Linearization of μCRL specifications

Y.S. Usenko

Report SEN-R0209 May 31, 2002

ISSN 1386-369X

Linearization of μCRL Specifications

Yaroslav S. Usenko
Yaroslav.Usenko@cwi.nl

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
We describe a linearization algorithm for μCRL processes, similar to the one described in [21] for a subset of the language called parallel pCRL. This algorithm finds its roots in formal language theory: the 'grammar' defining a process is transformed into a variant of Greibach Normal Form. Next, any such form is further reduced to *linear form*, i.e., to an equation that resembles a right-linear, data-parametric grammar. From the other perspective, linear specifications in μCRL resemble symbolic representations of transition systems, that can be further transformed and analyzed by many of the existing tools and techniques. We aim at proving the correctness of this linearization algorithm. To this end we use an equivalence relation on recursive specifications in μCRL that is model independent and does not involve an explicit notion of solution.

2000 Mathematics Subject Classification: 68Q10; 68Q42; 68Q65; 68Q85
2000 ACM Computing Classification System: D.2.1; D.2.4; D.3.1; D.3.3; F.3.2; I.1.1
Keywords and Phrases: μCRL, Process Algebra, Linearization of Recursive Specifications, Symbolic Representation of Transitions Systems, Program Transformation.

1. Introduction

In this paper we address the issue of linearization of recursive specifications in the specification language μCRL (micro Common Representation Language, [20, 16]) and extend the existing linearization techniques for a subset of the μCRL called parallel pCRL [21] to the full μCRL setting. The language μCRL has been developed under the assumption that an extensive and mathematically precise study of the basic constructs of specification languages is fundamental to an analytical approach of much richer (and more complicated) specification languages such as SDL [38], LOTOS [26], PSF [28, 29] and CRL [35]. Moreover, it is assumed that μCRL and its proof theory provide a solid basis for the design and construction of tools for analysis and manipulation of distributed systems.

The language μCRL offers a uniform framework for the specification of data and processes. Data are specified by equational specifications: one can declare sorts and functions working upon these sorts, and describe the meaning of these functions by equational axioms. Processes are described in process algebraic style, where the particular process syntax stems from ACP [4, 2, 15], extended with data-parametric ingredients: there are constructs for conditional composition, and for data-parametric choice and communication. As is common in process algebra, infinite processes are specified by means of (finite systems of) recursive equations. In μCRL such equations can also be data-parametric. As an example, for action a and adopting standard semantics for μCRL, each solution for the equation $\mathsf{X} = \mathsf{a} \cdot \mathsf{X}$ specifies (or "identifies") the process that can only repeatedly execute a, and so does each solution for $\mathsf{Y}(17)$ where $\mathsf{Y}(n)$ is defined by the data-parametric equation $\mathsf{Y}(n) = \mathsf{a} \cdot \mathsf{Y}(n+1)$ with $n \in Nat$. An interesting subclass of systems of recursive equations consists of those that contain only one *linear* equation. Such a system is called an LPE (Linear Process Equation). Here, linearity refers both to the form of recursion allowed, and to a restriction on the process operations allowed. The above examples $\mathsf{X} = \mathsf{a} \cdot \mathsf{X}$ and $\mathsf{Y}(n) = \mathsf{a} \cdot \mathsf{Y}(n+1)$ are both LPEs. The restriction to LPE format still yields an expressive setting (for example, it is not hard to show that each computable process over a finite set of actions can be simply defined using an LPE containing only computable functions over the natural numbers, cf. [32]). Moreover, in the design and construction of tools for μCRL, LPEs establish a basic and convenient representation format, that can be seen as symbolic representation of labelled transition systems. This applies, for example, to tools for generation of transition systems, or tools for optimization, deadlock checking, or simulation [8], all of which are based on term rewriting. However, the real potential of the LPE format is in symbolic techniques that enable analysis of large or infinite systems. Some of these are based on equational theorem prover [31], invariants [7], "cones and foci" method [23], or confluence reduction [9].

The LPE format stems from [7], in which the notion of a *process operator* is distinguished, and a proof technique for dealing with convergent LPEs is defined. Furthermore, there is a strong resemblance between LPEs and specifications in UNITY [12, 10]. The restriction to linear systems has a long tradition in process algebra. For instance, restricting to so-called linear *specifications*, i.e., linear systems that in some distinguished model have a unique solution per variable, various completeness results were proved in a simple fashion (cf. [30, 5]). However, without data-parametric constructs for process specification, the expressiveness is limited: only regular processes can be defined.

The language μCRL is considered to be a specification language because it contains ingredients that facilitate in a straightforward, natural way the modeling of distributed, communicating processes. In particular, it contains constructs for *parallelism*, *encapsulation* and *abstraction*. On the other hand, as mentioned above, LPEs constitute a basic fragment of μCRL in terms of expressiveness and tool support. This explains our interest in transforming any system of μCRL equations into an equivalent LPE, i.e., our interest to *linearize* μCRL process definitions.

We define the linearization algorithm on an abstract level, but in a very detailed manner. We do not concern ourselves with the question if and in what way systems of recursive equations over μCRL define processes as their unique solutions (per variable). Instead, we argue that the transformation is correct in a more general sense: we show that linearization "preserves all solutions". This means that if a particular μCRL system of recursive equations defines a series of solutions for its variables in some model, then the LPE resulting from linearization has (at least) the same solutions for the associated process terms. Consequently, if the resulting LPE is such that one can infer that these solutions are *unique* in some particular (process) model, then both systems define the same processes in that model. In our algorithm, most transformation steps satisfy a stronger property: the set of solutions is the same before and after the transformation. The presented linearization algorithm is developed with two additional goals in mind. We try to keep it optimal in terms of the size of generated LPE, briefly mentioning additional optimizations that could be applied. We also try to preserve the structure and the names of the initial specification as much as possible.

To the best of our knowledge, a first description of a transformation of (non-parallel) pCRL into an LPE like format was given in [6]. Transformation procedures from BPA to Greibach Normal Forms were outlined in [1] and presented in [25]. The implications and equivalences of regular systems of recursive equations and recursive program schemes w.r.t. their full sets of solutions were extensively studied by Courcelle in [13, 14] and Benson and Guessarian in [3]. The definitions in these papers have a lot in common with our approach, but they could not be directly applied to the μCRL setting.

Structure of the paper. In Section 2 we discuss the language μCRL. Furthermore, we define implication and equivalence between μCRL process terms defined over different μCRL specifications. Sections 3, 4 and 5 fully describe the linearization procedure. In Section 3 we describe in detail the first part of this transformation, which yields process definitions in so-called parallel extended Greibach normal form. In Section 4 we describe the transformation from parallel extended Greibach normal form into one equation which is quite similar to an LPE. Then, in Section 5 we introduce a special data type which is a list of multisets nested to an arbitrary depth, and explain how with the help of this data type we can achieve the LPE form. Section 6 contains some conclusions, comments on possible optimizations of our transformation, and identifies directions for future work. Appendices A and B contain detailed descriptions of the resulting LPEs that involve renaming operations of μCRL. Appendix C contains the full source code listing of the data type definitions used in Section 5.

2. Description of μCRL

In this section we first recall some general information about μCRL. Then we consider (recursive) process definitions in detail, and define various notions of equivalence, among which equivalence between process terms defined over different systems of equations. Next, we shortly discuss guardedness in process definitions. Finally, we introduce the notion of μCRL specifications and the formulation of the linearization problem.

2.1 Theory of μCRL

First we define the signature and axioms for booleans which are quite standard and can be found for instance in [11] (page 116). We use equational logic to prove boolean identities. Booleans are obligatory in any μCRL specification.

Definition 2.1. The signature of *Bool* consists of constants $\mathbf{t}, \mathbf{f}$, unary operation *not* and binary operations *and*, *or*, *eq*.

Note (Booleans). We use infix notation $\neg, \wedge, \vee, \leftrightarrow$ for *not*, *and*, *or*, *eq* respectively.

Definition 2.2. The axioms of *Bool* are the ones presented in Table 1.

$$\begin{array}{rclcrcl}
x \wedge y &=& y \wedge x & \qquad & x \vee y &=& y \vee x \\
(x \wedge y) \wedge z &=& x \wedge (y \wedge z) & & (x \vee y) \vee z &=& x \vee (y \vee z) \\
x \wedge x &=& x & & x \vee x &=& x \\
x \wedge (x \vee y) &=& x & & x \vee (x \wedge y) &=& x \\
(x \wedge y) \vee (x \wedge z) &=& x \wedge (y \vee z) & & (x \vee y) \wedge (x \vee z) &=& x \vee (y \wedge z) \\
x \wedge \mathbf{f} &=& \mathbf{f} & & x \vee \mathbf{t} &=& \mathbf{t} \\
x \wedge \neg x &=& \mathbf{f} & & x \vee \neg x &=& \mathbf{t} \\
x \leftrightarrow y &=& (x \wedge y) \vee (\neg x \wedge \neg y) & & & &
\end{array}$$

Table 1: Axioms of *Bool*.

Next we define the generalized equational theory of μCRL by defining its signature and the axioms. The axioms are taken from, or inspired by [18, 19].

Note (Vector Notation). Tuples occur a lot in the language, so we use a vector notation for them. Expression $\overrightarrow{d}$ is an abbreviation for $d^1, \ldots, d^n$, where d^k are data variables. Similarly, if type information is given, $\overrightarrow{d{:}D}$ is an abbreviation for $d^1{:}D^1, \ldots, d^n{:}D^n$ for some natural number n. In case $n = 0$ the whole vector vanishes as well as brackets (if any) surrounding it. For instance $\mathsf{a}(\overrightarrow{d})$ is an abbreviation for a in this case (here a is an *action*, this notion is introduced below). For all vectors $\overrightarrow{d}$ and $\overrightarrow{e}$ we have $\overrightarrow{d}, \overrightarrow{e} = \overrightarrow{d, e}$. Thus $\overrightarrow{d, e}$ is an abbreviation for $d^1, \ldots, d^n, e^1, \ldots, e^{n'}$. We also write $\overrightarrow{d{:}D}$ & $e{:}E$ for $d^1{:}D^1, \ldots, d^n{:}D^n, e{:}E$.

For any vector of variables $\overrightarrow{d}$, $\overrightarrow{f}(\overrightarrow{d})$ is an abbreviation for $f^1(\overrightarrow{d}), \ldots, f^m(\overrightarrow{d})$ for some $m \in \mathit{Nat}$ and $\overrightarrow{f} = f^1, \ldots, f^m$, where each $f^k(\overrightarrow{d})$ is a data term that may contain elements of $\overrightarrow{d}$ as free variables. As with vectors of variables, in case $m = 0$ the vector of data terms vanishes. We often use $\overrightarrow{t}$ to express a data term vector without explicitly denoting its variables.

Definition 2.3. The signature of μCRL consists of data sorts (or 'data types') including *Bool* as defined above, and a distinct sort *Proc* of processes. Each data sort D is assumed to be equipped with a binary function $eq : D \times D \to \mathit{Bool}$. (This requirement can be weakened by demanding such functions only for data sorts that are parameters of communicating actions). The operational signature of μCRL is parameterized by the finite set of action labels *ActLab* and a partial commutative and associative function $\gamma : \mathit{ActLab} \times \mathit{ActLab} \to \mathit{ActLab}$ such that $\gamma(\mathsf{a}_1, \mathsf{a}_2) \in \mathit{ActLab}$ implies that $\mathsf{a}_1, \mathsf{a}_2$ and $\gamma(\mathsf{a}_1, \mathsf{a}_2)$ have parameters of the same sorts. The process operations are the ones listed below:

- actions $\mathsf{a}(\overrightarrow{t})$ parameterized by data terms $\overrightarrow{t}$, where $\mathsf{a} \in \mathit{ActLab}$ is an action label. More precisely, a is an operation $\mathsf{a} : \overrightarrow{D_\mathsf{a}} \to \mathit{Proc}$. We write $\mathit{type}(\mathsf{a})$ for $\overrightarrow{D_\mathsf{a}}$.
- constants δ and τ of sort *Proc*.
- binary operations $+, \cdot, \|, \lfloor\!\lfloor, |$ defined on *Proc*, where $|$ is defined using γ.
- unary *Proc* operations $\partial_H, \tau_I, \rho_R$ for each set of action labels $H, I \subseteq \mathit{ActLab}$ and action label renaming function $R : \mathit{ActLab} \to \mathit{ActLab}$ such that a and $R(\mathsf{a})$ have parameters of the same sorts. Such functions R we call *well-defined* action label renaming functions.
- a ternary operation $\_ \lhd \_ \rhd \_ : \mathit{Proc} \times \mathit{Bool} \times \mathit{Proc} \to \mathit{Proc}$.
- binders $\sum_{d:D}$ defined on *Proc*, for each data variable d of sort D.

The partial commutative and associative function γ is called a *communication function.* If $\gamma(\mathsf{a}, \mathsf{b}) = \mathsf{c}$ this indicates that actions with labels a and b can synchronize, becoming action c, provided that the data parameters of these actions are equal. The case when $\gamma(\gamma(\mathsf{a}, \mathsf{b}), \mathsf{c})$ is undefined for all a,b and c, which means that at most two parties can communicate synchronously, is called *handshaking* communication (or simply handshaking). The constant δ represents a deadlocked process and the constant τ represents some internal or hidden activity. The *choice* operator $+$ and the *sequential composition* operator $\cdot$ are well known. The *merge* operator $\|$ represents *parallel composition.* The $\lfloor\!\lfloor$ (*left merge*) and $|$ (*communication merge*) are auxiliary operations used to equationally define $\|$. The *encapsulation* operator $\partial_H(q)$ blocks actions in q with action labels in the set H, which is especially used to enforce actions to communicate. The *hiding* operator $\tau_I(q)$ with a set of action labels $I = \{\mathsf{a}, \mathsf{b}, \dots\}$ hides actions with these labels in q by renaming them to τ. The *renaming* operator $\rho_R(q)$ where R is a function from action labels to action labels renames each action with label a in q to an action with label $R(\mathsf{a})$. The operator $p_1 \lhd c \rhd p_2$ is the *if c then p_1 else p_2* operator, where c is an expression of type *Bool*. The *sum operator* $\sum_{d:D} p$ expresses a (potentially infinite) choice $p[d := d_0] + p[d := d_1] + \cdots$ if data domain $D = \{d_0, d_1, \dots\}$, and $p[d := d_i]$ is the term p with all free occurrences of d replaced by d_i.

Definition 2.4. Axioms of μCRL are the ones presented in Tables 2,3,4,5,6, 7 and 8. We assume that

- the descending order of binding strength of operators is: $\cdot$, $\{\|, \lfloor\!\lfloor, |\}$, $\lhd\rhd$, $\sum$, $+$;
- x, y, z are variables of sort *Proc*;
- c, c_1, c_2 are variables of sort *Bool*;
- $d, d^1, d^n, d', \dots$ are data variables (but d in $\sum_{d:D}$ is not a variable);
- b stands for either $\mathsf{a}(\overrightarrow{d})$, or τ, or δ;

- $\overrightarrow{d} = \overrightarrow{d'}$ is an abbreviation for $eq(d^1, d'^1) \wedge \cdots \wedge eq(d^n, d'^n)$, where $\overrightarrow{d} = d^1, \ldots, d^n$ and $\overrightarrow{d'} = d'^1, \ldots, d'^n$;
- the axioms where p and q occur are schemata ranging over all terms p and q of sort *Proc*, *including* those in which d occurs freely;
- the axiom (SUM2) is a scheme ranging over all terms r of sort *Proc* in which d *does not* occur freely.

The axioms in Table 7 are used for the parallel composition elimination. From these axioms we can derive the identities $x \parallel y = y \parallel x$, $(x \parallel y) \parallel z = x \parallel (y \parallel z)$ and $x \parallel \delta = x \cdot \delta$ with the help of the axioms (A1),(A2),(A6),(A7),(CM1),(CM2),(CM4),(CM8) and (CD1). Note that due to (SC3), the axioms (CM6), (CM9), (CT2), (CD2), (Cond9′) and (SUM7′) become derivable. The axioms in Table 8 are used to simplify combinations of renaming, hiding and encapsulation. The axioms (B1) and (B2) are not used in the transformations described in this paper, so these transformations are also sound in models where these two axioms do not hold.

We use many sorted equational logic for processes and booleans, while other data types can have slightly different proof rules, which may include induction principles, quantifier introduction principles, etc. The proof theory of μCRL [19] consists of proof rules for the data sorts, the rules of equational logic for the booleans, and the rules of generalized equational logic [18] for the processes. Note that the rules of generalized equational logic do not allow to substitute terms containing free variables if they become bound. For example, in axiom (SUM1) we cannot substitute $\mathsf{a}(d)$ for x.

Definition 2.5. Two process terms p_1 and p_2 are *(unconditionally) equivalent* (notation $p_1 = p_2$) if $p_1 = p_2$ is derivable from the axioms of μCRL and boolean identities by using many sorted generalized equational logic. In this case we write $\{\mu\text{CRL}, \mathit{BOOL}\} \vdash p_1 = p_2$. Here *BOOL* is used to refer to the specification of the booleans, and the use of equational logic for deriving boolean identities.

Two process terms p_1 and p_2 are *conditionally equivalent* if $\{\mu\text{CRL}, \mathit{BOOL}, \mathit{DATA}\} \vdash p_1 = p_2$. Here *DATA* is used to refer to the specification of all data sorts involved, and all proof rules that may be applied.

$$x + y = y + x \tag{A1}$$
$$x + (y + z) = (x + y) + z \tag{A2}$$
$$x + x = x \tag{A3}$$
$$(x + y) \cdot z = x \cdot z + y \cdot z \tag{A4}$$
$$(x \cdot y) \cdot z = x \cdot (y \cdot z) \tag{A5}$$
$$x + \delta = x \tag{A6}$$
$$\delta \cdot x = \delta \tag{A7}$$
$$x \cdot \tau = x \tag{B1}$$
$$z \cdot (\tau \cdot (x + y) + x) = z \cdot (x + y) \tag{B2}$$

Table 2: Basic axioms of μCRL.

2.2 Systems of Recursion Equations

We assume a fixed and infinite set $\mathsf{Procnames} = \{\mathsf{X}, \mathsf{Y}, \mathsf{Z}, \ldots\}$ of *process names* with type information associated to them. We extend the sort *Proc* of processes by allowing the process names in $P \subseteq$

$$x \parallel y = (x \mathbin{⫫} y + y \mathbin{⫫} x) + x \mid y \tag{CM1}$$
$$b \mathbin{⫫} x = b \cdot x \tag{CM2}$$
$$(b \cdot x) \mathbin{⫫} y = b \cdot (x \parallel y) \tag{CM3}$$
$$(x + y) \mathbin{⫫} z = x \mathbin{⫫} z + y \mathbin{⫫} z \tag{CM4}$$
$$(b \cdot x) \mid b' = (b \mid b') \cdot x \tag{CM5}$$
$$b \mid (b' \cdot x) = (b \mid b') \cdot x \tag{CM6}$$
$$(b \cdot x) \mid (b' \cdot y) = (b \mid b') \cdot (x \parallel y) \tag{CM7}$$
$$(x + y) \mid z = x \mid z + y \mid z \tag{CM8}$$
$$x \mid (y + z) = x \mid y + x \mid z \tag{CM9}$$
$$\mathsf{a}(\overrightarrow{d}) \mid \mathsf{a}'(\overrightarrow{d'}) = \gamma(\mathsf{a},\mathsf{a}')(\overrightarrow{d}) \lhd \overrightarrow{d} = \overrightarrow{d'} \rhd \delta \quad \text{if } \gamma(\mathsf{a},\mathsf{a}') \text{ is defined} \tag{CF1}$$
$$\mathsf{a}(\overrightarrow{d}) \mid \mathsf{a}'(\overrightarrow{d'}) = \delta \quad \text{otherwise} \tag{CF2}$$
$$\tau \mid b = \delta \tag{CT1}$$
$$b \mid \tau = \delta \tag{CT2}$$
$$\delta \mid b = \delta \tag{CD1}$$
$$b \mid \delta = \delta \tag{CD2}$$

Table 3: Axioms for parallel composition in μCRL.

$$x \lhd \mathbf{t} \rhd y = x \tag{Cond1}$$
$$x \lhd \mathbf{f} \rhd y = y \tag{Cond2}$$
$$x \lhd c \rhd y = x \lhd c \rhd \delta + y \lhd \neg c \rhd \delta \tag{Cond3}$$
$$(x \lhd c_1 \rhd \delta) \lhd c_2 \rhd \delta = (x \lhd c_1 \wedge c_2 \rhd \delta) \tag{Cond4}$$
$$(x \lhd c_1 \rhd \delta) + (x \lhd c_2 \rhd \delta) = x \lhd c_1 \vee c_2 \rhd \delta \tag{Cond5}$$
$$(x \lhd c \rhd \delta) \cdot y = (x \cdot y) \lhd c \rhd \delta \tag{Cond6}$$
$$(x + y) \lhd c \rhd \delta = x \lhd c \rhd \delta + y \lhd c \rhd \delta \tag{Cond7}$$
$$(x \lhd c \rhd \delta) \mathbin{⫫} y = (x \mathbin{⫫} y) \lhd c \rhd \delta \tag{Cond8}$$
$$(x \lhd c \rhd \delta) \mid y = (x \mid y) \lhd c \rhd \delta \tag{Cond9}$$
$$x \mid (y \lhd c \rhd \delta) = (x \mid y) \lhd c \rhd \delta \tag{Cond9$'$}$$
$$(x \lhd c \rhd \delta) \cdot (y \lhd c \rhd \delta) = (x \cdot y) \lhd c \rhd \delta \tag{Sca}$$
$$p \lhd eq(d,e) \rhd \delta = p[e := d] \lhd eq(d,e) \rhd \delta \tag{PE}$$

Table 4: Axioms for conditions in μCRL.

Procnames as variables of type $\overrightarrow{D} \to \mathit{Proc}$. The terms in the signature of μCRL extended with P are further called *(μCRL) process terms* and the set of all of them is denoted by $\mathit{Terms}(P)$. The *free data variables* in a process term are those not bound by $\sum_{d:D}$ occurrences. We write DVar for the set of all free and bound data variables that can occur in a term.

Definition 2.6. A *process equation* is an equation of the form $\mathsf{X}(\overrightarrow{d_\mathsf{X}{:}D_\mathsf{X}}) = q_\mathsf{X}$, where X is a process name with a list of data parameters $\overrightarrow{d_\mathsf{X}{:}D_\mathsf{X}}$, and q_X is a process term, in which only the data variables

$$\sum_{d:D} x = x \tag{SUM1}$$

$$\sum_{e:D} r = \sum_{d:D} (r[e := d]) \tag{SUM2}$$

$$\sum_{d:D} p = \sum_{d:D} p + p \tag{SUM3}$$

$$\sum_{d:D} (p + q) = \sum_{d:D} p + \sum_{d:D} q \tag{SUM4}$$

$$\sum_{d:D} (p \cdot x) = (\sum_{d:D} p) \cdot x \tag{SUM5}$$

$$\sum_{d:D} (p \mathbin{\lfloor\!\lfloor} x) = (\sum_{d:D} p) \mathbin{\lfloor\!\lfloor} x \tag{SUM6}$$

$$\sum_{d:D} (p \mid x) = (\sum_{d:D} p) \mid x \tag{SUM7}$$

$$\sum_{d:D} (x \mid p) = x \mid (\sum_{d:D} p) \tag{SUM7$'$}$$

$$\sum_{d:D} (\partial_H(p)) = \partial_H(\sum_{d:D} p) \tag{SUM8}$$

$$\sum_{d:D} (\tau_I(p)) = \tau_I(\sum_{d:D} p) \tag{SUM9}$$

$$\sum_{d:D} (\rho_R(p)) = \rho_R(\sum_{d:D} p) \tag{SUM10}$$

$$\sum_{d:D} (p \lhd c \rhd \delta) = (\sum_{d:D} p) \lhd c \rhd \delta \tag{SUM12}$$

Table 5: Axioms for sums in μCRL.

from $\overrightarrow{d_{\mathsf{X}}}$ may occur freely. We write *rhs*(X) for q_{X}, *pars*(X) for $\overrightarrow{d_{\mathsf{X}}}$, and *type*($\mathsf{X}$) for $\overrightarrow{D_{\mathsf{X}}}$.

Definition 2.7. Let $P \subseteq \mathsf{Procnames}$ be a finite set of process names such that each process name is uniquely typed. A (finite) non-empty set G of process equations over *Terms*(P) is called a *(finite) system of process equations* if each process name in P occurs exactly once at the left. The set of process names (with types) that appear within G is denoted as $|G|$ (so, $|G| = P$). We use *rhs*(X, G), *pars*(X, G) and *type*(X, G) to refer to the corresponding parts of the equation for X in G.

Although the original definition of a μCRL specification allows to have the same process names with different types, we do not treat this possibility here as it would make the explanation only more long-winded.

Definition 2.8. Let G be a finite system of process equations, X be a process name in it, and $\overrightarrow{t}$ be a data term vector of type *type*(X, G). Then the pair $(\mathsf{X}(\overrightarrow{t}), G)$ is called a *process definition.* We use the abbreviation (X, G) for $(\mathsf{X}(\mathit{pars}(\mathsf{X}, G)), G)$.

Example 2.9. Both $G_1 = \{\mathsf{X} = \mathsf{a}{\cdot}\mathsf{Y}, \mathsf{Y} = \mathsf{b}{\cdot}\mathsf{X}, \mathsf{Z} = \mathsf{X}\|\mathsf{Y}\}$ and $G_2 = \{\mathsf{T}(n{:}\mathit{Nat}) = \mathsf{a}(\mathit{even}(n)){\cdot}\mathsf{T}(S(n))\}$ with $\mathit{even} : \mathit{Nat} \to \mathit{Bool}$ as expected and $S : \mathit{Nat} \to \mathit{Nat}$ the successor function, are examples of systems of process equations. All of $(\mathsf{X}, G_1), (\mathsf{T}, G_2), (\mathsf{T}(m), G_2)$ are process definitions.

$$\partial_H(b) = b \text{ if } b = \tau \text{ or } (b = \mathsf{a}(\overrightarrow{d}) \text{ and } \mathsf{a} \notin H) \tag{D1}$$
$$\partial_H(b) = \delta \text{ otherwise} \tag{D2}$$
$$\partial_H(x+y) = \partial_H(x) + \partial_H(y) \tag{D3}$$
$$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y) \tag{D4}$$
$$\partial_H(x \lhd c \rhd \delta) = \partial_H(x) \lhd c \rhd \delta \tag{D5}$$
$$\tau_I(b) = b \text{ if } b = \delta \text{ or } (b = \mathsf{a}(\overrightarrow{d}) \text{ and } \mathsf{a} \notin I) \tag{T1}$$
$$\tau_I(b) = \tau \text{ otherwise} \tag{T2}$$
$$\tau_I(x+y) = \tau_I(x) + \tau_I(y) \tag{T3}$$
$$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y) \tag{T4}$$
$$\tau_I(x \lhd c \rhd \delta) = \tau_I(x) \lhd c \rhd \delta \tag{T5}$$
$$\rho_R(\delta) = \delta \tag{RD}$$
$$\rho_R(\tau) = \tau \tag{RT}$$
$$\rho_R(\mathsf{a}(\overrightarrow{d})) = R(\mathsf{a})(\overrightarrow{d}) \tag{R1}$$
$$\rho_R(x+y) = \rho_R(x) + \rho_R(y) \tag{R3}$$
$$\rho_R(x \cdot y) = \rho_R(x) \cdot \rho_R(y) \tag{R4}$$
$$\rho_R(x \lhd c \rhd \delta) = \rho_R(x) \lhd c \rhd \delta \tag{R5}$$

Table 6: Axioms for renaming operators in μCRL.

$$(x \mathbin{⌊\!⌊} y) \mathbin{⌊\!⌊} z = x \mathbin{⌊\!⌊} (y \parallel z) \tag{SC1}$$
$$x \mid y = y \mid x \tag{SC3}$$
$$(x \mid y) \mid z = x \mid (y \mid z) \tag{SC4}$$
$$x \mid (y \mathbin{⌊\!⌊} z) = (x \mid y) \mathbin{⌊\!⌊} z \tag{SC5}$$
$$x \mathbin{⌊\!⌊} \delta = x \cdot \delta \tag{SCD}$$

Table 7: Axioms for Standard Concurrency in μCRL.

2.3 Equivalence of Process Definitions

We introduce the notion of *equivalence* over systems of process equations in a stepwise manner. Let G_1 and G_2 be systems of process equations, and assume that the common data sorts of G_1 and G_2 are equally defined. Let $\mathit{DATA}(G_1, G_2)$ represents all data specifications occurring in G_1 and G_2 and all proof rules adopted for these data. We first define (conditional) implication between process terms, and then the equivalence.

In the following definition, derivations of the form $\{\mu\text{CRL}, \mathit{BOOL}, \mathit{DATA}\} \cup G_1 \vdash \phi$ are required. In this case, the axioms from μCRL, BOOL and DATA may be used to derive ϕ, as well as the process equations in G_1. However, we restrict derivability by requiring that the (data-parametric) process names from G_1 are considered as (data-parametric) *constants*. For example, if $G_1 = \{\mathsf{X} = \mathsf{a} \cdot \mathsf{X}\}$, we may use $\mathsf{X} = \mathsf{a} \cdot \mathsf{X}$ as an axiom in $\{\mu\text{CRL}, \mathit{BOOL}, \mathit{DATA}\} \cup \{\mathsf{X} = \mathsf{a} \cdot \mathsf{X}\} \vdash \phi$, but X may *not* be used as a variable that can be instantiated (e.g., $\{\mu\text{CRL}, \mathit{BOOL}, \mathit{DATA}\} \cup \{\mathsf{X} = \mathsf{a} \cdot \mathsf{X}\} \nvdash \mathsf{a} = \mathsf{a} \cdot \mathsf{a}$).

Definition 2.10. Let G_1, G_2 be systems of process equations with $|G_1| = \{\mathsf{X}_1, \ldots, \mathsf{X}_n\}$ and $|G_2| =$

$$\partial_{H_1}(\partial_{H_2}(x)) = \partial_{H_1 \cup H_2}(x) \tag{DD}$$
$$\tau_{I_1}(\tau_{I_2}(x)) = \tau_{I_1 \cup I_2}(x) \tag{TT}$$
$$\rho_{R_1}(\rho_{R_2}(x)) = \rho_{R_1 \circ R_2}(x) \tag{RR}$$
$$\partial_H(\tau_I(x)) = \tau_I(\partial_{H \setminus I}(x)) \tag{DT}$$
$$\partial_H(\rho_R(x)) = \rho_R(\partial_{R^{-1}(H)}(x)) \tag{DR}$$
$$\tau_I(\rho_R(x)) = \rho_R(\tau_{R^{-1}(I)}(x)) \tag{TR}$$
$$\partial_{\emptyset}(x) = x \tag{D0}$$
$$\tau_{\emptyset}(x) = x \tag{T0}$$
$$\rho_{R_{ActLab}}(x) = x \tag{R0}$$
$$\rho_R(\partial_H(x)) = \rho_{R_H}(\partial_H(x)) \tag{RDO}$$
$$\rho_R(\tau_I(x)) = \rho_{R_H}(\tau_I(x)) \tag{RTO}$$

where $R_S(\mathsf{a})$ for $S \subseteq \mathit{ActLab}$ is defined to be equal to a if $\mathsf{a} \in S$ and to $R(\mathsf{a})$ otherwise.

Table 8: Axioms for combinations of renaming operators.

$\{\mathsf{Y}_1, \ldots, \mathsf{Y}_m\}$. Let furthermore *DATA* be such that it contains $\mathit{DATA}(G_1, G_2)$, i.e., *DATA* contains all data sorts and associated proof rules of $\mathit{DATA}(G_1, G_2)$.

We say that $(\mathsf{X}_1(\overrightarrow{t_1}), G_1)$ *conditionally implies* $(\mathsf{Y}_1(\overrightarrow{t_2}), G_2)$ (notation $(\mathsf{X}_1(\overrightarrow{t_1}), G_1) \Rightarrow_c (\mathsf{Y}_1(\overrightarrow{t_2}), G_2)$) for some (possibly open) data term vectors $\overrightarrow{t_1}, \overrightarrow{t_2}$ over *DATA* if for $j = 1, \ldots, m$ there is a set of mappings $g_{\mathsf{Y}_j} : \mathit{type}(\mathsf{Y}_j) \to \mathit{Terms}(\{\mathsf{X}_1, \ldots, \mathsf{X}_n\})$ such that

$$\{\mu\text{CRL}, \mathit{BOOL}, \mathit{DATA}\} \cup G_1 \vdash \mathsf{X}_1(\overrightarrow{t_1}) = g_{\mathsf{Y}_1}(\overrightarrow{t_2}) \text{ and}$$
$$\forall j \in 1 \ldots m \Big(\{\mu\text{CRL}, \mathit{BOOL}, \mathit{DATA}\} \cup G_1 \vdash$$
$$g_{\mathsf{Y}_j}(\overrightarrow{d_j'}) = \mathit{rhs}(\mathsf{Y}_j)\,[\forall k \in 1 \ldots m\ \mathsf{Y}_k(t') := g_{\mathsf{Y}_k}(t')]\Big)$$

If *DATA* identities are not used in these derivations we say that $(\mathsf{X}_1(\overrightarrow{t_1}), G_1)$ *(unconditionally) implies* $(\mathsf{Y}_1(\overrightarrow{t_2}), G_2)$ (notation $(\mathsf{X}_1(\overrightarrow{t_1}), G_1) \Rightarrow (\mathsf{Y}_1(\overrightarrow{t_2}), G_2)$). In case $(\mathsf{X}(\mathit{pars}(\mathsf{X}, G_1)), G_1)$ (conditionally) implies $(\mathsf{Y}(\mathit{pars}(\mathsf{Y}, G_2)), G_2)$ we say that (X, G_1) (conditionally) implies (Y, G_2) (notation $(\mathsf{X}, G_1) \Rightarrow (\mathsf{Y}, G_2)$ $((\mathsf{X}, G_1) \Rightarrow_c (\mathsf{Y}, G_2))$).

The adjective "conditional" could be replaced by "data-dependent", but we did not do this because it is used similarly in the guardedness definition (See Section 2.4).

Definition 2.11. Process definition $(\mathsf{X}(\overrightarrow{t_1}), G_1)$ is *equivalent* to process definition $(\mathsf{Y}(\overrightarrow{t_2}), G_2)$ (notation $(\mathsf{X}(\overrightarrow{t_1}), G_1) = (\mathsf{Y}(\overrightarrow{t_2}), G_2)$) if both $(\mathsf{X}(\overrightarrow{t_1}), G_1) \Rightarrow (\mathsf{Y}(\overrightarrow{t_2}), G_2)$ and $(\mathsf{Y}(\overrightarrow{t_2}), G_2) \Rightarrow (\mathsf{X}(\overrightarrow{t_1}), G_1)$. Similarly, if $(\mathsf{X}(\mathit{pars}(\mathsf{X}, G_1)), G_1) = (\mathsf{Y}(\mathit{pars}(\mathsf{Y}, G_2)), G_2)$ we say that (X, G_1) is *equivalent* to (Y, G_2). The *conditional* equivalence (notation $=_c$) is defined in the same way.

Finally, $G_1 = G_2$ if $|G_1| = |G_2|$ and for all $\mathsf{X} \in |G_1|$, $(\mathsf{X}, G_1) = (\mathsf{X}, G_2)$.

Note that on systems of process equations, the relations $=$ and $=_c$ are equivalences, and the relations $\Rightarrow$ and $\Rightarrow_c$ are reflexive and transitive. The following simple examples demonstrate the use of Definitions 2.11 and 2.10.

Example 2.12. Let $G_1 = \{\mathsf{X} = \mathsf{a} \cdot \mathsf{Y}, \mathsf{Y} = \mathsf{b} \cdot \mathsf{X}\}$ and $G_2 = \{\mathsf{X} = \mathsf{a} \cdot \mathsf{b} \cdot \mathsf{X}\}$. We can show that $(\mathsf{X}, G_1) = (\mathsf{X}, G_2)$. The implication from left to right can be shown by choosing $g_{\mathsf{X}} = \mathsf{X}$. The reverse direction can be shown by choosing $g_{\mathsf{X}} = \mathsf{X}$ and $g_{\mathsf{Y}} = \mathsf{b} \cdot \mathsf{X}$.

Example 2.13. Let $G_1 = \{\mathsf{X}(b{:}\mathit{Bool}) = \mathsf{a}(b) \cdot \mathsf{X}(\neg b)\}$ and $G_2 = \{\mathsf{Y}(n{:}\mathit{Nat}) = \mathsf{a}(\mathit{even}(n)) \cdot \mathsf{Y}(S(n))\}$. We can show that $(\mathsf{X}(\mathbf{t}), G_1) \Rightarrow_c (\mathsf{Y}(0), G_2)$ by choosing $g_\mathsf{Y}(n) = \mathsf{X}(\mathit{even}(n))$. In this case we need to show that $\mathsf{X}(\mathbf{t}) = g_\mathsf{Y}(0)$ (which follows from $\mathit{even}(0) = \mathbf{t}$) and that $\mathsf{X}(\mathit{even}(n)) = \mathsf{a}(\mathit{even}(n)) \cdot \mathsf{X}(\mathit{even}(S(n)))$. This latter identity follows from $\mathsf{X}(b) = \mathsf{a}(b){\cdot}\mathsf{X}(\neg b)$ and the data identity $\mathit{even}(S(n)) = \neg \mathit{even}(n)$. If we assume the existence of a function $n : \mathit{Bool} \to \mathit{Nat}$, defined by $n(\mathbf{t}) = 0$ and $n(\mathbf{f}) = 1$, we can prove that $(\mathsf{X}(b), G_1) \Rightarrow_c (\mathsf{Y}(n(b)), G_2)$ using the same function $g_\mathsf{Y}(n)$ and the data identities $\mathit{even}(n(b)) = b$ and $\mathit{even}(S(n(b))) = \neg b$, both of which seem reasonable.

We do not have any of the reverse implications: consider the model with carrier set Nat, in which $\mathsf{a}(b)$ is interpreted as 1, and sequential composition as $+$. Then $\mathsf{Y}(0)$ has many solutions, whereas $\mathsf{X}(\mathbf{t})$ has none.

It can be shown that the basic Definition 2.10 characterizes preservation of solutions of a process definition in every model of μCRL and data identities. For exact definitions and more details on this subject we refer to [33].

The following lemma shows that by applying a μCRL axiom to the right hand side of an equation we get an equivalent system.

Lemma 2.14 (Axioms). *Let p_1, p_2 be process terms such that $p_1 = p_2$. Let G be a system of process equations, and X be a process name in it such that p_1 is a subterm of $\mathit{rhs}(\mathsf{X}, G)$. Let G' consist of equations in G, but in the equation defining X an occurrence of p_1 is replaced by p_2. Then $G = G'$.*

The following lemma shows that by replacing a subterm of the right hand side of an equation by a fresh process name, and adding the equation for it, we get an equivalent process definition for each process name in the original system.

Lemma 2.15 (New equation). *Let G be a system of process equations, and X be a process name in it. Let p be a subterm of $\mathit{rhs}(\mathsf{X}, G)$ with free data variables $d^1{:}D^1, \ldots, d^n{:}D^n = \overrightarrow{d{:}D}$ in it. Let Y be a process name, $\mathsf{Y} \notin |G|$. Let G' consist of equations in G, but in the equation defining X an occurrence of p is replaced by $\mathsf{Y}(\overrightarrow{d})$, and the equation $\mathsf{Y}(\overrightarrow{d{:}D}) = p$ is added to G. Then for any $\mathsf{Z} \in |G|$ we have $(\mathsf{Z}, G) = (\mathsf{Z}, G')$.*

Proof. To prove that $(\mathsf{Z}, G) \Rightarrow (\mathsf{Z}, G')$ we take $g_\mathsf{Z}(\mathit{pars}(\mathsf{Z})) = \mathsf{Z}(\mathit{pars}(\mathsf{Z}))$ for all $\mathsf{Z} \in |G|$, and $g_\mathsf{Y} = p$. To prove the other direction we just take $g_\mathsf{Z}(\mathit{pars}(\mathsf{Z})) = \mathsf{Z}(\mathit{pars}(\mathsf{Z}))$ for all $\mathsf{Z} \in |G|$. □

The following lemma shows that under certain conditions we can substitute a process name by its right hand side in a right hand side of an equation.

Lemma 2.16 (Substitution). *Let G be a system of process equations, and X be a process name in it. Let $\mathsf{Y}(\overrightarrow{t})$ be a subterm of $\mathit{rhs}(\mathsf{X}, G)$ for some $\mathsf{Y} \neq \mathsf{X}$. Let G' consist of equations in G, but in the equation defining X an occurrence of $\mathsf{Y}(\overrightarrow{t})$ is replaced by $\mathit{rhs}(\mathsf{Y}, G)[\mathit{pars}(\mathsf{Y}, G) := \overrightarrow{t}]$. Then we have that $G = G'$.*

Proof. In both directions we take the mappings g_X to be the identity mapping. □

The following lemma says that we can add dummy data parameters to a process equation, or remove such parameters.

Lemma 2.17 (Extra parameters). *Let G be a system of process equations, and X be a process name in it with parameters $d^1, \ldots, d^n$. Suppose that d^i does not occur freely in $\mathit{rhs}(\mathsf{X}, G)$. Let G' be as G, but the process name X is replaced by X' and $\mathit{pars}(\mathsf{X}', G') = d^1, \ldots, d^{i-1}, d^{i+1}, \ldots, d^n$. Then for all $\mathsf{Y} \in |G| \wedge \mathsf{Y} \neq \mathsf{X}$ we have $(\mathsf{Y}, G) = (\mathsf{Y}, G')$, and $(\mathsf{X}(d^1, \ldots, d^n), G) = (\mathsf{X}'(d^1, \ldots, d^{i-1}, d^{i+1}, \ldots, d^n), G')$.*

Proof. In both directions we take the mappings g_Y (for $\mathsf{Y} \neq \mathsf{X}$) to be the identity mappings. In one direction $g_{\mathsf{X}'}(d^1, \ldots, d^{i-1}, d^{i+1}, \ldots, d^n) = \mathsf{X}(d^1, \ldots, d^n)$ and $g_\mathsf{X}(d^1, \ldots, d^n) = \mathsf{X}'(d^1, \ldots, d^{i-1}, d^{i+1}, \ldots, d^n)$. □

2.4 Guardedness

In this paper we use a slightly different notion of guardedness than the one in [19].

Definition 2.18. An occurrence of a process name X in a process term p is *completely guarded* if there is a subterm p' of p of the form $q \cdot p''$ containing this occurrence of X, where q is a process term containing no process names.

A process term is called *completely guarded* if every occurrence of a process name in it is completely guarded. Note that a term that contains no process names is completely guarded.

A system of process equations G is *completely guarded* if for any $\mathsf{X} \in |G|$, $\mathit{rhs}(\mathsf{X}, G)$ is a completely guarded term.

Definition 2.19. A process definition (X, G) is *(unconditionally) guarded* if there is a process definition (X', G') such that G' is a completely guarded system of process equations, and $(\mathsf{X}, G) = (\mathsf{X}', G')$.

Definition 2.20. Let G be a system of process equations. A *Process Name Unguarded-Dependency Graph (PNUDG)* is an oriented graph with the set of nodes $|G|$, and edges defined as follows: $\mathsf{X} \to \mathsf{Y}$ belongs to the graph if Y is not completely guarded in $\mathit{rhs}(\mathsf{X}, G)$.

Lemma 2.21. *If the PNUDG of a finite system of process equations G is acyclic, then G is guarded.*

Proof. Given a system G we replace each unguarded occurrence of a process name by its right hand side. By Lemma 2.16 we get an equivalent system. Due to the fact that PNUDG is acyclic, we need to perform the replacement only finitely many times, and after that we get a completely guarded system. □

The following example shows that the converse of Lemma 2.21 does not hold.

Example 2.22. System G consisting of one equation $\mathsf{X} = \mathsf{X} \lhd \mathbf{f} \rhd \delta$ is guarded, but its PNUDG contains the cycle $\mathsf{X} \to \mathsf{X}$.

2.5 μCRL Specifications

For the purpose of this paper we restrict to the μCRL specifications that do not contain left merge ($\lfloor\!\lfloor$) and communication ($|$) explicitly. These operators were introduced to allow the finite axiomatization of parallel composition ($\|$) in the bisimulation setting, and they are hardly used explicitly in μCRL specifications.

We consider systems of process equations with the right hand sides from the following subset of μCRL terms

$$p ::= \mathsf{a}(\overrightarrow{t}) \mid \delta \mid \mathsf{Y}(\overrightarrow{t}) \mid p+p \mid p \cdot p \mid p \| p \mid \sum_{d:D} p \mid p \lhd c \rhd p \mid \partial_H(p) \mid \tau_I(p) \mid \rho_R(p) \quad (2.1)$$

The combination of the given data specification with a process definition $(\mathsf{X}(\overrightarrow{t}), G)$ of process equations determines a μCRL *specification* in the sense as defined in [20]. Such a specification depends on a finite subset **act** of *ActLab* and on **comm**, an enumeration of γ restricted to the labels in **act**. So a finite system G implicitly describes a finitary based language.

For a consistent (meaningful) specification, i.e., a *Statically Semantically Correct* specification, it is necessary that all objects are specified only once, that all typing is respected and that the communications in **comm** are specified in a functional way. Furthermore, the *eq* functions for the data sorts should have the following properties:

$$\{\mathit{DATA}, \mathit{eq}(d, e) = \mathbf{t}\} \vdash d = e \quad \text{and} \quad \{\mathit{DATA}, d = e\} \vdash \mathit{eq}(d, e) = \mathbf{t}$$

All data sorts that are introduced during the linearization must have *eq* functions satisfying these properties.

The problem of linearization of a μCRL specification defined by $(\mathsf{X}(\overrightarrow{t}), G)$ consists of generation of a new μCRL specification which

- depends on the same **act** and **comm**,
- contains all data definitions of the original one, and, possibly, definitions of the auxiliary data types,
- is defined by $(\mathsf{Z}(\mathrm{m}_\mathsf{X}(\overrightarrow{t})), L)$, where L contains exactly one process equation for Z in linear form (defined later), and m_X is a mapping from $\mathit{pars}(\mathsf{X}, G)$ to $\mathit{pars}(\mathsf{Z}, L)$,

such that $(\mathsf{X}(\overrightarrow{t}), G) \Rightarrow_c (\mathsf{Z}(\mathrm{m}_\mathsf{X}(\overrightarrow{t})), L)$.

It is not possible to linearize a μCRL specification which in unguarded. In this paper we describe the linearization procedure for specifications, where the system of the equations has acyclic PNUDG. (Conditionally) guarded systems with cyclic PNUDG are not treated in the current paper. We note that in some cases cycles can be removed, for example because they are not reachable, or using properties of data types (cf. [24]). The elimination of cycles is not treated here.

3. Transformation to Parallel Extended Greibach Normal Form

As the input for the linearization procedure we take a μCRL process definition $(\mathsf{X}(\overrightarrow{t}), G)$ such that PNUDG of G is acyclic. In this section we transform G into a system of process equations G_4 in Parallel Extended Greibach Normal Form. The resulting system will contain process equations for all process names in $|G|$ with the same names and types of data parameters involved, as well as, possibly, other process equations.

3.1 Normal Forms

Below we define two normal forms for systems of process equations in μCRL: pre-Parallel Extended Greibach Normal Form (pre-PEGNF) and Parallel Extended Greibach Normal Form. Later on, in Section 4 we define an even more restricted form called post-Parallel Extended Greibach Normal Form (post-PEGNF). A system is said to be in one of these forms if all of its equations are in the respective form.

From this point on we assume that $\mathsf{a}(\overrightarrow{t})$ with possible indices can also be an abbreviation for τ. This is done to make the normal form representations more concise.

Definition 3.1. A μCRL process equation is in *pre-PEGNF* if it is of the form:

$$\mathsf{X}(\overrightarrow{d{:}D}) = \sum_{i\in I} \sum_{\overrightarrow{e_i:E_i}} p_i(\overrightarrow{d, e_i}) \lhd c_i(\overrightarrow{d, e_i}) \rhd \delta$$

where $p_i(\overrightarrow{d, e_i})$ are terms of the following syntax:

$$p ::= \mathsf{a}(\overrightarrow{t}) \mid \delta \mid \mathsf{Y}(\overrightarrow{t}) \mid p \cdot p \mid p \parallel p \mid \rho_R(\tau_I(\partial_H(p \parallel p))) \mid \rho_R(\tau_I(\partial_H(\mathsf{Y}(\overrightarrow{t})))) \tag{3.1}$$

A μCRL process equation is in *PEGNF* iff it is of the form:

$$\begin{aligned}\mathsf{X}(\overrightarrow{d{:}D}) =& \sum_{i\in I} \sum_{\overrightarrow{e_i:E_i}} \mathsf{a}_i(\overrightarrow{f_i}(\overrightarrow{d, e_i})) \cdot p_i(\overrightarrow{d, e_i}) \lhd c_i(\overrightarrow{d, e_i}) \rhd \delta \\ &+ \sum_{j\in J} \sum_{\overrightarrow{e_j:E_j}} \mathsf{a}_j(\overrightarrow{f_j}(\overrightarrow{d, e_j})) \lhd c_j(\overrightarrow{d, e_j}) \rhd \delta\end{aligned}$$

where I and J are disjoint, and all $p_i(\overrightarrow{d, e_i})$ are terms having the syntax (3.1)

Note (Sum Notation). Apart from functions $\sum_{d:D} p$ that are included in the syntax of process terms, we use the following abbreviations. Expression $\sum_{\overrightarrow{d:D}}$ is an abbreviation for $\sum_{d^1:D^1} \cdots \sum_{d^n:D^n}$. In case $n = 0$, $\sum_{\overrightarrow{d:D}} p$ is an abbreviation for p. Expression $\sum_{i \in I} p_i$, where I is a finite set, is an abbreviation for $p_{i_1} + \cdots + p_{i_n}$ such that $\{i_1, \ldots, i_n\} = I$. In case $I = \emptyset$, $\sum_{i \in I} p_i$ is an abbreviation for δ.

Note (Conditions). As follows from the above definition, any process equation in (pre-)(post-)PEGNF must have a condition in each summand. However, this is not a necessary restriction. In case a summand q does not have a condition, it is an abbreviation for $q \lhd \mathbf{t} \rhd \delta$.

We also mention here that pre-PEGNF could be achieved by an algorithm similar to the one presented in Proposition 7.2 of [13]. There it is proved that every system of equations can be transformed to a *quasi-uniform* one by the introduction of new variables. In a quasi-uniform system each equation has at most one function symbol (in our case one function symbol of sort *Proc*) in the right hand side, which means that every such system is in pre-PEGNF. In our case such an algorithm would generate many more additional equations than necessary, many of which would become unreachable after performing the transformation in Subsection 3.5.

3.2 Preprocessing

We first transform G into G_1. This can be seen as a preprocessing step that possibly renames bound data variables. For instance $\sum_{d:D}((\sum_{d:E} \mathsf{a}(d)) \cdot \mathsf{b}(d))$ is replaced by $\sum_{d:D}((\sum_{e:E} \mathsf{a}(e)) \cdot \mathsf{b}(d))$, where e is a fresh variable. We replace each equation $\mathsf{X}(\overrightarrow{d_\mathsf{X}{:}D_\mathsf{X}}) = p_\mathsf{X}$ in G_1 with the equation $\mathsf{X}(\overrightarrow{d_\mathsf{X}{:}D_\mathsf{X}}) = S_0(\{\overrightarrow{d_\mathsf{X}}\}, p_\mathsf{X})$, where $S_0 : DVar \times Terms(|G_1|) \to Terms(|G_1|)$ is defined in the following way:

$$S_0(S, f(p^1, \ldots, p^n)) \to f(S_0(S, p^1), \ldots, S_0(S, p^n)) \text{ if } f \text{ is not } \sum_{d:D}$$

$$S_0\Big(S, \sum_{d:D} p\Big) \to \begin{cases} \sum_{d:D} S_0(S \cup \{d\}, p) & \text{if } d \notin S \\ \sum_{e:D} S_0(S \cup \{e\}, p[d := e]) & \text{if } d \in S \end{cases}$$

where e is a fresh variable.

Proposition 3.2. *Let G_1 be the result of applying the preprocessing to G. Then $G_1 = G$.*

Proof. The statement follows from Lemma 2.14 if we apply axiom (SUM2). □

As can easily be seen, the preprocessing step does not increase the size or the number of equations in the system.

3.3 Reduction by Simple Rewriting

By applying term rewriting we get an equivalent set of process equations to the given one, but with terms in right hand sides having the more restricted form as presented in Table 9.

The rewrite rules that we apply to the right hand sides of the equations are listed in Tables 10 and 11. The symbols $\sum_{d:D}$ are treated in this rewrite system as function symbols, not as binders. This is justified by the fact that we have renamed all nested bound variables, which allows the use of first order term rewriting. The mapping induced by the rewrite rules for a given system of process equations G is called $rewr : Terms(|G|) \to Terms(|G|)$.

Before applying rewriting we eliminate all terms of the form $\_ \lhd \_ \rhd \_$ with the third argument different from δ, with the following rule:

$$y \not\equiv \delta \implies x \lhd c \rhd y \to x \lhd c \rhd \delta + y \lhd \neg c \rhd \delta \qquad \text{(RCOND3)}$$

Rewriting is performed modulo the identities presented in Table 12

$$
\begin{aligned}
p &::= p_1 \mid \delta \\
p_1 &::= \mathsf{a}(\overrightarrow{t}) \mid \mathsf{Y}(\overrightarrow{t}) \mid p_1 + p_1 \mid p_2 \cdot p \mid p_1 \parallel p_1 \mid \sum_{d:D} p_3 \mid p_4 \lhd c \rhd \delta \mid \partial_H(p_5) \mid \tau_I(p_6) \\
&\qquad \mid \rho_R(p_7) \\
p_2 &::= \mathsf{a}(\overrightarrow{t}) \mid \mathsf{Y}(\overrightarrow{t}) \mid p_1 + p_1 \mid p_2 \cdot p \mid p_1 \parallel p_1 \mid \partial_H(p_5) \mid \tau_I(p_6) \mid \rho_R(p_7) \\
p_3 &::= \mathsf{a}(\overrightarrow{t}) \mid \mathsf{Y}(\overrightarrow{t}) \mid p_2 \cdot p \mid p_1 \parallel p_1 \mid \sum_{d:D} p_3 \mid p_4 \lhd c \rhd \delta \mid \partial_H(p_5) \mid \tau_I(p_6) \mid \rho_R(p_7) \\
p_4 &::= \mathsf{a}(\overrightarrow{t}) \mid \mathsf{Y}(\overrightarrow{t}) \mid p_2 \cdot p \mid p_1 \parallel p_1 \mid \partial_H(p_5) \mid \tau_I(p_6) \mid \rho_R(p_7) \\
p_5 &::= \mathsf{Y}(\overrightarrow{t}) \mid p_1 \parallel p_1 \\
p_6 &::= p_5 \mid \partial_H(p_5) \\
p_7 &::= p_6 \mid \tau_I(p_6)
\end{aligned}
$$

Table 9: Syntax of terms after simple rewriting.

$$
\begin{aligned}
x + \delta &\to x && \text{(RA6)} \\
x \parallel \delta &\to x \cdot \delta && \text{(SC6)} \\
\delta \cdot x &\to \delta && \text{(RA7)} \\
\Big(\sum_{d:D} x\Big) \cdot y &\to \sum_{d:D} (x \cdot y) && \text{(RSUM5)} \\
(x \lhd c \rhd \delta) \cdot y &\to (x \cdot y) \lhd c \rhd \delta && \text{(RCOND6)} \\
\sum_{d:D} \delta &\to \delta && \text{(RSUM1}'\text{)} \\
\sum_{d:D} (x + y) &\to \sum_{d:D} x + \sum_{d:D} y && \text{(RSUM4)} \\
\delta \lhd c \rhd \delta &\to \delta && \text{(RCOND0}'\text{)} \\
(x + y) \lhd c \rhd \delta &\to x \lhd c \rhd \delta + y \lhd c \rhd \delta && \text{(RCOND7)} \\
\Big(\sum_{d:D} x\Big) \lhd c \rhd \delta &\to \sum_{d:D} x \lhd c \rhd \delta && \text{(RSUM12)} \\
(x \lhd c_1 \rhd \delta) \lhd c_2 \rhd \delta &\to x \lhd c_1 \wedge c_2 \rhd \delta && \text{(RCOND4)}
\end{aligned}
$$

Table 10: Rewrite rules defining *rewr* (Part 1).

The optimization rules presented in Table 13 are not needed to get the desired restricted syntactic form, but can be used to simplify the terms. They could be applied with higher priority than the rules in Tables 10 and 11 to achieve possible reductions. Note that the rule (RSCA′) could lead to optimizations only in cases where x is completely guarded, and y or z are not.

Proposition 3.3. *The commutative/associative term rewriting system of Tables 10 and 11 is strongly terminating.*

Proof. Termination can be proved using the AC-RPO technique [34] for following order on the operations:

$$\partial_H > \tau_I > \rho_R > \parallel > \cdot > \_ \lhd c \rhd \delta > \sum > + > \mathsf{a}(\overrightarrow{t}) > \delta$$

$$
\begin{array}{rcll}
\partial_H(\mathsf{a}(\overrightarrow{t})) & \to & \delta \quad \text{if } \mathsf{a} \in H & \text{(RD2)}\\
\partial_H(\mathsf{a}(\overrightarrow{t})) & \to & \mathsf{a}(\overrightarrow{t}) \quad \text{if } \mathsf{a} \notin H & \text{(RD1)}\\
\partial_H(\tau) & \to & \tau & \text{(RD1}'\text{)}\\
\partial_H(\delta) & \to & \delta & \text{(RD2}'\text{)}\\
\partial_H(x+y) & \to & \partial_H(x)+\partial_H(y) & \text{(RD3)}\\
\partial_H(x\cdot y) & \to & \partial_H(x)\cdot\partial_H(y) & \text{(RD4)}\\
\partial_H\Big(\sum_{d:D} x\Big) & \to & \sum_{d:D}\partial_H(x) & \text{(RSUM8)}\\
\partial_H(x \lhd c \rhd \delta) & \to & \partial_H(x) \lhd c \rhd \delta & \text{(RD5)}\\
\partial_{H_1}(\partial_{H_2}(x)) & \to & \partial_{H_1\cup H_2}(x) & \text{(RDD)}\\
\partial_H(\tau_I(x)) & \to & \tau_I(\partial_{H\setminus I}(x)) & \text{(RDT)}\\
\partial_H(\rho_R(x)) & \to & \rho_R(\partial_{R^{-1}(H)}(x)) & \text{(RDR)}\\
\tau_I(\mathsf{a}(\overrightarrow{t})) & \to & \tau \quad \text{if } \mathsf{a} \in I & \text{(RT2)}\\
\tau_I(\mathsf{a}(\overrightarrow{t})) & \to & \mathsf{a}(\overrightarrow{t}) \quad \text{if } \mathsf{a} \notin I & \text{(RT1)}\\
\tau_I(\tau) & \to & \tau & \text{(RT2}'\text{)}\\
\tau_I(\delta) & \to & \delta & \text{(RT1}'\text{)}\\
\tau_I(x+y) & \to & \tau_I(x)+\tau_I(y) & \text{(RT3)}\\
\tau_I(x\cdot y) & \to & \tau_I(x)\cdot\tau_I(y) & \text{(RT4)}\\
\tau_I\Big(\sum_{d:D} x\Big) & \to & \sum_{d:D}\tau_I(x) & \text{(RSUM9)}\\
\tau_I(x \lhd c \rhd \delta) & \to & \tau_I(x) \lhd c \rhd \delta & \text{(RT5)}\\
\tau_{I_1}(\tau_{I_2}(x)) & \to & \tau_{I_1\cup I_2}(x) & \text{(RTT)}\\
\tau_I(\rho_R(x)) & \to & \rho_R(\tau_{R^{-1}(I)}(x)) & \text{(RTR)}\\
\rho_R(\mathsf{a}(\overrightarrow{t})) & \to & R(\mathsf{a})(\overrightarrow{t}) & \text{(RR1)}\\
\rho_R(\tau) & \to & \tau & \text{(RRT)}\\
\rho_R(\delta) & \to & \delta & \text{(RRD)}\\
\rho_R(x+y) & \to & \rho_R(x)+\rho_R(y) & \text{(RR3)}\\
\rho_R(x\cdot y) & \to & \rho_R(x)\cdot\rho_R(y) & \text{(RR4)}\\
\rho_R\Big(\sum_{d:D} x\Big) & \to & \sum_{d:D}\rho_R(x) & \text{(RSUM10)}\\
\rho_R(x \lhd c \rhd \delta) & \to & \rho_R(x) \lhd c \rhd \delta & \text{(RR5)}\\
\rho_{R_1}(\rho_{R_2}(x)) & \to & \rho_{R_1\circ R_2}(x) & \text{(RRR)}
\end{array}
$$

Table 11: Rewrite rules defining *rewr* (Part 2).

□

Lemma 3.4. *For any process term p not containing $p_1 \lhd c \rhd p_2$, where $p_2 \not\equiv \delta$, we have that $rewr(p)$ has the syntax defined in Table 9.*

Proof. Let $q = rewr(p)$. It can be seen from the rewrite rules that they preserve the syntax (2.1).

$$\begin{aligned}
x + y &= y + x \\
x + (y + z) &= (x + y) + z \\
(x \cdot y) \cdot z &= x \cdot (y \cdot z) \\
x \parallel y &= y \parallel x \\
x \parallel (y \parallel z) &= (x \parallel y) \parallel z
\end{aligned}$$

Table 12: The rewriting is performed modulo these identities.

$$\begin{aligned}
x + x &\to x && \text{(RA3)} \\
x \lhd c \rhd x &\to x && \text{(RCOND0)} \\
x \lhd \mathbf{t} \rhd y &\to x && \text{(RCOND1)} \\
x \lhd \mathbf{f} \rhd y &\to y && \text{(RCOND2)} \\
x \lhd c_1 \rhd \delta + x \lhd c_2 \rhd \delta &\to x \lhd c_1 \vee c_2 \rhd \delta && \text{(RCOND5)} \\
(x_1 \lhd c \rhd x_2) \cdot (y_1 \lhd c \rhd y_2) &\to x_1 \cdot y_1 \lhd c \rhd x_2 \cdot y_2 && \text{(RSCA)} \\
x \cdot (y \lhd c \rhd z) &\to x \cdot y \lhd c \rhd x \cdot z && \text{(RSCA}'\text{)} \\
\tau_I(\partial_H(x)) &\to \tau_{I \setminus H}(\partial_H(x)) && \text{(RTD)} \\
\rho_R(\tau_I(\partial_H(x))) &\to \rho_{R_{I \cup H}}(\tau_I(\partial_H(x))) && \text{(RRTD)} \\
\rho_R(\tau_I(x)) &\to \rho_{R_I}(\tau_I(x)) && \text{(RRT}'\text{)} \\
\rho_R(\partial_H(x)) &\to \rho_{R_H}(\partial_H(x)) && \text{(RRD}'\text{)} \\
\partial_\emptyset(x) &\to x && \text{(RD0)} \\
\tau_\emptyset(x) &\to x && \text{(RT0)} \\
\rho_{R_{ActLab}}(x) &\to x && \text{(RR0)}
\end{aligned}$$

where $R_S(\mathsf{a})$ for $S \subseteq \mathit{ActLab}$ is defined to be equal to a if $\mathsf{a} \in S$ and to $R(\mathsf{a})$ otherwise.

Table 13: Optimization rules.

Suppose q does not satisfy the syntax defined in Table 9. All of the possibilities for q that exist imply that q is reducible. We give some of the possibilities below; for the rest the appropriate rules can be easily found in Table 11.

- $q = \delta + p_1$. Can be reduced by (RA6).
- $q = \delta \parallel p_1$. Can be reduced by (SC6).
- $q = \delta \cdot p_1$. Can be reduced by (RA7).
- $q = (\sum_{d:D} p_1) \cdot p_2$. Can be reduced by (RSUM5).
- $q = (p_1 \lhd c \rhd \delta) \cdot p_2$. Can be reduced by (RCOND6).
- $q = \sum_{d:D} \delta$. Can be reduced by (RSUM1$'$).
- $q = \sum_{d:D}(p_1 + p_2)$. Can be reduced by (RSUM4).
- $q = \delta \lhd c \rhd \delta$. Can be reduced by (RCOND0$'$).

- $q = (p_1 + p_2) \lhd c \rhd \delta$. Can be reduced by (RCOND7).
- $q = (\sum_{d:D} p_1) \lhd c \rhd \delta$. Can be reduced by (RSUM12).
- $q = (p_1 \lhd c_1 \rhd \delta) \lhd c_2 \rhd \delta$. Can be reduced by (RCOND4).

□

Proposition 3.5. *Let G_2 be the result of applying the rewriting to G_1. Then $G_2 = G_1$.*

Proof. Taking into account that G_1 does not contain nested occurrences of bound variables, each rewrite rule is a consequence of the axioms of μCRL. By Lemma 2.14 we get $G_2 = G_1$. □

As a result of applying simple rewriting the number of equations obviously remains the same. The right hand sides of the equations may grow in a linear fashion with respect to the number of operation symbols of sort *Proc* occurrences. This is because a number of rules copy operation symbols when distributing over $+$ or $\cdot$ (for example the rule (RSUM4) copies the summation symbol). It can be checked that the total number of $+, \cdot$ and $\|$ occurrences does not increase during the rewriting (except for certain optimization rules). Therefore the number of such copyings is linear in the term size. The number of occurrences of action labels and process names does not increase during the rewriting.

3.4 Adding New Process Equations

In this step we reduce the complexity of terms in the right hand sides of the G_2 equations even further by the introduction of new process equations. In some cases we take a subterm of a right hand side and substitute it by a fresh process name parameterized by (at least) all free variables that appear in that subterm. As the result we get a system of process equations G_3 with equations in pre-PEGNF. Such a transformation can be performed for all equations $\mathsf{X}(\overrightarrow{d_\mathsf{X}{:}D_\mathsf{X}}) = p_\mathsf{X}$ by replacing them with $\mathsf{X}(\overrightarrow{d_\mathsf{X}{:}D_\mathsf{X}}) = S_1(\overrightarrow{d_\mathsf{X}{:}D_\mathsf{X}}, p_\mathsf{X})$.

The transformations S_1 and S_2 are defined in the Table 14, where *fresh_var* represents a fresh process name, and *add* represents addition of the equation to the resulting system. Formally, S_1 and S_2 induce operations $\hat{S}_1$ and $\hat{S}_2$ that operate on sets of equations and are defined in the expected way (those operations actually transform the system of recursive equations).

The transformation S_1 distributes over all operations that preserve the form of right hand side of equations in pre-PEGNF. These are all operations except for parallel and sequential compositions, hiding, renaming and encapsulation, for which we apply the transformation S_2. The transformation S_2 distributes over all operations that preserve the syntax (3.1). These are all operations except for alternative composition, sums and conditions, for which we introduce new equations, as preserving them would break pre-PEGNF. In the following we provide a simple example of the transformation.

Example 3.6. Let $G = \{\mathsf{X}(d{:}D) = \mathsf{a}(d) \cdot (\mathsf{b}(d) + \mathsf{X}(f(d)))\}$ be a given system of process equations. After applying the transformation S_1 we get the system $G' = \{\mathsf{X}(d{:}D) = \mathsf{a}(d) \cdot \mathsf{Y}(d), \mathsf{Y}(d{:}D) = \mathsf{b}(d) + \mathsf{X}(f(d))\}$ which is in pre-PEGNF.

Proposition 3.7. *The functions S_1 and S_2 are well-defined.*

Proof. Using the order on the operations $S_1 > +, S_1 > \sum, S_2 > \cdot, S_2 > \|, S_2 > \rho_R, S_2 > \tau_I, S_2 > \partial_H$ it can be shown that infinite recursion is not possible for any admissible arguments given. □

Lemma 3.8. *All process equations in G_3 are in pre-PEGNF.*

Proof. It is easy to see that S_2 produces terms that satisfy the syntax (3.1) from Definition 3.1. The transformation S_1 can add only $+$, $\sum$ or $\lhd\rhd$ operations to them at the correct places, with regard to the syntax (3.1). The only interesting transformation to consider is $S_1\Big(S, \sum_{d:D} p\Big) \rightarrow$

$$\begin{aligned}
S_1(S,\mathsf{a}(\overrightarrow{t})) &\to \mathsf{a}(\overrightarrow{t})\\
S_1(S,\delta) &\to \delta\\
S_1(S,\mathsf{X}(\overrightarrow{t})) &\to \mathsf{X}(\overrightarrow{t})\\
S_1(S,p_1\cdot p_2) &\to S_2(S,p_1\cdot p_2)\\
S_1(S,p_1\parallel p_2) &\to S_2(S,p_1\parallel p_2)\\
S_1(S,p_1+p_2) &\to S_1(S,p_1)+S_1(S,p_2)\\
S_1(S,p\lhd c\rhd\delta) &\to S_2(S,p)\lhd c\rhd\delta\\
S_1\Big(S,\sum_{d:D}p\Big) &\to \sum_{d:D}S_1(S\mathbin{\&}d{:}D,p)\\
S_1(S,\partial_H(p)) &\to S_2(S,\partial_H(p))\\
S_1(S,\tau_I(p)) &\to S_2(S,\tau_I(p))\\
S_1(S,\rho_R(p)) &\to S_2(S,\rho_R(p))
\end{aligned}$$

$$\begin{aligned}
S_2(S,\mathsf{a}(\overrightarrow{t})) &\to \mathsf{a}(\overrightarrow{t})\\
S_2(S,\delta) &\to \delta\\
S_2(S,\mathsf{X}(\overrightarrow{t})) &\to \mathsf{X}(\overrightarrow{t})\\
S_2(S,p_1\cdot p_2) &\to S_2(S,p_1)\cdot S_2(S,p_2)\\
S_2(S,p_1\parallel p_2) &\to S_2(S,p_1)\parallel S_2(S,p_2)\\
S_2(S,p_1+p_2) &\to (\mathsf{Y}:=\mathit{fresh\_var})(S);\\
&\qquad \mathit{add}\big(\mathsf{Y}(S)=S_1(S,p_1+p_2)\big)\\
S_2(S,p\lhd c\rhd\delta) &\to (\mathsf{Y}:=\mathit{fresh\_var})(S);\\
&\qquad \mathit{add}\big(\mathsf{Y}(S)=S_1(S,p\lhd c\rhd\delta)\big)\\
S_2\Big(S,\sum_{d:D}p\Big) &\to (\mathsf{Y}:=\mathit{fresh\_var})(S);\\
&\qquad \mathit{add}\Big(\mathsf{Y}(S)=S_1\Big(S,\textstyle\sum_{d:D}p\Big)\Big)\\
S_2(S,\partial_H(p)) &\to \partial_H(S_2(S,p))\\
S_2(S,\tau_I(p)) &\to \tau_I(S_2(S,p))\\
S_2(S,\rho_R(p)) &\to \rho_R(S_2(S,p))
\end{aligned}$$

Table 14: Transformations S_1 and S_2.

$\sum_{d:D} S_1(S \mathbin{\&} d{:}D, p)$, as we need to show that p is not of the form $p_1 + p_2$. This follows from the fact that p satisfies the syntax defined in Table 9. □

Proposition 3.9. *For any process name* X *in* G_2 *we have* $(\mathsf{X}, G_3) = (\mathsf{X}, G_2)$.

Proof. The statement follows from Lemma 2.15. □

The transformation described in this subsection does not increase the size of terms. The number of process equations may increase linearly in the size of terms in the original system.

3.5 Guarding

Next we transform the equations of G_3 to PEGNF. To this end, we use the function $\mathit{guard} : \mathit{DVar} \times \mathit{Terms}(|G|) \to \mathit{Terms}(|G|)$, which replaces unguarded occurrences of process names with the right hand sides of their defining equations. It is defined as follows:

$$\begin{aligned}
&\mathit{guard}\Big(S,\sum_{i\in I}\sum_{\overrightarrow{e_i:E_i}}p_i\lhd c_i\rhd\delta\Big)=\mathit{rewr}\Big(\sum_{i\in I}\sum_{\overrightarrow{e_i:E_i}}\mathit{guard}(S\cup\{\overrightarrow{e_i}\},p_i)\lhd c_i\rhd\delta\Big)\\
&\mathit{guard}(S,\mathsf{a}(\overrightarrow{t}))=\mathsf{a}(\overrightarrow{t})\\
&\mathit{guard}(S,\delta)=\delta\\
&\mathit{guard}(S,\mathsf{Y}(\overrightarrow{t}))=\mathit{guard}\Big(S,S_0\big(S\setminus\{\mathit{pars}(\mathsf{Y})\},\mathit{rhs}(\mathsf{Y})\big)\big[\mathit{pars}(\mathsf{Y}):=\overrightarrow{t}\big]\Big)\\
&\mathit{guard}(S,p_1\cdot p_2)=\mathit{rewr}\Big(\mathit{simpl}\big(\mathit{guard}(S,p_1)\cdot p_2\big)\Big)\\
&\mathit{guard}(S,\rho_R\circ\tau_I\circ\partial_H(p))=\mathit{rewr}\big(\rho_R\circ\tau_I\circ\partial_H(\mathit{guard}(S,p))\big)
\end{aligned}$$

$$\begin{aligned}\mathit{guard}\big(S, p_1 \parallel p_2\big) = \mathit{rewr}\Big(&\mathit{simpl}\big(\mathit{guard}(S,p_1) \mathbin{\lfloor\!\lfloor} p_2\big) + \mathit{simpl}\big(\mathit{guard}(S,p_2) \mathbin{\lfloor\!\lfloor} p_1\big)\\ &+ \mathit{simpl}\big(\mathit{guard}(S,p_1) \mid \mathit{guard}(S,p_2)\big)\Big)\end{aligned}$$

Here we use the function *rewr* from Subsection 3.3 and the function S_0 from Subsection 3.2. The function *guard* keeps track of the free variables that can occur in a term that is being guarded. In case we do the replacement of a process name by the right hand side of its defining equation (fourth clause), we first rename its bound variables so that they do not become bound twice, then we substitute the values of the parameters, and then apply *guard* to the resulting term. The function *simpl* is defined as follows:

$$\begin{aligned}&\mathit{simpl}\Bigg(\Big(\sum_{i\in I}\sum_{\overrightarrow{e_i:E_i}} \mathsf{a}_i(\overrightarrow{t_i})\cdot p_i \lhd c_i \rhd \delta + \sum_{j\in J}\sum_{\overrightarrow{e_j:E_j}} \mathsf{a}_j(\overrightarrow{t_j}) \lhd c_j \rhd \delta\Big)\cdot p\Bigg)\\ &\quad= \sum_{i\in I}\sum_{\overrightarrow{e_i:E_i}} \mathsf{a}_i(\overrightarrow{t_i})\cdot \big(p_i\cdot p\big) \lhd c_i \rhd \delta + \sum_{j\in J}\sum_{\overrightarrow{e_j:E_j}} \mathsf{a}_j(\overrightarrow{t_j})\cdot p \lhd c_j \rhd \delta\end{aligned}$$

$$\begin{aligned}&\mathit{simpl}\Bigg(\Big(\sum_{i\in I}\sum_{\overrightarrow{e_i:E_i}} \mathsf{a}_i(\overrightarrow{t_i})\cdot p_i \lhd c_i \rhd \delta + \sum_{j\in J}\sum_{\overrightarrow{e_j:E_j}} \mathsf{a}_j(\overrightarrow{t_j}) \lhd c_j \rhd \delta\Big) \mathbin{\lfloor\!\lfloor} p\Bigg)\\ &\quad= \sum_{i\in I}\sum_{\overrightarrow{e_i:E_i}} \mathsf{a}_i(\overrightarrow{t_i})\cdot \big(p_i \parallel p\big) \lhd c_i \rhd \delta + \sum_{j\in J}\sum_{\overrightarrow{e_j:E_j}} \mathsf{a}_j(\overrightarrow{t_j})\cdot p \lhd c_j \rhd \delta\end{aligned}$$

$$\begin{aligned}&\mathit{simpl}\Bigg(\Big(\sum_{i\in I}\sum_{\overrightarrow{e_i:E_i}} \mathsf{a}_i(\overrightarrow{f_i}(\overrightarrow{d,e_i}))\cdot p_i(\overrightarrow{d,e_i}) \lhd c_i(\overrightarrow{d,e_i}) \rhd \delta\\ &\qquad\qquad + \sum_{j\in J}\sum_{\overrightarrow{e_j:E_j}} \mathsf{a}_j(\overrightarrow{f_j}(\overrightarrow{d,e_j})) \lhd c_j(\overrightarrow{d,e_j}) \rhd \delta\Big)\\ &\qquad \mid \Big(\sum_{i\in I'}\sum_{\overrightarrow{e'_i:E'_i}} \mathsf{a}'_i(\overrightarrow{f'_i}(\overrightarrow{d',e'_i}))\cdot p'_i(\overrightarrow{d',e'_i}) \lhd c'_i(\overrightarrow{d',e'_i}) \rhd \delta\\ &\qquad\qquad + \sum_{j\in J'}\sum_{\overrightarrow{e'_j:E'_j}} \mathsf{a}'_j(\overrightarrow{f'_j}(\overrightarrow{d',e'_j})) \lhd c'_j(\overrightarrow{d',e'_j}) \rhd \delta\Big)\Bigg)\\ &\quad= \sum_{(k,l)\in I\gamma I'}\ \sum_{\overrightarrow{e_k:E_k,e'_l:E'_l}} \gamma(\mathsf{a}_k,\mathsf{a}'_l)(\overrightarrow{f_k}(\overrightarrow{d,e_k}))\cdot \big(p_k(\overrightarrow{d,e_k}) \parallel p_l(\overrightarrow{d',e'_l})\big)\\ &\qquad\qquad \lhd \overrightarrow{f_k}(\overrightarrow{d,e_k}) = \overrightarrow{f'_l}(\overrightarrow{d',e'_l}) \wedge c_k(\overrightarrow{d,e_k}) \wedge c'_l(\overrightarrow{d',e'_l}) \rhd \delta\\ &\quad+ \sum_{(k,l)\in I\gamma J'}\ \sum_{\overrightarrow{e_k:E_k,e'_l:E'_l}} \gamma(\mathsf{a}_k,\mathsf{a}'_l)(\overrightarrow{f_k}(\overrightarrow{d,e_k}))\cdot p_k(\overrightarrow{d,e_k})\\ &\qquad\qquad \lhd \overrightarrow{f_k}(\overrightarrow{d,e_k}) = \overrightarrow{f'_l}(\overrightarrow{d',e'_l}) \wedge c_k(\overrightarrow{d,e_k}) \wedge c'_l(\overrightarrow{d',e'_l}) \rhd \delta\\ &\quad+ \sum_{(k,l)\in J\gamma I'}\ \sum_{\overrightarrow{e_k:E_k,e'_l:E'_l}} \gamma(\mathsf{a}_k,\mathsf{a}'_l)(\overrightarrow{f_k}(\overrightarrow{d,e_k}))\cdot p_l(\overrightarrow{d',e'_l})\\ &\qquad\qquad \lhd \overrightarrow{f_k}(\overrightarrow{d,e_k}) = \overrightarrow{f'_l}(\overrightarrow{d',e'_l}) \wedge c_k(\overrightarrow{d,e_k}) \wedge c'_l(\overrightarrow{d',e'_l}) \rhd \delta\\ &\quad+ \sum_{(k,l)\in J\gamma J'}\ \sum_{\overrightarrow{e_k:E_k,e'_l:E'_l}} \gamma(\mathsf{a}_k,\mathsf{a}'_l)(\overrightarrow{f_k}(\overrightarrow{d,e_k}))\\ &\qquad\qquad \lhd \overrightarrow{f_k}(\overrightarrow{d,e_k}) = \overrightarrow{f'_l}(\overrightarrow{d',e'_l}) \wedge c_k(\overrightarrow{d,e_k}) \wedge c'_l(\overrightarrow{d',e'_l}) \rhd \delta\end{aligned}$$

where $P\gamma Q = \{(p,q) \in P \times Q \mid \gamma(\mathsf{a}_p, \mathsf{a}'_q)$ is defined$\}$. The function *simpl* shows that for any term p^1 and p^2 in the form of a right hand side of an equation in PEGNF, and for any term p having syntax (3.1) we can transform $p^1 \cdot p$, $p^1 \mathbin{\lfloor\!\lfloor} p$ and $p^1 \mid p^2$ to the form of a right hand side of an equation in PEGNF by applying the axioms of μCRL.

Proposition 3.10. *For any finite system G_3 in pre-PEGNF with acyclic PNUDG, and any process name X in it, the function guard is well-defined on $rhs(\mathsf{X}, G_3)$.*

Proof. Let n be the number of equations in G_3. The only clause that makes the argument of *guard* larger is the third one. Due to the fact that PNUDG is acyclic, this rule cannot be applied more than n times deep (otherwise for some process name Z we would have a cycle). $\square$

We define the system G_4 in the following way. For each equation

$$\mathsf{X}(\overrightarrow{d{:}D}) = \sum_{i\in I} \sum_{\overrightarrow{e_i{:}E_i}} p_i(\overrightarrow{d, e_i}) \lhd c_i(\overrightarrow{d, e_i}) \rhd \delta$$

in G_3 we add

$$\mathsf{X}(\overrightarrow{d{:}D}) = guard\Big(\{\,\overrightarrow{d}\,\}, \sum_{i\in I} \sum_{\overrightarrow{e_i{:}E_i}} p_i(\overrightarrow{d, e_i}) \lhd c_i(\overrightarrow{d, e_i}) \rhd \delta\Big)$$

to G_4.

Lemma 3.11. *The equations in G_4 are in PEGNF.*

Proof. Due to Proposition 3.10 we can apply induction on the definition of *guard*. The second and third clause of the *guard* definition are the induction base and they are trivially in PEGNF. The fourth clause is also trivial. In the first clause the only rules in Tables 10 and 11 that can be applied are (RCOND7), (RSUM12), (RCOND4) and (RSUM4), which bring the right hand side to the desired form. (In case the inner *guard* returns δ, the rewrite rules that can be applied are (RCOND0′), (RSUM1′) and (RA6).)

For the sixth clause *rewr* can be applied with all the rules for renaming, hiding and encapsulation, which preserve PEGNF. For the fifth and seventh clauses we use the fact that *simpl* produces terms in PEGNF. $\square$

Proposition 3.12. *Let G_3 and G_4 be defined as above. Then $G_3 = G_4$.*

Proof. It was already noted before that the transformations performed by *rewr* and S_0 are derivable from the axioms of μCRL. It is easy to see that the transformations performed by *simpl* are derivable from the axioms as well. According to Lemma 2.16 and Lemma 2.14 all transformations performed by *guard* lead to equivalent systems. We note that care has been taken to rename some data variables during the substitution (in the third clause of *guard* definition) in order to make the substitution and the following applications of the axioms sound. $\square$

The transformation performed in this step does not increase the number of equations, but their sizes may grow exponentially, due to application of (A4). An example of such an exponential growth can be found in [21]. We also note that similar growth is possible due to application of axioms (CM4) for the left merge, and (CM8) and (CM9) for communication.

Summarizing, the initial and the current μCRL specification are related by $(\mathsf{X}, G) = (\mathsf{X}, G_4)$, and we have not added any extra data type definitions to the current specification up till now.

4. From PEGNF to One Equation

In this section we transform the system of process equations G_4 in PEGNF (cf. Definition 3.1) into G_7 which consists of a single process equation in post-PEGNF with a specially constructed parameter list.

4.1 Transformation to post-PEGNF

First, we transform all equations of G_4 into post-PEGNF.

Definition 4.1. A μCRL process equation is in *post-PEGNF* iff it is of the form:

$$\begin{aligned}\mathsf{X}(\overrightarrow{d{:}D}) = &\sum_{i\in I}\sum_{\overrightarrow{e_i:E_i}} \mathsf{a}_i(\overrightarrow{f_i}(\overrightarrow{d,e_i}))\cdot p_i(\overrightarrow{d,e_i}) \lhd c_i(\overrightarrow{d,e_i}) \rhd \delta \\ &+\sum_{j\in J}\sum_{\overrightarrow{e_j:E_j}} \mathsf{a}_j(\overrightarrow{f_j}(\overrightarrow{d,e_j})) \lhd c_j(\overrightarrow{d,e_j}) \rhd \delta\end{aligned}$$

where I and J are disjoint, and all $p_i(\overrightarrow{d,e_i})$ are terms of the following syntax:

$$p ::= \mathsf{Y}(\overrightarrow{t}) \mid p\cdot p \mid p \parallel p \mid \rho_R(\tau_I(\partial_H(p\parallel p))) \mid \rho_R(\tau_I(\partial_H(\mathsf{Y}(\overrightarrow{t})))) \tag{4.1}$$

In order to do this we need to eliminate all actions and δ that appear in terms p_i in PEGNF. This is achieved by introducing a new process name X_a for each action a that occurs inside the process terms p_i, with parameters corresponding to those of the action (and a new process name X_δ for δ). Thus we add equations $\mathsf{X}_\mathsf{a}(\overrightarrow{d_\mathsf{a}{:}D_\mathsf{a}}) = \mathsf{a}(\overrightarrow{d_\mathsf{a}})$ and $\mathsf{X}_\delta = \delta$ to the system, and replace the occurrences of actions $\mathsf{a}(\overrightarrow{t})$ by $\mathsf{X}_\mathsf{a}(\overrightarrow{t})$, and δ by X_δ.

Proposition 4.2. *Let the system G_5 of process equations be obtained after postprocessing the system G_4 as described above. Then for all $\mathsf{X}\in|G_4|$ we have $(\mathsf{X},G_5) = (\mathsf{X},G_4)$ and G_5 is in post-PEGNF.*

Proof. According to Lemma 2.15 this transformation is correct and leads to a system that obviously is in PEGNF. □

As a possible optimization during this postprocessing step, the following slightly different strategy can be applied. If we encounter a subterm $\mathsf{a}\cdot\mathsf{Y}$ in p_i, we replace it by a new process name (with parameters for both a and Y), and add the equation for it to the system. This optimization goes along the lines of a *regular linearization procedure* (see the Conclusions), which is a more general case of such an optimization.

It is also possible to eliminate renaming, hiding and encapsulation operations that do not have parallel composition in their arguments by introducing more terms of the form $\rho_R(\tau_I(\partial_H(p_1\parallel p_2)))$, thus removing $\rho_R(\tau_I(\partial_H(\mathsf{Y}(\overrightarrow{t}))))$ from the grammar (4.1). This can be done by introducing a fresh process name Z for every different $\rho_R(\tau_I(\partial_H(\mathsf{Y}(\overrightarrow{t}))))$ together with the defining equation $\mathsf{Z}(\overrightarrow{d_\mathsf{Y}{:}D_\mathsf{Y}}) = \rho_R(\tau_I(\partial_H(\mathsf{Y}(\overrightarrow{t}))))$. By taking the *rhs*$(\mathsf{Y})$ and applying the rewrite rules for the renaming operators we either get rid of the construct, or get a new instance of it, possibly with different R, I, and/or H. Given the fact that the set of actions is finite, the number of different R, I, and H is also finite, and therefore we cannot introduce an infinite number of fresh process names in this way.

An interesting question is whether we can eliminate $\rho_R(\tau_I(\partial_H(p_1\parallel p_2)))$ by introducing more process equations and renamings of the form $\rho_R(\tau_I(\partial_H(\mathsf{Y}(\overrightarrow{t}))))$. An interesting example would be $\mathsf{X} = \mathsf{a}\cdot\partial_{\{\mathsf{b}\}}(\mathsf{X}\parallel\partial_{\{\mathsf{b}\}}(\mathsf{X}\parallel\mathsf{X}))$ with $\gamma(\mathsf{a},\mathsf{a}) = \mathsf{a}$.

It remains an interesting question whether all renaming operations can be eliminated without the use of infinite data types. We conjecture that it is not possible. The partial elimination of renaming operators do not lead to simplifications of the data type that we need to encode. Total elimination of renaming operations would provide such a simplification.

4.2 Formal Parameters Harmonization

In this subsection we make the formal parameters of all μCRL process names in G_5 uniform, in order to compress all equations in one. This harmonization is defined by the following steps.

1. We rename the data variables with the same names but with different types in different processes. This can easily be done (see Section 3.2).

2. We create the common list of data parameters $\overrightarrow{d{:}D}$ by taking the *set* of all data parameters in all equations, and giving some order to it.

3. For each process name X in G we define a mapping M_X from its parameter list $\overrightarrow{D_\mathsf{X}}$ to the common parameter list $\overrightarrow{D}$. This mapping is such that each newly created parameter is a constant. (Recall that a correct μCRL specification contains constants for each declared data sort.)

4. Then we replace all left hand sides of process equations $\mathsf{X}(\overrightarrow{d_\mathsf{X}{:}D_\mathsf{X}})$ by $\mathsf{X}(\overrightarrow{d{:}D})$, and all process terms of the form $\mathsf{Y}(\overrightarrow{t})$ in right hand sides of process equations by $\mathsf{Y}(M_\mathsf{Y}(\overrightarrow{t}))$.

Proposition 4.3. *Let the system G_6 of process equations be obtained after harmonization of the system G_5 as described above. Then for all $\mathsf{X} \in |G_5|$ we have $(\mathsf{X}(M_\mathsf{X}(\overrightarrow{d_\mathsf{X}})), G_6) = (\mathsf{X}(\overrightarrow{d_\mathsf{X}}), G_5)$.*

Proof. By Lemma 2.17 it follows that this transformation yields an equivalent system of equations. □

We remark that a more optimal strategy in terms of the number of data parameters, than 'global harmonization', is to merge as many parameters as possible. This can be achieved by renaming parameters of some processes so that they match the parameters of other processes, and therefore are not introduced in the general parameter list. In this case the number of parameters of some type s in the general list will be the maximal number of parameters of this type in an equation. A drawback of this optimization is the fact that we may lose parameter name information for some process names.

4.3 Making One Process Equation

In this subsection we combine n μCRL process equations from G_6 with the same formal parameters into one equation. This is done by adding a data parameter $s{:}\mathit{StateN}$ that represents the process names from $|G_6|$ to the parameters; adding a condition to each summand of each equation which checks that the value of data parameter s is the appropriate one; and combining all right hand sides into one alternative composition. The data type StateN is an enumerated data type with equality predicate. Natural numbers could be used for StateN. A finite data type is sufficient though.

More precisely, let G_6 be a system of n μCRL process equations in (post-)PEGNF with the same formal parameters.

$$\begin{aligned}
\mathsf{X}^1(\overrightarrow{d{:}D}) = &\sum_{i\in I^1}\sum_{\overrightarrow{e_i{:}E_i^1}} \mathsf{a}_i^1(\overrightarrow{f_i^1}(\overrightarrow{d,e_i}))\cdot p_i^1(\overrightarrow{d,e_i}) \lhd c_i^1(\overrightarrow{d,e_i}) \rhd \delta\\
&+\sum_{j\in J^1}\sum_{\overrightarrow{e_j{:}E_j^1}} \mathsf{a}_j^1(\overrightarrow{f_j^1}(\overrightarrow{d,e_j})) \lhd c_j^1(\overrightarrow{d,e_j}) \rhd \delta\\
&\vdots\\
\mathsf{X}^n(\overrightarrow{d{:}D}) = &\sum_{i\in I^n}\sum_{\overrightarrow{e_i{:}E_i^n}} \mathsf{a}_i^n(\overrightarrow{f_i^n}(\overrightarrow{d,e_i}))\cdot p_i^n(\overrightarrow{d,e_i}) \lhd c_i^n(\overrightarrow{d,e_i}) \rhd \delta\\
&+\sum_{j\in J^n}\sum_{\overrightarrow{e_j{:}E_j^n}} \mathsf{a}_j^n(\overrightarrow{f_j^n}(\overrightarrow{d,e_j})) \lhd c_j^n(\overrightarrow{d,e_j}) \rhd \delta
\end{aligned}$$

We define the system G_7 as a single (post-)PEGNF process equation in the following way:

$$\begin{aligned}
&\mathsf{X}(s{:}StateN, \overrightarrow{d{:}D})\\
&\quad = \sum_{i\in I^1} \sum_{\overrightarrow{e_i{:}E_i^1}} \mathsf{a}_i^1(\overrightarrow{f_i^1}(\overrightarrow{d, e_i})) \cdot S(p_i^1(\overrightarrow{d, e_i})) \lhd c_i^1(\overrightarrow{d, e_i}) \wedge s = 1 \rhd \delta\\
&\qquad + \sum_{j\in J^1} \sum_{\overrightarrow{e_j{:}E_j^1}} \mathsf{a}_j^1(\overrightarrow{f_j^1}(\overrightarrow{d, e_j})) \lhd c_j^1(\overrightarrow{d, e_j}) \wedge s = 1 \rhd \delta\\
&\qquad \vdots\\
&\qquad + \sum_{i\in I^n} \sum_{\overrightarrow{e_i{:}E_i^n}} \mathsf{a}_i^n(\overrightarrow{f_i^n}(\overrightarrow{d, e_i})) \cdot S(p_i^n(\overrightarrow{d, e_i})) \lhd c_i^n(\overrightarrow{d, e_i}) \wedge s = n \rhd \delta\\
&\qquad + \sum_{j\in J^n} \sum_{\overrightarrow{e_j{:}E_j^n}} \mathsf{a}_j^n(\overrightarrow{f_j^n}(\overrightarrow{d, e_j})) \lhd c_j^n(\overrightarrow{d, e_j}) \wedge s = n \rhd \delta
\end{aligned}$$

where $S(\mathsf{X}^s(\overrightarrow{t})) = \mathsf{X}(s, \overrightarrow{t})$ and distributes over $\cdot$, $\|$, ρ_R, τ_I and ∂_H.

Proposition 4.4. *Let G_6 and G_7 be as defined above, and let StateN enumerate $1, \ldots, n$. Then for any $s{:}StateN$, data term vector $\overrightarrow{t}$, and any $\mathsf{X}^s \in |G'|$ we have $(\mathsf{X}(s, \overrightarrow{t}), G_7) =_c (\mathsf{X}^s(\overrightarrow{t}), G_6)$.*

Proof. The equivalence is easy to derive with the following functions: $g_{\mathsf{X}^i}(\overrightarrow{t}) = \mathsf{X}(i, \overrightarrow{t})$ for each $i{:}StateN$, and $g_{\mathsf{X}}(s, \overrightarrow{t}) = \mathsf{X}^s(\overrightarrow{t})$. Note that identities of sort *StateN* are used in the derivations. □

Summarizing, for any X^s from the initial μCRL specification we have

$$(\mathsf{X}^s(\overrightarrow{t}), G) =_c (\mathsf{X}(s, M_{\mathsf{X}^s}(\overrightarrow{t})), G_7)$$

and the current specification additionally contains definitions of the *StateN* data type.

5. Introduction of Lists-of-Multisets

The final step in the linearization of μCRL processes consists of the introduction of a data parameter, that allows to model sequential and parallel compositions of process names with parameters, as a single process term. The data parameter should also encode renaming, hiding and encapsulation operations. In the case that no such sequential or parallel composition occurs in the equation, we do not apply this step. The renaming, hiding and encapsulation operations can, in this case, be eliminated using the transformation described in Section 4.1. We note that if no parallel composition operations were present, we could also eliminate the renaming, hiding and encapsulation operations and arrive at the pCRL case (see [21]). In this case the stack data type would be sufficient.

Definition 5.1. A process equation is called a *Linear Process Equation (LPE)* if it is of the form

$$\begin{aligned}
\mathsf{X}(\overrightarrow{d{:}D}) = &\sum_{i\in I} \sum_{\overrightarrow{e_i{:}E_i}} \mathsf{a}_i(\overrightarrow{f_i}(\overrightarrow{d, e_i})) \cdot \mathsf{X}(\overrightarrow{g_i}(\overrightarrow{d, e_i})) \lhd c_i(\overrightarrow{d, e_i}) \rhd \delta\\
&+ \sum_{j\in J} \sum_{\overrightarrow{e_j{:}E_j}} \mathsf{a}_j(\overrightarrow{f_j}(\overrightarrow{d, e_j})) \lhd c_j(\overrightarrow{d, e_j}) \rhd \delta
\end{aligned}$$

where I and J are disjoint sets of indices.

For the particular transformation described here, it is necessary that the process equation to be transformed has data parameters. This need not be the case after application of all preceding transformation steps. For instance the equation $\mathsf{X} = \mathsf{a} \cdot \mathsf{X} \cdot \ldots \cdot \mathsf{X} + \mathsf{b}$ does not have a data parameter. In this case we add a dummy data parameter (over a singleton data type, cf. Lemma 2.17) to apply the following transformation.

In the case of pCRL processes the data type needed was a stack (see Subsection 4.3 [21]). The case of μCRL is complicated in the following ways.

- Parallel composition is present in addition to sequential composition.
- Instead of a single process that was ready to be executed in the sequential case, we can have many parallel components represented by their state vectors, and the number of components can change during process execution.
- The components may communicate; thus simultaneous execution of two (handshaking) or more (multi-party communication) components is possible.
- The renaming, hiding and encapsulation operations can influence the way in which a component (or more than one of them) can be executed.

As a first step we consider the case with handshaking and no renaming, hiding and encapsulation operations; after that we add these operations, and finally outline the multi-party communication case. This is done in order to divide the explanation of the data type into smaller and more understandable parts. In addition to that, for each particular specification the appropriate data type can be used, depending on presence of the renaming operations and the type of communication used.

5.1 Parallel and Sequential Compositions with Handshaking

Assuming that no renaming operators are present, let G_7 contain a single μCRL process equation in post-PEGNF:

$$\begin{aligned}\mathsf{X}(\overrightarrow{d{:}D}) = &\sum_{i\in I}\sum_{\overrightarrow{e_i:E_i}} \mathsf{a}_i(\overrightarrow{f_i}(\overrightarrow{d,e_i}))\cdot p_i(\overrightarrow{d,e_i}) \lhd c_i(\overrightarrow{d,e_i}) \rhd \delta\\ &+\sum_{j\in J}\sum_{\overrightarrow{e_j:E_j}} \mathsf{a}_j(\overrightarrow{f_j}(\overrightarrow{d,e_j})) \lhd c_j(\overrightarrow{d,e_j}) \rhd \delta\end{aligned} \tag{5.1}$$

where $p_i(\overrightarrow{d,e_i})$ are terms of the following syntax:

$$p ::= \mathsf{X}(\overrightarrow{t}) \mid p\cdot p \mid p \parallel p \tag{5.2}$$

The form above differs from the LPE in having the sequential and parallel compositions of recursive calls instead of a single recursive call. We define the data type *State* (Appendix C.1) to represent the state vector $\overrightarrow{d{:}D}$. It is a simple tuple data type, that has a constructor $\mathit{state} : \overrightarrow{D} \to \mathit{State}$, projection functions $\mathit{pr}_i : \mathit{State} \to D_i$, equality predicate, if-then-else construction, and a greater-than predicate gt.[1]

The data type LM is used to represent a list containing state vectors $\overrightarrow{d}$ and/or multisets of elements of type LM. For the latter multisets we use the data type ML (see Appendix C.2 for the implementation details). The main idea is to represent a number of associative sequential compositions as a list, and a number of associative parallel compositions as a multiset. These lists and multisets can be nested up to arbitrary depth, as the terms can contain arbitrarily nested parallel and sequential compositions. A single state vector is represented as the list containing it. Thus the sort LM has three constructors:

[1]In the text, often we do not distinguish between $\overrightarrow{D}$ and *State*, and do not use *state* and pr_i, but use vector notation instead.

- $\mathit{LM0} :\to \mathit{LM}$, representing the empty list,
- $\mathit{seq1} : \mathit{State} \times \mathit{LM} \to \mathit{LM}$, with $\mathit{seq1}(d, \mathit{lm})$ representing the list with the state vector d added as the head of lm,
- $\mathit{seqM} : \mathit{ML} \times \mathit{LM} \to \mathit{LM}$, with $\mathit{seqM}(\mathit{ml}, \mathit{lm})$ representing the list with the multiset ml added as the head of lm,

and the sort ML has two constructors:

- $\mathit{ML} : \mathit{LM} \to \mathit{ML}$, representing the multiset containing one list lm,
- $\mathit{par} : \mathit{LM} \times \mathit{ML} \to \mathit{ML}$, with $\mathit{par}(\mathit{lm}, \mathit{ml})$ representing the multiset with the list lm added to ml.

We note however, that with these constructors we can have different terms representing the same semantical value. For instance the following equivalent terms can be identified using the definitions in Appendix C.2:

- $\mathit{seqM}(\mathit{ML}(\mathit{LM0}), \mathit{lm}) = \mathit{lm}$,
- $\mathit{seqM}(\mathit{ML}(\mathit{seq1}(d, \mathit{lm1})), \mathit{lm}) = \mathit{seq1}(d, \mathit{conc}(\mathit{lm1}, \mathit{lm}))$,
- $\mathit{seqM}(\mathit{ML}(\mathit{seqM}(\mathit{ml}, \mathit{lm1})), \mathit{lm}) = \mathit{seqM}(\mathit{ml}, \mathit{conc}(\mathit{lm1}, \mathit{lm}))$,
- $\mathit{ML}(\mathit{seqM}(\mathit{ml}, \mathit{LM0})) = \mathit{ml}$,
- $\mathit{par}(\mathit{LM0}, \mathit{ml}) = \mathit{ml}$,
- $\mathit{par}(\mathit{lm}, \mathit{ML}(\mathit{lm1})) = \mathit{par}(\mathit{lm1}, \mathit{ML}(\mathit{lm}))$,
- $\mathit{par}(\mathit{seqM}(\mathit{ml}, \mathit{LM0}), \mathit{ml1}) = \mathit{comp}(\mathit{ml}, \mathit{ml1})$,

where the functions *conc* and *comp* are explained below. The first three identities are due to the fact that a multiset at the left hand side of a sequential composition is only needed if it contains at least two elements. The fourth identity says that putting a multiset into a list and then putting this list into a multiset does not change anything. The sixth identity is due to commutativity of parallel composition. The fifth and the last one say that a list at the left hand side of a parallel composition is only needed if it contains at least two elements.

There are more such identities, and we want to operate with the right hand sides of these identities only. We define the *normal forms* for lists and multisets in the following way. A term of sort LM is in normal form if it is in one of the following three forms:

- $\mathit{LM0}$,
- $\mathit{seq1}(d, \mathit{lm})$,
- $\mathit{seqM}(\mathit{ml}, \mathit{lm})$,

where

- d is a term of sort State,
- lm is a term of sort LM in normal form,
- ml is a term of sort ML in normal form having *par* as its outermost symbol.

A term of sort ML is in normal form if it is in one of the following two forms:

- $\mathit{ML}(\mathit{lm})$,

- $par(lm_1, \ldots par(lm_n, ML(lm_{n+1})) \ldots)$,

where for all $i \in \{1, \ldots, n+1\}$:

- lm, lm_i are terms of sort LM in normal form, and not of the form $seqM(ml, LM0)$,
- $lm_i \neq LM0$,
- $\neg gt(lm_i, lm_{i+1})$.

The gt function (greater than) is defined on LM and ML using the function gt on the sort $State$.

Preservation of normal forms is achieved by defining auxiliary functions that guarantee the generation of normal forms only, if the arguments are in normal forms:

- $conc : LM \times LM \rightarrow LM$,
- $conp : ML \times LM \rightarrow LM$,
- $mkml : LM \rightarrow ML$,
- $comp : ML \times ML \rightarrow ML$.

The first one is used to concatenate two lists. The second – to prepend a multiset to a list. The third – to make a multiset out of a list, and the last one – to concatenate two multisets. The implementation of these functions can be found in Appendix C.2. It can be shown by induction that if the arguments of the auxiliary functions are in normal form, then the result also rewrites to a term in normal form. In addition, this property can be shown for all functions in C that generate terms of sort LM or ML.

Preservation of normal forms gives us a simple way to define equality on the LM and ML data types. We can also check that the following properties are preserved for any lm and ml in normal form:

- $mkml(conp(ml, LM0)) = ml$,
- $conp(mkml(lm), LM0) = lm$.

We use the functions $seqc$ and $parc$ to represent sequential and parallel compositions on the sort LM, respectively. The following properties of these functions can be checked, under the assumption that all arguments are in normal form: associativity of $seqc$, associativity and commutativity of $parc$, $LM0$ is zero element for both functions.

For each term p_i from equation (5.1) we construct the term $\mathbf{mklm}_i[p_i] : State \times \overrightarrow{E_i} \rightarrow LM$, which gives us a way to represent the terms p_i as the terms of sort LM, in the following way:

$$\begin{aligned}
\mathbf{mklm}_i[\mathsf{X}(\overrightarrow{t})](\overrightarrow{t_d, t_{e_i}}) &= seq1(\overrightarrow{t}[\overrightarrow{d, e_i} := \overrightarrow{t_d, t_{e_i}}], LM0) \\
\mathbf{mklm}_i[p^1 \cdot p^2](\overrightarrow{t_d, t_{e_i}}) &= seqc(\mathbf{mklm}_i[p^1](\overrightarrow{t_d, t_{e_i}}), \mathbf{mklm}_i[p^2](\overrightarrow{t_d, t_{e_i}})) \\
\mathbf{mklm}_i[p^1 \parallel p^2](\overrightarrow{t_d, t_{e_i}}) &= parc(\mathbf{mklm}_i[p^1](\overrightarrow{t_d, t_{e_i}}), \mathbf{mklm}_i[p^2](\overrightarrow{t_d, t_{e_i}}))
\end{aligned}$$

As an example, if $p_i = (\mathsf{X}(n) \parallel \mathsf{X}(s(n))) \cdot \mathsf{X}(s(s(n)))$, then $\mathbf{mklm}_i[p_i](n) = \langle \{n, s(n)\}, s(s(n)) \rangle$ (or as a term $seqM(par(seq1(n, LM0), ML(seq1(s(n), LM0)), seq1(s(s(n)), LM0))))$.

As explained earlier, the data type LM represents a nesting of sequential and parallel compositions of the state vectors of process X defined by equation (5.1). For a given $lm{:}LM$, an important notion is the multiset of the state vectors of X that are *ready to be executed*. In other words, these are state vectors of X that are not prepended by other state vectors of X with a sequential composition. We call this multiset of state vectors of X from lm the *first layer* of lm. More formally, an occurrence of $d{:}State$ belongs to the *first layer* of lm if lm has no subterm of the form $seq1(d_1, lm_1)$ or $seqM(ml_1, lm_1)$ such that this occurrence of d is in lm_1.

The following functions involving the notion of the first layer are used in the definitions of the resulting LPE:

$\mathit{lenf} : LM \to Nat$	– the number of elements in the first layer
$\mathit{getf1} : LM \times Nat \to \overrightarrow{D}$	– get n-th element
$\mathit{replf1} : LM \times Nat \times LM \to LM$	– replace n-th element with an *lm*
$\mathit{remf1} : LM \times Nat \to LM$	– remove n-th element
$\mathit{replf2} : LM \times Nat \times Nat \times LM \times LM \to LM$	– replace two elements
$\mathit{replremf2} : LM \times Nat \times Nat \times LM \to LM$	– replace one and remove the other element
$\mathit{remf2} : LM \times Nat \times Nat \to LM$	– remove two elements

As can be seen from the implementation (Appendix C.2), removing an element from an *lm*:*LM* is equivalent to replacing it with *LM0*. In the example considered earlier, we have two elements in the first layer, where n is has number zero, and $s(n)$ has number one.

Assume the system G_7 consists of process equation X as defined in (5.1). We can now define a system L consisting of process equation Z, that mimics the behavior of X, in the following way:

$$
\begin{aligned}
&\mathsf{Z}(lm{:}LM) = \\
&\quad \sum_{i\in I}\sum_{n:Nat}\sum_{\overrightarrow{e_i:E_i}} \mathsf{a}_i(\overrightarrow{f_i}(\overrightarrow{\mathit{getf1}(lm,n),e_i}))\cdot \mathsf{Z}(\mathit{replf1}(lm,n,\mathbf{mklm}_i[p_i](\overrightarrow{\mathit{getf1}(lm,n),e_i}))) \\
&\qquad\quad \lhd\, n < \mathit{lenf}(lm) \wedge c_i(\overrightarrow{\mathit{getf1}(lm,n),e_i}) \rhd \delta \\
&\quad +\sum_{j\in J}\sum_{n:Nat}\sum_{\overrightarrow{e_j:E_j}} \mathsf{a}_j(\overrightarrow{f_j}(\overrightarrow{\mathit{getf1}(lm,n),e_j}))\cdot \mathsf{Z}(\mathit{remf1}(lm,n)) \\
&\qquad\quad \lhd\, n < \mathit{lenf}(lm) \wedge \mathit{remf1}(lm,n) \neq \langle\,\rangle \wedge c_j(\overrightarrow{\mathit{getf1}(lm,n),e_j}) \rhd \delta \\
&\quad +\sum_{j\in J}\sum_{n:Nat}\sum_{\overrightarrow{e_j:E_j}} \mathsf{a}_j(\overrightarrow{f_j}(\overrightarrow{\mathit{getf1}(lm,n),e_j})) \\
&\qquad\quad \lhd\, n < \mathit{lenf}(lm) \wedge \mathit{remf1}(lm,n) = \langle\,\rangle \wedge c_j(\overrightarrow{\mathit{getf1}(lm,n),e_j}) \rhd \delta \\
&\quad +\sum_{(k,l)\in I\gamma I}\sum_{n:Nat}\sum_{m:Nat}\sum_{\overrightarrow{e_k:E_k}}\sum_{\overrightarrow{e'_l:E_l}} \gamma(\mathsf{a}_k,\mathsf{a}_l)(\overrightarrow{f_k}(\overrightarrow{\mathit{getf1}(lm,n),e_k})) \\
&\qquad \cdot \mathsf{Z}(\mathit{replf2}(lm,n,m,\mathbf{mklm}_k[p_k](\overrightarrow{\mathit{getf1}(lm,n),e_k}),\mathbf{mklm}_l[p_l](\overrightarrow{\mathit{getf1}(lm,m),e'_l}))) \\
&\qquad\quad \lhd\, n < m \wedge m < \mathit{lenf}(lm) \wedge \overrightarrow{f_k}(\overrightarrow{\mathit{getf1}(lm,n),e_k}) = \overrightarrow{f_l}(\overrightarrow{\mathit{getf1}(lm,m),e'_l}) \\
&\qquad\qquad \wedge c_k(\overrightarrow{\mathit{getf1}(lm,n),e_k}) \wedge c_l(\overrightarrow{\mathit{getf1}(lm,m),e'_l}) \rhd \delta \\
&\quad +\sum_{(k,l)\in I\gamma J}\sum_{n:Nat}\sum_{m:Nat}\sum_{\overrightarrow{e_k:E_k}}\sum_{\overrightarrow{e'_l:E_l}} \gamma(\mathsf{a}_k,\mathsf{a}_l)(\overrightarrow{f_k}(\overrightarrow{\mathit{getf1}(lm,n),e_k})) \\
&\qquad \cdot \mathsf{Z}(\mathit{replremf2}(lm,n,m,\mathbf{mklm}_k[p_k](\overrightarrow{\mathit{getf1}(lm,n),e_k}))) \\
&\qquad\quad \lhd\, n \neq m \wedge n < \mathit{lenf}(lm) \wedge m < \mathit{lenf}(lm) \wedge \overrightarrow{f_k}(\overrightarrow{\mathit{getf1}(lm,n),e_k}) = \overrightarrow{f_l}(\overrightarrow{\mathit{getf1}(lm,m),e'_l}) \\
&\qquad\qquad \wedge c_k(\overrightarrow{\mathit{getf1}(lm,n),e_k}) \wedge c_l(\overrightarrow{\mathit{getf1}(lm,m),e'_l}) \rhd \delta
\end{aligned}
$$

$$+ \sum_{(k,l)\in J\gamma J} \sum_{n:Nat} \sum_{m:Nat} \sum_{\overrightarrow{e_k:E_k}} \sum_{\overrightarrow{e'_l:E_l}} \gamma(\mathsf{a}_k,\mathsf{a}_l)(\overrightarrow{f_k}(\overrightarrow{getf1(lm,n),e_k})) \cdot \mathsf{Z}(remf2(lm,n,m))$$

$$\lhd\, n < m \wedge m < lenf(lm) \wedge \overrightarrow{f_k}(\overrightarrow{getf1(lm,n),e_k}) = \overrightarrow{f_l}(\overrightarrow{getf1(lm,m),e'_l})$$

$$\wedge\, c_k(\overrightarrow{getf1(lm,n),e_k}) \wedge c_l(\overrightarrow{getf1(lm,m),e'_l}) \wedge remf2(lm,n,m) \neq \langle\,\rangle \rhd \delta$$

$$+ \sum_{(k,l)\in J\gamma J} \sum_{n:Nat} \sum_{m:Nat} \sum_{\overrightarrow{e_k:E_k}} \sum_{\overrightarrow{e'_l:E_l}} \gamma(\mathsf{a}_k,\mathsf{a}_l)(\overrightarrow{f_k}(\overrightarrow{getf1(lm,n),e_k}))$$

$$\lhd\, n < m \wedge m < lenf(lm) \wedge \overrightarrow{f_k}(\overrightarrow{getf1(lm,n),e_k}) = \overrightarrow{f_l}(\overrightarrow{getf1(lm,m),e'_l})$$

$$\wedge\, c_k(\overrightarrow{getf1(lm,n),e_k}) \wedge c_l(\overrightarrow{getf1(lm,m),e'_l}) \wedge remf2(lm,n,m) = \langle\,\rangle \rhd \delta$$

where $P\gamma Q = \{(k,l) \in P \times Q \mid \gamma(\mathsf{a}_k,\mathsf{a}_l) \text{ is defined}\}$.

The first three sets of summands of the equation represent the singular executions of the ready components (elements of the first layer), which are sometimes called interleavings. The process $\mathsf{Z}(lm)$ can execute any action the original process $\mathsf{X}(\overrightarrow{d})$ can execute, provided that $\overrightarrow{d}$ belongs to the first layer of lm. After that the state of Z becomes lm with the first layer occurrence of $\overrightarrow{d}$ replaced by the LM representation of the resulting parallel/sequential composition generated from the terms p_i taken from the equation for X. The second and third sets represent the case where the ready component terminates. In this case we remove the component from lm and, depending on whether this was the last element of lm, either terminate, or not.

The last four sets of summands represent the dual executions of the ready components by means of synchronous communication of them, sometimes called handshakings. Here we take two different ready components, say $\overrightarrow{d}$ and $\overrightarrow{d'}$ and execute the actions that $\mathsf{X}(\overrightarrow{d}) \mid \mathsf{X}(\overrightarrow{d'})$ could execute. These are the actions that communicate and have equal parameter vectors. Due to the commutativity of communication function and parallel composition, it is enough to consider only ordered pairs of elements of the first layer (that is why the condition $n < m$ is present if both components perform terminating actions of X, or both do not). In order to determine the next state of Z, we either replace both of the components by the future behavior of both $\mathsf{X}(\overrightarrow{d})$ and $\mathsf{X}(\overrightarrow{d'})$, respectively (fourth set of summands), or replace one and remove the other (fifth set), or remove both components (last two sets). The last two sets of summands only differ in the fact that the first one does not terminate, and the second one does. This behavior is determined on whether the two communicating components were the last two elements of lm, or not.

The following theorem states the correctness of our construction.

Theorem 5.2. $(\mathsf{X}(\overrightarrow{d}), G_7) \Rightarrow_c (\mathsf{Z}(seq1(\overrightarrow{d}, LM0)), L)$.

Proof. The statement can be proved similarly to Proposition 49 in [21]. Here we define g_{Z} in the following way:

$$\begin{aligned}
&g_{\mathsf{Z}}(lm) = \delta \lhd lm = \langle\,\rangle \rhd \\
&\quad (\mathsf{X}(\overrightarrow{getf1(lm,0)}) \lhd remf1(lm,0) = \langle\,\rangle \rhd \\
&\quad\quad (\mathsf{X}(\overrightarrow{getf1(lm,0)}) \cdot g_{\mathsf{Z}}(remf1(lm,0)) \lhd lenf(lm) = 1 \rhd \\
&\quad\quad\quad (g_{\mathsf{Z}}(getflm(lm)) \parallel g_{\mathsf{Z}}(remflm(lm)) \lhd \neg is\_seq(lm) \rhd \\
&\quad\quad\quad\quad\quad (g_{\mathsf{Z}}(getflm(lm)) \parallel g_{\mathsf{Z}}(remflm(getseql(lm)))) \cdot g_{\mathsf{Z}}(getseqr(lm)))))
\end{aligned}$$

with the additional functions (see Appendix C.2 for precise definitions) having the following meaning:

- $is\_seq(lm)$ is a predicate that checks if lm is a sequential composition of two non-empty LMs;

- *getflm*(lm) returns the first element of the first multiset of the list lm (undefined in case there is no first multiset in lm);
- *remflm*(lm) removes the above mentioned element;
- *getseql*(lm) and *getseqr*(lm) split lm into two sequential parts, with the former one returning the first multiset and the latter one returning the rest (undefined in case there is no first multiset in lm).

From this definition, assuming $lm \neq \langle\,\rangle$, it can be shown that for any $n > 0$:

$$g_Z(seq1(\overrightarrow{d}, LM0)) = \mathsf{X}(\overrightarrow{d})$$
$$g_Z(seq1(\overrightarrow{d}, lm)) = \mathsf{X}(\overrightarrow{d}) \cdot g_Z(lm)$$
$$g_Z(seqM(par(lm_1, \dots par(lm_n, ML(lm_{n+1})) \dots), LM0)) = g_Z(lm_1) \parallel \cdots \parallel g_Z(lm_{n+1})$$
$$g_Z(seqM(par(lm_1, \dots par(lm_n, ML(lm_{n+1})) \dots), lm)) = (g_Z(lm_1) \parallel \cdots \parallel g_Z(lm_{n+1})) \cdot g_Z(lm)$$

Furthermore, we can show that for all the terms p_i from the equation (5.1)

$$g_Z(\mathbf{mklm}_i[p_i](\overrightarrow{t})) = p_i[\overrightarrow{d, e_i} := \overrightarrow{t}]$$

Using all these facts, correctness of necessary proof obligations can be derived from the axioms of μCRL and the data types defined in Appendix C. □

5.2 Renaming Operators

In this subsection we still assume that only handshaking communication is possible, but allow the renaming operations to be present. Taking into account that $x = \rho_{R_{ActLab}}(\tau_\emptyset(\partial_\emptyset(x)))$, where R_{ActLab} is the identity mapping, we assume that G_7 contains a single μCRL process equation in post-PEGNF of the following form:

$$\begin{aligned}\mathsf{X}(\overrightarrow{d{:}D}) = &\sum_{i\in I}\sum_{\overrightarrow{e_i:E_i}} \mathsf{a}_i(\overrightarrow{f_i}(\overrightarrow{d,e_i})) \cdot p_i(\overrightarrow{d,e_i}) \lhd c_i(\overrightarrow{d,e_i}) \rhd \delta \\ &+ \sum_{j\in J}\sum_{\overrightarrow{e_j:E_j}} \mathsf{a}_j(\overrightarrow{f_j}(\overrightarrow{d,e_j})) \lhd c_j(\overrightarrow{d,e_j}) \rhd \delta\end{aligned} \tag{5.3}$$

where $p_i(\overrightarrow{d,e_i})$ are terms of the following syntax:

$$p ::= p \cdot p \mid \rho_R(\tau_I(\partial_H(\mathsf{X}(\overrightarrow{t})))) \mid \rho_R(\tau_I(\partial_H(p \parallel p))) \tag{5.4}$$

We reuse the *State* data type defined in the previous subsection and extend the *LM* and *ML* data types to contain information about renaming operations surrounding a recursive call or a parallel composition, which we call *annotation* (cf. Appendix C.3).

To capture the annotations in the form of a data type, we first need to turn actions into a data type. Let the set of action labels *ActLab* be equal to $\{\mathsf{a}_0, \dots, \mathsf{a}_n\}$. We define the data types *Act*, *ActSet*, *ActMap* and *Annote* (see Appendix C.3), to represent actions, sets of actions, mappings of actions, and triples (R, I, H), respectively. For each action label $\mathsf{a} \in ActLab$ we define $\mathbf{mka}[\mathsf{a}] :\rightarrow Act$ to be equal to $a(i)$, where i is such that $\mathsf{a} = \mathsf{a}_i$. For each $S \subseteq ActLab$ we define $\mathbf{mkas}[S] :\rightarrow ActSet$ such that $\mathbf{mkas}[\{\mathsf{a}_0, \dots, \mathsf{a}_m\}] = add(\mathbf{mka}[\mathsf{a}_0], \dots add(\mathbf{mka}[\mathsf{a}_m], ActSet0) \dots)$. For every well-defined action renaming function R (cf. Definition 2.3) we define $\mathbf{mkam}[R] :\rightarrow ActMap$ to have the property that for any action $\mathsf{a} \in Act$ $appl(\mathbf{mka}[\mathsf{a}], \mathbf{mkam}[R]) = \mathbf{mka}[R(\mathsf{a})]$, where $appl : Act \times ActMap \rightarrow Act$ gives the result of application of a mapping to an action label.

The data types *ALM* (annotated *LM*) and *AML* (annotated *ML*) have the same constructors as *LM* and *ML*, respectively, with the following two type differences that concern the annotations:

- *seq1* : $\mathit{Annote} \times \mathit{State} \times \mathit{ALM} \to \mathit{ALM}$, with $\mathit{seq1}(\mathit{ann}, d, \mathit{lm})$ representing the list with the state vector d, annotated with ann, added to the head of lm,
- *par* : $\mathit{Annote} \times \mathit{ALM} \times \mathit{AML} \to \mathit{AML}$, with $\mathit{par}(\mathit{ann}, \mathit{lm}, \mathit{ml})$ representing the multiset with the list lm added to ml and this parallel composition annotated with ann.

Normal forms of the ALM and AML terms are defined as follows. A term of sort ALM is in normal form if it is of the form:

- $\mathit{ALM0}$,
- $\mathit{seq1}(\mathit{ann}, d, \mathit{lm})$,
- $\mathit{seqM}(\mathit{ml}, \mathit{lm})$,

where

- d is a term of sort State and ann is a term of sort Annote,
- lm is a term of sort ALM in normal form,
- ml is a term of sort AML in normal form having par as outermost symbol.

A term of sort AML is in normal form if it is of the form:

- $\mathit{AML}(\mathit{lm})$,
- $\mathit{par}(\mathit{ann}_1, \mathit{lm}_1, \ldots \mathit{par}(\mathit{ann}_n, \mathit{lm}_n, \mathit{AML}(\mathit{lm}_{n+1}))\ldots)$,

where for all $i \in \{1, \ldots, n+1\}$:

- $\mathit{lm}, \mathit{lm}_i$ are terms of sort ALM in normal form, not of the form $\mathit{seqM}(\mathit{par}(\mathit{Ann0}, \mathit{lm}', \mathit{ml}), \mathit{ALM0})$,
- $\mathit{lm}_i \neq \mathit{ALM0}$,
- lm_n is not of the form $\mathit{seqM}(\mathit{ml}, \mathit{ALM0})$,
- $\neg \mathit{gt}(\mathit{lm}_i, \mathit{lm}_{i+1})$.

The *gt* function (greater than) is defined on ALM and AML using the functions *gt* on the sorts *State* and *Annote*.

As in the case without annotations, normal forms are preserved by the auxiliary functions *conc*, *conp*, *mkml* and *comp*. In addition to that we have the function *annote* to emulate the application of the renaming operations to an ALM. The preservation of normal forms can be shown for all functions that generate terms of sort ALM or AML. Also, the properties of combinations of *mkml* and *conp*, as well as the properties of *seqc* and *parc* compositions are also valid in the setting with annotations. It is also easy to check that *annote* distributes over *seqc*.

For each term p_i from the equation for X we construct the term $\mathbf{mklm}_i[p_i] : \mathit{State} \times \overrightarrow{E_i} \to \mathit{ALM}$ in the following way:

$$\begin{aligned}
&\mathbf{mklm}_i[\rho_R(\tau_I(\partial_H(\mathsf{X}(\overrightarrow{t}))))](\overrightarrow{t_d, t_{e_i}}) = \\
&\qquad \mathit{seq1}(\mathit{ann}(\mathbf{mkam}[R], \mathbf{mkas}[I], \mathbf{mkas}[H]), \overrightarrow{t}[\overrightarrow{d, e_i} := \overrightarrow{t_d, t_{e_i}}], \mathit{LM0}) \\
&\mathbf{mklm}_i[p^1 \cdot p^2](\overrightarrow{t_d, e_i}) = \mathit{seqc}(\mathbf{mklm}_i[p^1](\overrightarrow{t_d, t_{e_i}}), \mathbf{mklm}_i[p^2](\overrightarrow{t_d, t_{e_i}})) \\
&\mathbf{mklm}_i[\rho_R(\tau_I(\partial_H(p^1 \parallel p^2)))](\overrightarrow{t_d, t_{e_i}}) = \\
&\qquad \mathit{parc}(\mathit{ann}(\mathbf{mkam}[R], \mathbf{mkas}[I], \mathbf{mkas}[H]), \mathbf{mklm}_i[p^1](\overrightarrow{t_d, t_{e_i}}), \mathbf{mklm}_i[p^2](\overrightarrow{t_d, t_{e_i}}))
\end{aligned}$$

As an example, if $p_i = \rho_R(\partial_H\,(\mathsf{X}(n) \parallel \mathsf{X}(s(n)))) \cdot \tau_I(\partial_{H_1}(\mathsf{X}(s(s(n)))))$, then

$$\begin{aligned}\mathbf{mklm}_i[p_i](n) = \mathit{seqM}(\mathit{par}(\mathit{ann}(\mathbf{mkam}[R], \mathit{ActSet0}, \mathbf{mkas}[H]), \mathit{seq1}(\mathit{Ann0}, n, \mathit{LM0}),\\ \mathit{ML}(\mathit{seq1}(\mathit{Ann0}, s(n), \mathit{LM0}))), \mathit{seq1}(\mathit{ann}(\mathit{ActMap0}, \mathbf{mkas}[I], \mathbf{mkas}[H_1]), s(s(n)), \mathit{LM0}))\end{aligned}$$

For the precise definition of the *ALM* and *AML* data types we refer to Appendix C.3.

The notion of the first layer is preserved for the case with annotations, but in addition to the state vector, each element of the first layer has its individual annotation, which is a composition of all annotations in the scope of which it appears. In case we are interested in a pair of state vectors from the first layer, we have to consider three annotations. For example (considering just the encapsulations), $\partial_H(\partial_{H_1}(\mathsf{X}(1)) \parallel \partial_{H_2}(\mathsf{X}(2)))$ leads to the pair of the first layer elements (1 and 2), and three annotations (H, H_1, and H_2). The following additional functions involving the notions of the first layer and annotations are used in the definition of the resulting LPE:

$\mathit{getf1d} : \mathit{ALM} \times \mathit{Nat} \to \overrightarrow{D}$	– get n-th element
$\mathit{getf1a} : \mathit{ALM} \times \mathit{Nat} \to \mathit{Annote}$	– get n-th element's annotation
$\mathit{getf2a0} : \mathit{ALM} \times \mathit{Nat} \times \mathit{Nat} \to \mathit{Annote}$	– get n-th element's annotation up to the junction with the m-th element
$\mathit{getf2a1} : \mathit{ALM} \times \mathit{Nat} \times \mathit{Nat} \to \mathit{Annote}$	– get m-th element's annotation up to the junction with the n-th element
$\mathit{getf2a} : \mathit{ALM} \times \mathit{Nat} \times \mathit{Nat} \to \mathit{Annote}$	– get (n,m)-th elements' annotation (from the junction upwards)

And as in the case without annotations, removing an element is equivalent to replacing it with *ALM0*.

Assume the system G_7 consists of process equation X as defined in (5.3). A system L consisting of process equation Z, which mimics behavior of X, is defined in Appendix A. The following theorem states the correctness of our construction.

Theorem 5.3. $(\mathsf{X}(\overrightarrow{d}), G_7) \Rightarrow_c (\mathsf{Z}(\mathit{seq1}(\mathit{Ann0}, \overrightarrow{d}, \mathit{LM0})), L)$.

5.3 Multi-Party Communication

In this subsection we define the LPE for the case when an arbitrary number of parallel components can be executed synchronously. The number is unknown a priori and is only bound by the number of the elements of the first layer in a particular state. On the other hand, the number of different action labels is finite, and, as will be shown later, so is the number of possible communication configurations.

We start from the simpler sub-case where no renaming operations are present. First of all we introduce some abbreviations to make dealing with the commutative associative partial communication function γ a bit more liberal. We assume that ϵ is such that for any $\mathsf{a} \in \mathit{ActLab}$ $\gamma(\mathsf{a}, \epsilon) = \mathsf{a}$, and recall that $\gamma(\tau, \mathsf{a})$ is undefined. Moreover, taking associativity of γ into account, we define $\gamma(\mathsf{a}_1, \ldots, \mathsf{a}_n) = \gamma(\mathsf{a}_1, \ldots \gamma(\mathsf{a}_{n-1}, \mathsf{a}_n) \ldots)$. For any action label $\mathsf{a} \in \mathit{ActLab}$, we define $\mathsf{a}^0 = \epsilon$, $\mathsf{a}^1 = \mathsf{a}$, and $\mathsf{a}^{n+1} = \gamma(\mathsf{a}, \mathsf{a}^n)$. Similarly, $\tau^0 = \epsilon$, $\tau^1 = \tau$, and τ^n is undefined for all $n > 1$. From the finiteness of *ActLab* it can easily be seen that for any action $\mathsf{a} \in \mathit{ActLab}$ there are minimal natural numbers $p(\mathsf{a})$ (prefix of a) and $c(\mathsf{a})$ (cycle of a) such that the sequence a^n repeats itself after $p(\mathsf{a})$ steps with the period $c(\mathsf{a})$. More precisely, taking into account that a^n may become undefined for some n and all greater powers, we define the numbers $p(\mathsf{a})$ and $c(\mathsf{a})$ as follows:

$$p(\mathsf{a}) = \mathbf{min}\{n \in \mathbb{N} \mid \mathsf{a}^n \text{ is undefined } \;\vee\; \exists m{>}n\ \mathsf{a}^n = \mathsf{a}^m\}$$

$$c(\mathsf{a}) = \begin{cases} 0 & \text{if } \mathsf{a}^{p(\mathsf{a})} \text{ is undefined} \\ \mathbf{min}\{n \in \mathbb{N} \mid n{>}0 \;\wedge\; \mathsf{a}^{p(\mathsf{a})} = \mathsf{a}^{p(\mathsf{a})+n}\} & \text{otherwise} \end{cases}$$

which means that if a^n is undefined for some n, then $p(\mathsf{a})$ is minimal with respect to such n, and in this case we put $c(\mathsf{a}) = 0$. In accordance to this, we define $p(\tau) = 2$ and $c(\tau) = 0$.

Considering the equation for X as defined in (5.1), we take the sets of indices I and J and for all $i \in I \cup J$ we define $p(i) = p(\mathsf{a}_i)$ and $c(i) = c(\mathsf{a}_i)$. For this equation for X we define a notion of *configuration* as a function $conf : I \cup J \to \mathbb{N}$. A particular configuration specifies how many occurrences of an action label take part in a communication. We consider only the configurations that for each action label a_i have no more than $p(i) + c(i) - 1$ occurrences. Moreover we only consider the configurations that are defined. Assuming that $\gamma(conf) = \gamma(\mathsf{a}_0^{conf(0)}, \dots, \mathsf{a}_m^{conf(m)})$, where $I \cup J = \{0, \dots, m\}$, we define the set of configurations in the following way:

$$Conf = \{conf \mid \forall i \in I \cup J \ \ \big(0 \leq conf(i) < p(i) + c(i)\big) \ \wedge \sum_{i \in I \cup J} conf(i) > 0 \ \wedge \ \gamma(conf) \text{ is defined}\}$$

The set of configurations that do not lead to termination is defined as

$$Conf1 = \{conf \in Conf \mid \sum_{i \in I} conf(i) > 0\}$$

and the set of all others is named $Conf2 = Conf \setminus Conf1$. Now, for a given n we can check whether a_i^n conforms to a configuration as follows ($n \mid m$ represents the "n divides m" predicate):

$$\textbf{is\_conf}[conf, i](n) = \ (n = conf(i)) \vee \big(conf(i) > 0 \wedge c(i) > 0 \wedge n > p(i) \wedge c(i) \mid (n - conf(i))\big)$$

which says that n should either be the exact number specified in the configuration, or be greater than it by a multiple of $c(\mathsf{a}_i)$.

As one can expect, we need several list data types to deal with multi-party communications. In addition to the sorts *State* and *Nat* defined in Appendix C.1 we use the sorts *LState* and *LNat* to represent lists of natural numbers and states, respectively (see Appendix C.4). We also use the sort *ActPars* to represent different action parameter tuples that occur in the initial specification. Different actions may be parameterized by the same parameter sorts. In this case the values of the actual parameters have equal representations in the sort *ActPars*. The sorts E_i are used to represent the tuples of sorts that occur in the sum sequences of the equation (5.1) for X. These data types are tuple data types similar to *State*, with the exception that *ActPars* preserves a type information for tuples. The sorts *LActPars* and LE_i represent lists of *ActPars* and E_i, respectively. All the list data types have the functions *len*, *cat* and *head*, representing the length of the list, concatenation of two list, and the first element of the list (undefined for the empty list), respectively. The following additional functions involving these data types are used in the definitions below:

$$is\_unique : LNat \to Bool$$
$$is\_sorted : LNat \to Bool$$
$$is\_each\_lower : LNat \times Nat \to Bool$$
$$EQ : LActPars \to Bool$$
$$F_i : LState \times LE_i \to LActPars$$
$$C_i : LState \times LE_i \to Bool$$

The function *is_unique* checks if all list elements are unique, the function *is_sorted* checks if the list is sorted, and the function *is_each_lower* checks if each of the list elements is less than some natural number. The functions F_i model application of the terms $\overrightarrow{f_i}$ to each pair of elements in the argument lists, the functions C_i model conjunction of c_i applied to each pair of the elements, and the function EQ checks if all of the list elements are equal.

In addition to the data types LM and ML we use the sort LLM to represent lists of LMs (see Appendix C.5). The following additional functions involving this data type are used in the definitions below:

$getfn : LM \times LNat \to LState$	– get n first layer elements
$replfn : LM \times LNat \times LLM \to LM$	– replace n first layer elements with elements of LLM
$remfn : LM \times LNat \to LM$	– remove n first layer elements
$mkllm_i : LState \times LE_i \to LLM$	

The function $mkllm_i$ applies the term $\mathbf{mklm}_i[p_i]$ to each pair of elements in the argument lists.

We use the following meta-symbols in the resulting LPE definition:

$$\mathbf{cat}[l_0, \dots, l_m] = cat(l_0, \dots cat(l_{m-1}, l_m) \dots)$$
$$\mathbf{mkllm}[p_i](ld, \overrightarrow{le_i}) = mkllm_i(ld, \overrightarrow{le_i}) \text{ for } i \in I$$
$$\mathbf{mkllm}[p_j](ld, \overrightarrow{le_j}) = add(LM0, LLM0) \text{ for } j \in J$$

Assume the system G_7 consists of process equation X as defined in (5.1) with the sets of indices $J = \{0, \dots, k\}$ and $I = \{k+1, \dots, m\}$. We can now define a system L consisting of process equation Z, which mimics behavior of X, in the following way:

$$\begin{aligned}
&\mathsf{Z}(lm{:}LM) = \\
&\quad \sum_{conf \in Conf1} \sum_{ln_0:LNat} \cdots \sum_{ln_m:LNat} \sum_{\overrightarrow{le_0:LE_0}} \cdots \sum_{\overrightarrow{le_m:LE_m}} \gamma(conf)(\overrightarrow{f_{mc}}(\overrightarrow{getf1(lm, head(ln))}, head(\overrightarrow{le_{mc}}))) \\
&\qquad \cdot \mathsf{Z}(replfn(lm, ln, \\
&\qquad\quad \mathbf{cat}[\mathbf{mkllm}[p_0](\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}), \dots, \mathbf{mkllm}[p_m](\overrightarrow{getfn(lm, ln_m)}, \overrightarrow{le_m})])) \\
&\qquad\quad \lhd\ ln \neq LNat0 \wedge len(ln) \leq lenf(lm) \wedge is\_unique(ln) \\
&\qquad\qquad \wedge \bigwedge\nolimits_{0 \leq i \leq m} is\_sorted(ln_i) \wedge \bigwedge\nolimits_{0 \leq i \leq m} is\_each\_lower(lenf(lm), ln_i) \\
&\qquad\qquad \wedge \bigwedge\nolimits_{0 \leq i \leq m} \mathbf{is\_conf}[conf, i](len(ln_i)) \wedge \bigwedge\nolimits_{0 \leq i \leq m} len(ln_i) = len(\overrightarrow{le_i}) \\
&\qquad\qquad \wedge EQ(\mathbf{cat}[F_0(\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}), \dots, F_m(\overrightarrow{getfn(lm, ln_m)}, \overrightarrow{le_m})]) \\
&\qquad\qquad \wedge C_0(\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}) \wedge \cdots \wedge C_m(\overrightarrow{getfn(lm, ln_m)}, \overrightarrow{le_m}) \rhd \delta \\
&+ \sum_{conf \in Conf2} \sum_{ln_0:LNat} \cdots \sum_{ln_k:LNat} \sum_{\overrightarrow{le_0:LE_0}} \cdots \sum_{\overrightarrow{le_k:LE_k}} \gamma(conf)(\overrightarrow{f_{mc}}(\overrightarrow{getf1(lm, head(lnJ))}, head(\overrightarrow{le_{mc}}))) \\
&\qquad \cdot \mathsf{Z}(remfn(lm, lnJ)) \\
&\qquad\quad \lhd\ lnJ \neq LNat0 \wedge len(lnJ) \leq lenf(lm) \wedge is\_unique(lnJ) \\
&\qquad\qquad \wedge \bigwedge\nolimits_{0 \leq j \leq k} is\_sorted(ln_j) \wedge \bigwedge\nolimits_{0 \leq j \leq k} is\_each\_lower(lenf(lm), ln_j) \\
&\qquad\qquad \wedge \bigwedge\nolimits_{0 \leq j \leq k} \mathbf{is\_conf}[conf, j](len(ln_j)) \wedge \bigwedge\nolimits_{0 \leq j \leq k} len(ln_j) = len(\overrightarrow{le_j}) \\
&\qquad\qquad \wedge EQ(\mathbf{cat}[F_0(\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}), \dots, F_k(\overrightarrow{getfn(lm, ln_k)}, \overrightarrow{le_k})]) \\
&\qquad\qquad \wedge C_0(\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}) \wedge \cdots \wedge C_k(\overrightarrow{getfn(lm, ln_k)}, \overrightarrow{le_k}) \\
&\qquad\qquad \wedge remfn(lm, lnJ) \neq \langle\,\rangle \rhd \delta
\end{aligned}$$

$$
\begin{aligned}
+ \sum_{conf\in Conf2} \sum_{ln_0:LNat} \cdots \sum_{ln_k:LNat} \sum_{\overrightarrow{le_0}:\overrightarrow{LE_0}} \cdots \sum_{\overrightarrow{le_k}:\overrightarrow{LE_k}} \gamma(conf)(\overrightarrow{f_{mc}}(\overrightarrow{getf1(lm, head(lnJ))}), head(\overrightarrow{le_{mc}})))\\
\triangleleft\, lnJ \neq LNat0 \wedge len(lnJ) \leq lenf(lm) \wedge is\_unique(lnJ)\\
\wedge \bigwedge\nolimits_{0\leq j\leq k} is\_sorted(ln_j) \wedge \bigwedge\nolimits_{0\leq j\leq k} is\_each\_lower(lenf(lm), ln_j)\\
\wedge \bigwedge\nolimits_{0\leq j\leq k} \mathbf{is\_conf}[conf, j](len(ln_j)) \wedge \bigwedge\nolimits_{0\leq j\leq k} len(ln_j) = len(\overrightarrow{le_j})\\
\wedge EQ(\mathbf{cat}[F_0(\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}), \ldots, F_k(\overrightarrow{getfn(lm, ln_k)}, \overrightarrow{le_k})])\\
\wedge C_0(\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}) \wedge \cdots \wedge C_k(\overrightarrow{getfn(lm, ln_k)}, \overrightarrow{le_k})\\
\wedge remfn(lm, lnJ) = \langle\,\rangle \triangleright \delta
\end{aligned}
$$

where $ln = \mathbf{cat}[ln_0, \ldots, ln_m]$, $lnJ = \mathbf{cat}[ln_0, \ldots, ln_k]$, and $mc = min\{n \in Nat \mid conf(n) > 0\}$.

The first set of summands of the LPE represents the case when the process cannot terminate, because at least one of the communicating components is not terminating (for some $i \in I$ we have $conf(i) > 0$). The sum variables $ln_0, \ldots, ln_m$ represent lists of numbers of ready components that will communicate by performing actions $\mathsf{a}_0, \ldots, \mathsf{a}_m$ from the process equation for X, respectively. The condition of the summand makes sure that the total number of communicating components is not zero and not bigger than the total number of first layer elements. Moreover, the same component should not occur more than once, the order of the components is not important, and the numbers, the components are indexed by, are in range (smaller than $lenf(lm)$). Finally it is checked that the number of components performing each particular action conforms to the chosen configuration. The variables $\overrightarrow{le_0}, \ldots, \overrightarrow{le_m}$ represent lists of the sum parameter vectors $\overrightarrow{e_i}$ from the process equation for X. The length of each list should be equal to the number of components performing the corresponding action. We note that not all of the sums for $ln_0, \ldots, ln_m$ and $\overrightarrow{le_0}, \ldots, \overrightarrow{le_m}$ are needed for each configuration. For instance if in a particular configuration we have $conf(i) = 0$, then the sums for ln_i and $\overrightarrow{le_i}$ can be dropped. This is because the only valid representations for ln_i and $\overrightarrow{le_i}$ will be the empty lists, and all other conjuncts of the condition involving them will be equal to true.

Furthermore, the other conditions necessary to make the communication possible are: the initial conditions c_i are satisfied for all of the components, and the parameters of communicating actions are equal. We use the function $\overrightarrow{f_{mc}}$ applied to the first communicating component to get the values of the action parameters. To figure out what the next state of the process Z is, we replace the elements of the first layer of lm that took part in the communication with the next states these components would have in the process X ($LM0$ in case a particular component terminates).

The other two sets of summands represent the configurations that only involve the terminating actions of the equation for X. The difference between the two is in whether after this communication the lm becomes equal to $LM0$. If this is the case, then the LPE Z terminates, and otherwise continues the execution.

The following theorem states the correctness of our construction.

Theorem 5.4. $(\mathsf{X}(\overrightarrow{d}), G_7) \Rightarrow_c (\mathsf{Z}(seq1(\overrightarrow{d}, LM0)), L)$.

5.4 Multi-Party Communication with Renaming

For the case with the renaming operations we cannot use the communication configurations because we do not know to what action labels the initial action labels performed by the components will be renamed. That is why we have to expect that the resulting action can be any action to which one of the actions a_i can be renamed by a renaming function.

In addition to the data types ALM and AML we use the sort $LALM$ to represent lists of ALMs, the sort $LAct$ to represent lists of Acts and the sort $ActDT$ to represent either an action label, or τ, or δ (see Appendix C.6). The following additional functions involving these data types are used in the definitions of the resulting LPE:

$$
\begin{aligned}
&\mathit{is\_act} : \mathit{Act} \times \mathit{LALM} \times \mathit{LNat} \times \mathit{LAct} \to \mathit{Bool}\\
&\mathit{is\_tau} : \mathit{LALM} \times \mathit{LNat} \times \mathit{LAct} \to \mathit{Bool}\\
&\mathit{mklact} : \mathit{Nat} \times \mathit{Act} \to \mathit{LAct}\\
&\overrightarrow{f0} : \mathit{ALM} \times \mathit{LNAT} \times \ldots \times \mathit{LNAT} \times E_0 \times \ldots \times E_n \to \mathit{ActPars}\\
&\mathit{mkllm}_i : \mathit{LState} \times \mathit{LE}_i \to \mathit{LALM}
\end{aligned}
$$

The function $\mathit{is\_act}$ checks if a list of components can communicate by performing action from the list, and the result of this communication is the given action. The function $\mathit{is\_tau}$ does the same, but checks that the result is τ. The function mklact generates the list of n actions a. The function $\overrightarrow{f0}$ can be defined as:

$$\overrightarrow{f0}(\mathit{lm}, \mathit{ln}_0, \ldots, \mathit{ln}_n, \overrightarrow{e_0}, \ldots, \overrightarrow{e_n}) = \overrightarrow{f_l}(\overrightarrow{\mathit{getf1d}(\mathit{lm}, \mathit{head}(\mathit{ln}_l))}, \overrightarrow{e_l}) \text{ for } l = \mathbf{min}\{i \mid \mathit{len}(\mathit{ln}_i) > 0 \vee i = n\}$$

The meaning of this definition is that we find the number l of the first ready component taking part in the communication, and apply the corresponding function vector $\overrightarrow{f_l}$ to get the values of the action parameters. The function mkllm_i applies the term $\mathbf{mklm}_i[p_i]$ to each pair of elements in the argument lists.

Assume the system G_7 consists of process equation X as defined in (5.4). A system L consisting of process equation Z, which mimics behavior of X, is defined in Appendix B. The correctness statement is similar to the case with handshaking:

Theorem 5.5. $(\mathsf{X}(\overrightarrow{d}), G_7) \Rightarrow_c (\mathsf{Z}(\mathit{seq1}(\mathit{Ann0}, \overrightarrow{d}, \mathit{LM0})), L)$.

Summarizing Section 5 and the entire transformation, for any X^s from the initial μCRL specification we have

$$(\mathsf{X}^s(\overrightarrow{t}), G) \Rightarrow_c (\mathsf{Z}(\mathit{seq1}(\mathit{Ann0}, (s, M_{\mathsf{X}^s}(\overrightarrow{t})), \mathit{LM0})), L)$$

and the current specification contains definitions of the data types from Appendix C (for the data type dependencies we refer to Figure 1 in that Appendix).

6. Conclusions

We described a transformation of μCRL process definitions into a linear format, and argued that this transformation is correct. Our correctness argument is not tied to some particular model, and also applies to process definitions that do not necessarily imply that the models have unique solutions. Furthermore, this transformation is idempotent in the following sense: applying the transformation to an LPE yields the same LPE.

During the process of linearization many optimizations are conceivable, some of which can only be applied in a certain context. We have already mentioned some optimization rewrite rules (Table 13) that can be applied during one of the linearization steps. Another optimization can be performed in the cases where a new process name is introduced. There can be a choice of what parameters to use for the new process name in order to fetch the complicated structure of data terms involved (see Subsection II.6.3 of [37] for a detailed example). Furthermore, there are many (minor) optimizations, such as the rewriting of conditions or the elimination of constant parameters. Due to the fact that the LPE format provides such a simple process structure, we feel that this type of optimizations can be best performed after the transformation into the LPE format. Such optimizations include rewriting of data terms, eliminations of redundant variables and constants, abstract interpretation, and so on, some of which have been described in [17] and implemented in the μCRL Toolset [37].

One particular optimization that we want to mention is called *regular linearization.* By regular linearization we mean the linearization process that does not deploy infinite data types to encode process behavior. The regular linearization procedures can take the equations we have before introduction of the infinite LM data type (Section 5) and try to achieve the LPE form without this data type introduction. This is not always possible: for instance $\mathsf{X} = \mathsf{a} \cdot \mathsf{X} \cdot \mathsf{X} + \mathsf{a}$ cannot be linearized without introducing an infinite data type, even if we restrict to the bisimulation model. This follows from the fact that X represents an infinite graph in the bisimulation model (cf. [27]), but an LPE without infinite data types can only represent a finite graph in that model (cf. [15], page 40). One of the possibilities for regular linearization is based on [27], and applies to the situation where regularity follows from the absence of termination in a recursion, like in $\mathsf{X} = \mathsf{a} \cdot \mathsf{X} \cdot \mathsf{X}$. Restricting to standard process semantics for μCRL, an LPE that specifies the same behavior is $\mathsf{X} = \mathsf{a} \cdot \mathsf{X}$. However, this optimization is model dependent, as there can be models in which the two equations have different sets of solutions. For some other cases, also dealt with in [27] and used in the μCRL Toolset, these optimizations can be justified on a general level using the equivalence of systems of process equations. For example, the system $G_1 = \{\mathsf{X} = \mathsf{a} \cdot \mathsf{Y} \cdot \mathsf{X}, \mathsf{Y} = \mathsf{b}\}$ can be transformed into $G_2 = \{\mathsf{X} = \mathsf{a} \cdot \mathsf{Z}, \mathsf{Z} = \mathsf{b} \cdot \mathsf{X}\}$, and we can prove that $(X, G_1) = (X, G_2)$, thus showing that this transformation is sound in every model. More on regular linearization, as it is implemented in the μCRL Toolset, can be found in Subsection II.6 of [37].

Another particular optimization that we want to mention is called *clustering of actions.* We refer to Definition 2.7, Theorem 2.8 and Theorem A.4 in [23]. This transformation allows to optimize an LPE to a form in which every action label occurs at most twice (either as a termination action or not). The constructed LPE is equivalent (in every model) to the original one. During the transformation the sums $\sum_{i\in I}$ and $\sum_{j\in J}$, which in Definition 5.1 represent abbreviations for alternative compositions, are changed to 'real' sums over enumerated data types. A similar transformation could be applied before introducing the data type LM in Section 5, which would lead to smaller resulting LPEs. More on clustering of actions, as it is implemented in the μCRL Toolset, can be found in Subsection 3.1 (page 13) of [37].

In the future we plan to work on extending the linearization procedure to cover the timed version of μCRL [22]. A precise definition of the regular linearization procedure, as well as some regularity, reachability and guardedness analysis methods could lead to better linearization results. Additional extensions to the language like interrupts, process creation and priorities could be investigated, as they seem to be useful for applications. An implementation of the linearization procedure using rewriting strategies [36] is currently under development.

Acknowledgments. Thanks go to Wan Fokkink and Jan Friso Groote for careful proofreading of the manuscript, and to Jan Bergstra, Bert Lisser, Bas Luttik, Faron Moller, Vincent van Oostrom, Jaco van de Pol, Alban Ponse, Konstantin Savenkov, and Mark van der Zwaag for helpful discussions and comments.

References

[1] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Decidability of bisimulation equivalence for processes generating context-free languages. *JACM*, 40(3):653–682, 1993.

[2] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in TCS 18. Cambridge University Press, 1990.

[3] D.B. Benson and I. Guessarian. Algebraic solutions to recursion schemes. *JCSS*, 35:365–400, 1987.

[4] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60(1/3):109–137, 1984.

[5] J.A. Bergstra and J.W. Klop. A complete inference system for regular processes with silent moves. In F.R. Drake and J.K. Truss, editors, *Proceedings Logic Colloquium 1986*, pages 21–81, Hull, 1988. North-Holland. First appeared as: Report CS-R8420, CWI, Amsterdam, 1984.

[6] J.A. Bergstra and A. Ponse. Translation of a muCRL-fragment to I-CRL. In *Methods for the Transformation and analysis of CRL. Deliverable 46/SPE/WP5/DS/A/007/b1*, SPECS RACE Project no. 1046, pages 125–148. Available through GSI Tecsi, May 1991.

[7] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In B. Jonsson and J. Parrow, editors, *Proc. CONCUR'94*, LNCS 836, pages 401–416. Springer, 1994.

[8] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I.A. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: a toolset for analysing algebraic specifications. In *Proc. CAV'2001*, LNCS 2102, pages 250–254. Springer, July 2001.

[9] S.C.C. Blom and J.C. van de Pol. State space reduction by proving confluence. Report, CWI, Amsterdam, To appear. Available from `http://www.cwi.nl/~vdpol/papers/confotf.ps.Z`.

[10] J.J. Brunekreef. Process specification in a UNITY format. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes, Utrecht 1994*, Workshops in Computing, pages 319–337. Springer, 1995.

[11] S. Burris and H.P. Sankappanavar. *A Course in Universal Algebra.* Number 78 in Graduate Texts in Mathematics. Springer, 1981.

[12] K.M. Chandy and J. Misra. *Parallel Program Design. A Foundation.* Addison-Wesley, 1988.

[13] B. Courcelle. Equivalences and transformations of regular systems–applications to recursive program schemes and grammars. *TCS*, 42:1–122, 1986.

[14] B. Courcelle. Recursive applicative program schemes. In J. van Leeuwen, editor, *Handbook of TCS*, volume B, chapter 9, pages 459–492. Elsevier, 1990.

[15] W.J. Fokkink. *Introduction to Process Algebra.* Texts in TCS. An EATCS Series. Springer, 2000.

[16] J.F. Groote. The syntax and semantics of timed μCRL. Report SEN-R9709, CWI, Amsterdam, 1997.

[17] J.F. Groote and B. Lisser. Computer assisted manipulation of algebraic process specifications. Report SEN-R0117, CWI, Amsterdam, May 2001.

[18] J.F. Groote and S.P. Luttik. Undecidability and completeness results for process algebras with alternative quantification over data. Report SEN-R9806, CWI, Amsterdam, July 1998. Available from `http://www.cwi.nl/~luttik/`.

[19] J.F. Groote and A. Ponse. Proof theory for μCRL: A language for processes with data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Semantics of Specification Languages*, Workshop in Computing, pages 232–251. Springer, 1994.

[20] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes 1994*, Workshop in Computing, pages 26–62. Springer, 1995.

[21] J.F. Groote, A. Ponse, and Y.S. Usenko. Linearization in parallel pCRL. *JLAP*, 48(1-2):39–72, June 2001.

[22] J.F. Groote, M.A. Reniers, J.J. van Wamel, and M.B. van der Zwaag. Completeness of timed μCRL. Report SEN-R0034, CWI, Amsterdam, November 2000.

[23] J.F. Groote and J. Springintveld. Focus points and convergent process operators. A proof strategy for protocol verification. Report 142, Department of Philosophy, Utrecht University, 1995. Available from `ftp://ftp.phil.uu.nl/pub/logic/PREPRINTS/preprint142.ps.Z`.

[24] J.F. Groote and J. van Wamel. The parallel composition of uniform processes with data. *TCS*, 266(1-2):631–652, 2001.

[25] Y. Hirshfeld and F. Moller. Decidability results in automata and process theory. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency: Structure versus Automata*, LNCS 1043, pages 102–148, Berlin, 1996. Springer.

[26] ISO/IEC. *LOTOS — a formal description technique based on the temporal ordering of observational behaviour.* International Standard 8807. International Organization for Standardization, — Information Processing Systems — Open Systems Interconection, Genève, September 1988.

[27] S. Mauw and J.C. Mulder. Regularity of BPA-systems is decidable. In B. Jonsson and J. Parrow, editors, *Proc. CONCUR '94*, LNCS 836, pages 34–47. Springer, 1994.

[28] S. Mauw and G.J. Veltink. A process specification formalism. *Fund. Inf.*, XIII:85–139, 1990.

[29] S. Mauw and G.J. Veltink, editors. *Algebraic Specification of Communication Protocols.* Cambridge Tracts in TCS 36. Cambridge University Press, 1993.

[30] R. Milner. A complete inference system for a class of regular behaviours. *JCSS*, 28:439–466, 1984.

[31] J.C. van de Pol. A prover for the mucrl toolset with applications – version 0.1. Report SEN-R0106, CWI, Amsterdam, April 2001.

[32] A. Ponse. Computable processes and bisimulation equivalence. *Formal Aspects of Computing*, 8(6):648–678, 1996.

[33] A. Ponse and Y.S. Usenko. Equivalence of recursive specifications in process algebra. *IPL*, 80(1):59–65, October 2001.

[34] Albert Rubio. A fully syntactic AC-RPO. *Information and Computation*, 2002. To appear.

[35] SPECS-Semantics and Analysis. *Definition of MR and CRL Version 2.1. Deliverable 46/SPE/WP5/DS/A/017/b1.* SPECS RACE Project no. 1046. Available through GSI Tecsi, 1990.

[36] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Proc. RTA'01*, LNCS 2051, pages 357–361. Springer, May 2001.

[37] A.G. Wouters. Manual for the μCRL tool set (version 2.8.2). Report SEN-R0130, CWI, Amsterdam, December 2001.

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

A. Resulting LPE for the Case with the Renaming Operations and Handshaking

$$
\begin{aligned}
&\mathsf{Z}(lm{:}ALM) = \\
&\quad\sum_{i\in I\setminus I_\tau}\sum_{\mathsf{a}\in\mathbf{R}(i)}\sum_{n:Nat}\sum_{\overrightarrow{e_i:E_i}} \mathsf{a}(\overrightarrow{f_i}(\overrightarrow{getf1d(lm,n),e_i}))\cdot\mathsf{Z}(replf1(lm,n,\mathbf{mklm}_i[p_i](\overrightarrow{getf1d(lm,n),e_i})))\\
&\qquad\qquad \lhd\, n < lenf(lm)\wedge c_i(\overrightarrow{getf1d(lm,n),e_i})\\
&\qquad\qquad\quad \wedge\,\mathbf{mka}[\mathsf{a}_i]\notin getH(getf1a(lm,n))\cup getI(getf1a(lm,n))\\
&\qquad\qquad\quad \wedge\,\mathbf{mka}[\mathsf{a}] = appl(\mathbf{mka}[\mathsf{a}_i],getR(getf1a(lm,n)))\rhd\delta\\
&\quad+\sum_{i\in I\setminus I_\tau}\sum_{n:Nat}\sum_{\overrightarrow{e_i:E_i}}\tau\cdot\mathsf{Z}(replf1(lm,n,\mathbf{mklm}_i[p_i](\overrightarrow{getf1d(lm,n),e_i})))\\
&\qquad\qquad \lhd\, n < lenf(lm)\wedge c_i(\overrightarrow{getf1d(lm,n),e_i})\\
&\qquad\qquad\quad \wedge\,\mathbf{mka}[\mathsf{a}_i]\in getI(getf1a(lm,n))\setminus getH(getf1a(lm,n))\rhd\delta\\
&\quad+\sum_{i\in I_\tau}\sum_{n:Nat}\sum_{\overrightarrow{e_i:E_i}}\tau\cdot\mathsf{Z}(replf1(lm,n,\mathbf{mklm}_i[p_i](\overrightarrow{getf1d(lm,n),e_i})))\\
&\qquad\qquad \lhd\, n < lenf(lm)\wedge c_i(\overrightarrow{getf1d(lm,n),e_i})\rhd\delta\\
&\quad+\sum_{j\in J\setminus J_\tau}\sum_{\mathsf{a}\in\mathbf{R}(j)}\sum_{n:Nat}\sum_{\overrightarrow{e_j:E_j}} \mathsf{a}(\overrightarrow{f_j}(\overrightarrow{getf1d(lm,n),e_j}))\cdot\mathsf{Z}(remf1(lm,n))\\
&\qquad\qquad \lhd\, n < lenf(lm)\wedge remf1(lm,n)\neq\langle\,\rangle\wedge c_j(\overrightarrow{getf1d(lm,n),e_j})\\
&\qquad\qquad\quad \wedge\,\mathbf{mka}[\mathsf{a}_j]\notin getH(getf1a(lm,n))\cup getI(getf1a(lm,n))\\
&\qquad\qquad\quad \wedge\,\mathbf{mka}[\mathsf{a}] = appl(\mathbf{mka}[\mathsf{a}_j],getR(getf1a(lm,n)))\rhd\delta\\
&\quad+\sum_{j\in J\setminus J_\tau}\sum_{n:Nat}\sum_{\overrightarrow{e_j:E_j}}\tau\cdot\mathsf{Z}(remf1(lm,n))\\
&\qquad\qquad \lhd\, n < lenf(lm)\wedge remf1(lm,n)\neq\langle\,\rangle\wedge c_j(\overrightarrow{getf1d(lm,n),e_j})\\
&\qquad\qquad\quad \wedge\,\mathbf{mka}[\mathsf{a}_j]\in getI(getf1a(lm,n))\setminus getH(getf1a(lm,n))\rhd\delta\\
&\quad+\sum_{j\in J_\tau}\sum_{n:Nat}\sum_{\overrightarrow{e_j:E_j}}\tau\cdot\mathsf{Z}(remf1(lm,n))\\
&\qquad\qquad \lhd\, n < lenf(lm)\wedge remf1(lm,n)\neq\langle\,\rangle\wedge c_j(\overrightarrow{getf1d(lm,n),e_j})\rhd\delta\\
&\quad+\sum_{j\in J\setminus J_\tau}\sum_{\mathsf{a}\in\mathbf{R}(j)}\sum_{n:Nat}\sum_{\overrightarrow{e_j:E_j}} \mathsf{a}(\overrightarrow{f_j}(\overrightarrow{getf1d(lm,n),e_j}))\\
&\qquad\qquad \lhd\, n < lenf(lm)\wedge remf1(lm,n)=\langle\,\rangle\wedge c_j(\overrightarrow{getf1d(lm,n),e_j})\\
&\qquad\qquad\quad \wedge\,\mathbf{mka}[\mathsf{a}_j]\notin getH(getf1a(lm,n))\cup getI(getf1a(lm,n))\\
&\qquad\qquad\quad \wedge\,\mathbf{mka}[\mathsf{a}] = appl(\mathbf{mka}[\mathsf{a}_j],getR(getf1a(lm,n)))\rhd\delta\\
&\quad+\sum_{j\in J\setminus J_\tau}\sum_{n:Nat}\sum_{\overrightarrow{e_j:E_j}}\tau\lhd n < lenf(lm)\wedge remf1(lm,n)=\langle\,\rangle\wedge c_j(\overrightarrow{getf1d(lm,n),e_j})\\
&\qquad\qquad\qquad\qquad \wedge\,\mathbf{mka}[\mathsf{a}_j]\in getI(getf1a(lm,n))\setminus getH(getf1a(lm,n))\rhd\delta\\
&\quad+\sum_{j\in J_\tau}\sum_{n:Nat}\sum_{\overrightarrow{e_j:E_j}}\tau\lhd n < lenf(lm)\wedge remf1(lm,n)=\langle\,\rangle\wedge c_j(\overrightarrow{getf1d(lm,n),e_j})\rhd\delta
\end{aligned}
$$

$$
\begin{aligned}
&+ \sum_{(k,l)\in(I\setminus I_\tau)^2} \sum_{(\mathsf{a},\mathsf{b},\mathsf{c})\in\mathbf{R}^{\mathbf{3}}_\gamma(k,l)} \sum_{n:Nat} \sum_{m:Nat} \sum_{\overrightarrow{e_k:E_k}} \sum_{\overrightarrow{e'_l:E_l}} \mathsf{a}(\overrightarrow{f_k}(\overrightarrow{getf1d(lm,n),e_k}))\\
&\quad \cdot \mathsf{Z}(replf2(lm,n,m,\mathbf{mklm}_k[p_k](\overrightarrow{getf1d(lm,n),e_k}),\mathbf{mklm}_l[p_l](\overrightarrow{getf1d(lm,m),e'_l})))\\
&\quad\ \lhd n<m \wedge m<lenf(lm) \wedge \overrightarrow{f_k}(\overrightarrow{getf1d(lm,n),e_k}) = \overrightarrow{f_l}(\overrightarrow{getf1d(lm,m),e'_l})\\
&\qquad \wedge c_k(\overrightarrow{getf1d(lm,n),e_k}) \wedge c_l(\overrightarrow{getf1d(lm,m),e'_l})\\
&\qquad \wedge \mathbf{mka}[\mathsf{a}_k] \notin getH(getf2a0(lm,n,m)) \cup getI(getf2a0(lm,n,m))\\
&\qquad \wedge \mathbf{mka}[\mathsf{a}_l] \notin getH(getf2a1(lm,n,m)) \cup getI(getf2a1(lm,n,m))\\
&\qquad \wedge \mathbf{mka}[\mathsf{b}] = appl(\mathbf{mka}[\mathsf{a}_k], getR(getf2a0(lm,n,m)))\\
&\qquad \wedge \mathbf{mka}[\mathsf{c}] = appl(\mathbf{mka}[\mathsf{a}_l], getR(getf2a1(lm,n,m)))\\
&\qquad \wedge \mathbf{mka}[\gamma(\mathsf{b},\mathsf{c})] \notin getH(getf2a(lm,n,m)) \cup getI(getf2a(lm,n,m))\\
&\qquad \wedge \mathbf{mka}[\mathsf{a}] = appl(\mathbf{mka}[\gamma(\mathsf{b},\mathsf{c})], getR(getf2a(lm,n,m))) \rhd \delta\\
&+ \sum_{(k,l)\in(I\setminus I_\tau)^2} \sum_{(\mathsf{b},\mathsf{c})\in\mathbf{R}^{\mathbf{2}}_\gamma(k,l)} \sum_{n:Nat} \sum_{m:Nat} \sum_{\overrightarrow{e_k:E_k}} \sum_{\overrightarrow{e'_l:E_l}} \tau\\
&\quad \cdot \mathsf{Z}(replf2(lm,n,m,\mathbf{mklm}_k[p_k](\overrightarrow{getf1d(lm,n),e_k}),\mathbf{mklm}_l[p_l](\overrightarrow{getf1d(lm,m),e'_l})))\\
&\quad\ \lhd n<m \wedge m<lenf(lm) \wedge \overrightarrow{f_k}(\overrightarrow{getf1d(lm,n),e_k}) = \overrightarrow{f_l}(\overrightarrow{getf1d(lm,m),e'_l})\\
&\qquad \wedge c_k(\overrightarrow{getf1d(lm,n),e_k}) \wedge c_l(\overrightarrow{getf1d(lm,m),e'_l})\\
&\qquad \wedge \mathbf{mka}[\mathsf{a}_k] \notin getH(getf2a0(lm,n,m)) \cup getI(getf2a0(lm,n,m))\\
&\qquad \wedge \mathbf{mka}[\mathsf{a}_l] \notin getH(getf2a1(lm,n,m)) \cup getI(getf2a1(lm,n,m))\\
&\qquad \wedge \mathbf{mka}[\mathsf{b}] = appl(\mathbf{mka}[\mathsf{a}_k], getR(getf2a0(lm,n,m)))\\
&\qquad \wedge \mathbf{mka}[\mathsf{c}] = appl(\mathbf{mka}[\mathsf{a}_l], getR(getf2a1(lm,n,m)))\\
&\qquad \wedge \mathbf{mka}[\gamma(\mathsf{b},\mathsf{c})] \in getH(getf2a(lm,n,m)) \cup getI(getf2a(lm,n,m)) \rhd \delta\\
&+ \sum_{(k,l)\in(I\setminus I_\tau)\times(J\setminus J_\tau)} \sum_{(\mathsf{a},\mathsf{b},\mathsf{c})\in\mathbf{R}^{\mathbf{3}}_\gamma(k,l)} \sum_{n:Nat} \sum_{m:Nat} \sum_{\overrightarrow{e_k:E_k}} \sum_{\overrightarrow{e'_l:E_l}} \mathsf{a}(\overrightarrow{f_k}(\overrightarrow{getf1d(lm,n),e_k}))\\
&\quad \cdot \mathsf{Z}(replremf2(lm,n,m,\mathbf{mklm}_k[p_k](\overrightarrow{getf1d(lm,n),e_k})))\\
&\quad\ \lhd n\neq m \wedge n<lenf(lm) \wedge m<lenf(lm) \wedge \overrightarrow{f_k}(\overrightarrow{getf1d(lm,n),e_k}) = \overrightarrow{f_l}(\overrightarrow{getf1d(lm,m),e'_l})\\
&\qquad \wedge c_k(\overrightarrow{getf1d(lm,n),e_k}) \wedge c_l(\overrightarrow{getf1d(lm,m),e'_l})\\
&\qquad \wedge \mathbf{mka}[\mathsf{a}_k] \notin getH(getf2a0(lm,n,m)) \cup getI(getf2a0(lm,n,m))\\
&\qquad \wedge \mathbf{mka}[\mathsf{a}_l] \notin getH(getf2a1(lm,n,m)) \cup getI(getf2a1(lm,n,m))\\
&\qquad \wedge \mathbf{mka}[\mathsf{b}] = appl(\mathbf{mka}[\mathsf{a}_k], getR(getf2a0(lm,n,m)))\\
&\qquad \wedge \mathbf{mka}[\mathsf{c}] = appl(\mathbf{mka}[\mathsf{a}_l], getR(getf2a1(lm,n,m)))\\
&\qquad \wedge \mathbf{mka}[\gamma(\mathsf{b},\mathsf{c})] \notin getH(getf2a(lm,n,m)) \cup getI(getf2a(lm,n,m))\\
&\qquad \wedge \mathbf{mka}[\mathsf{a}] = appl(\mathbf{mka}[\gamma(\mathsf{b},\mathsf{c})], getR(getf2a(lm,n,m))) \rhd \delta
\end{aligned}
$$

$$
\begin{aligned}
+ & \sum_{(k,l)\in(I\setminus I_\tau)\times(J\setminus J_\tau)} \sum_{(\mathsf{b},\mathsf{c})\in\mathbf{R}^{\mathbf{2}}_{\gamma}(k,l)} \sum_{n:Nat} \sum_{m:Nat} \sum_{\overrightarrow{e_k:E_k}} \sum_{\overrightarrow{e'_l:E_l}} \tau \\
& \cdot \mathsf{Z}(\mathit{replremf2}(lm,n,m,\mathbf{mklm}_k[p_k](\overrightarrow{\mathit{getf1d}(lm,n),e_k}))) \\
& \quad \lhd n \neq m \wedge n < \mathit{lenf}(lm) \wedge m < \mathit{lenf}(lm) \wedge \overrightarrow{f_k}(\overrightarrow{\mathit{getf1d}(lm,n),e_k}) = \overrightarrow{f_l}(\overrightarrow{\mathit{getf1d}(lm,m),e'_l}) \\
& \quad\quad \wedge c_k(\overrightarrow{\mathit{getf1d}(lm,n),e_k}) \wedge c_l(\overrightarrow{\mathit{getf1d}(lm,m),e'_l}) \\
& \quad\quad \wedge \mathbf{mka}[\mathsf{a}_k] \notin \mathit{getH}(\mathit{getf2a0}(lm,n,m)) \cup \mathit{getI}(\mathit{getf2a0}(lm,n,m)) \\
& \quad\quad \wedge \mathbf{mka}[\mathsf{a}_l] \notin \mathit{getH}(\mathit{getf2a1}(lm,n,m)) \cup \mathit{getI}(\mathit{getf2a1}(lm,n,m)) \\
& \quad\quad \wedge \mathbf{mka}[\mathsf{b}] = \mathit{appl}(\mathbf{mka}[\mathsf{a}_k], \mathit{getR}(\mathit{getf2a0}(lm,n,m))) \\
& \quad\quad \wedge \mathbf{mka}[\mathsf{c}] = \mathit{appl}(\mathbf{mka}[\mathsf{a}_l], \mathit{getR}(\mathit{getf2a1}(lm,n,m))) \\
& \quad\quad \wedge \mathbf{mka}[\gamma(\mathsf{b},\mathsf{c})] \in \mathit{getH}(\mathit{getf2a}(lm,n,m)) \cup \mathit{getI}(\mathit{getf2a}(lm,n,m)) \rhd \delta \\
+ & \sum_{(k,l)\in(J\setminus J_\tau)^2} \sum_{(\mathsf{a},\mathsf{b},\mathsf{c})\in\mathbf{R}^{\mathbf{3}}_{\gamma}(k,l)} \sum_{n:Nat} \sum_{m:Nat} \sum_{\overrightarrow{e_k:E_k}} \sum_{\overrightarrow{e'_l:E_l}} \mathsf{a}(\overrightarrow{\mathit{getf1d}(lm,n),e_k}) \cdot \mathsf{Z}(\mathit{remf2}(lm,n,m)) \\
& \quad \lhd n < m \wedge m < \mathit{lenf}(lm) \wedge \overrightarrow{f_k}(\overrightarrow{\mathit{getf1d}(lm,n),e_k}) = \overrightarrow{f_l}(\overrightarrow{\mathit{getf1d}(lm,m),e'_l}) \\
& \quad\quad \wedge c_k(\overrightarrow{\mathit{getf1d}(lm,n),e_k}) \wedge c_l(\overrightarrow{\mathit{getf1d}(lm,m),e'_l}) \wedge \mathit{remf2}(lm,n,m) \neq \langle\,\rangle \\
& \quad\quad \wedge \mathbf{mka}[\mathsf{a}_k] \notin \mathit{getH}(\mathit{getf2a0}(lm,n,m)) \cup \mathit{getI}(\mathit{getf2a0}(lm,n,m)) \\
& \quad\quad \wedge \mathbf{mka}[\mathsf{a}_l] \notin \mathit{getH}(\mathit{getf2a1}(lm,n,m)) \cup \mathit{getI}(\mathit{getf2a1}(lm,n,m)) \\
& \quad\quad \wedge \mathbf{mka}[\mathsf{b}] = \mathit{appl}(\mathbf{mka}[\mathsf{a}_k], \mathit{getR}(\mathit{getf2a0}(lm,n,m))) \\
& \quad\quad \wedge \mathbf{mka}[\mathsf{c}] = \mathit{appl}(\mathbf{mka}[\mathsf{a}_l], \mathit{getR}(\mathit{getf2a1}(lm,n,m))) \\
& \quad\quad \wedge \mathbf{mka}[\gamma(\mathsf{b},\mathsf{c})] \notin \mathit{getH}(\mathit{getf2a}(lm,n,m)) \cup \mathit{getI}(\mathit{getf2a}(lm,n,m)) \\
& \quad\quad \wedge \mathbf{mka}[\mathsf{a}] = \mathit{appl}(\mathbf{mka}[\gamma(\mathsf{b},\mathsf{c})], \mathit{getR}(\mathit{getf2a}(lm,n,m))) \rhd \delta \\
+ & \sum_{(k,l)\in(J\setminus J_\tau)^2} \sum_{(\mathsf{b},\mathsf{c})\in\mathbf{R}^{\mathbf{2}}_{\gamma}(k,l)} \sum_{n:Nat} \sum_{m:Nat} \sum_{\overrightarrow{e_k:E_k}} \sum_{\overrightarrow{e'_l:E_l}} \tau \cdot \mathsf{Z}(\mathit{remf2}(lm,n,m)) \\
& \quad \lhd n < m \wedge m < \mathit{lenf}(lm) \wedge \overrightarrow{f_k}(\overrightarrow{\mathit{getf1d}(lm,n),e_k}) = \overrightarrow{f_l}(\overrightarrow{\mathit{getf1d}(lm,m),e'_l}) \\
& \quad\quad \wedge c_k(\overrightarrow{\mathit{getf1d}(lm,n),e_k}) \wedge c_l(\overrightarrow{\mathit{getf1d}(lm,m),e'_l}) \wedge \mathit{remf2}(lm,n,m) \neq \langle\,\rangle \\
& \quad\quad \wedge \mathbf{mka}[\mathsf{a}_k] \notin \mathit{getH}(\mathit{getf2a0}(lm,n,m)) \cup \mathit{getI}(\mathit{getf2a0}(lm,n,m)) \\
& \quad\quad \wedge \mathbf{mka}[\mathsf{a}_l] \notin \mathit{getH}(\mathit{getf2a1}(lm,n,m)) \cup \mathit{getI}(\mathit{getf2a1}(lm,n,m)) \\
& \quad\quad \wedge \mathbf{mka}[\mathsf{b}] = \mathit{appl}(\mathbf{mka}[\mathsf{a}_k], \mathit{getR}(\mathit{getf2a0}(lm,n,m))) \\
& \quad\quad \wedge \mathbf{mka}[\mathsf{c}] = \mathit{appl}(\mathbf{mka}[\mathsf{a}_l], \mathit{getR}(\mathit{getf2a1}(lm,n,m))) \\
& \quad\quad \wedge \mathbf{mka}[\gamma(\mathsf{b},\mathsf{c})] \in \mathit{getH}(\mathit{getf2a}(lm,n,m)) \cup \mathit{getI}(\mathit{getf2a}(lm,n,m)) \rhd \delta
\end{aligned}
$$

$$
\begin{aligned}
+ \sum_{(k,l)\in(J\setminus J_\tau)^2} \sum_{(\mathsf{a},\mathsf{b},\mathsf{c})\in\mathbf{R}^{\mathbf{3}}_\gamma(k,l)} \sum_{n:Nat} \sum_{m:Nat} \sum_{\overrightarrow{e_k:E_k}} \sum_{\overrightarrow{e'_l:E_l}} & \mathsf{a}(\overrightarrow{f_k}(\overrightarrow{getf1d(lm,n),e_k})) \\
& \lhd n < m \wedge m < lenf(lm) \wedge \overrightarrow{f_k}(\overrightarrow{getf1d(lm,n),e_k}) = \overrightarrow{f_l}(\overrightarrow{getf1d(lm,m),e'_l}) \\
& \wedge c_k(\overrightarrow{getf1d(lm,n),e_k}) \wedge c_l(\overrightarrow{getf1d(lm,m),e'_l}) \wedge remf2(lm,n,m) = \langle\,\rangle \\
& \wedge \mathbf{mka}[\mathsf{a}_k] \notin getH(getf2a0(lm,n,m)) \cup getI(getf2a0(lm,n,m)) \\
& \wedge \mathbf{mka}[\mathsf{a}_l] \notin getH(getf2a1(lm,n,m)) \cup getI(getf2a1(lm,n,m)) \\
& \wedge \mathbf{mka}[\mathsf{b}] = appl(\mathbf{mka}[\mathsf{a}_k], getR(getf2a0(lm,n,m))) \\
& \wedge \mathbf{mka}[\mathsf{c}] = appl(\mathbf{mka}[\mathsf{a}_l], getR(getf2a1(lm,n,m))) \\
& \wedge \mathbf{mka}[\gamma(\mathsf{b},\mathsf{c})] \notin getH(getf2a(lm,n,m)) \cup getI(getf2a(lm,n,m)) \\
& \wedge \mathbf{mka}[\mathsf{a}] = appl(\mathbf{mka}[\gamma(\mathsf{b},\mathsf{c})], getR(getf2a(lm,n,m))) \rhd \delta \\
+ \sum_{(k,l)\in(J\setminus J_\tau)^2} \sum_{(\mathsf{b},\mathsf{c})\in\mathbf{R}^{\mathbf{2}}_\gamma(k,l)} \sum_{n:Nat} \sum_{m:Nat} \sum_{\overrightarrow{e_k:E_k}} \sum_{\overrightarrow{e'_l:E_l}} & \tau \\
& \lhd n < m \wedge m < lenf(lm) \wedge \overrightarrow{f_k}(\overrightarrow{getf1d(lm,n),e_k}) = \overrightarrow{f_l}(\overrightarrow{getf1d(lm,m),e'_l}) \\
& \wedge c_k(\overrightarrow{getf1d(lm,n),e_k}) \wedge c_l(\overrightarrow{getf1d(lm,m),e'_l}) \wedge remf2(lm,n,m) = \langle\,\rangle \\
& \wedge \mathbf{mka}[\mathsf{a}_k] \notin getH(getf2a0(lm,n,m)) \cup getI(getf2a0(lm,n,m)) \\
& \wedge \mathbf{mka}[\mathsf{a}_l] \notin getH(getf2a1(lm,n,m)) \cup getI(getf2a1(lm,n,m)) \\
& \wedge \mathbf{mka}[\mathsf{b}] = appl(\mathbf{mka}[\mathsf{a}_k], getR(getf2a0(lm,n,m))) \\
& \wedge \mathbf{mka}[\mathsf{c}] = appl(\mathbf{mka}[\mathsf{a}_l], getR(getf2a1(lm,n,m))) \\
& \wedge \mathbf{mka}[\gamma(\mathsf{b},\mathsf{c})] \in getH(getf2a(lm,n,m)) \cup getI(getf2a(lm,n,m)) \rhd \delta
\end{aligned}
$$

where

$$
\begin{aligned}
& I_\tau = \{i \in I \mid \mathsf{a}_i = \tau\} \qquad J_\tau = \{j \in J \mid \mathsf{a}_j = \tau\} \\
& \mathbf{R}(i) = \{\mathsf{a} \in ActLab \mid type(\mathsf{a}) = type(\mathsf{a}_i)\} \\
& \mathbf{R}^{\mathbf{2}}_\gamma(k,l) = \{(\mathsf{b},\mathsf{c}) \in ActLab^2 \mid type(\mathsf{b}) = type(\mathsf{a}_k) = type(\mathsf{c}) = type(\mathsf{a}_l) \wedge \gamma(\mathsf{a},\mathsf{b}) \text{ is defined}\} \\
& \mathbf{R}^{\mathbf{3}}_\gamma(k,l) = \{(\mathsf{a},\mathsf{b},\mathsf{c}) \in ActLab \times \mathbf{R}^{\mathbf{2}}_\gamma(k,l) \mid type(\mathsf{a}) = type(\mathsf{b})\}
\end{aligned}
$$

The LPE Z is in a sense an extension of the LPE we obtained for the case without the remaining operations. The first nine summands correspond to the first three summands of the latter LPE, so each of the interleaving possibilities is represented by three summands. The first one represents the case when the action (not τ) is not encapsulated or hidden, but can be renamed. The second one represents the case when the action (not τ) is not encapsulated, but hidden. And the third one represents the τ summands (we treat them separately, because τ cannot be encapsulated, hidden or renamed). There is no summand for the encapsulated actions, as they all become equal to δ and vanish.

In the case of handshakings, we get only two summands for each summand in the case without the renaming operations. This is because τ does not communicate and we do not need an additional summand for it.

B. Resulting LPE for the Case with the Renaming Operations and Multi-Party Communication

Without loss of generality, we assume that $J \setminus J_\tau = \{0, \ldots, k\}$ and $I \setminus I_\tau = \{k+1, \ldots, m\}$.

$$
\begin{aligned}
&\mathsf{Z}(lm{:}ALM) = \\
&\sum_{i \in I \setminus I_\tau} \sum_{\mathsf{a} \in \mathbf{R}(i)} \sum_{ln_0:LNat} \cdots \sum_{ln_m:LNat} \sum_{\overrightarrow{le_0:LE_0}} \cdots \sum_{\overrightarrow{le_m:LE_m}} \mathsf{a}(\overrightarrow{f0}(lm, ln_0, \ldots, ln_m, head(\overrightarrow{le_0}), \ldots, head(\overrightarrow{le_m}))) \\
&\quad \cdot \mathsf{Z}(replfn(lm, ln, \mathbf{cat}[\mathbf{mkllm}[p_0](\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}), \ldots, \mathbf{mkllm}[p_m](\overrightarrow{getfn(lm, ln_m)}, \overrightarrow{le_m})])) \\
&\qquad \lhd lnI \neq LNat0 \wedge len(ln) \leq lenf(lm) \wedge is_unique(ln) \wedge \bigwedge\nolimits_{0 \leq l \leq m} is_sorted(ln_l) \\
&\qquad\quad \wedge \bigwedge\nolimits_{0 \leq l \leq m} is_each_lower(lenf(lm), ln_l) \wedge \bigwedge\nolimits_{0 \leq l \leq m} len(ln_l) = len(\overrightarrow{le_l}) \\
&\qquad\quad \wedge EQ(\mathbf{cat}[F_0(\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}), \ldots, F_m(\overrightarrow{getfn(lm, ln_m)}, \overrightarrow{le_m})]) \\
&\qquad\quad \wedge C_0(\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}) \wedge \cdots \wedge C_m(\overrightarrow{getfn(lm, ln_m)}, \overrightarrow{le_m}) \\
&\qquad\quad \wedge is_act(\mathbf{mka}[\mathsf{a}], lm, ln, \\
&\qquad\qquad \mathbf{cat}[mklact(len(ln_0), \mathbf{mka}[\mathsf{a}_0]), \ldots, mklact(len(ln_m), \mathbf{mka}[\mathsf{a}_m])]) \rhd \delta \\
&+ \sum_{i \in I \setminus I_\tau} \sum_{ln_0:LNat} \cdots \sum_{ln_m:LNat} \sum_{\overrightarrow{le_0:LE_0}} \cdots \sum_{\overrightarrow{le_m:LE_m}} \tau \\
&\quad \cdot \mathsf{Z}(replfn(lm, ln, \mathbf{cat}[\mathbf{mkllm}[p_0](\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}), \ldots, \mathbf{mkllm}[p_m](\overrightarrow{getfn(lm, ln_m)}, \overrightarrow{le_m})])) \\
&\qquad \lhd lnI \neq LNat0 \wedge len(ln) \leq lenf(lm) \wedge is_unique(ln) \wedge \bigwedge\nolimits_{0 \leq l \leq m} is_sorted(ln_l) \\
&\qquad\quad \wedge \bigwedge\nolimits_{0 \leq l \leq m} is_each_lower(lenf(lm), ln_l) \wedge \bigwedge\nolimits_{0 \leq l \leq m} len(ln_l) = len(\overrightarrow{le_l}) \\
&\qquad\quad \wedge EQ(\mathbf{cat}[F_0(\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}), \ldots, F_m(\overrightarrow{getfn(lm, ln_m)}, \overrightarrow{le_m})]) \\
&\qquad\quad \wedge C_0(\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}) \wedge \cdots \wedge C_m(\overrightarrow{getfn(lm, ln_m)}, \overrightarrow{le_m}) \\
&\qquad\quad \wedge is_tau(lm, ln, \\
&\qquad\qquad \mathbf{cat}[mklact(len(ln_0), \mathbf{mka}[\mathsf{a}_0]), \ldots, mklact(len(ln_m), \mathbf{mka}[\mathsf{a}_m])]) \rhd \delta \\
&+ \sum_{i \in I_\tau} \sum_{n:Nat} \sum_{\overrightarrow{e_i:E_i}} \tau \cdot \mathsf{Z}(replf1(lm, n, \mathbf{mklm}_i[p_i](\overrightarrow{getf1d(lm, n), e_i}))) \\
&\qquad \lhd n < lenf(lm) \wedge c_i(\overrightarrow{getf1d(lm, n), e_i}) \rhd \delta \\
&+ \sum_{j \in J \setminus J_\tau} \sum_{\mathsf{a} \in \mathbf{R}(j)} \sum_{ln_0:LNat} \cdots \sum_{ln_k:LNat} \sum_{\overrightarrow{le_0:LE_0}} \cdots \sum_{\overrightarrow{le_k:LE_k}} \mathsf{a}(\overrightarrow{f0}(lm, ln_0, \ldots, ln_k, head(\overrightarrow{le_0}), \ldots, head(\overrightarrow{le_k}))) \\
&\quad \cdot \mathsf{Z}(remfn(lm, lnJ)) \\
&\qquad \lhd lnJ \neq LNat0 \wedge len(lnJ) \leq lenf(lm) \wedge is_unique(lnJ) \wedge \bigwedge\nolimits_{0 \leq l \leq k} is_sorted(ln_l) \\
&\qquad\quad \wedge \bigwedge\nolimits_{0 \leq l \leq k} is_each_lower(lenf(lm), ln_l) \wedge \bigwedge\nolimits_{0 \leq l \leq k} len(ln_l) = len(\overrightarrow{le_l}) \\
&\qquad\quad \wedge EQ(\mathbf{cat}[F_0(\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}), \ldots, F_k(\overrightarrow{getfn(lm, ln_k)}, \overrightarrow{le_k})]) \\
&\qquad\quad \wedge C_0(\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}) \wedge \cdots \wedge C_k(\overrightarrow{getfn(lm, ln_k)}, \overrightarrow{le_k}) \\
&\qquad\quad \wedge is_act(\mathbf{mka}[\mathsf{a}], lm, lnJ, \\
&\qquad\qquad \mathbf{cat}[mklact(len(ln_0), \mathbf{mka}[\mathsf{a}_0]), \ldots, mklact(len(ln_k), \mathbf{mka}[\mathsf{a}_k])]) \\
&\qquad\quad \wedge remfn(lm, lnJ) \neq \langle\, \rangle \rhd \delta
\end{aligned}
$$

$$
\begin{aligned}
&+ \sum_{j\in J\setminus J_\tau} \sum_{ln_0:LNat} \cdots \sum_{ln_k:LNat} \sum_{\overrightarrow{le_0:LE_0}} \cdots \sum_{\overrightarrow{le_k:LE_k}} \tau \cdot \mathsf{Z}(remfn(lm, lnJ)) \\
&\qquad \lhd lnJ \neq LNat0 \wedge len(lnJ) \leq lenf(lm) \wedge is\_unique(lnJ) \wedge \bigwedge\nolimits_{0\leq l\leq k} is\_sorted(ln_l) \\
&\qquad\quad \wedge \bigwedge\nolimits_{0\leq l\leq k} is\_each\_lower(lenf(lm), ln_l) \wedge \bigwedge\nolimits_{0\leq l\leq k} len(ln_l) = len(\overrightarrow{le_l}) \\
&\qquad\quad \wedge EQ(\mathbf{cat}[F_0(\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}), \ldots, F_k(\overrightarrow{getfn(lm, ln_k)}, \overrightarrow{le_k})]) \\
&\qquad\quad \wedge C_0(\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}) \wedge \cdots \wedge C_k(\overrightarrow{getfn(lm, ln_k)}, \overrightarrow{le_k}) \\
&\qquad\quad \wedge is\_tau(lm, lnJ, \\
&\qquad\qquad\quad \mathbf{cat}[mklact(len(ln_0), \mathbf{mka}[\mathsf{a}_0]), \ldots, mklact(len(ln_k), \mathbf{mka}[\mathsf{a}_k])]) \\
&\qquad\quad \wedge remfn(lm, lnJ) \neq \langle\,\rangle \rhd \delta \\
&+ \sum_{j\in J_\tau} \sum_{n:Nat} \sum_{\overrightarrow{e_j:E_j}} \tau \cdot \mathsf{Z}(remf1(lm, n)) \\
&\qquad \lhd n < lenf(lm) \wedge remf1(lm, n) \neq \langle\,\rangle \wedge c_j(\overrightarrow{getf1d(lm, n), e_j}) \rhd \delta \\
&+ \sum_{j\in J\setminus J_\tau} \sum_{\mathsf{a}\in\mathbf{R}(j)} \sum_{ln_0:LNat} \cdots \sum_{ln_k:LNat} \sum_{\overrightarrow{le_0:LE_0}} \cdots \sum_{\overrightarrow{le_k:LE_k}} \mathsf{a}(\overrightarrow{f0}(lm, ln_0, \ldots, ln_k, head(\overrightarrow{le_0}), \ldots, head(\overrightarrow{le_k}))) \\
&\qquad \lhd lnJ \neq LNat0 \wedge len(lnJ) \leq lenf(lm) \wedge is\_unique(lnJ) \wedge \bigwedge\nolimits_{0\leq l\leq k} is\_sorted(ln_l) \\
&\qquad\quad \wedge \bigwedge\nolimits_{0\leq l\leq k} is\_each\_lower(lenf(lm), ln_l) \wedge \bigwedge\nolimits_{0\leq l\leq k} len(ln_l) = len(\overrightarrow{le_l}) \\
&\qquad\quad \wedge EQ(\mathbf{cat}[F_0(\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}), \ldots, F_k(\overrightarrow{getfn(lm, ln_k)}, \overrightarrow{le_k})]) \\
&\qquad\quad \wedge C_0(\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}) \wedge \cdots \wedge C_k(\overrightarrow{getfn(lm, ln_k)}, \overrightarrow{le_k}) \\
&\qquad\quad \wedge is\_act(\mathbf{mka}[\mathsf{a}], lm, lnJ, \\
&\qquad\qquad\quad \mathbf{cat}[mklact(len(ln_0), \mathbf{mka}[\mathsf{a}_0]), \ldots, mklact(len(ln_k), \mathbf{mka}[\mathsf{a}_k])]) \\
&\qquad\quad \wedge remfn(lm, lnJ) = \langle\,\rangle \rhd \delta \\
&+ \sum_{j\in J\setminus J_\tau} \sum_{ln_0:LNat} \cdots \sum_{ln_k:LNat} \sum_{\overrightarrow{le_0:LE_0}} \cdots \sum_{\overrightarrow{le_k:LE_k}} \tau \\
&\qquad \lhd lnJ \neq LNat0 \wedge len(lnJ) \leq lenf(lm) \wedge is\_unique(lnJ) \wedge \bigwedge\nolimits_{0\leq l\leq k} is\_sorted(ln_l) \\
&\qquad\quad \wedge \bigwedge\nolimits_{0\leq l\leq k} is\_each\_lower(lenf(lm), ln_l) \wedge \bigwedge\nolimits_{0\leq l\leq k} len(ln_l) = len(\overrightarrow{le_l}) \\
&\qquad\quad \wedge EQ(\mathbf{cat}[F_0(\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}), \ldots, F_k(\overrightarrow{getfn(lm, ln_k)}, \overrightarrow{le_k})]) \\
&\qquad\quad \wedge C_0(\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}) \wedge \cdots \wedge C_k(\overrightarrow{getfn(lm, ln_k)}, \overrightarrow{le_k}) \\
&\qquad\quad \wedge is\_tau(lm, lnJ, \\
&\qquad\qquad\quad \mathbf{cat}[mklact(len(ln_0), \mathbf{mka}[\mathsf{a}_0]), \ldots, mklact(len(ln_k), \mathbf{mka}[\mathsf{a}_k])]) \\
&\qquad\quad \wedge remfn(lm, lnJ) = \langle\,\rangle \rhd \delta \\
&+ \sum_{j\in J_\tau} \sum_{n:Nat} \sum_{\overrightarrow{e_j:E_j}} \tau \lhd n < lenf(lm) \wedge remf1(lm, n) = \langle\,\rangle \wedge c_j(\overrightarrow{getf1d(lm, n), e_j}) \rhd \delta
\end{aligned}
$$

where $lnI = \mathbf{cat}[ln_{k+1}, \ldots, ln_m]$, $lnJ = \mathbf{cat}[ln_0, \ldots, ln_k]$, and $lnJ = cat(lnJ, lnI)$.

The first three sets of summands represent multi-party communications of several components with at least one of them not terminating. In the third set we separate the actions a_i that are equal to τ – they cannot communicate and can only be executed in the interleaving way. In the first set of summands we consider all non τ actions a_i and all possible renamings of them. We do not need to consider the renamings of actions a_j here because at least one of the components will be executing an a_i action, and therefore the resulting action will be a renaming of it.

As in the case of multi-party communications without renaming, we take a number of lists to identify which first layer elements will communicate by performing which actions. The condition $lnI \neq LNat0$ ensures that at least one of the elements will not terminate. Instead of checking the conformance to a chosen configuration, we use the function is_act to see if the result of the multi-party communication is the chosen action. The rest of the conditions are the same as in the case without renaming operations. The second set of summands is similar to the first one and captures the case when communication results in τ.

The following six summands capture the case when all components terminate after performing a communication. The first three represent the sub-case when the LPE Z does not terminate in such a situation, and the last three represent the sub-case when the LPE Z terminates.

In case the LPE Z performs an action, its parameters are the parameters of any of the communicating actions, so we take the first one. We could skip the definition of the function $\overrightarrow{f0}$ and use the following expression instead:

$$head(\mathbf{cat}[F_0(\overrightarrow{getfn(lm, ln_0)}, \overrightarrow{le_0}), \ldots, F_k(\overrightarrow{getfn(lm, ln_k)}, \overrightarrow{le_k})])$$

which, however, is a more complex expression.

C. μCRL Code of *LM* and *ML* Data Types

The source code is split into six parts (Figure 1): two basic parts and four terminal parts corresponding to the cases with or without the renaming operations, and with handshaking or with multi-party communication. For each terminal part all of the parts it depends upon are needed (only once in case of multiple dependencies).

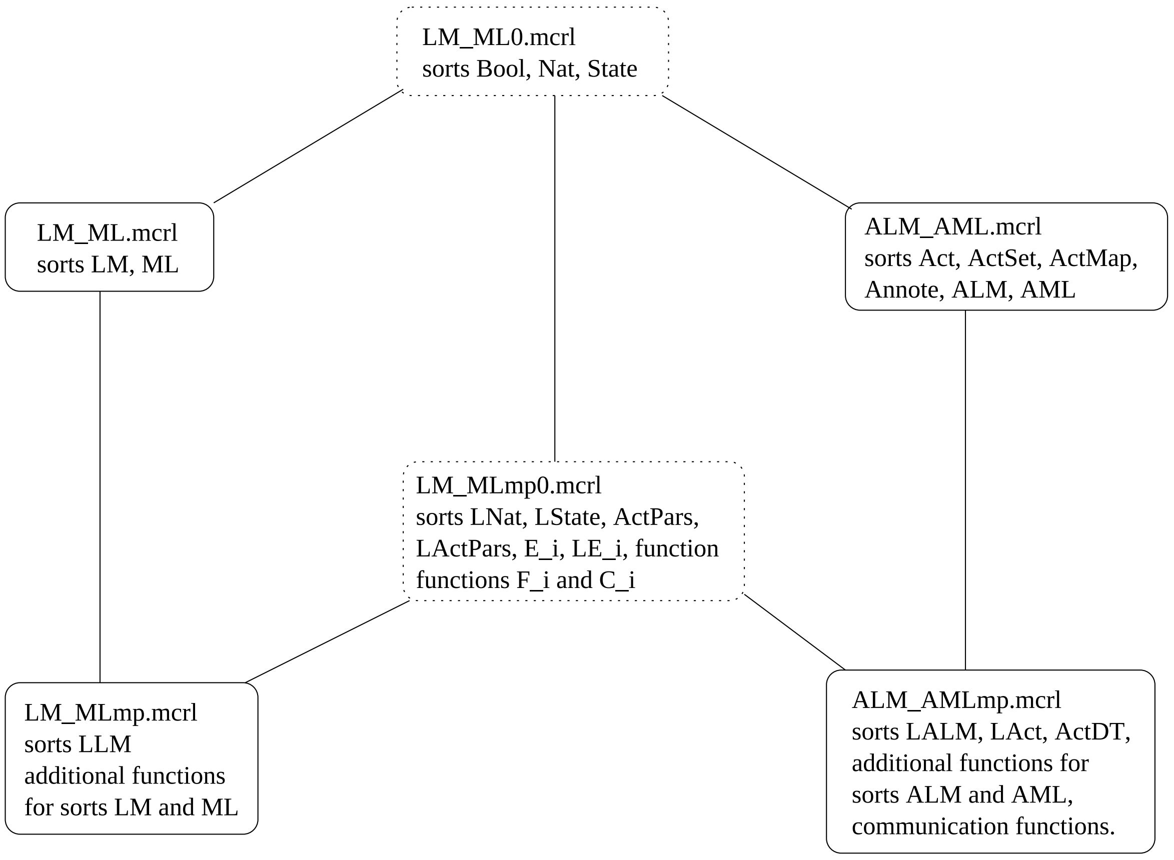
LM_ML0.mcrl
sorts Bool, Nat, State
LM_ML.mcrl
sorts LM, ML
ALM_AML.mcrl
sorts Act, ActSet, ActMap, Annote, ALM, AML
LM_MLmp0.mcrl
sorts LNat, LState, ActPars, LActPars, E_i, LE_i, function functions F_i and C_i
LM_MLmp.mcrl
sorts LLM
additional functions for sorts LM and ML
ALM_AMLmp.mcrl
sorts LALM, LAct, ActDT, additional functions for sorts ALM and AML, communication functions.

Figure 1: Code Files Dependencies.

C.1 Basic Data Types

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%% sorts Bool, Nat, State(generated)                            %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  sort  Bool (Booleans)                                       %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Bool
func
  T,F: -> Bool
map
  and: Bool#Bool      -> Bool
  or:  Bool#Bool      -> Bool
  not: Bool           -> Bool
  if:  Bool#Bool#Bool -> Bool
  eq:  Bool#Bool      -> Bool
  gt:  Bool#Bool      -> Bool
var
  b,b1,b2: Bool
rew
  and(T,b)=b                    and(b,T)=b
  and(b,F)=F                    and(F,b)=F
  and(b,b)=b
  and(b,not(b))=F               and(not(b),b)=F
  and(or(b,b1),b2)=or(and(b,b2),and(b1,b2))
  and(b,or(b1,b2))=or(and(b,b1),and(b,b2))

  or(T,b)=T                     or(b,T)=T
  or(b,F)=b                     or(F,b)=b
  or(b,b)=b
  or(b,not(b))=T                or(not(b),b)=T

  not(F)=T                      not(T)=F
  not(not(b))=b
  not(or(b,b1))=and(not(b),not(b1))
  not(and(b,b1))=or(not(b),not(b1))

  if(T,b1,b2)=b1                if(F,b1,b2)=b2
  if(b,b1,b1)=b1                if(not(b),b1,b2)=if(b,b2,b1)
  if(b,T,b2)=or(b,b2)           if(b,F,b2)=and(not(b),b2)
  if(b,b1,T)=or(not(b),b1)      if(b,b1,F)=and(b,b1)
  if(b,b1,b2)=or(or(and(b,b1),and(not(b),b2)),and(b1,b2))

  eq(b,b)=T eq(b,not(b))=F eq(not(b),b)=F eq(not(b),not(b1))=eq(b,b1)
  eq(F,b)=not(b) eq(b,F)=not(b) eq(T,b)=b eq(b,T)=b
  eq(b,b1)=or(and(b,b1),and(not(b),not(b1)))

  gt(b,b)=F gt(T,F)=T gt(b,T)=F

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  sort   Nat (Natural numbers with binary representations)   %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Nat
func
  0:                -> Nat
  x2p1:         Nat -> Nat      % 2n+1
  x2p2:         Nat -> Nat      % 2n+2
map
  eq:           Nat#Nat         -> Bool
  1,2,3,4,5,6:                  -> Nat  % useful abbreviations
  x2p0:         Nat             -> Nat  % 2n
  succ:         Nat             -> Nat  % n+1
  gt:           Nat#Nat         -> Bool % greater than
  if:           Bool#Nat#Nat    -> Nat
  add,sub,csub: Nat#Nat         -> Nat  % addition, subtraction (partial), cut-off subtraction
  divides:      Nat#Nat         -> Bool % does the first argument divide the second? (partial)
var
  n,m: Nat b:Bool
rew
  gt(n,n)=F gt(0,n)=F gt(x2p1(n),0)=T gt(x2p2(n),0)=T

  gt(x2p1(n),x2p2(m))=gt(n,m)
  gt(x2p2(n),x2p1(m))=not(gt(m,n))

  gt(x2p1(n),x2p1(m))=gt(n,m)
  gt(x2p2(n),x2p2(m))=gt(n,m)

  % eq(n,m)=not(or(gt(n,m),gt(m,n)))    % sane, but inefficient

  eq(n,n)=T
  eq(x2p1(n),0)=F
  eq(0,x2p1(n))=F
  eq(x2p2(n),0)=F
  eq(0,x2p2(n))=F
  eq(x2p1(n),x2p2(m))=F
  eq(x2p2(n),x2p1(m))=F
  eq(x2p2(n),x2p2(m))=eq(n,m)
  eq(x2p1(n),x2p1(m))=eq(n,m)

  1=x2p1(0)  2=x2p2(0)                  % 1=2*0+1 2=2*0+2
  3=x2p1(1)  4=x2p2(1)                  % 3=2*1+1 4=2*1+2
  5=x2p1(2)  6=x2p2(2)                  % 5=2*2+1 6=2*2+2

  x2p0(0)=0                             % 2*0=0
  x2p0(x2p1(n))=x2p2(x2p0(n))           % 2(2n+1)=2(2n)+2
  x2p0(x2p2(n))=x2p2(x2p1(n))           % 2(2n+2)=2((2n+1)+1)=2(2n+1)+2

  succ(0)=x2p1(0)                       % 0+1=2*0+1
  succ(x2p1(n))=x2p2(n)                 % (2n+1)+1=2n+2
  succ(x2p2(n))=x2p1(succ(n))           % (2n+2)+1=2(n+1)+1

  add(0,n)=n add(n,0)=n
  add(x2p1(n),x2p1(m))=x2p2(add(n,m))           % (2n+1)+(2m+1)=2(n+m)+2
  add(x2p2(n),x2p2(m))=x2p2(succ(add(n,m)))     % (2n+2)+(2m+2)=2(n+m)+4=2(n+m+1)+2
  add(x2p1(n),x2p2(m))=x2p1(succ(add(n,m)))     % (2n+1)+(2m+2)=2(n+m)+3=2(n+m+1)+1
  add(x2p2(n),x2p1(m))=x2p1(succ(add(n,m)))     % (2n+2)+(2m+1)=2(n+m)+3=2(n+m+1)+1

  sub(n,0)=n sub(n,n)=0                         % sub(0,x2p{1,2}) is undefined
  sub(x2p1(n),x2p1(m))=x2p0(sub(n,m))           % (2n+1)-(2m+1)=2(n-m)
  sub(x2p2(n),x2p2(m))=x2p0(sub(n,m))           % (2n+2)-(2m+2)=the same
  sub(x2p1(n),x2p2(m))=x2p1(sub(n,succ(m)))     % (2n+1)-(2m+2)=2(n-m)-1=2(n-(m+1))+1 -- undef if n=m!
  sub(x2p2(n),x2p1(m))=x2p1(sub(n,m))           % (2n+2)-(2m+1)=2(n-m)+1

  csub(n,m)=if(gt(n,m),sub(n,m),0)

  divides(x2p1(n),0)=T divides(x2p2(n),0)=T         % any n>0 divides 0; divides(0,n) is undefined
  divides(x2p1(n),x2p1(m))=                         % n divides m whenever it divides m-n
        and(not(gt(n,m)),divides(x2p1(n),sub(x2p1(m),x2p1(n))))
  divides(x2p1(n),x2p2(m))=
        and(not(gt(n,m)),divides(x2p1(n),sub(x2p2(m),x2p1(n))))
  divides(x2p2(n),x2p1(m))=F                        % even never divides odd.
  divides(x2p2(n),x2p2(m))=divides(succ(n),succ(m)) % (2n+2)|(2m+2) iff (n+1)|(m+1)

  if(T,n,m)=n if(F,n,m)=m if(b,n,n)=n if(not(b),n,m)=if(b,m,n)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%  To be generated from the spec                              %%%
%%%%  The parts that do not parse before actual generation       %%%
%%%%  are commented out                                          %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  sort  State (pre-LPE process parameters tuple)             %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort State
% func
  % state:D_0#...#D_n->State
map
  eq: State#State->Bool
  gt: State#State->Bool
  if: Bool#State#State->State
  % pr_0:State->D_0 ... pr_n:State->D_n
var
  d,e:State b:Bool
rew
  if(T,d,e)=d if(F,d,e)=e if(b,d,d)=d if(not(b),d,e)=if(b,e,d)
  % gt(state(d0,...,dn),state(e0,...,en))=
  %     or(gt(d0,e0),and(eq(d0,e0),...or(gt(d{n-1},e{n-1}),and(eq(d{n-1},e{n-1}),gt(dn,en)))...))
  eq(d,d)=T
  % eq(state(d0,...,dn),state(e0,...,en))=and(eq(d0,e0),...,and(eq(dn,en))...)
```

C.2 Handshaking *LM* and *ML* Data Types

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%  LM And ML data types                                       %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  sort  LM                                                   %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort LM                         % list of ML or State elements
func
  LM0: ->LM                     % empty list
  seq1: State#LM->LM            % add one State element to the head of the list
  seqM: ML#LM->LM               % add one ML to the head of the list
                                %(first argument never ML(x))
```

```
map
  eq: LM#LM->Bool               % equality on LM
  if: Bool#LM#LM->LM

  gt: LM#LM->Bool
  conc: LM#LM->LM               % concatenate 2 LMs in a wf way.
  conp: ML#LM->LM               % prepend an ML to an LM

  lenf:LM->Nat                  % number of "ready" components
  getf1:LM#Nat->State           % get n-th component
  replf1:LM#Nat#LM->LM          % replace n-th component
  remf1:LM#Nat->LM              % remove n-th component
  replf2:LM#Nat#Nat#LM#LM->LM   % replace n-th and m-th components
  replremf2:LM#Nat#Nat#LM->LM   % replace n-th and remove m-th components
  remf2:LM#Nat#Nat->LM          % remove n-th and m-th components

  parc: LM#LM->LM               % compose 2 LMs parallely
  seqc: LM#LM->LM               % compose 2 LMs sequentially

  is_seq:LM->Bool               % is it a sequential composition of smth. with a nonempty lm?
  getflm:LM->LM                 % only defined if =seqM(ml,lm1): get first elem from ml
  remflm:LM->LM                 % only defined if =seqM(ml,lm1): remove first elem from ml
  getseql:LM->LM                % only defined if =seqM(ml,lm1): get conp(ml,lm0)
  getseqr:LM->LM                % only defined if =seqM(ml,lm1): get lm1
var
  d,d1: State lm,lm1,lm2:LM ml,ml1:ML n,m:Nat b:Bool
rew
  gt(LM0,lm)=F
  gt(seq1(d,lm),LM0)=T
  gt(seq1(d,lm),seq1(d1,lm1))
        =if(eq(lm,LM0),
           if(eq(lm1,LM0),gt(d,d1),F),
           if(eq(lm1,LM0),T,or(gt(d,d1),and(eq(d,d1),gt(lm,lm1)))))
  gt(seq1(d,lm),seqM(ml,lm1))=F
  gt(seqM(ml,lm),LM0)=T
  gt(seqM(ml,lm),seq1(d,lm1))=T
  gt(seqM(ml,lm),seqM(ml1,lm1))
        =if(eq(lm,LM0),
           if(eq(lm1,LM0),gt(ml,ml1),F),
           if(eq(lm1,LM0),T,or(gt(ml,ml1),and(eq(ml,ml1),gt(lm,lm1)))))

  conc(LM0,lm)=lm conc(lm,LM0)=lm
  conc(seq1(d,lm),lm1)=seq1(d,conc(lm,lm1))
  conc(seqM(ml,lm),lm1)=seqM(ml,conc(lm,lm1))

  conp(ML(lm),lm1)=conc(lm,lm1)
  conp(par(lm,ml),lm1)=seqM(par(lm,ml),lm1)

  eq(LM0,seq1(d,lm))=F eq(seq1(d,lm),LM0)=F
  eq(LM0,seqM(ml,lm))=F eq(seqM(ml,lm),LM0)=F
  eq(seq1(d,lm),seqM(ml,lm1))=F eq(seqM(ml,lm1),seq1(d,lm))=F

  eq(lm,lm)=T
  eq(seq1(d,lm),seq1(d1,lm1))=and(eq(d,d1),eq(lm,lm1))
  eq(seqM(ml,lm),seqM(ml1,lm1))=and(eq(ml,ml1),eq(lm,lm1))

  if(T,lm,lm1)=lm if(F,lm,lm1)=lm1 if(b,lm,lm)=lm if(not(b),lm,lm1)=if(b,lm1,lm)

  lenf(LM0)=0
  lenf(seq1(d,lm))=1
  lenf(seqM(ml,lm))=lenf(ml)

  % undefined getf1(LM0,n)=
  getf1(seq1(d,lm),0)=d
  getf1(seqM(ml,lm),n)=getf1(ml,n)

  replf1(seq1(d,lm),0,lm1)=conc(lm1,lm)
  replf1(seqM(ml,lm),n,lm1)=conp(replf1(ml,n,lm1),lm)

  remf1(lm,n)=replf1(lm,n,LM0)

  replf2(seqM(ml,lm),n,m,lm1,lm2)=conp(replf2(ml,n,m,lm1,lm2),lm)
  replremf2(lm,n,m,lm1)=replf2(lm,n,m,lm1,LM0)
  remf2(lm,n,m)=replf2(lm,n,m,LM0,LM0)

  seqc(lm,lm1)=conc(lm,lm1)
  parc(lm,lm1)=conp(comp(mkml(lm),mkml(lm1)),LM0)

  is_seq(LM0)=F is_seq(seq1(d,LM0))=F is_seq(seqM(ml,LM0))=F
  is_seq(seq1(d,seq1(d1,lm)))=T is_seq(seq1(d,seqM(ml,lm)))=T
  is_seq(seqM(ml,seq1(d,lm)))=T is_seq(seqM(ml,seqM(ml1,lm)))=T

  getflm(seqM(par(lm,ml),lm1))=lm
  remflm(seqM(par(lm,ml),lm1))=conp(ml,lm1)
  getseql(seqM(ml,lm))=conp(ml,LM0)
  getseqr(seqM(ml,lm))=lm

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  sort ML                                                     %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort ML                         % Multiset of LM
func
  ML: LM->ML                    % Multiset with one list
  par: LM#ML->ML                % Add a list to the multiset (first argument never LM0)
map
  eq: ML#ML->Bool               % equality on ML
  if:Bool#ML#ML->ML

  mkml :LM->ML                  % Make a proper ML out of an LM
  comp :ML#ML->ML               % Compose 2 MLs in a wf way.
  gt   :ML#ML->Bool

  in: LM#ML->Bool               % test if an lm is in ml (on the first level, of course).
  rem: LM#ML->ML                % remove an lm from ml if it is on the first level, don't change otherwise

  lenf: ML->Nat
  getf1: ML#Nat->State
  replf1: ML#Nat#LM->ML
  replf2:ML#Nat#Nat#LM#LM->ML   % replace n-th and m-th components
var
  d,d1: State lm,lm1,lm2:LM ml,ml1:ML n,m:Nat b:Bool
rew
  gt(ML(lm),ML(lm1))=gt(lm,lm1)
  gt(ML(lm),par(lm1,ml))=F
  gt(par(lm1,ml),ML(lm))=T
  gt(par(lm,ml),par(lm1,ml1))=or(gt(lm,lm1),and(eq(lm,lm1),gt(ml,ml1)))

  mkml(LM0)=ML(LM0)
  mkml(seq1(d,lm))=ML(seq1(d,lm))
  mkml(seqM(ml,lm))=if(eq(lm,LM0),ml,ML(seqM(ml,lm)))

  comp(ML(LM0),ml)=ml
  comp(ml,ML(LM0))=ml

  comp(ML(seq1(d,lm)),ML(seq1(d1,lm1)))=
        if(gt(seq1(d,lm),seq1(d1,lm1)),
          par(seq1(d1,lm1),ML(seq1(d,lm))),
          par(seq1(d,lm),ML(seq1(d1,lm1))))
  comp(ML(seq1(d,lm)),ML(seqM(ml,lm1)))=par(seq1(d,lm1),ML(seqM(ml,lm1)))
  comp(ML(seqM(ml,lm)),ML(seq1(d,lm1)))=comp(ML(seq1(d,lm1)),ML(seqM(ml,lm)))
  comp(ML(seqM(ml,lm)),ML(seqM(ml1,lm1)))=
        if(gt(seqM(ml,lm),seqM(ml1,lm1)),
          par(seqM(ml1,lm1),ML(seqM(ml,lm))),
          par(seqM(ml,lm),ML(seqM(ml1,lm1))))

  comp(ML(seq1(d,lm)),par(lm1,ml))=
        if(gt(seq1(d,lm),lm1),
          par(lm1,comp(ML(seq1(d,lm)),ml)),
          par(seq1(d,lm),par(lm1,ml)))
  comp(par(lm1,ml),ML(seq1(d,lm)))=comp(ML(seq1(d,lm)),par(lm1,ml))
  comp(ML(seqM(ml,lm)),par(lm1,ml1))=
        if(gt(seqM(ml,lm),lm1),
          par(lm1,comp(ML(seqM(ml,lm)),ml1)),
          par(seqM(ml,lm),par(lm1,ml1)))
  comp(par(lm1,ml1),ML(seqM(ml,lm)))=comp(ML(seqM(ml,lm)),par(lm1,ml1))
  comp(par(lm,ml),par(lm1,ml1))=
        if(gt(lm,lm1),
          par(lm1,comp(ml1,par(lm,ml))),
          par(lm,comp(ml,par(lm1,ml1))))

  eq(ML(lm),par(lm1,ml))=F eq(par(lm1,ml),ML(lm))=F
  eq(ML(lm1),ML(lm2))=eq(lm1,lm2)
  eq(par(lm,ml),par(lm1,ml1))=                  % ML par(lm1,ml1) has at least 2 elements
        and(in(lm,par(lm1,ml1)),eq(ml,rem(lm,par(lm1,ml1))))
  eq(ml,ml)=T

  if(T,ml,ml1)=ml if(F,ml,ml1)=ml1 if(b,ml,ml)=ml if(not(b),ml,ml1)=if(b,ml1,ml)

  in(lm,ML(lm1))=eq(lm,lm1)
  in(lm,par(lm1,ml))=or(eq(lm,lm1),in(lm,ml))

  % undefined (not needed) rem(lm,ML(lm1))=if(eq(lm,lm1),ML(LM0),ML(lm1))
  rem(lm,par(lm1,ML(lm2)))=if(eq(lm,lm1),ML(lm2),if(eq(lm,lm2),ML(lm1),par(lm1,ML(lm2))))
  rem(lm,par(lm1,par(lm2,ml)))=if(eq(lm,lm1),par(lm2,ml),par(lm1,rem(lm,par(lm2,ml))))

  lenf(ML(lm))=lenf(lm)
  lenf(par(lm,ml))=add(lenf(lm),lenf(ml))

  getf1(ML(lm),n)=getf1(lm,n)
  getf1(par(lm,ml),n)=if(gt(lenf(lm),n),getf1(lm,n),getf1(ml,sub(n,lenf(lm))))

  replf1(ML(lm),n,lm1)=mkml(replf1(lm,n,lm1))
  replf1(par(lm,ml),n,lm1)=if(gt(lenf(lm),n),
                             comp(mkml(replf1(lm,n,lm1)),ml),
                             comp(ML(lm),replf1(ml,sub(n,lenf(lm)),lm1)))

  replf2(ML(lm),n,m,lm1,lm2)=mkml(replf2(lm,n,m,lm1,lm2))
  replf2(par(lm,ml),n,m,lm1,lm2)=
        if(gt(m,n),
          if(gt(lenf(lm),n),
            if(gt(lenf(lm),m),
              comp(mkml(replf2(lm,n,m,lm1,lm2)),ml),
              comp(mkml(replf1(lm,n,lm1)),replf1(ml,sub(m,lenf(lm)),lm2))),
            comp(ML(lm),replf2(ml,sub(n,lenf(lm)),sub(m,lenf(lm)),lm1,lm2))),
          if(gt(lenf(lm),m),
            if(gt(lenf(lm),n),
              comp(mkml(replf2(lm,n,m,lm1,lm2)),ml),
              comp(mkml(replf1(lm,m,lm2)),replf1(ml,sub(n,lenf(lm)),lm1))),
            comp(ML(lm),replf2(ml,sub(n,lenf(lm)),sub(m,lenf(lm)),lm1,lm2))))
```

C.3 *ALM* and *AML* Data Types

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  ALM And AML data types                                     %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  sort  Act (Actions)                                        %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Act
func
  a:Nat->Act
map
  eq:Act#Act->Bool
  if:Bool#Act#Act->Act
  gt:Act#Act->Bool
var a,a1:Act n,m:Nat b:Bool
rew
  eq(a(n),a(m))=eq(n,m)
  if(T,a,a1)=a if(F,a,a1)=a1 if(b,a,a)=a if(not(b),a,a1)=if(b,a1,a)
  gt(a(n),a(m))=gt(n,m)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  sort  ActSet (Sets of action Actions)                      %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort ActSet
func
  ActSet0:->ActSet
  _add:Act#ActSet->ActSet
map
  eq:ActSet#ActSet->Bool
  gt:ActSet#ActSet->Bool
  if:Bool#ActSet#ActSet->ActSet
  add:Act#ActSet->ActSet                % add an element
  add1:Act#ActSet->ActSet               % add an element assuming it is not in the set
  in:Act#ActSet->Bool                   % is an element in the set?
  rem:Act#ActSet->ActSet                % remove an element (if present)
  union:ActSet#ActSet->ActSet           % set union
  minus:ActSet#ActSet->ActSet           % set minus
  intersect:ActSet#ActSet->ActSet       % set intersection
var a,a1:Act as,as1:ActSet b:Bool
rew
  eq(as,as)=T eq(ActSet0,_add(a,as))=F eq(_add(a,as),ActSet0)=F
  eq(_add(a,as),_add(a1,as1))=and(in(a,_add(a1,as1)),eq(as,rem(a,_add(a1,as1))))

  gt(ActSet0,as)=F
  gt(_add(a,as),ActSet0)=T
  gt(_add(a,as),_add(a1,as1))=or(gt(a,a1),and(eq(a,a1),gt(as,as1)))

  if(T,as,as1)=as if(F,as,as1)=as1 if(b,as,as)=as if(not(b),as,as1)=if(b,as1,as)

  add(a,as)=if(in(a,as),as,add1(a,as))
  add1(a,ActSet0)=_add(a,ActSet0)
  add1(a,_add(a1,as))=if(gt(a,a1),_add(a1,add1(a,as)),_add(a,add(a1,as)))

  in(a,ActSet0)=F in(a,_add(a1,as))=or(eq(a,a1),in(a,as))

  rem(a,ActSet0)=ActSet0
  rem(a,_add(a1,as))=if(eq(a,a1),as,_add(a1,rem(a,as)))

  union(ActSet0,as)=as union(as,ActSet0)=as
  union(_add(a,as),as1)=union(as,add(a,as1))

  minus(ActSet0,as)=ActSet0 minus(as,ActSet0)=as
  minus(_add(a,as),as1)=if(in(a,as1),minus(as,as1),_add(a,minus(as,as1)))

  intersect(ActSet0,as)=ActSet0 intersect(as,ActSet0)=ActSet0
  intersect(_add(a,as),as1)=if(in(a,as1),_add(a,intersect(as,as1)),intersect(as,as1))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  sort  ActMap (Function from Act to Act)                    %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort ActMap
func
  ActMap0:->ActMap
  _add:Act#Act#ActMap->ActMap
map
  eq:ActMap#ActMap->Bool
  gt:ActMap#ActMap->Bool
  if:Bool#ActMap#ActMap->ActMap
  mod:Act#Act#ActMap->ActMap            % modify the mapping with the pair
  mod0:Act#Act#ActMap->ActMap           % modify the mapping with the pair assuming the arg is there
  mod1:Act#Act#ActMap->ActMap           % modify the mapping with the pair assuming the arg is not there
  in:Act#ActMap->Bool                   % is an element in the map's args?
  in:Act#Act#ActMap->Bool               % is a pair in the map?
  rem:Act#ActMap->ActMap                % remove a pair by the arg (if present)
  appl:Act#ActMap->Act                  % apply the mapping
  comp:ActMap#ActMap->ActMap            % compose 2 maps
  rimage:ActSet#ActMap->ActSet          % F^{-1}(AS)
  simpl:ActSet#ActMap->ActMap           % transform am not to change as
var a,a1,a2,a3:Act as:ActSet am,am1:ActMap b:Bool
rew
  eq(am,am)=T eq(ActMap0,_add(a,a1,am))=F eq(_add(a,a1,am),ActMap0)=F
  eq(_add(a,a1,am),_add(a2,a3,am1))=and(in(a,a1,_add(a1,a2,am1)),eq(am,rem(a,_add(a2,a3,am1))))

  gt(ActMap0,am)=F
  gt(_add(a,a1,am),ActMap0)=T
  gt(_add(a,a1,am),_add(a2,a3,am1))=or(gt(a2,a),and(eq(a2,a),or(gt(a1,a3),and(eq(a1,a3),gt(am,am1)))))

  if(T,am,am1)=am if(F,am,am1)=am1 if(b,am,am)=am if(not(b),am,am1)=if(b,am1,am)

  mod(a,a1,am)=if(in(a,am),mod0(a,a1,am),mod1(a,a1,am))

  mod0(a,a1,_add(a2,a3,am))=if(eq(a,a2),_add(a2,a1,am),_add(a2,a3,mod0(a,a1,am)))
  mod1(a,a1,ActMap0)=_add(a,a1,ActMap0)
  mod1(a,a1,_add(a2,a3,am))=if(gt(a,a2),_add(a2,a3,mod1(a,a1,am)),_add(a,a1,_add(a2,a3,am)))

  in(a,ActMap0)=F in(a,_add(a2,a3,am))=or(eq(a,a2),in(a,am))
  in(a,a1,ActMap0)=F in(a,a1,_add(a2,a3,am))=or(and(eq(a,a2),eq(a1,a3)),in(a,a1,am))

  rem(a,ActMap0)=ActMap0
  rem(a,_add(a2,a3,am))=if(eq(a,a2),am,_add(a2,a3,rem(a,am)))

  appl(a,ActMap0)=a appl(a,_add(a2,a3,am))=if(eq(a,a2),a3,appl(a,am))

  comp(ActMap0,am)=am comp(am,ActMap0)=am
  comp(_add(a,a1,am),am1)=if(eq(appl(a1,am1),a),rem(a,comp(am,am1)),mod1(a,appl(a1,am1),rem(a,comp(am,am1))))

  rimage(ActSet0,am)=ActSet0 rimage(as,ActMap0)=as
  rimage(as,_add(a,a1,am))=if(in(a1,as),add(a,rimage(as,am)),rem(a,rimage(as,am)))

  simpl(ActSet0,am)=am simpl(as,ActMap0)=ActMap0
  simpl(as,_add(a,a1,am))=if(in(a,as),simpl(as,am),_add(a,a1,simpl(as,am)))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  sort  Annote (Triple of one ActMap and two ActSets)        %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Annote
func
  ann:ActMap#ActSet#ActSet->Annote
map
  eq:Annote#Annote->Bool
  gt:Annote#Annote->Bool
  if:Bool#Annote#Annote->Annote
  Ann0:->Annote
  comp:Annote#Annote->Annote
  getH:Annote->ActSet
  getI:Annote->ActSet
  getR:Annote->ActMap
var as,as1,as2,as3:ActSet am,am1:ActMap ann1,ann2:Annote b:Bool
rew
  eq(ann(am,as,as1),ann(am1,as2,as3))=and(and(eq(am,am1),eq(as,as2)),eq(as1,as3))

  gt(ann(am,as,as1),ann(am1,as2,as3))=or(gt(as1,as3),and(eq(as1,as3),or(gt(as,as2),and(eq(as,as2),gt(am,am1)))))

  if(T,ann1,ann2)=ann1 if(F,ann1,ann2)=ann2 if(b,ann1,ann1)=ann1 if(not(b),ann1,ann2)=if(b,ann2,ann1)

  Ann0=ann(ActMap0,ActSet0,ActSet0)

  comp(ann(am,as,as1),ann(am1,as2,as3))=
        ann(comp(am1,am),union(as2,rimage(as,am1)),union(as3,minus(rimage(as1,am1),as2)))

  getH(ann(am,as,as1))=as1 getI(ann(am,as,as1))=as getR(ann(am,as,as1))=am

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  sort  ALM                                                  %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort ALM                                % List of AML or State elements
func
  ALM0: ->ALM                           % Empty list
  seq1: Annote#State#ALM->ALM           % Add one State element to the head of the list
  seqM: AML#ALM->ALM                    % Add one AML to the head of the list (first argument never AML(x))
map
  eq: ALM#ALM->Bool                     % Equality on ALM
  if: Bool#ALM#ALM->ALM

  gt: ALM#ALM->Bool
  conc: ALM#ALM->ALM                    % Concatenate 2 ALMs in a wf way.
  conp: AML#ALM->ALM                    % Prepend an AML to an ALM
  annote: Annote#ALM->ALM               % add annotation

  lenf:ALM->Nat                         % number of "ready" components
  getf1d:ALM#Nat->State                 % get n-th component
  getf1a:ALM#Nat->Annote                % get n-th component's annotation
  replf1:ALM#Nat#ALM->ALM               % replace n-th component
  remf1:ALM#Nat->ALM                    % remove n-th component

  getf2a0:ALM#Nat#Nat->Annote           % get n-th component's annotation
  getf2a1:ALM#Nat#Nat->Annote           % get m-th component's annotation
  getf2a:ALM#Nat#Nat->Annote            % get (n,m)-th components' annotation

  replf2:ALM#Nat#Nat#ALM#ALM->ALM       % replace n-th and m-th components
  replremf2:ALM#Nat#Nat#ALM->ALM        % replace n-th and remove m-th components
  remf2:ALM#Nat#Nat->ALM                % remove n-th and m-th components

  parc: Annote#ALM#ALM->ALM             % Compose 2 ALMs parallely
  seqc: ALM#ALM->ALM                    % Compose 2 ALMs sequentially
```

```
var
  d,d1: State lm,lm1,lm2:ALM ml,ml1:AML n,m:Nat b:Bool ann,ann1:Annote a:Act
rew
  gt(ALM0,lm)=F
  gt(seq1(ann,d,lm),ALM0)=T
  gt(seq1(ann,d,lm),seq1(ann1,d1,lm1))
        =if(eq(eq(lm,ALM0),eq(lm1,ALM0)),
          if(eq(eq(ann,Ann0),eq(ann1,Ann0)),
            or(gt(d,d1),and(eq(d,d1),or(gt(lm,lm1),and(eq(lm,lm1),gt(ann,ann1))))),
            eq(ann1,Ann0)),
          eq(lm1,ALM0))
  gt(seq1(ann,d,lm),seqM(ml,lm1))=F
  gt(seqM(ml,lm),ALM0)=T
  gt(seqM(ml,lm),seq1(ann,d,lm1))=T
  gt(seqM(ml,lm),seqM(ml1,lm1))
        =if(eq(eq(lm,ALM0),eq(lm1,ALM0)),
          or(gt(ml,ml1),and(eq(ml,ml1),gt(lm,lm1))),
          eq(lm1,ALM0))

  conc(ALM0,lm)=lm conc(lm,ALM0)=lm
  conc(seq1(ann,d,lm),lm1)=seq1(ann,d,conc(lm,lm1))
  conc(seqM(ml,lm),lm1)=seqM(ml,conc(lm,lm1))

  conp(AML(lm),lm1)=conc(lm,lm1)
  conp(par(ann,lm,ml),lm1)=seqM(par(ann,lm,ml),lm1)

  annote(ann,ALM0)=ALM0 annote(Ann0,lm)=lm
  annote(ann,seq1(ann1,d,lm))=seq1(comp(ann,ann1),d,annote(ann,lm))
  annote(ann,seqM(ml,lm))=conp(annote(ann,ml),annote(ann,lm))

  eq(ALM0,seq1(ann,d,lm))=F eq(seq1(ann,d,lm),ALM0)=F
  eq(ALM0,seqM(ml,lm))=F eq(seqM(ml,lm),ALM0)=F
  eq(seq1(ann,d,lm),seqM(ml,lm1))=F eq(seqM(ml,lm1),seq1(ann,d,lm))=F

  eq(lm,lm)=T
  eq(seq1(ann,d,lm),seq1(ann1,d1,lm1))=and(and(eq(d,d1),eq(lm,lm1)),eq(ann,ann1))
  eq(seqM(ml,lm),seqM(ml1,lm1))=and(eq(ml,ml1),eq(lm,lm1))

  if(T,lm,lm1)=lm if(F,lm,lm1)=lm1 if(b,lm,lm)=lm if(not(b),lm,lm1)=if(b,lm1,lm)

  lenf(ALM0)=0
  lenf(seq1(ann,d,lm))=1
  lenf(seqM(ml,lm))=lenf(ml)

  % undefined getf1(ALM0,n)=
  getf1d(seq1(ann,d,lm),0)=d
  getf1d(seqM(ml,lm),n)=getf1d(ml,n)
  getf1a(seq1(ann,d,lm),0)=ann
  getf1a(seqM(ml,lm),n)=getf1a(ml,n)

  replf1(seq1(ann,d,lm),0,lm1)=conc(annote(ann,lm1),lm)
  replf1(seqM(ml,lm),n,lm1)=conp(replf1(ml,n,lm1),lm)

  remf1(lm,n)=replf1(lm,n,ALM0)

  getf2a0(seqM(ml,lm),n,m)=getf2a0(ml,n,m)
  getf2a1(seqM(ml,lm),n,m)=getf2a1(ml,n,m)
  getf2a(seqM(ml,lm),n,m)=getf2a(ml,n,m)

  replf2(seqM(ml,lm),n,m,lm1,lm2)=conp(replf2(ml,n,m,lm1,lm2),lm)
  replremf2(lm,n,m,lm1)=replf2(lm,n,m,lm1,ALM0)
  remf2(lm,n,m)=replf2(lm,n,m,ALM0,ALM0)

  seqc(lm,lm1)=conc(lm,lm1)
  parc(ann,lm,lm1)=annote(ann,conp(comp(mkml(lm),mkml(lm1)),ALM0))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  sort AML                                                    %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort AML                                % Multiset of ALM
func
  AML: ALM->AML                         % Multiset with one list
  par: Annote#ALM#AML->AML              % Add a list to the multiset (first argument never ALM0)
map
  eq: AML#AML->Bool                     % equality on AML
  if:Bool#AML#AML->AML

  mkml :ALM->AML                        % Make a proper AML out of an ALM
  comp :AML#AML->AML                    % Compose 2 AMLs in a wf way.
  annote: Annote#AML->AML               % add annotation
  gt   :AML#AML->Bool

  in: ALM#AML->Bool                     % test if an lm is in ml (on the first level, of course).
  rem: ALM#AML->AML                     % remove an lm from ml if it is on the first level, don't change otherwise

  lenf: AML->Nat
  getf1d: AML#Nat->State
  getf1a: AML#Nat->Annote
  replf1: AML#Nat#ALM->AML

  getf2a0:AML#Nat#Nat->Annote           % get n-th component's annotation
  getf2a1:AML#Nat#Nat->Annote           % get m-th component's annotation
  getf2a:AML#Nat#Nat->Annote            % get (n,m)-th components' annotation
  replf2:AML#Nat#Nat#ALM#ALM->AML       % replace n-th and m-th components
var
  d,d1: State lm,lm1,lm2:ALM ml,ml1:AML n,m:Nat b:Bool ann,ann1,ann2:Annote a:Act
rew
  gt(AML(lm),AML(lm1))=gt(lm,lm1)
  gt(AML(lm),par(ann,lm1,ml))=F
  gt(par(ann,lm1,ml),AML(lm))=T
  gt(par(ann,lm,ml),par(ann1,lm1,ml1))
        =if(eq(eq(ann,Ann0),eq(ann1,Ann0)),
          or(gt(lm,lm1),and(eq(lm,lm1),gt(ml,ml1))),
          eq(ann1,Ann0))

  mkml(ALM0)=AML(ALM0)
  mkml(seq1(ann,d,lm))=AML(seq1(ann,d,lm))
  mkml(seqM(ml,lm))=if(eq(lm,ALM0),ml,AML(seqM(ml,lm)))

  comp(AML(ALM0),ml)=ml
  comp(ml,AML(ALM0))=ml

  comp(AML(seq1(ann,d,lm)),AML(seq1(ann1,d1,lm1)))=
        if(gt(seq1(ann,d,lm),seq1(ann1,d1,lm1)),
          par(Ann0,seq1(ann1,d1,lm1),AML(seq1(ann1,d,lm))),
          par(Ann0,seq1(ann,d,lm),AML(seq1(ann1,d1,lm1))))
  comp(AML(seq1(ann,d,lm)),AML(seqM(ml,lm1)))=par(Ann0,seq1(ann,d,lm1),AML(seqM(ml,lm1)))
  comp(AML(seqM(ml,lm)),AML(seq1(ann,d,lm1)))=comp(AML(seq1(ann,d,lm1)),AML(seqM(ml,lm)))
  comp(AML(seqM(ml,lm)),AML(seqM(ml1,lm1)))=
        if(gt(seqM(ml,lm),seqM(ml1,lm1)),
          par(Ann0,seqM(ml1,lm1),AML(seqM(ml,lm))),
          par(Ann0,seqM(ml,lm),AML(seqM(ml1,lm1))))

  comp(AML(seq1(ann,d,lm)),par(ann1,lm1,ml))=
        if(and(eq(ann1,Ann0),gt(seq1(ann,d,lm),lm1)),
          par(Ann0,lm1,comp(AML(seq1(ann,d,lm)),ml)),
          par(Ann0,seq1(ann,d,lm),par(ann1,lm1,ml)))
  comp(par(ann1,lm1,ml),AML(seq1(ann,d,lm)))=comp(AML(seq1(ann,d,lm)),par(ann1,lm1,ml))
  comp(AML(seqM(ml,lm)),par(ann1,lm1,ml1))=
        if(and(eq(ann1,Ann0),gt(seqM(ml,lm),lm1)),
          par(Ann0,lm1,comp(AML(seqM(ml,lm)),ml1)),
          par(Ann0,seqM(ml,lm),par(ann1,lm1,ml1)))
  comp(par(ann1,lm1,ml1),AML(seqM(ml,lm)))=comp(AML(seqM(ml,lm)),par(ann1,lm1,ml1))
  comp(par(ann,lm,ml),par(ann1,lm1,ml1))=
        if(eq(ann,Ann0),
          if(eq(ann1,Ann0),
            if(gt(lm,lm1),par(Ann0,lm1,comp(ml1,par(ann,lm,ml))),par(Ann0,lm,comp(ml,par(ann1,lm1,ml1)))),
            par(Ann0,lm,comp(ml,par(ann1,lm1,ml1)))),
          if(eq(ann1,Ann0),
            par(Ann0,lm1,comp(ml1,par(ann,lm,ml))),
            if(gt(par(ann,lm,ml),par(ann1,lm1,ml1)),
              par(Ann0,seqM(par(ann1,lm1,ml1),ALM0),par(ann,lm,ml)),
              par(Ann0,seqM(par(ann,lm,ml),ALM0),par(ann1,lm1,ml1)))))

  annote(ann,AML(lm))=mkml(annote(ann,lm))
  annote(ann,par(ann1,lm,ml))=par(comp(ann,ann1),lm,ml)

  eq(AML(lm),par(ann1,lm1,ml))=F eq(par(ann1,lm1,ml),AML(lm))=F
  eq(AML(lm1),AML(lm2))=eq(lm1,lm2)
  eq(par(ann,lm,ml),par(ann1,lm1,ml1))=
        and(eq(ann,ann1),and(in(lm,par(Ann0,lm1,ml1)),eq(ml,rem(lm,par(Ann0,lm1,ml1)))))
                                                % AML par(Ann0,lm1,ml1) has at least 2 elements
```

```
eq(ml,ml)=T

if(T,ml,ml1)=ml if(F,ml,ml1)=ml1 if(b,ml,ml)=ml if(not(b),ml,ml1)=if(b,ml1,ml)

in(lm,AML(lm1))=eq(lm,lm1)
in(lm,par(ann1,lm1,ml))=and(eq(ann1,Ann0),or(eq(lm,lm1),in(lm,ml)))

% undefined (not needed) rem(lm,AML(lm1))=if(eq(lm,lm1),AML(ALM0),AML(lm1))
rem(lm,par(ann1,lm1,AML(lm2)))=
      if(eq(ann1,Ann0),
        if(eq(lm,lm1),AML(lm2),if(eq(lm,lm2),AML(lm1),par(ann1,lm1,AML(lm2)))),
        par(ann1,lm1,AML(lm2)))
rem(lm,par(ann1,lm1,par(ann2,lm2,ml)))=
      if(eq(ann1,Ann0),
        if(eq(lm,lm1),par(ann2,lm2,ml),par(ann1,lm1,rem(lm,par(ann2,lm2,ml)))),
        par(ann1,lm1,par(ann2,lm2,ml)))

lenf(AML(lm))=lenf(lm)
lenf(par(ann,lm,ml))=add(lenf(lm),lenf(ml))

getf1d(AML(lm),n)=getf1d(lm,n)
getf1d(par(ann,lm,ml),n)=if(gt(lenf(lm),n),getf1d(lm,n),getf1d(ml,sub(n,lenf(lm))))
getf1a(AML(lm),n)=getf1a(lm,n)
getf1a(par(ann,lm,ml),n)=comp(ann,if(gt(lenf(lm),n),getf1a(lm,n),getf1a(ml,sub(n,lenf(lm)))))

replf1(AML(lm),n,lm1)=mkml(replf1(lm,n,lm1))
replf1(par(ann,lm,ml),n,lm1)=annote(ann,if(gt(lenf(lm),n),
                            comp(mkml(replf1(lm,n,lm1)),ml),
                            comp(AML(lm),replf1(ml,sub(n,lenf(lm)),lm1))))

getf2a0(AML(lm),n,m)=getf2a0(lm,n,m)
getf2a0(par(ann,lm,ml),n,m)=
      if(eq(gt(lenf(lm),n),gt(lenf(lm),m)),
        if(gt(lenf(lm),n),getf2a0(lm,n,m),getf2a0(ml,sub(n,lenf(lm)),sub(m,lenf(lm)))),
        if(gt(lenf(lm),n),getf1a(lm,n),getf1a(ml,sub(n,lenf(lm)))))
getf2a1(AML(lm),n,m)=getf2a1(lm,n,m)
getf2a1(par(ann,lm,ml),n,m)=
      if(eq(gt(lenf(lm),n),gt(lenf(lm),m)),
        if(gt(lenf(lm),n),getf2a1(lm,n,m),getf2a1(ml,sub(n,lenf(lm)),sub(m,lenf(lm)))),
        if(gt(lenf(lm),m),getf1a(lm,n),getf1a(ml,sub(m,lenf(lm)))))
getf2a(AML(lm),n,m)=getf2a(lm,n,m)
getf2a(par(ann,lm,ml),n,m)=
      if(eq(gt(lenf(lm),n),gt(lenf(lm),m)),
        comp(ann,if(gt(lenf(lm),n),getf2a(lm,n,m),getf2a(ml,sub(n,lenf(lm)),sub(m,lenf(lm))))),
        ann)

replf2(AML(lm),n,m,lm1,lm2)=mkml(replf2(lm,n,m,lm1,lm2))
replf2(par(ann,lm,ml),n,m,lm1,lm2)=annote(ann,
      if(gt(m,n),
        if(gt(lenf(lm),n),
          if(gt(lenf(lm),m),
            comp(mkml(replf2(lm,n,m,lm1,lm2)),ml),
            comp(mkml(replf1(lm,n,lm1)),replf1(ml,sub(m,lenf(lm)),lm2))),
          comp(AML(lm),replf2(ml,sub(n,lenf(lm)),sub(m,lenf(lm)),lm1,lm2))),
        if(gt(lenf(lm),m),
          if(gt(lenf(lm),n),
            comp(mkml(replf2(lm,n,m,lm1,lm2)),ml),
            comp(mkml(replf1(lm,m,lm2)),replf1(ml,sub(n,lenf(lm)),lm1))),
          comp(AML(lm),replf2(ml,sub(n,lenf(lm)),sub(m,lenf(lm)),lm1,lm2)))))
```

C.4 Basic Data Types for Multi-Party Communications

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%  Sorts  LNat, LState; ActPars, E_i, LActPars, LE_i, functions on them (all generated)  %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  sort  LNat (list of Naturals)                              %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort LNat
func
  LNat0:->LNat
  add:Nat#LNat->LNat
map
```

```
  eq:LNat#LNat->Bool
  if:Bool#LNat#LNat->LNat
  len:LNat->Nat
  cat:LNat#LNat->LNat
  head:LNat->Nat                % return the head of the list
  in:Nat#LNat->Bool
  lower:LNat#Nat->LNat          % return a sublist containing elems <n
  upper:LNat#Nat->LNat          % return a sublist containing elems >=n
  sub:LNat#Nat->LNat            % subtract n from each elem.
  is_unique:LNat->Bool          % are all the elems different?
  is_sorted:LNat->Bool          % is the list sorted?
  is_each_lower:Nat#LNat->Bool  % is each of the elems lower than the first arg?
  gen0Mm1:Nat->LNat             % generate list 0..M-1 (if M=0 then return LNat0)
var
  lnat,lnat1:LNat n,m:Nat b:Bool
rew
  eq(lnat,lnat)=T eq(LNat0,add(n,lnat))=F
  eq(add(n,lnat),LNat0)=F eq(add(n,lnat),add(m,lnat1))=and(eq(n,m),eq(lnat,lnat1))
  if(T,lnat,lnat1)=lnat if(F,lnat,lnat1)=lnat1
  if(b,lnat,lnat)=lnat if(not(b),lnat,lnat1)=if(b,lnat1,lnat)
  len(LNat0)=0
  len(add(n,lnat))=succ(len(lnat))
  cat(LNat0,lnat)=lnat cat(lnat,LNat0)=lnat
  cat(add(n,lnat),lnat1)=add(n,cat(lnat,lnat1))
  head(add(n,lnat))=n
  in(n,LNat0)=F
  in(n,add(m,lnat))=or(eq(n,m),in(n,lnat))
  lower(LNat0,n)=LNat0
  lower(add(m,lnat),n)=if(gt(n,m),add(m,lower(lnat,n)),lower(lnat,n))
  upper(LNat0,n)=LNat0
  upper(add(m,lnat),n)=if(gt(n,m),upper(lnat,n),add(m,upper(lnat,n)))
  sub(LNat0,n)=LNat0
  sub(add(m,lnat),n)=add(sub(m,n),sub(lnat,n))
  is_unique(LNat0)=T
  is_unique(add(n,lnat))=and(in(n,lnat),is_unique(lnat))
  is_sorted(LNat0)=T is_sorted(add(n,LNat0))=T
  is_sorted(add(n,add(m,lnat)))=and(not(gt(n,m)),is_sorted(add(m,lnat)))
  is_each_lower(n,LNat0)=T
  is_each_lower(n,add(m,lnat))=and(gt(n,m),is_each_lower(n,lnat))
  gen0Mm1(0)=LNat0
  gen0Mm1(x2p1(n))=cat(gen0Nm1(x2p0(n)),add(n,LNat0))
  gen0Mm1(x2p2(n))=cat(gen0Nm1(x2p1(n)),add(n,LNat0))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  sort  LState  (list of States)                              %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort LState
func
  LState0:->LState
  add:State#LState->LState
map
  eq:LState#LState->Bool
  if:Bool#LState#LState->LState
  len:LState->Nat
var
  ld,ld1:LState d,d1:State b:Bool
rew
  eq(ld,ld)=T eq(LState0,add(d,ld))=F
  eq(add(d,ld),LState0)=F eq(add(d,ld),add(d1,ld1))=and(eq(d,d1),eq(ld,ld1))
  if(T,ld,ld1)=ld if(F,ld,ld1)=ld1 if(b,ld,ld)=ld if(not(b),ld,ld1)=if(b,ld1,ld)
  len(LState0)=0
  len(add(d,ld))=succ(len(ld))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Sorts LActPars (list of ActPars, defined below)            %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort LActPars
func
  LActPars0:->LActPars
  add:ActPars#LActPars->LActPars
map
  eq:LActPars#LActPars->Bool
  if:Bool#LActPars#LActPars->LActPars
  len:LActPars->Nat
  head:LActPars->ActPars        % return the head of the list
  EQ:LActPars->Bool             % are all of the elements equal?
```

```
var
  laa,laa1:LActPars aa,aa1:ActPars b:Bool
rew
  eq(laa,laa)=T eq(LActPars0,add(aa,laa))=F
  eq(add(aa,laa),LActPars0)=F eq(add(aa,laa),add(aa1,laa1))=and(eq(aa,aa1),eq(laa,laa1))
  if(T,laa,laa1)=laa if(F,laa,laa1)=laa1 if(b,laa,laa)=laa if(not(b),laa,laa1)=if(b,laa1,laa)
  len(LActPars0)=0
  len(add(aa,laa))=succ(len(laa))
  head(add(aa,laa))=aa
  EQ(LActPars0)=T
  EQ(add(aa,LActPars0))=T
  EQ(add(aa,add(aa1,laa)))=and(eq(aa,aa1),EQ(add(aa1,laa)))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%  To be generated from the spec                              %%%
%%%%  The parts that do not parse before actual generation       %%%
%%%%  are commented out                                          %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Sort ActPars (unique action parameters tuples)                     %%%
%%% if parameters of act(m) are a_k(...),                               %%%
%%% it means that act(m) and act(k) have the same parameter sorts       %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort ActPars
func
  % a_k:D_0#...#D_n->ActPars
map
  eq: ActPars#ActPars->Bool
  gt: ActPars#ActPars->Bool
  if: Bool#ActPars#ActPars->ActPars
  % pr_k_0:ActPars->D_0 ... pr_k_n:ActPars->D_n
var
  aa,aa1:ActPars b:Bool
rew
  if(T,aa,aa1)=aa if(F,aa,aa1)=aa1 if(b,aa,aa)=aa if(not(b),aa,aa1)=if(b,aa1,aa)
  % gt(a_k(d0,...,dn),a_k(e0,...,en))=
  %     or(gt(d0,e0),and(eq(d0,e0),...or(gt(d{n-1},e{n-1}),and(eq(d{n-1},e{n-1}),gt(dn,en)))...))
  % gt(a_k(d0,...,dn),a_m(e0,...,el))="k>m"
  eq(aa,aa)=T
  % eq(a_k(d0,...,dn),a_k(e0,...,en))=and(eq(d0,e0),...,and(eq(dn,en))...)
  % eq(a_k(d0,...,dn),a_m(e0,...,el))=F    (k!=m)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Sorts E_i (sum types tuples)                              %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort E_0
func
  % e_i:D_0#...#D_n->E_i
map
  eq: E_0#E_0->Bool
  gt: E_0#E_0->Bool
  if: Bool#E_0#E_0->E_0
  % pr_0:E_i->D_0 ... pr_n:E_i->D_n
var
  ee,ee1:E_0 b:Bool
rew
  if(T,ee,ee1)=ee if(F,ee,ee1)=ee1 if(b,ee,ee)=ee if(not(b),ee,ee1)=if(b,ee1,ee)
  % gt(e_i(d0,...,dn),e_i(e0,...,en))=
  %     or(gt(d0,e0),and(eq(d0,e0),...or(gt(d{n-1},e{n-1}),and(eq(d{n-1},e{n-1}),gt(dn,en)))...))
  eq(ee,ee)=T
  % eq(e_i(d0,...,dn),e_i(e0,...,en))=and(eq(d0,e0),...,and(eq(dn,en))...)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Sorts LE_i (list of E_i)                                   %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort LE_0
func
  LE0_0:->LE_0
  add:E_0#LE_0->LE_0
map
  eq:LE_0#LE_0->Bool
  if:Bool#LE_0#LE_0->LE_0
  len:LE_0->Nat
  head:LE_0->E_0
var
```

```
  lee,lee1:LE_0 ee,ee1:E_0 b:Bool
rew
  eq(lee,lee)=T eq(LE0_0,add(ee,lee))=F
  eq(add(ee,lee),LE0_0)=F eq(add(ee,lee),add(ee1,lee1))=and(eq(ee,ee1),eq(lee,lee1))
  if(T,lee,lee1)=lee if(F,lee,lee1)=lee1 if(b,lee,lee)=lee if(not(b),lee,lee1)=if(b,lee1,lee)
  len(LE0_0)=0
  len(add(ee,lee))=succ(len(lee))
  head(add(ee,lee))=ee

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Functions F_i and C_i (use the terms vectors f_i and c_i)  %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
map
  F_0:LState#LE_0->LActPars
  C_0:LState#LE_0->Bool
var
  d:State ld:LState e:E_0 le:LE_0
rew
  F_0(LState0,LE0_0)=LActPars0
  % F_i(add(d,ld),add(e,le))=add([meta(f_i)](pr_k(d),pr_k(e)),F_i(ld,le))
  C_0(LState0,LE0_0)=T
  % C_i(add(d,ld),add(e,le))=and([meta(c_i)](pr_k(d),pr_k(e)),C_i(ld,le))
```

C.5 Data Types for Multi-Party Communications with LM and ML

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  sort  LLM  (list of LMs)                                   %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort LLM
func
  LLM0:->LLM
  add:LM#LLM->LLM
map
  eq:LLM#LLM->Bool
  if:Bool#LLM#LLM->LLM
  len:LLM->Nat
  cat:LLM#LLM->LLM
  lower:LLM#LNat#Nat->LLM       % return a sublist containing elems whose places are <n
  upper:LLM#LNat#Nat->LLM       % return a sublist containing elems whose places are >=n
  LEmptyLM:Nat->LLM             % returns the list consisting of n LM0s.
var
  llm,llm1:LLM lnat:LNat lm,lm1:LM b:Bool n,m:Nat
rew
  eq(llm,llm)=T eq(LLM0,add(lm,llm))=F
  eq(add(lm,llm),LLM0)=F eq(add(lm,llm),add(lm1,llm1))=and(eq(lm,lm1),eq(llm,llm1))
  if(T,llm,llm1)=llm if(F,llm,llm1)=llm1 if(b,llm,llm)=llm if(not(b),llm,llm1)=if(b,llm1,llm)
  len(LLM0)=0
  len(add(lm,llm))=succ(len(llm))
  cat(LLM0,llm)=llm cat(llm,LLM0)=llm
  cat(add(lm,llm),llm1)=add(lm,cat(llm,llm1))
  lower(LLM0,LNat0,n)=LLM0
  lower(add(lm,llm),add(m,lnat),n)=if(gt(n,m),add(lm,lower(llm,lnat,n)),lower(llm,lnat,n))
  upper(LLM0,LNat0,n)=LLM0
  upper(add(lm,llm),add(m,lnat),n)=if(gt(n,m),upper(llm,lnat,n),add(lm,upper(llm,lnat,n)))
  LEmptyLM(0)=LLM0
  LEmptyLM(x2p1(n))=add(LM0,cat(LEmptyLM(n),LEmptyLM(n)))
  LEmptyLM(x2p2(n))=add(LM0,LEmptyLM(x2p1(n)))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Additional parts for the sorts LM and ML                    %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
map
  getfn:LM#LNat->LState         % get the list of states.
  replfn:LM#LNat#LLM->LM        % replace the components with indices from LNat with the elements of LLM
  replfn:ML#LNat#LLM->ML
  remfn:LM#LNat->LM             % remove the components with indices from LNat
var
  llm:LLM lnat:LNat lm,lm1:LM ml:ML n:Nat
rew
  getfn(lm,LNat0)=LState0
  getfn(lm,add(n,lnat))=add(getf1(lm,n),getfn(lm,lnat))

  replfn(lm,add(n,LNat0),add(lm1,LLM0))=replf1(lm,n,lm1)
  replfn(seqM(ml,lm),add(n,lnat),add(lm1,llm))=conp(replfn(ml,add(n,lnat),add(lm1,llm)),lm)

  replfn(ML(lm),lnat,llm)=mkml(replfn(lm,lnat,llm))
  replfn(par(lm,ml),lnat,llm)=
    comp(if(eq(lower(lnat,lenf(lm)),LNat0),
           ML(LM0),
           mkml(replfn(lm,lower(lnat,lenf(lm)),lower(llm,lnat,lenf(lm))))),
         if(eq(upper(lnat,lenf(lm)),LNat0),
           ML(LM0),
           replfn(ml,sub(upper(lnat,lenf(lm)),lenf(lm)),upper(llm,lnat,lenf(lm)))))

  remfn(lm,lnat)=replfn(lm,lnat,LEmptyLM(len(lnat)))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%  To be generated from the spec                             %%%
%%%%  The parts that do not parse before actual generation      %%%
%%%%  are commented out                                         %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Functions mkllm_i                                          %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
map
  mkllm_0:LState#LE_0->LLM
var
  d:State ld:LState ee:E_0 lee:LE_0
rew
  mkllm_0(LState0,LE0_0)=add(LM0,LLM0)
  % mkllm_0(add(d,ld),add(ee,lee))=add([meta(mklm_0)](pr_k(d),pr_k(ee)),mkllm_0(ld,lee))

```

C.6 Data Types for Multi-Party Communications with ALM and AML

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  sort  LALM  (list of ALMs)                                 %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort LALM
func
  LALM0:->LALM
  add:ALM#LALM->LALM
map
  eq:LALM#LALM->Bool
  if:Bool#LALM#LALM->LALM
  len:LALM->Nat
  cat:LALM#LALM->LALM
  lower:LALM#LNat#Nat->LALM     % return a sublist containing elems whose places are <n
  upper:LALM#LNat#Nat->LALM     % return a sublist containing elems whose places are >=n
  LEmptyALM:Nat->LALM           % returns the list consisting of n ALM0s.
var
  llm,llm1:LALM lnat:LNat lm,lm1:ALM b:Bool n,m:Nat
rew
  eq(llm,llm)=T eq(LALM0,add(lm,llm))=F
  eq(add(lm,llm),LALM0)=F eq(add(lm,llm),add(lm1,llm1))=and(eq(lm,lm1),eq(llm,llm1))
  if(T,llm,llm1)=llm if(F,llm,llm1)=llm1 if(b,llm,llm)=llm if(not(b),llm,llm1)=if(b,llm1,llm)
  len(LALM0)=0
  len(add(lm,llm))=succ(len(llm))
  cat(LALM0,llm)=llm cat(llm,LALM0)=llm
  cat(add(lm,llm),llm1)=add(lm,cat(llm,llm1))
  lower(LALM0,LNat0,n)=LALM0
  lower(add(lm,llm),add(m,lnat),n)=if(gt(n,m),add(lm,lower(llm,lnat,n)),lower(llm,lnat,n))
  upper(LALM0,LNat0,n)=LALM0
  upper(add(lm,llm),add(m,lnat),n)=if(gt(n,m),upper(llm,lnat,n),add(lm,upper(llm,lnat,n)))
  LEmptyALM(0)=LALM0
  LEmptyALM(x2p1(n))=add(ALM0,cat(LEmptyALM(n),LEmptyALM(n)))
  LEmptyALM(x2p2(n))=add(ALM0,LEmptyALM(x2p1(n)))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  sort  LAct  (list of Actions)                              %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort LAct
func
  LAct0:->LAct
  add:Act#LAct->LAct
map
  eq:LAct#LAct->Bool
  if:Bool#LAct#LAct->LAct
  len:LAct->Nat
  cat:LAct#LAct->LAct
  lower:LAct#LNat#Nat->LAct     % return a sublist containing elems whose places are <n
  upper:LAct#LNat#Nat->LAct     % return a sublist containing elems whose places are >=n
  mklact:Nat#Act->LAct          % generate the list of n actions a
var
  la,la1:LAct lnat:LNat a,a1:Act b:Bool n,m:Nat
rew
  eq(la,la)=T eq(LAct0,add(a,la))=F
  eq(add(a,la),LAct0)=F eq(add(a,la),add(a1,la1))=and(eq(a,a1),eq(la,la1))
  if(T,la,la1)=la if(F,la,la1)=la1 if(b,la,la)=la if(not(b),la,la1)=if(b,la1,la)
  len(LAct0)=0
  len(add(a,la))=succ(len(la))
  cat(LAct0,la)=la cat(la,LAct0)=la
  cat(add(a,la),la1)=add(a,cat(la,la1))
  lower(LAct0,LNat0,n)=LAct0
  lower(add(a,la),add(m,lnat),n)=if(gt(n,m),add(a,lower(la,lnat,n)),lower(la,lnat,n))
  upper(LAct0,LNat0,n)=LAct0
  upper(add(a,la),add(m,lnat),n)=if(gt(n,m),upper(la,lnat,n),add(a,upper(la,lnat,n)))
  mklact(0,a)=LAct0
  mklact(x2p1(n),a)=add(a,cat(mklact(n,a),mklact(n,a)))
  mklact(x2p2(n),a)=add(a,mklact(x2p1(n),a))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Sort ActDT (action or delta or tau)                         %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort ActDT
func
  adt_a:Act->ActDT
  adt_d:->ActDT
  adt_t:->ActDT
map
  eq:ActDT#ActDT->Bool
  if:Bool#ActDT#ActDT->ActDT
  gamma: ActDT#ActDT->ActDT
  annote: Annote#ActDT->ActDT
var
  a,a1:Act adt,adt1:ActDT b:Bool ann:Annote
rew
  eq(adt,adt)=T eq(adt_a(a),adt_a(a1))=eq(a,a1)
  eq(adt_a(a),adt_d)=F eq(adt_a(a),adt_t)=F
  eq(adt_d,adt_a(a))=F eq(adt_d,adt_t)=F
  eq(adt_t,adt_a(a))=F eq(adt_t,adt_d)=F

  if(T,adt,adt1)=adt if(F,adt,adt1)=adt1 if(b,adt,adt)=adt if(not(b),adt,adt1)=if(b,adt1,adt)

  gamma(adt_a(a),adt_a(a1))=if(cannot_communicate(a,a1),adt_d,adt_a(gamma(a,a1)))
  gamma(adt_d,adt)=adt_d gamma(adt_t,adt)=adt_d gamma(adt,adt_d)=adt_d gamma(adt,adt_t)=adt_d

  annote(ann,adt_a(a))=if(in(a,getH(ann)),adt_d,if(in(a,getI(ann)),adt_t,adt_a(appl(a,getR(ann)))))
  annote(ann,adt_d)=adt_d annote(ann,adt_t)=adt_t

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Additional parts for the sorts ALM and AML                   %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
map
  getfnd:ALM#LNat->LState               % get the components with indices from LNat
  replfn:ALM#LNat#LALM->ALM             % replace the components with indices from LNat with the elements of LALM
  replfn:AML#LNat#LALM->AML
  remfn:ALM#LNat->ALM                   % remove the components with indices from LNat.
  getActDT:ALM#LNat#LAct->ActDT         % get the action a list of ready components performing the list of action
  getActDT:AML#LNat#LAct->ActDT         % will communicate into
  is_act:Act#ALM#LNat#LAct->Bool        % is this action a?
  is_tau:ALM#LNat#LAct->Bool            % is this tau?
var
  lm,lm1:ALM ml:AML lnat,lnat1:LNat llm:LALM ann:Annote a,a1:Act la,la1:LAct n,n1:Nat
rew
  getfnd(lm,LNat0)=LState0
  getfnd(lm,add(n,lnat))=add(getf1d(lm,n),getfnd(lm,lnat))

  replfn(lm,add(n,LNat0),add(lm1,LALM0))=replf1(lm,n,lm1)
  replfn(seqM(ml,lm),add(n,lnat),add(lm1,llm))=conp(replfn(ml,add(n,lnat),add(lm1,llm)),lm)

  replfn(AML(lm),lnat,llm)=mkml(replfn(lm,lnat,llm))
  replfn(par(ann,lm,ml),lnat,llm)=
    annote(ann,comp(if(eq(lower(lnat,lenf(lm)),LNat0),
                      AML(ALM0),
                      mkml(replfn(lm,lower(lnat,lenf(lm)),lower(llm,lnat,lenf(lm))))),
                    if(eq(upper(lnat,lenf(lm)),LNat0),
                      AML(ALM0),
                      replfn(ml,sub(upper(lnat,lenf(lm)),lenf(lm)),upper(llm,lnat,lenf(lm))))))

  remfn(lm,lnat)=replfn(lm,lnat,LEmptyALM(len(lnat)))

  getActDT(lm,add(n,LNat0),add(a,LAct0))=annote(getf1a(lm,n),adt_a(a))
  getActDT(seqM(ml,lm),add(n,add(n1,lnat1)),add(a,add(a1,la1)))=
        getActDT(ml,add(n,add(n1,lnat1)),add(a,add(a1,la1)))

  getActDT(AML(lm),lnat,la)=getActDT(lm,lnat,la)
  getActDT(par(ann,lm,ml),lnat,la)=
        annote(ann,
          if(eq(lower(lnat,lenf(lm)),LNat0),
            getActDT(ml,sub(lnat,lenf(lm)),la),
            if(eq(upper(lnat,lenf(lm)),LNat0),
              getActDT(lm,lnat,la),
              gamma(getActDT(lm,lower(lnat,lenf(lm)),lower(la,lnat,lenf(lm))),
                    getActDT(ml,sub(upper(lnat,lenf(lm)),lenf(lm)),upper(la,lnat,lenf(lm)))))))

  is_act(a,lm,lnat,la)=eq(adt_a(a),getActDT(lm,lnat,la))
  is_tau(lm,lnat,la)=eq(adt_t,getActDT(lm,lnat,la))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%  To be generated from the spec                              %%%
%%%%  The parts that do not parse before actual generation       %%%
%%%%  are commented out                                          %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Communication functions                                    %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
map
  cannot_communicate:Act#Act->Bool
  gamma:Act#Act->Act

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  Functions mkllm_i and f0                                   %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
map
  mkllm_0:LState#LE_0->LALM
  % f0:ALM#LNat#...#LNat#E_0#E_1#...#E_n->ActPars
var
  d:State ld:LState ee:E_0 lee:LE_0
rew
  mkllm_0(LState0,LE0_0)=add(ALM0,LALM0)
  % mkllm_0(add(d,ld),add(ee,lee))=add([meta(mklm_0)](pr_k(d),pr_k(ee)),mkllm_0(ld,lee))

  % f0(lm,ln_0,...,ln_n,e_0,...,e_n)=
  %     if(not(eq(ln_0,LNat0)),[meta]f_0(getf1(lm,head(ln_0)),e_0),
  %       if(not(eq(ln_1,LNat0)),[meta]f_1(getf1(lm,head(ln_1)),e_1),
  %         if(not(eq(ln_2,LNat0)),[meta]f_2(getf1(lm,head(ln_2)),e_2),
  %           .......
  %             if(not(eq(ln_{n-1},LNat0)),[meta]f_{n-1}(getf1(lm,head(ln_{n-1})),e_{n-1}),
  %               [meta]f_n(getf1(lm,head(ln_n)),e_n)
  %             )
  %           .......
  %         )
  %       )
  %     )
```