p_1 and
  - for 1 ≤ i ≤ m both n_i, n'_i ∈ Sig.ActNames.
  - for each 1 ≤ i < j ≤ m it holds that n_i ≠ n_j,
– for $1 \leq i \leq m$, the types of $n_i$ and $n'_i$ are the same in $\text{Sig}$.
– $p_1$ is SSC wrt. $\text{Sig}$ and $\mathcal{V}$.

\[ p \equiv \Gamma_{(n_1 \rightarrow n'_1, \ldots, n_m \rightarrow n'_m)} p_1 \]

– for $1 \leq i \leq m$, if $n_i = n_{i,1} | \cdots | n_{i,k}$, then $n_{i,j} \in \text{Sig}.\text{ActNames}$.
– for $1 \leq i \leq m$ either $n'_i \in \text{Sig}.\text{ActNames}$ or $n'_i = \tau$.
– for each $1 \leq i \neq j \leq m$ it holds that $n_i \nsubseteq n_j$.
– for $1 \leq i \leq m$ it holds that, if $n_i = n_{i,1} | \cdots | n_{i,k}$, then the types of all $n_{i,j}$ and $n'_i$ are the same in $\text{Sig}$.
– $p_1$ is SSC wrt. $\text{Sig}$ and $\mathcal{V}$.

- $p \equiv \Sigma_{x:S} p_1$ and iff
- $(\forall \{\langle x:S' \rangle \mid S' \text{ is a Name} \}) \cup \{\langle x:S \rangle \}$ is a set of variables over $\text{Sig}$.
- $p_1$ is SSC wrt. $\text{Sig}$ and $(\forall \{\langle x:S' \rangle \mid S' \text{ is a Name} \}) \cup \{\langle x:S \rangle \}$.

- $p \equiv n$ and $n = p' \in \text{Sig}.\text{Proc}$ for some Process-term $p'$, or $n \in \text{Sig}.\text{Act}$.
- $p \equiv n(t_1, \ldots, t_m)$ with $m \geq 1$ and
  - $n(x_1: \text{sort}_{\text{Sig},\mathcal{V}}(t_1), \ldots, x_m: \text{sort}_{\text{Sig},\mathcal{V}}(t_m)) = p' \in \text{Sig}.\text{Proc}$ for Names $x_1, \ldots, x_m$ and Process-term $p'$, or $n: \text{sort}_{\text{Sig},\mathcal{V}}(t_1) \times \cdots \times \text{sort}_{\text{Sig},\mathcal{V}}(t_m) \in \text{Sig}.\text{Act}$.
  - for $1 \leq i \leq m$ the Data-term $t_i$ is SSC wrt. $\text{Sig}$ and $\mathcal{V}$.

\section*{Sort expressions}

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

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

\section*{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 $\kappa$, we denote the static context as $\text{Sig}(\kappa)$ and the dynamic part as $\text{Vars}(\kappa)$. The static context is a tuple

\[ (\text{BasicSorts}, \text{DefinedSorts}, \text{Operations}, \text{Actions}, \text{Processes}) \]
which represents the names and types of the sorts, operations, actions and processes defined in the specification. The types of the context operatings are defined below:

\[
\text{BasicSorts} = \{ \text{String} \} \\
\text{DefinedSorts : String} \rightarrow \text{Type} \\
\text{Operations} \in \text{String} \times \text{Type} \\
\text{Actions} \in \text{String} \times \text{Type} \\
\text{Processes} : \text{String} \rightarrow \text{Type}
\]

The sort \text{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 \text{basicType} : \text{Type} \rightarrow \text{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 \text{Variables} : \text{String} \rightarrow \text{Type}.

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

- \( t \equiv n \) with \( n \) a Name and \( \langle n:S \rangle \in \text{V} \) for some \( S \), or \( n \rightarrow \text{sort}_{\text{Sig}, \text{V}}(n) \in \text{Sig.Fun} \cup \text{Sig.Map} \).
- \( t \equiv n(t_1, \ldots, t_m) \) (\( m \geq 1 \)) and \( n: \text{sort}_{\text{Sig}, \text{V}}(t_1) \times \cdots \times \text{sort}_{\text{Sig}, \text{V}}(t_m) \rightarrow \text{sort}_{\text{Sig}, \text{V}}(n(t_1, \ldots, t_m)) \in \text{Sig.Fun} \cup \text{Sig.Map} \) and all \( t_i \) (\( 1 \leq i \leq m \)) are SSC wrt. \text{Sig} and \text{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: \text{is.well_named} – all ids are defined, \text{id.vars} – all possible types the term can be, \text{is.well_typed} – is the term well-typed?

The function \( \text{T}_n \) is defined defined as (well-namedness of the arguments is assumed):

<table>
<thead>
<tr>
<th>DataVar(String)</th>
<th>type(k, String)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpId(String)</td>
<td>type(k, String)</td>
</tr>
<tr>
<td>Number(NumberString)</td>
<td>PNI(NumberString)</td>
</tr>
<tr>
<td>ListEnum(d_0, \ldots, d_n)</td>
<td>List(\text{minC}(\text{T}(d_0), \ldots, \text{T}(d_n)))</td>
</tr>
<tr>
<td>SetEnum(d_0, \ldots, d_n)</td>
<td>Set(\text{minC}(\text{T}(d_0), \ldots, \text{T}(d_n)))</td>
</tr>
<tr>
<td>BagEnum(BagEnumElmt(d_0, d'_n), \ldots, BagEnumElmt(d_n, d'_m))</td>
<td>Bag(\text{minC}(\text{T}(d_0), \ldots, \text{T}(d_n)))</td>
</tr>
<tr>
<td>SetBagComp(IdDecl(v, s), d)</td>
<td>Set(s) if \text{T}_n(d) = \text{Bool}</td>
</tr>
<tr>
<td>DataApp(d, d_0, \ldots, d_n)</td>
<td>Set(s) if \text{T}_n(d) = \text{Bool}</td>
</tr>
<tr>
<td>Forall([IdDecl(v_0, s_0)], \ldots, [IdDecl(v_n, s_n)], d)</td>
<td>\text{B}</td>
</tr>
<tr>
<td>Exists([IdDecl(v_0, s_0)], \ldots, [IdDecl(v_n, s_n)], d)</td>
<td>\text{Bool}</td>
</tr>
<tr>
<td>Lambda([IdDecl(v_0, s_0)], \ldots, [IdDecl(v_n, s_n)], d)</td>
<td>\text{len}(v_0), \ldots, \text{len}(v_n) \rightarrow \text{T}_n(d)</td>
</tr>
<tr>
<td>Whr(d, [v_0, d_0, \ldots, v_n, d_n])</td>
<td>\text{T}_n(d)</td>
</tr>
</tbody>
</table>

7
The following internal, or system, identifiers have the corresponding (polymorphic) types:

- **EmptyList()**
  - Type is `List(TypeAny)`

- **EmptySetBag()**
  - Type is `SB(TypeAny)`

- **NotOrCompl(d)**
  - Type is ` Bool ∨ T(d) ≡_t SB(TypeAny)`

- **Neg(d)**
  - Type is ` T(d) ≡_t PNI`

- **Size(d)**
  - Type is ` T(d) ≡_t LSB(TypeAny)`

- **ListAt(d, d')**
  - Type is `T(d) ≡_t List(TypeAny) ∧ T(d') ≡_t PN`

- **Div(d, d')**
  - Type is `T(d) ≡_t PNI ≡ T(d')`

- **Mod(d, d')**
  - Type is `T(d) ≡_t PNI ∧ T(d') ≡_t PPos`

- **AddOrUnion(d, d')**
  - Type is `maxMoI(T(d), T(d'))`

- **SubtOrDiff(d, d')**
  - Type is `Bool`

- **LTOrPropSubset(d, d')**
  - Type is `Bool`

- **GTOrPropSupset(d, d')**
  - Type is `Bool`

- **LTEOrSubset(d, d')**
  - Type is `Bool`

- **GEOrSupSet(d, d')**
  - Type is `Bool`

- **ln(d, d')**
  - Type is `LSB(T(d)) ≡_t T(d')`

- **Cons(d, d')**
  - Type is `List(T(d')) ≡_t T(d')`

- **Snoc(d, d')**
  - Type is `List(T(d')) ≡ T(d)`

- **Concat(d, d')**
  - Type is `List(T(d')) ≡_t List(TypeAny) ≡_t T(d')`

- **EqNeq(d, d')**
  - Type is `T(d) ≡_t T(d')`

- **True()**
  - Type is `Bool`

- **False()**
  - Type is `Bool`

- **Imp(d, d')**
  - Type is `T(d) ≡_t T(d') = Bool`

- **And(d, d')**
  - Type is `T(d) ≡_t T(d') = Bool`

### C.1 The signature of a specification

**Definition C.1.** The signature $\text{Sig}(E)$ of a specification $E$ consists of a seven-tuple

$$(\text{Sort, Fun, Map, Act, Comm, Proc, Init})$$

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

- If $E \equiv \text{sort } n_1 \cdots n_m$ with $m \geq 1$, then $\text{Sig}(E) \overset{\text{def}}{=} \{\{n_1, \ldots, n_m\}, 0, 0, 0, 0, 0, 0\}$.

- If $E \equiv : fd_1 \rightarrow \cdots \rightarrow fd_m$ with $m \geq 1$, then $\text{Sig}(E) \overset{\text{def}}{=} \{0, \text{Fun}, 0, 0, 0, 0, 0\}$, where

  $\text{Fun} \overset{\text{def}}{=} \{n_{ij}: \rightarrow S_i | fd_i \equiv n_{i1}, \ldots, n_{il}; \rightarrow S_i, 1 \leq i \leq m, 1 \leq j \leq l_i\} \cup \{n_{ij}: S_{i1} \times \cdots \times S_{ik_i} \rightarrow S_i | fd_i \equiv n_{i1}, \ldots, n_{il}; S_{i1} \times \cdots \times S_{ik_i} \rightarrow S_i, 1 \leq i \leq m, 1 \leq j \leq l_i\}$.

- If $E \equiv \text{map } md_1 \cdots md_m$ with $m \geq 1$, then $\text{Sig}(E) \overset{\text{def}}{=} \{0, 0, \text{Map}, 0, 0, 0, 0\}$, where

  $\text{Map} \overset{\text{def}}{=} \{n_{ij}: \rightarrow S_i | md_i \equiv n_{i1}, \ldots, n_{il}; \rightarrow S_i, 1 \leq i \leq m, 1 \leq j \leq l_i\} \cup \{n_{ij}: S_{i1} \times \cdots \times S_{ik_i} \rightarrow S_i | md_i \equiv n_{i1}, \ldots, n_{il}; S_{i1} \times \cdots \times S_{ik_i} \rightarrow S_i, 1 \leq i \leq m, 1 \leq j \leq l_i\}$. 

8
If $E$ is a Equation-specification, then $\text{Sig}(E) \eqdef (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$.

If $E \equiv ad_1 \cdots ad_m$ with $m \geq 1$, then $\text{Sig}(E) \eqdef (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \{ad_i \mid 1 \leq i \leq m\}, \emptyset, \emptyset)$, where

\[
\begin{align*}
\text{Act} & \eqdef \{n_i \mid ad_i \equiv n_i, 1 \leq i \leq m\} \\
& \cup \{n_i; S_{i1} \times \cdots \times S_{ik_i} \mid ad_i \equiv n_{i1}, \ldots, n_{ik_i}; S_{i1} \times \cdots \times S_{ik_i}, 1 \leq i \leq m, 1 \leq j \leq l_i\}.
\end{align*}
\]

If $E \equiv \text{comm} cd_1 \cdots cd_m$ with $m \geq 1$, then $\text{Sig}(E) \eqdef (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \{cd_i \mid 1 \leq i \leq m\}, \emptyset, \emptyset)$.

If $E \equiv \text{proc} pd_1 \cdots pd_m$ with $m \geq 1$, then $\text{Sig}(E) \eqdef (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \{pd_i \mid 1 \leq i \leq m\}, \emptyset)$.

If $E \equiv E_1 \ E_2$ with $\text{Sig}(E_i) = (\text{Sort}_i, \text{Fun}_i, \text{Map}_i, \text{Act}_i, \text{Comm}_i, \text{Proc}_i, \text{Init}_i)$ for $i = 1, 2$, then

\[
\text{Sig}(E) \eqdef (\text{Sort}_1 \cup \text{Sort}_2, \text{Fun}_1 \cup \text{Fun}_2, \text{Map}_1 \cup \text{Map}_2, \\
\text{Act}_1 \cup \text{Act}_2, \text{Comm}_1 \cup \text{Comm}_2, \text{Proc}_1 \cup \text{Proc}_2, \text{Init}_1 \cup \text{Init}_2).
\]

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

\[
\begin{align*}
\text{Sig.Sort} & \text{ for Sort, } \\
\text{Sig.Fun} & \text{ for Fun, } \\
\text{Sig.Map} & \text{ for Map, } \\
\text{Sig.Act} & \text{ for Act, } \\
\text{Sig.Comm} & \text{ for Comm, } \\
\text{Sig.Proc} & \text{ for Proc, } \\
\text{Sig.Init} & \text{ for Init.}
\end{align*}
\]

**C.2 Variables**

Variables play an important role in specifications. The next definition says given a specification $E$, which elements from $\text{Name}$ 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 $\text{Sig}$ be a signature. A set $\mathcal{V}$ containing pairs $\langle x:S \rangle$ with $x$ and $S$ from $\text{Name}$, is called a set of variables over $\text{Sig}$ iff for each $\langle x:S \rangle \in \mathcal{V}$:

- for each $\text{Name} S'$ and Process-term $p$ it holds that $x: \to S' \notin \text{Sig.Fun} \cup \text{Sig.Map}, x \notin \text{Sig.Act}$ and $x = p \notin \text{Sig.Proc},$
- $S \in \text{Sig.Sort}$, and
- for each $\text{Name} S'$ such that $S' \neq S$ it holds that $\langle x:S' \rangle \notin \mathcal{V}$.

**Definition C.4.** Let $vd$ be a Variable-declaration. The function $\text{Vars}$ is defined by:

\[
\text{Vars}(vd) \eqdef \begin{cases} \\
\emptyset & \text{if } vd \text{ is empty,} \\
\{\langle x_i:S_i \rangle \mid 1 \leq i \leq m, 1 \leq j \leq l_i\} & \text{if } vd \equiv \text{var} \ x_{i1}, \ldots, x_{il_i}; S_1 \cdots x_{m1}, \ldots, x_{ml_m}; S_m.
\end{cases}
\]

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

9
Definition C.5. Let $t$ be a data-term and $\text{Sig}$ a signature. Let $\mathcal{V}$ be a set of variables over $\text{Sig}$. We define:

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

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

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

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

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

### C.3 Well-formed $\mu$CRL 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 $\text{Sig}$ be a signature. We call a Name $S$ a constructor sort iff $S \in \text{Sig.Sort}$ and there exists Names $S_1, \ldots, S_k, f$ ($k \geq 0$) such that $f: S_1 \times \cdots \times S_k \to S \in \text{Sig.Sorts}$.

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 $f: S_1 \times \cdots \times S_k \to S \in \text{Sig.Fun}$ ($k \geq 0$) 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. $A = \text{Set}(B)$, $B = \text{Ref}(A)$,
- There is no empty sort due to nonterminating struct recursion. $C = \text{struct}(\text{leaf}(C), \text{node}(C, C))$
- If $\text{Time} \in \text{Sig}(E).\text{Sort}$, then $0: \to \text{Time} \in \text{Sig}(E).\text{Fun} \cup \text{Sig}(E).\text{Map}$ and $\leq : \text{Time} \times \text{Time} \to \text{Bool} \in \text{Sig}(E).\text{Map}$.

10
D A Term representation format for MTLPSs

A MTLPS is stored as an A Term with the following functions. The sort of stored MTLPS is \text{MTLPS}.

\[
\begin{align*}
\text{spec2gen} & : \text{DataTypes} \times \text{ActionSpec}^* \times \text{InitProcSpec} \rightarrow \text{MTLPS} \\
\text{actspec} & : \text{String} \times \text{String}^* \rightarrow \text{ActionSpec} \\
\text{initprocspec} & : \text{TermAppl} \times \text{Variable}^* \times \text{Summand}^* \rightarrow \text{InitProcSpec} \\
\text{smd} & : \text{Variable}^* \times \text{Action}^* \times \text{Time} \times \text{IndexedTerm}^* \times \text{TermAppl} \rightarrow \text{Summand} \\
\text{act} & : \text{String} \times \text{TermAppl} \rightarrow \text{Action} \\
\text{time} & : \text{TermAppl} \rightarrow \text{Time} \\
\text{notime} & : \rightarrow \text{Time} \\
\text{it} & : \text{Nat} \times \text{TermAppl} \rightarrow \text{IndexedTerm} \\
\text{dc} & : \text{Nat} \rightarrow \text{IndexedTerm} \\
\text{d} & : \text{Signature} \times \text{Equation}^* \rightarrow \text{DataTypes} \\
\text{e} & : \text{Variable}^* \times \text{TermAppl} \times \text{TermAppl} \times \text{TermAppl} \rightarrow \text{Equation} \\
\text{v} & : \text{String} \times \text{String} \rightarrow \text{Variable} \\
\text{s} & : \text{String}^* \times \text{Function}^* \times \text{Function}^* \rightarrow \text{Signature} \\
\text{f} & : \text{String} \times \text{String}^* \times \text{String} \rightarrow \text{Function}
\end{align*}
\]

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

The constructor of sort \text{InitProcSpec} contains the actual LPS parameters (from \text{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 \text{smd} is the term of sort \text{Time} representing the time at which the multiaction happens, or \text{notime}, indicating that no time info is given. The last parameter of \text{smd} is the boolean term representing the condition.

The second parameter of \text{e} is the boolean condition used for conditional term rewriting.

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

If the delta summand of the TLPS is present, \text{δ} has to be represented by the A Term string \text{"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 $\mu$CRL v1)

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

\[ spec2gen : DataTypes \times InitProcSpec \rightarrow LPS \]
\[ initprocspec : Term^* \times Variable^* \times Summand^* \rightarrow InitProcSpec \]
\[ smd : Variable^* \times String \times Term^* \times NextState \times Term \rightarrow Summand \]
\[ terminated : \rightarrow NextState \]
\[ i : Term^* \rightarrow NextState \]
\[ d : Signature \times Equation^* \rightarrow DataTypes \]
\[ e : Variable^* \times Term \times Term \rightarrow Equation \]
\[ v : String \times String \rightarrow Variable \]
\[ s : String^* \times Function^* \times Function^* \rightarrow Signature \]
\[ f : String \times String^* \times String \rightarrow Function \]

The sort $Term$ 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 $\text{init}$) 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.
**A Term representation format for input muCRL (for \(\mu\text{CRL} v1\))**

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

- **spec2gen**: \(DataTypes \times InitProcSpec \rightarrow LPS\)
- **initprocspec**: \(Term^* \times Variable^* \times Summand^* \rightarrow InitProcSpec\)
- **smd**: \(Variable^* \times String \times Term^* \times NextState \times Term \rightarrow Summand\)
- **terminated**: \(\rightarrow NextState\)
- **i**: \(Term^* \rightarrow NextState\)
- **d**: \(Signature \times Equation^* \rightarrow DataTypes\)
- **e**: \(Variable^* \times Term \times Term \rightarrow Equation\)
- **v**: \(String \times String \rightarrow Variable\)
- **s**: \(String^* \times Function^* \times Function^* \rightarrow Signature\)
- **f**: \(String \times String^* \times String \rightarrow Function\)

The sort \(Term\) 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 \(\text{init}\)) 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.