The Parallel Composition of Uniform Processes with Data

Jan Friso Groote[1,2] & Jos van Wamel[1]

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

2. Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

E-mail: {jfg,jos}@cwi.nl

ABSTRACT

A general basis for the definition of a finite but unbounded number of parallel processes is the equation $S(n, dt) = P(0, get(0, dt)) \lhd eq(n,0) \rhd (P(n, get(n, dt)) \parallel S(n-1, dt))$. In this formula $eq(n, 0)$ is an equality test, and $get(n, dt)$ denotes the n-th data element in table dt. We derive a linear process equation with the same behaviour as $S(n, dt)$, and show that this equation is well-defined, provided one adopts the principle CL-RSP from [4].
In order to demonstrate the strength of our result, we use it for the analysis of a standard example. We show that $n+1$ concatenated buffers form a *queue* of capacity $n+1$.

1 Introduction

Distributed algorithms are often configured as an arbitrarily large but finite set of processors that run a similar program. Using the formalism μCRL (*micro* Common Representation Language [9]) this can be described, using recursion and operators for parallelism.

Several benchmark verifications in μCRL and process algebra are therefore based on the parallel composition of an arbitrary large, but finite number of processes, that basically fit the same description, modulo some data parameters. We mention Grid Protocols [3], a Leader Election Protocol [6], a Summing Protocol [8], Milner's Scheduler [15], and the IEEE 1394 Tree Identify Protocol [19].

This observation led to the question whether a more general theorem for handling such processes would be feasible, and if so, whether it would be useful. Let us first describe the problem in more detail.

Assume that the individual processes are given by $P(k)$, where $k \in \mathbb{N}$ is the index of the process. The following equation puts $n+1$ of these processes in parallel:

$$S(n{:}\mathbb{N}) = P(0) \lhd eq(n,0) \rhd (P(n) \parallel S(n-1)) \qquad (1)$$

where the function eq is used for an equality test, and the expression $x \lhd b \rhd y$ denotes *"if b then x else y"*. (For convenience, data parameters are not considered in equation 1.) Clearly, the process $S(n)$ stands for $P(n) \parallel (\ldots (P(1) \parallel P(0)) \ldots)$.

The description in equation (1) gives rise to two issues. The first one is whether the equation unambiguously defines that $S(n)$ is the parallel composition of the processes $P(k)$.

It is clear that the parallel composition of processes $P(k)$ is a solution for $S(n)$, but is it the only one? In this paper we show, assuming the principle CL-RSP (Convergent Linear Recursive Specification Principle [4]), that this is the only solution for $S(n)$. So, an equation in the form of (1) is indeed a proper definition.

The second issue is to transform the description in (1) to a format that is more suitable for verification purposes. Many case studies have shown that for the analysis and verification of μCRL processes, the so-called *linear format* is an adequate basis for verifications. We already have a long record of verifications based on linear process descriptions, see for example [5, 6, 8, 11, 19]. For this reason, our μCRL tool set [17] is particularly tailored for the analysis of linear process descriptions. The results in this paper will therefore not only be helpful in manual verifications, but they will also contribute to the further development of the tool set.

The uniformity and relative simplicity of the linear format also allows a more uniform approach to the theory for verification. For instance, in this paper we use the *cones and foci* technique [11], which is also based on the linear format.

Now, assuming processes $P(k)$ in a linear format, we derive a linear process equivalent to $S(n)$. Actually, the "Composition Theorem" (3.7) we thus provide is rather straightforward. However, it takes care of several details that are easily overlooked when carrying out the tedious act of linearisation of a set of processes without the help of such general theorems. It should be noted that all our proofs are fully syntactic in nature, and only depend on the Recursive Specification Principle, induction, and data and process axioms.

We think that Theorem 3.7 is a convenient tool for the verification of distributed systems with an unbounded, but finite number of uniform processes. As an illustration we concatenate $n+1$ 1-place buffers, linearise the overall process using Theorem 3.7, and show that it is equal to a queue of size $n+1$.

In our view, "expansion" is the 'extraction' of initial actions from a set of parallel processes. This is usually achieved using semantic arguments, and without taking data parameters into consideration. On this type of expansion we found a number of results. In [16], for instance, there are some classical theorems for expanding a set of parallel processes. Furthermore, there are quite some results on algorithms for the model checking of specifications with multiple, similar processes. See e.g. [14, 18].

Purely syntactic, algebraic results on the composition of multiple parallel processes into a single process, seem to be scarce. In [1], however, there is a theorem for the composition of two processes, and in [3] general results can be found for the parallel composition of small computational units, with multiple input and output ports, into larger networks; *Grid Protocols*. Such networks can be used for modeling all kinds of algorithms (see also [20]).

We have found no results where – as in our work – a large part of the control structure of a process is removed from the process expression, and coded again using data parameters. We therefore think that the main result in this paper is quite unique in its sort, but also that it is not necessarily restricted to the setting of μCRL.

Acknowledgements. We thank Wan Fokkink, François Monin and Alban Ponse for their comments.

2 μCRL

The axiom system pCRL (*pico* CRL, which is μCRL without operators for parallelism) is presented first. It serves as the basic framework for our studies. The following step is to incorporate operators for parallelism and introduce μCRL. Also the concept of linear processes is introduced, as well as the Recursive Specification Principle.

2.1 Axioms for pCRL

Atomic actions are the building blocks of processes. Therefore, axiom systems in process algebra have a set of atomic actions A as a parameter. The actions are parameterised with data, and w.l.o.g. we may assume that all actions have exactly one such parameter. The set of action labels is denoted AL. For process variables we use $x, y, z, \ldots$ and for process terms we use $p, q, r, \ldots$. Choice or alternative composition is 0 by $+$, and sequential composition by $\cdot$, which is often omitted from expressions. We write $\cdot$ only in the tables of axioms. Deadlock is 0 by δ, and silent step by τ. We use $a, b, c, \ldots$ to denote elements from either AL, $AL \cup \{\tau\}$ (AL_τ), A or $A \cup \{\delta, \tau\}$ ($A_{\delta\tau}$).

Table 1 lists the axioms of pCRL. Axioms A1–A7 are well known from process algebra. The $\sum$-operator and the use of capital X will be explained below.

A1	$x + y = y + x$	SUM1	$\sum_{d:D} x = x$
A2	$x + (y + z) = (x + y) + z$	SUM3	$\sum X = \sum X + Xd$
A3	$x + x = x$	SUM4	$\sum_{d:D}(Xd + Yd) = \sum X + \sum Y$
A4	$(x + y)\cdot z = x\cdot z + y\cdot z$	SUM5	$(\sum X)\cdot x = \sum_{d:D}(Xd\cdot x)$
A5	$(x\cdot y)\cdot z = x\cdot(y\cdot z)$	SUM11	$(\forall d{\in}D\ Xd = Yd) \to \sum X = \sum Y$
A6	$x + \delta = x$		
A7	$\delta\cdot x = \delta$	C1	$x \lhd \mathsf{t} \rhd y = x$
		C2	$x \lhd \mathsf{f} \rhd y = y$
Bool1	$\neg(\mathsf{t} = \mathsf{f})$		
Bool2	$\neg(b = \mathsf{t}) \to b = \mathsf{f}$		

Table 1: Axioms of pCRL

Data types in μCRL are algebraically specified in the standard way using sorts, functions and axioms. For data sorts we use $D, E, \ldots$, and for data variables of the respective sorts we use $d, e, \ldots$. In μCRL we assume sort **Bool** for booleans.

Sort **Bool** contains the constants t ("true") and f ("false"). Typical boolean variables are $b, c, \ldots$, and the use of booleans in process expressions may become clear from the axioms C1 and C2 for the conditional construct $_ \lhd _ \rhd _$. For sort **Bool** we assume connectives $\neg, \wedge, \vee, \to$ with straightforward interpretations, and for the construction of proofs we (implicitly) use the proof theory for μCRL [10], which also provides a rule for structural induction on data terms. For booleans, this implies that we may use the principle of *case distinction* in proofs, i.e., if a formula ϕ holds for both $b = \mathsf{t}$ and $b = \mathsf{f}$ then ϕ holds in general.

We moreover assume the presence of the sort $\mathbb{N}$ of natural numbers with the constants $0, 1, 2, \ldots$, and standard operations such as $+$, $-$, $\leq$, $\geq$, $>$, $<$ and eq (equality, also written "$=$"). We use standard induction on natural numbers.

If we want to use induction in proving properties of data terms, we have to assume that the models for the data specifications involved are minimal. For the booleans this implies that there are not more than two booleans (axiom Bool2). We also have to assume that t and f are different (Bool1), because otherwise the conditional construct would not be well-defined.

In general, for $n > 0$ finite sums $p_1 + \ldots + p_n$ are abbreviated by $\sum_{i \in I} p_i$, where $I = \{1, \ldots, n\}$. In μCRL, a summation construct of the form $\sum_{d:D} p$ is a binder of variable d of data sort D in p. D may be infinite.

In axioms SUMx distinction is made between *sum operators* $\sum$ and *sum constructs* $\sum_{d:D} p$. The X in $\sum X$ may be instantiated with functions from some data sort to the sort of processes, such as $\lambda d{:}D.p$, where variable d in p may not become bound by $\sum$. We also have expressions $\sum_{d:D} x$, where some term p that is substituted for x may not contain free variable d. Data terms are considered modulo α-*conversion*, e.g., the terms $\sum_{d:D} p(d)$ and $\sum_{e:E} p(e)$ are equal.

In our calculations we work modulo associativity and commutativity of $+$, and we do not explicitly state the use of simple algebraic properties of the operators on booleans and natural numbers. Also the axioms C1 and C2 are used implicitly. As a rule, brackets are omitted from boolean expressions according to the convention that $\neg$ binds stronger than $\vee$ and $\wedge$, and that these in turn bind stronger than $\rightarrow$.

Lemma 2.1 (*Sum Elimination*). *Let $e{:}D$, where D is some arbitrary data sort with equality function eq. We have that:*

$$\sum_{d:D} Xd \triangleleft eq(d,e) \triangleright \delta = Xe \qquad (SE)$$

Proof. See e.g. [7]. □

2.2 Parallelism

The axioms of μCRL are the axioms of pCRL, combined with the axioms in Table 2. The signature $\Sigma(\mu\text{CRL})$ is as $\Sigma(p\text{CRL})$, extended with the operators for parallelism and *renaming.* (We only incorporate the operators ∂_H and τ_I, and not the general operators for renaming [4].)

For communication we have a binary function γ, which is only defined on *action labels*. In order for a communication to occur between actions $c(d), c'(e) \in A$, $\gamma(c, c')$ should be defined, and the data parameters of the actions should match according to axiom CF. By definition, the function γ is commutative and associative.

In this paper we assume the so-called *handshaking axiom*, which says that no more than two actions can synchronise. In other words, for all action labels a_1, a_2 and a_3, $\gamma(a_1, \gamma(a_2, a_3)) = \delta$ (c.f. [1]). This assumption is essential for our results, but also very reasonable. In most practical cases where synchronous communication occurs, only two parties are involved.

Concurrency is basically described by three operators: the *merge* $\|$, the *left merge* $⌊\!⌊$ and the *communication merge* $|$. The process $p \| q$ symbolises the parallel execution of p and q. It 'starts' with an action of either p or q, or with a communication, or *synchronisation*, between

p and q. $p \mathbin{\lfloor\!\lfloor} q$ is as $p \parallel q$, but the first action that is performed comes from p. Process $p \mid q$ is as $p \parallel q$, but the first action is a communication between p and q. We will tacitly assume associativity and commutativity of $\parallel$ and $\mid$, as it can be derived from the axioms for all closed instances.

Encapsulation operators ∂_H block atomic actions with labels in H by renaming them to δ. They are used to enforce communication between processes. Abstraction operators τ_I are particularly useful for hiding communication actions, by renaming them to τ.

The various operators of $\Sigma(\mu\text{CRL})$ are listed in order of decreasing binding strength:

$$\cdot \quad \{\lhd\ \rhd, \parallel, \mathbin{\lfloor\!\lfloor}, \mid\} \quad \sum_{d:D} \quad +$$

Brackets are omitted from expressions according to this convention.

SUM6	$(\sum X) \mathbin{\lfloor\!\lfloor} x = \sum_{d:D}(Xd \mathbin{\lfloor\!\lfloor} x)$		
SUM7	$(\sum X) \mid x = \sum_{d:D}(Xd \mid x)$		
SUM7′	$x \mid (\sum X) = \sum_{d:D}(x \mid Xd)$	CF	$c(d) \mid c'(e) = \begin{cases} \gamma(c,c')(d) \lhd eq(d,e) \rhd \delta & \text{if } \gamma(c,c') \text{ defined} \\ \delta & \text{otherwise} \end{cases}$
SUM8	$\partial_H(\sum X) = \sum_{d:D} \partial_H(Xd)$		
SUM9	$\tau_I(\sum X) = \sum_{d:D} \tau_I(Xd)$		
CM1	$x \parallel y = x \mathbin{\lfloor\!\lfloor} y + y \mathbin{\lfloor\!\lfloor} x + x \mid y$	CD1	$\delta \mid a = \delta$
CM2	$a \mathbin{\lfloor\!\lfloor} x = a \cdot x$	CD2	$a \mid \delta = \delta$
CM3	$a \cdot x \mathbin{\lfloor\!\lfloor} y = a \cdot (x \parallel y)$	CT1	$\tau \mid x = \delta$
CM4	$(x+y) \mathbin{\lfloor\!\lfloor} z = x \mathbin{\lfloor\!\lfloor} z + y \mathbin{\lfloor\!\lfloor} z$	CT2	$x \mid \tau = \delta$
CM5	$a \cdot x \mid b = (a \mid b) \cdot x$		
CM6	$a \mid b \cdot x = (a \mid b) \cdot x$	D1	$\partial_H(c(d)) = c(d) \quad \text{if } c \notin H$
CM7	$a \cdot x \mid b \cdot y = (a \mid b) \cdot (x \parallel y)$	D2	$\partial_H(c(d)) = \delta \quad \text{if } c \in H$
CM8	$(x+y) \mid z = x \mid z + y \mid z$	D3	$\partial_H(x+y) = \partial_H(x) + \partial_H(y)$
CM9	$x \mid (y+z) = x \mid y + x \mid z$	D4	$\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$
DD	$\partial_H(\delta) = \delta$	TI1	$\tau_I(c(d)) = c(d) \quad \text{if } c \notin I$
DT	$\partial_H(\tau) = \tau$	TI2	$\tau_I(c(d)) = \tau \quad \text{if } c \in I$
TID	$\tau_I(\delta) = \delta$	TI3	$\tau_I(x+y) = \tau_I(x) + \tau_I(y)$
TIT	$\tau_I(\tau) = \tau$	TI4	$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$

Table 2: Axioms for parallelism and renaming of μCRL, where $a, b \in A_{\delta\tau}$ and $c, c' \in AL$

2.3 Linear processes

We introduce *recursion* as well as the notion of *linear processes*. For theoretical purposes, it is often convenient to use the related notion of *Linear Process Operators* (*LPOs*), instead of linear processes. LPOs may be considered as descriptions of *process graphs*, with terms p as states, and actions a_i as *transition labels*. The conditions c_i determine when the corresponding transitions may take place.

Definition 2.2. A *Linear Process Operator* (*LPO*) Ψ is an expression of the form:

$$\lambda p{:}D \to \mathbb{P}.\lambda d{:}D. \sum_{i\in I}\sum_{e_i:D_i} a_i(f_i(d,e_i))\, p(g_i(d,e_i)) \lhd c_i(d,e_i) \rhd \delta$$

for some finite index set I, action labels $a_i \in AL_\tau$, data types D_i and D_{a_i}, functions $f_i : D \to D_i \to D_{a_i}$, $g_i : D \to D_i \to D$ and $c_i : D \to D_i \to \mathbf{Bool}$. $\mathbb{P}$ is the sort of processes.

Note that in general, processes have more than one parameter. Using pairing and projection functions, it is easy to see that this is a non-essential extension. In [4] linear processes were also equipped with a termination option. For reasons of conciseness, we omit this here. From an LPO Ψ one can easily derive its associated Linear Process Equation $p(d) = \Psi pd$. The other way, from LPE to LPO, is also easy.

Definition 2.3. A linear process operator Ψ written in the form above is called *convergent* iff there is a well-founded ordering $<$ on D, such that when $a_i \equiv \tau$ and $c_i(d,e_i)$, it holds that $g_i(d,e_i) < d$, for all $d \in D$, $i \in I$ and $e_i \in D_i$.

Convergence of an LPO guarantees that there are no cyclic τ-paths from certain states of the process to itself. Cyclic τ-paths would give rise to non-unique solutions, and consequently make the defining LPO ambiguous. The notion of convergence is closely related to the more standard notions of *guardedness* [1, 9].

We furthermore state the validity of the following principles, which are restricted variants of the corresponding principles in [4], as basic assumptions.

Definition 2.4. The *Recursive Definition Principle* (*RDP*) says that every linear process operator Ψ has at least one fixed point, i.e. there exists a $p : D \to \mathbb{P}$ such that $p = \Psi p$.

The idea behind CL-RSP is that whenever two process graphs have the same basic structure, as determined by the transition labels and the conditions at the transitions, there must be a 1-1 correspondence between the states of these two processes.

Definition 2.5. The *Convergent Linear Recursive Specification Principle* (*CL-RSP*) says that every convergent linear process operator has at most one fixed point, i.e. for all $p : D \to \mathbb{P}$ and $q : D \to \mathbb{P}$ if $p = \Psi p$ and $q = \Psi q$, then $p = q$.

3 Linearisation of parallel processes

3.1 Definition

We provide the linearisation of the parallel composition of $n+1$ linear processes of the form $P(k,d)$, i.e., we derive an LPE for such a process. The natural number k $(0 \leq k \leq n)$ is the index of the process, and the parameter d of some arbitrary sort D denotes other parameters. We assume that each process $P(k,d)$ is defined according to the following Linear Process Equation:

$$P(k{:}\mathbb{N}, d{:}D) = \sum_{i\in I}\sum_{e_i:E_i} a_i(f_i(k,d,e_i))\, P(k, g_i(k,d,e_i)) \lhd c_i(k,d,e_i) \rhd \delta \tag{2}$$

We also assume that this equation is convergent, so that it defines a unique process.

In order to define the parallel composition we will use a new sort *DTable*, which defines tables indexed by natural numbers, and contains elements of the sort D. Tables are simple structures, with sufficient functionality for our purposes. In order to define tables, we also need an auxiliary function $if : \mathbf{Bool} \times D \times D \to D$ reflecting $if - then - else$. In the sequel we also use $eq : D_{a_i} \times D_{a_i} \to \mathbf{Bool}$, expressing equality of the elements in D_{a_i} (only necessary for actions that may communicate). We do not explicitly provide defining equations for these functions.

The constant *emD* of sort *DTable* denotes the empty table. The function *upd* enters a new data element in the table and the function *get* gets a specific element from an entry of the table. These operators are characterised by a single equation. We do not specify what happens if an element from the empty table is being read, as we simply do not encounter this situation. We refer to the characterising axiom for tables as the *Table Axiom* (*TA*).

sort $DTable$
func $emD : \to DTable$
$upd : \mathbb{N} \times D \times DTable \to DTable$
$get : \mathbb{N} \times DTable \to D$
var $n, m{:}\mathbb{N},\ d{:}D,\ dt{:}DTable$
rew $get(n, upd(m, d, dt)) = if(eq(n, m), d, get(n, dt))$ *TA*

We can use the following process definition to put $n+1$ processes $P(k, d)$ in parallel:

$$S(n{:}\mathbb{N}, dt{:}DTable) = P(0, get(0, dt)) \triangleleft eq(n, 0) \triangleright (P(n, get(n, dt)) \parallel S(n-1, dt)) \tag{3}$$

In this equation dt denotes a table with initial values of the parameters of processes P. Obviously, the n-th table entry contains the value for the process with index n.

3.2 Composition

In this section we derive a linear description of $S(n, dt)$ (Lemma 3.3). As a bonus we get that $S(n, dt)$ has at most one solution (Corollary 3.5). In the following lemmas we present some facts that are used in the calculations to follow.

Lemma 3.1. *It holds that:*

1. $m > n = \mathsf{t} \to S(n, dt) = S(n, upd(m, d, dt))$;
2. $k_1 \neq k_2 \to get(n, upd(k_1, d_1, upd(k_2, d_2, dt))) = get(n, upd(k_2, d_2, upd(k_1, d_1, dt)))$.

Proof. The first fact is proven by induction on n. The second follows from axiom *TA*. □

Lemma 3.2. *Let $n \geq k, k_1, k_2$. We have that:*

1. $P(n+1, g_i(n+1, get(n+1, dt), e_i)) \parallel S(n, dt)$
$= S(n+1, upd(n+1, g_i(n+1, get(n+1, dt), e_i), dt))$;
2. $P(n+1, get(n+1, dt)) \parallel S(n, upd(k, g_i(k, get(k, dt), e_i), dt))$
$= S(n+1, upd(k, g_i(k, get(k, dt), e_i), dt))$;

3. $P(n+1, get(n+1, dt))$
$\| S(n, upd(k_1, g_{i_1}(k_1, get(k_1, dt), e_{i_1}), upd(k_2, g_{i_2}(k_2, get(k_2, dt), e_{i_2}), dt)))$
$= S(n+1, upd(k_1, g_{i_1}(k_1, get(k_1, dt), e_{i_1}), upd(k_2, g_{i_2}(k_2, get(k_2, dt), e_{i_2}), dt)))$;

4. $P(n+1, g_{i_1}(n+1, get(n+1, dt), e_{i_1})) \| S(n, upd(k_2, g_{i_2}(k_2, get(k_2, dt), e_{i_2}), dt))$
$= S(n+1, upd(n+1, g_{i_1}(n+1, get(n+1, dt), e_{i_1}), upd(k_2, g_{i_2}(k_2, get(k_2, dt), e_{i_2}), dt)))$.

Proof. Straightforward. Use axiom *TA*, equation (3) and Lemma 3.1.1. □

Below we present the core lemma of this paper. It gives an expansion of S, where all operators for parallelism have been removed. The resulting process has the index n and the table dt as parameters. In essence, the complexity of process S is now coded using the simple table operations *upd* and *get*.

Lemma 3.3 says that process *Par* may act on data of the k-th component, whose data state is represented by the k-th table entry. Such an action a_i leads to an update of the k-th table entry. Process *Par* may also do some internal action $\gamma(a_{i_1}, a_{i_2})$ and exchange data $f_{i_1}(k_1, get(k_1, dt), e_{i_1})$ between the components k_1 and k_2. The k_1-th and k_2-th table entries are updated as a result of this action.

Lemma 3.3. *The process S as defined in equation (3) is a solution for Par in equation (4) below, where the set I and the functions f_i, g_i and c_i are those that occur in equation (2).*

$$\begin{array}{l} Par(n{:}\mathbb{N}, dt{:}DTable) = \\ \quad \sum_{i \in I} \sum_{k:\mathbb{N}} \sum_{e_i:E_i} a_i(f_i(k, get(k, dt), e_i))\, Par(n, upd(k, g_i(k, get(k, dt), e_i), dt)) \\ \qquad \triangleleft\, c_i(k, get(k, dt), e_i) \wedge k \leq n \triangleright \delta\, + \\ \quad \sum_{i_1 \in I} \sum_{i_2 \in I} \sum_{k_1:\mathbb{N}} \sum_{k_2:\mathbb{N}} \sum_{e_{i_1}:E_{i_1}} \sum_{e_{i_2}:E_{i_2}} \gamma(a_{i_1}, a_{i_2})(f_{i_1}(k_1, get(k_1, dt), e_{i_1})) \\ \qquad Par(n, upd(k_1, g_{i_1}(k_1, get(k_1, dt), e_{i_1}), upd(k_2, g_{i_2}(k_2, get(k_2, dt), e_{i_2}), dt))) \\ \qquad\quad \triangleleft\, c_{i_1}(k_1, get(k_1, dt), e_{i_1}) \wedge c_{i_2}(k_2, get(k_2, dt), e_{i_2}) \wedge \\ \qquad\qquad eq(f_{i_1}(k_1, get(k_1, dt), e_{i_1}), f_{i_2}(k_2, get(k_2, dt), e_{i_2})) \wedge k_1 > k_2 \wedge k_1 \leq n \triangleright \delta. \end{array} \tag{4}$$

Proof. It is sufficient to show that the equation above holds, with all occurrences of *Par* replaced by S:

$$\begin{array}{l} S(n, dt) = \\ \quad \sum_{i \in I} \sum_{k:\mathbb{N}} \sum_{e_i:E_i} a_i(f_i(k, get(k, dt), e_i))\, S(n, upd(k, g_i(k, get(k, dt), e_i), dt)) \\ \qquad \triangleleft\, c_i(k, get(k, dt), e_i) \wedge k \leq n \triangleright \delta\, + \\ \quad \sum_{i_1 \in I} \sum_{i_2 \in I} \sum_{k_1:\mathbb{N}} \sum_{k_2:\mathbb{N}} \sum_{e_{i_1}:E_{i_1}} \sum_{e_{i_2}:E_{i_2}} \gamma(a_{i_1}, a_{i_2})(f_{i_1}(k_1, get(k_1, dt), e_{i_1})) \\ \qquad S(n, upd(k_1, g_{i_1}(k_1, get(k_1, dt), e_{i_1}), upd(k_2, g_{i_2}(k_2, get(k_2, dt), e_{i_2}), dt))) \\ \qquad\quad \triangleleft\, c_{i_1}(k_1, get(k_1, dt), e_{i_1}) \wedge c_{i_2}(k_2, get(k_2, dt), e_{i_2}) \wedge \\ \qquad\qquad eq(f_{i_1}(k_1, get(k_1, dt), e_{i_1}), f_{i_2}(k_2, get(k_2, dt), e_{i_2})) \wedge k_1 > k_2 \wedge k_1 \leq n \triangleright \delta. \end{array} \tag{5}$$

We do this with induction on n.

Base, $n = 0$.

$$
\begin{array}{ll}
S(0, dt) & \\
\stackrel{(3)}{=} & P(0, get(0, dt)) \\
\stackrel{(2)}{=} & \sum_{i \in I} \sum_{e_i : E_i} a_i(f_i(0, get(0, dt), e_i)) P(0, g_i(0, get(0, dt), e_i)) \\
& \qquad \triangleleft\, c_i(0, get(0, dt), e_i) \,\triangleright \delta \\
\stackrel{TA}{=} & \sum_{i \in I} \sum_{e_i : E_i} a_i(f_i(0, get(0, dt), e_i)) \\
& \qquad P(0, get(0, upd(0, g_i(0, get(0, dt), e_i), dt))) \\
& \qquad \triangleleft\, c_i(0, get(0, dt), e_i) \,\triangleright \delta \\
\stackrel{(3)}{=} & \sum_{i \in I} \sum_{e_i : E_i} a_i(f_i(0, get(0, dt), e_i)) S(0, upd(0, g_i(0, get(0, dt), e_i), dt)) \\
& \qquad \triangleleft\, c_i(0, get(0, dt), e_i) \,\triangleright \delta \\
\stackrel{\mathrm{SUM1}, SE}{=} & \sum_{i \in I} \sum_{k:\mathbb{N}} \sum_{e_i : E_i} a_i(f_i(k, get(k, dt), e_i)) S(k, upd(k, g_i(k, get(k, dt), e_i), dt)) \\
& \qquad \triangleleft\, c_i(k, get(k, dt), e_i) \wedge k \leq 0 \,\triangleright \delta.
\end{array}
$$

We now have the first main summand of equation (5). Moreover there are no k_1, k_2 that satisfy $k_2 < k_1 \leq 0$, so the second main summand of (5) equals δ. As $x + \delta = x$ we may conclude this part of the proof.

Induction step. Suppose equation (5) holds for some $n \geq 0$. We show that it also holds for $n + 1$. So, we must derive (5) with occurrences of n replaced by $n + 1$.

We expand the equation for $S(n + 1, dt)$ a little:

$$
\begin{array}{lcll}
S(n+1, dt) & \stackrel{(3)}{=} & P(n+1, get(n+1, dt)) \,\|\, S(n, dt) & \\
& \stackrel{\mathrm{CM1}}{=} & P(n+1, get(n+1, dt)) \,\|\!_\, S(n, dt) + & (A) \\
& & S(n, dt) \,\|\!_\, P(n+1, get(n+1, dt)) + & (B) \\
& & P(n+1, get(n+1, dt)) \mid S(n, dt) & (C)
\end{array}
$$

We analyse the terms A, B and C separately.

A. By equation (2) we have

$$
\begin{array}{ll}
P(n+1, get(n+1, dt)) \,\|\!_\, S(n, dt) & \\
= & \sum_{i \in I} \sum_{e_i : E_i} a_i(f_i(n+1, get(n+1, dt), e_i)) \\
& \qquad (P(n+1, g_i(n+1, get(n+1, dt), e_i)) \,\|\, S(n, dt)) \\
& \qquad \triangleleft\, c_i(n+1, get(n+1, dt), e_i) \,\triangleright \delta \\
\stackrel{3.2.1, SE}{=} & \sum_{i \in I} \sum_{k:\mathbb{N}} \sum_{e_i : E_i} a_i(f_i(k, get(k, dt), e_i)) \\
& \qquad S(n+1, upd(k, g_i(k, get(k, dt), e_i), dt)) \\
& \qquad \triangleleft\, c_i(k, get(k, dt), e_i) \wedge k = n+1 \,\triangleright \delta \qquad (a)
\end{array}
$$

B. By the induction hypothesis we have a linear form for $S(n, dt)$ (c.f. equation (5)). Expansion of B:

$$
\begin{aligned}
& S(n,dt) \mathbin{\lfloor\!\lfloor} P(n+1, get(n+1,dt)) \\
= \quad & \textstyle\sum_{i\in I}\sum_{k:\mathbb{N}}\sum_{e_i:E_i} a_i(f_i(k, get(k,dt), e_i)) \\
& \quad (S(n, upd(k, g_i(k, get(k,dt), e_i), dt)) \parallel P(n+1, get(n+1,dt))) \\
& \qquad \lhd c_i(k, get(k,dt), e_i) \wedge k \le n \rhd \delta + \\
& \textstyle\sum_{i_1\in I}\sum_{i_2\in I}\sum_{k_1:\mathbb{N}}\sum_{k_2:\mathbb{N}}\sum_{e_{i_1}:E_{i_1}}\sum_{e_{i_2}:E_{i_2}} \gamma(a_{i_1}, a_{i_2})(f_{i_1}(k_1, get(k_1,dt), e_{i_1})) \\
& \quad (S(n, upd(k_1, g_{i_1}(k_1, get(k_1,dt), e_{i_1}), upd(k_2, g_{i_2}(k_2, get(k_2,dt), e_{i_2}), dt))) \\
& \quad \parallel P(n+1, get(n+1,dt))) \\
& \qquad \lhd c_{i_1}(k_1, get(k_1,dt), e_{i_1}) \wedge c_{i_2}(k_2, get(k_2,dt), e_{i_2}) \wedge \\
& \qquad\ eq(f_{i_1}(k_1, get(k_1,dt), e_{i_1}), f_{i_2}(k_2, get(k_2,dt), e_{i_2})) \wedge \\
& \qquad\ k_1 > k_2 \wedge k_1 \le n \rhd \delta \\
\overset{3.2.\{2,3\}}{=} \quad & \textstyle\sum_{i\in I}\sum_{k:\mathbb{N}}\sum_{e_i:E_i} a_i(f_i(k, get(k,dt), e_i)) \\
& \quad S(n+1, upd(k, g_i(k, get(k,dt), e_i), dt)) \\
& \qquad \lhd c_i(k, get(k,dt), e_i) \wedge k \le n \rhd \delta + && (b1) \\
& \textstyle\sum_{i_1\in I}\sum_{i_2\in I}\sum_{k_1:\mathbb{N}}\sum_{k_2:\mathbb{N}}\sum_{e_{i_1}:E_{i_1}}\sum_{e_{i_2}:E_{i_2}} \gamma(a_{i_1}, a_{i_2})(f_{i_1}(k_1, get(k_1,dt), e_{i_1})) \\
& \quad S(n+1, upd(k_1, g_{i_1}(k_1, get(k_1,dt), e_{i_1}), \\
& \qquad\qquad upd(k_2, g_{i_2}(k_2, get(k_2,dt), e_{i_2}), dt))) \\
& \qquad \lhd c_{i_1}(k_1, get(k_1,dt), e_{i_1}) \wedge c_{i_2}(k_2, get(k_2,dt), e_{i_2}) \wedge \\
& \qquad\ eq(f_{i_1}(k_1, get(k_1,dt), e_{i_1}), f_{i_2}(k_2, get(k_2,dt), e_{i_2})) \wedge \\
& \qquad\ k_1 > k_2 \wedge k_1 \le n \rhd \delta && (b2)
\end{aligned}
$$

C. We may again use the induction hypothesis. We use the fact (axiom CF) that the communication of two actions is not a δ only if the arguments are equal. Also note that ternary communication is not allowed. In P we use indices $i_1, k_1 = n+1$, and in the first main summand of S we use indices i_2, k_2.

$$
\begin{aligned}
& P(n+1, get(n+1,dt)) \mid S(n,dt) \\
= \quad & \textstyle\sum_{i_1\in I}\sum_{i_2\in I}\sum_{k_2:\mathbb{N}}\sum_{e_{i_1}:E_{i_1}}\sum_{e_{i_2}:E_{i_2}} \\
& \quad \gamma(a_{i_1}, a_{i_2})(f_{i_1}(n+1, get(n+1,dt), e_{i_1})) \\
& \qquad (P(n+1, g_{i_1}(n+1, get(n+1,dt), e_{i_1})) \parallel \\
& \qquad\ S(n, upd(k_2, g_{i_2}(k_2, get(k_2,dt), e_{i_2}), dt))) \\
& \qquad\quad \lhd c_{i_1}(n+1, get(n+1,dt), e_{i_1}) \wedge c_{i_2}(k_2, get(k_2,dt), e_{i_2}) \wedge \\
& \qquad\quad\ eq(f_{i_1}(n+1, get(n+1,dt), e_{i_1}), f_{i_2}(k_2, get(k_2,dt), e_{i_2})) \wedge \\
& \qquad\quad\ k_2 \le n \rhd \delta \\
\overset{3.2.4,SE}{=} \quad & \textstyle\sum_{i_1\in I}\sum_{i_2\in I}\sum_{k_1:\mathbb{N}}\sum_{k_2:\mathbb{N}}\sum_{e_{i_1}:E_{i_1}}\sum_{e_{i_2}:E_{i_2}} \gamma(a_{i_1}, a_{i_2})(f_{i_1}(k_1, get(k_1,dt), e_{i_1})) \\
& \quad S(n+1, upd(k_1, g_{i_1}(k_1, get(k_1,dt), e_{i_1}), \\
& \qquad\qquad upd(k_2, g_{i_2}(k_2, get(k_2,dt), e_{i_2}), dt))) \\
& \qquad\quad \lhd c_{i_1}(k_1, get(k_1,dt), e_{i_1}) \wedge c_{i_2}(k_2, get(k_2,dt), e_{i_2}) \wedge \\
& \qquad\quad\ eq(f_{i_1}(k_1, get(k_1,dt), e_{i_1}), f_{i_2}(k_2, get(k_2,dt), e_{i_2})) \wedge \\
& \qquad\quad\ k_1 > k_2 \wedge k_1 = n+1 \rhd \delta && (c)
\end{aligned}
$$

Now take $a + b1$ and $b2 + c$, and combine the results. We then exactly have the desired right-hand side of $S(n+1, dt)$, which proves the induction step. We conclude that $S(n, dt)$ is a solution for $Par(n, dt)$. □

Lemma 3.4. *Equation (4) is convergent.*

Proof. As (2) is convergent, there is a well-founded relation $< \subseteq \langle \mathbb{N} \times D \rangle \times \langle \mathbb{N} \times D \rangle$, such that if $c_i(k, d, e_i) = \mathsf{t}$ and $a_i = \tau$, then $\langle k, g_i(k, d, e_i) \rangle < \langle k, d \rangle$.

Using $<$ we can define a well-founded relation $\prec$ as follows:

$$\langle \langle n_1, dt_1 \rangle, \langle n_2, dt_2 \rangle \rangle \subseteq \prec \text{ iff } \begin{cases} eq(n_1, n_2) \wedge \\ \text{for all } 0 \leq k \leq n_1 : \langle k, get(k, dt_1) \rangle \leq \langle k, get(k, dt_2) \rangle \wedge \\ \text{for some } 0 \leq k \leq n_1 : \langle k, get(k, dt_1) \rangle < \langle k, get(k, dt_2) \rangle \end{cases}$$

where $\langle k_1, d_1 \rangle \leq \langle k_2, d_2 \rangle$ iff $\langle k_1, d_1 \rangle < \langle k_2, d_2 \rangle$, or $eq(k_1, k_2) \wedge eq(d_1, d_2)$.

Now consider equation (4). The second main summand of Par can never 'start' with a τ-step, so only the first has to be taken into account. Using $\prec$ it is straightforward to see that convergence is a fact. □

Corollary 3.5 (*Parallel Specification Principle*). *Equation (3) has at most one solution for the variable S.*

Proof. Lemma 3.3 says that any solution for S in (3) is a solution for Par in (4). Using Lemma 3.4 CL-RSP expresses that there is at most one solution for Par. Consequently, also equation (3) has at most one solution. □

3.3 Main theorem

For practical use we find the form of equation (4) not very convenient, since it has the condition $k_1 > k_2$ in its second main summand. A more convenient form, stated in our main result, Theorem 3.7, has this condition coded in the indices i_1 and i_2. Another reason to rewrite equation (4) to the form in Theorem 3.7, is that the latter has only half the number of summands in its second main term.

We first present a lemma.

Lemma 3.6. *Let $k_1 \neq k_2$. It holds that*

$$Par(n, upd(k_1, g_{i_1}(k_1, get(k_1, dt), e_{i_1}), upd(k_2, g_{i_2}(k_2, get(k_2, dt), e_{i_2}), dt))) =$$
$$Par(n, upd(k_2, g_{i_2}(k_2, get(k_2, dt), e_{i_2}), upd(k_1, g_{i_1}(k_1, get(k_1, dt), e_{i_1}), dt))).$$

Proof. We abbreviate the term $g_{i_1}(k_1, get(k_1, dt), e_{i_1})$ by d_1 and $g_{i_2}(k_2, get(k_2, dt), e_{i_2})$ by d_2. As $Par(n, dt) = S(n, dt)$ it suffices to prove

$$S(n, upd(k_1, d_1, upd(k_2, d_2, dt))) = S(n, upd(k_2, d_2, upd(k_1, d_1, dt))).$$

We prove this fact by induction on n.

Base, $n = 0$. We have that

$$\begin{aligned} S(0, upd(k_1, d_1, upd(k_2, d_2, dt))) &\overset{(3)}{=} P(0, get(0, upd(k_1, d_1, upd(k_2, d_2, dt)))) \\ &\overset{3.1.2}{=} P(0, get(0, upd(k_2, d_2, upd(k_1, d_1, dt)))). \end{aligned}$$

Induction step. Using a similar argument and the induction hypothesis, this part of the proof also follows easily.

□

In order to obtain the following result, we have to assume that there is a total reflexive ordering $\leq$ on the index set I.

Theorem 3.7 (*Composition Theorem*). *The process S as defined in equation (3) is the (unique) solution for Par in the (convergent) equation below; so for all $n{:}\mathbb{N}$ and $dt{:}DTable$, $S(n, dt) = Par(n, dt)$, where the set I and the functions f_i, g_i and c_i are those that occur in equation (2).*

$$\begin{aligned}
&Par(n{:}\mathbb{N}, dt{:}DTable) = \\
&\quad \textstyle\sum_{i\in I}\sum_{k:\mathbb{N}}\sum_{e_i:E_i} a_i(f_i(k, get(k, dt), e_i))\, Par(n, upd(k, g_i(k, get(k, dt), e_i), dt)) \\
&\qquad \lhd c_i(k, get(k, dt), e_i) \land k \leq n \rhd \delta + \\
&\quad \textstyle\sum_{i_1\in I}\sum_{i_2\in I\land i_2\leq i_1}\sum_{k_1:\mathbb{N}}\sum_{k_2:\mathbb{N}}\sum_{e_{i_1}:E_{i_1}}\sum_{e_{i_2}:E_{i_2}} \gamma(a_{i_1}, a_{i_2})(f_{i_1}(k_1, get(k_1, dt), e_{i_1})) \\
&\qquad Par(n, upd(k_1, g_{i_1}(k_1, get(k_1, dt), e_{i_1}), upd(k_2, g_{i_2}(k_2, get(k_2, dt), e_{i_2}), dt))) \\
&\qquad\quad \lhd c_{i_1}(k_1, get(k_1, dt), e_{i_1}) \land c_{i_2}(k_2, get(k_2, dt), e_{i_2}) \land \\
&\qquad\quad\; eq(f_{i_1}(k_1, get(k_1, dt), e_{i_1}), f_{i_2}(k_2, get(k_2, dt), e_{i_2})) \land \\
&\qquad\quad\; \neg eq(k_1, k_2) \land k_1 \leq n \land k_2 \leq n \rhd \delta.
\end{aligned}$$

Proof. We show that the right-hand side of equation (4) may be transformed to the right-hand side of the equation above. Actually, as the first main summands of both equations are equal, we only have to show that the second main summands are equal. To keep the argument short we introduce the following two abbreviations:

$$\begin{aligned}
P_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2}) &= \gamma(a_{i_1}, a_{i_2})(f_{i_1}(k_1, get(k_1, dt), e_{i_1})) \\
&\quad Par(n, upd(k_1, g_{i_1}(k_1, get(k_1, dt), e_{i_1}), upd(k_2, g_{i_2}(k_2, get(k_2, dt), e_{i_2}), dt))), \\
C_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2}) &= c_{i_1}(k_1, get(k_1, dt), e_{i_1}) \land c_{i_2}(k_2, get(k_2, dt), e_{i_2}) \land \\
&\quad eq(f_{i_1}(k_1, get(k_1, dt), e_{i_1}), f_{i_2}(k_2, get(k_2, dt), e_{i_2})).
\end{aligned}$$

An essential observation is that $C_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2}) = C_{i_2 i_1}(k_2, k_1, e_{i_2}, e_{i_1})$, and if $k_1 \neq k_2$ and $f_{i_1}(k_1, get(k_1, dt), e_{i_1}) = f_{i_2}(k_2, get(k_2, dt), e_{i_2})$ then using Lemma 3.6 it follows that $P_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2}) = P_{i_2 i_1}(k_2, k_1, e_{i_2}, e_{i_1})$.

The second main summand of (4) can now be written as

$$\begin{aligned}
&\textstyle\sum_{i_1\in I}\sum_{i_2\in I}\sum_{k_1:\mathbb{N}}\sum_{k_2:\mathbb{N}}\sum_{e_{i_1}:E_{i_1}}\sum_{e_{i_2}:E_{i_2}} P_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2}) \\
&\qquad\qquad \lhd C_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2}) \land k_1 > k_2 \land k_1 \leq n \rhd \delta.
\end{aligned}$$

By splitting this summand into $i_2 \leq i_1$ and $i_2 \geq i_1$, splitting $k_1 > k_2$ into $k_1 \geq k_2$ and $\neg eq(k_1, k_2)$ and adding the redundant condition $k_2 \leq n$ this term is equal to:

$$\begin{aligned}
&\textstyle\sum_{i_1\in I}\sum_{i_2\in I\land i_2\leq i_1}\sum_{k_1:\mathbb{N}}\sum_{k_2:\mathbb{N}}\sum_{e_{i_1}:E_{i_1}}\sum_{e_{i_2}:E_{i_2}} P_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2}) \\
&\qquad \lhd C_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2}) \land k_1 \geq k_2 \land \neg eq(k_1, k_2) \land k_1 \leq n \land k_2 \leq n \rhd \delta + \\
&\textstyle\sum_{i_1\in I}\sum_{i_2\in I\land i_2\geq i_1}\sum_{k_1:\mathbb{N}}\sum_{k_2:\mathbb{N}}\sum_{e_{i_1}:E_{i_1}}\sum_{e_{i_2}:E_{i_2}} P_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2}) \\
&\qquad \lhd C_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2}) \land k_1 \geq k_2 \land \neg eq(k_1, k_2) \land k_1 \leq n \land k_2 \leq n \rhd \delta.
\end{aligned}$$

By changing the order of the summands and by exchanging the names of i_1 and i_2, k_1 and k_2, and e_{i_1} and e_{i_2} in the second main summand, we obtain:

$$\begin{aligned}
&\textstyle\sum_{i_1\in I}\sum_{i_2\in I\land i_2\leq i_1}\sum_{k_1:\mathbb{N}}\sum_{k_2:\mathbb{N}}\sum_{e_{i_1}:E_{i_1}}\sum_{e_{i_2}:E_{i_2}} P_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2}) \\
&\qquad \lhd C_{i_1 i_2}(k_1, k_2, , e_{i_1}, e_{i_2}) \land k_1 \geq k_2 \land \neg eq(k_1, k_2) \land k_1 \leq n \land k_2 \leq n \rhd \delta + \\
&\textstyle\sum_{i_1\in I}\sum_{i_2\in I\land i_2\leq i_1}\sum_{k_1:\mathbb{N}}\sum_{k_2:\mathbb{N}}\sum_{e_{i_1}:E_{i_1}}\sum_{e_{i_2}:E_{i_2}} P_{i_2 i_1}(k_2, k_1, e_{i_2}, e_{i_1}) \\
&\qquad \lhd C_{i_2 i_1}(k_2, k_1, e_{i_2}, e_{i_1}) \land k_2 \geq k_1 \land \neg eq(k_2, k_1) \land k_2 \leq n \land k_1 \leq n \rhd \delta.
\end{aligned}$$

By the observation stated above we may put some of the variables i_1, i_2, k_1 and k_2 in the second main summand back to their original places. A term results with two main summands, only differing in that one contains the condition $k_1 \geq k_2$, and the other contains $k_1 \leq k_2$. As either of the conditions must be the case, we may take both main summands together, and obtain

$$\sum_{i_1 \in I} \sum_{i_2 \in I \wedge i_2 \leq i_1} \sum_{k_1 : \mathbb{N}} \sum_{k_2 : \mathbb{N}} \sum_{e_{i_1} : E_{i_1}} \sum_{e_{i_2} : E_{i_2}} P_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2}) \\ \lhd C_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2}) \wedge \neg eq(k_1, k_2) \wedge k_1 \leq n \wedge k_2 \leq n \rhd \delta,$$

which is the desired right-hand side. □

4 Example verification

In this section we give an example of the application of Theorem 3.7. We concatenate $n+1$ buffers and prove that the total system exactly behaves as a queue of capacity $n+1$. In order to abstract from internal activity the *cones and foci* method is used [11].

4.1 The cones and foci method

In process algebra it is common to verify the correctness of a process – referred to as the *implementation* – by proving it equivalent to a more abstract process, the *specification*. Data parameters, which often impose control structures on a process, can make such equivalence proofs very complex.

The cones and foci technique addresses this problem. The main idea behind this technique is that there are usually many internal events in an implementation, but that they are only significant in the sense that they must somehow progress towards a state where visible events are possible. These events should match with a visible event in the corresponding specification. It may be, however, that external actions take place while internal activity is still possible.

A state of the implementation where no internal actions are enabled is called a *focus point*. Focus points are characterised by a condition on the data of the process called the *focus condition*. The focus condition is the negation of the condition which allows τ-actions to occur. The *cone* belonging to a focus point is that part of the state space, from which the focus can be reached by doing only internal actions.

Figure 1 may give some more intuition about cones and foci. Imagine that the transition system has a cone or 'funnel' which points towards the focus (F). In the funnel only internal process activity (τ-steps) takes place. This internal activity ultimately reaches a point where the implementation has to do external steps (a, b, c, d).

In a verification of processes with data, there may also be unreachable states in the implementation. These can be excluded using an *invariant*. As we do not need invariants in the example verification, we omit further references to invariants.

The crucial element in the technique is a mapping from the data states of the implementation to the data states of the specification. This mapping is surjective, but almost certainly not injective, since the data of the specification is very likely to be simpler than that of the implementation. So in terms of data structures we have a refinement, but in terms of actions we have an equivalence.

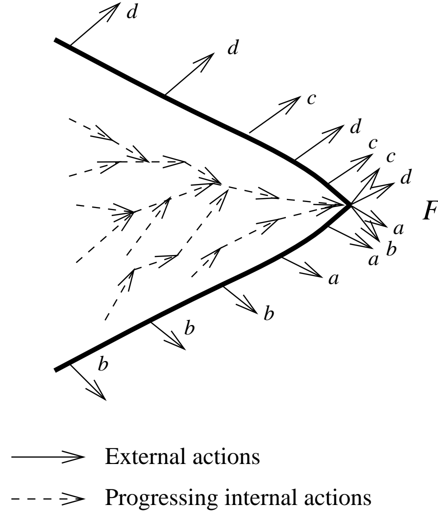

Figure 1: A cone and a focus point

Consider the following two LPOs (see also Definition 2.2), and let Ψ be the LPO of the implementation and Φ the LPO of the specification.

$$\Psi \stackrel{\text{def}}{=} \lambda p{:}D_\Psi \to \mathbb{P}.\lambda d{:}D_\Psi . \sum_{i\in I} \sum_{e_i:D_i} a_i(f_i(d,e_i))\, p(g_i(d,e_i)) \triangleleft c_i(d,e_i) \triangleright \delta$$

with action labels $a_i \in AL_\tau$, and

$$\Phi \stackrel{\text{def}}{=} \lambda q{:}D_\Phi \to \mathbb{P}.\lambda d{:}D_\Phi . \sum_{i\in I'} \sum_{e_i:D_i} a_i(f'_i(d,e_i))\, q(g'_i(d,e_i)) \triangleleft c'_i(d,e_i) \triangleright \delta$$

with action labels $a_i \in AL$. $I' \subseteq I$ contains the indices of all visible actions of the implementation. An equality proof for solutions of Ψ and Φ amounts to finding a *state mapping* $h : D_\Psi \to D_\Phi$ such that the following six *matching criteria* are satisfied.

1. The implementation Ψ must be convergent;

2. Internal actions in the implementation preserve the mapping. If for some $i \in I$ it holds that $a_i \equiv \tau$ then

   $c_i(d,e_i) \to h(d) = h(g_i(d,e_i))$;

3. If the implementation can do a visible action then the specification can do a similar one. Let $i \in I'$.

   $c_i(d,e_i) \to c'_i(h(d),e_i)$;

4. If the specification can do a visible action and the focus condition holds, then the implementation can do a similar one. Let $i \in I'$, and let $FC_\Psi(d)$ denote the focus condition.

   $FC_\Psi(d) \wedge c'_i(h(d),e_i) \to c_i(d,e_i)$;

5. The implementation and the specification have the same data parameters on visible actions. Let $i \in I'$.

   $c_i(d, e_i) \to f_i(d, e_i) = f'_i(h(d), e_i)$;

6. If the implementation and specification perform a visible action, then the mapping on the (data) state of the process is altered in a similar way. Let $i \in I'$.

   $c_i(d, e_i) \to h(g_i(d, e_i)) = g'_i(h(d), e_i)$.

If these six criteria are satisfied, then the General Equality Theorem from [11] states how the specification and the implementation are related in *branching bisimulation* semantics.

Theorem 4.1 (*General Equality Theorem*). *Let r and q be solutions of the LPOs Ψ and Φ, respectively. It holds that*

$$r(d) \triangleleft FC_\Psi(d) \triangleright \tau\, r(d) = q(h(d)) \triangleleft FC_\Psi(d) \triangleright \tau\, q(h(d)).$$

4.2 Concatenated buffers form a queue

We connect $n+1$ buffers of capacity 1, and prove that the external behaviour of the whole system equals that of a queue of size $n+1$. The $n+1$ concatenated buffers form an excellent example to demonstrate our main result. However, there are other convenient ways to prove this fact. Various approaches to concatenate queues with queues or buffers can be found in the literature, see e.g. [1, 2, 12, 13]. A basic inductive argument for such a proof is that a queue of size $n > 0$, concatenated with a buffer, should behave as a queue of size $n+1$. None of the references mentioned uses our approach in an implicit or explicit way.

The full benefit of the main theorem will rather be gathered in the verification of processes with a more complicated interaction, and where the combined behaviour of n processes does not so easily imply the behaviour of $n+1$ processes as in our example. A number of more realistic, but for this context too complicated examples were already mentioned in the introduction: [3, 6, 8, 15, 19].

Consider the following specification of a buffer:

sort *BElt*
func $\langle _ , _ \rangle : D \times \mathbf{Bool} \to \mathit{BElt}$
$\mathit{dat} : \mathit{BElt} \to D$
$\mathit{empty} : \mathit{BElt} \to \mathbf{Bool}$
var $b{:}\mathbf{Bool}, d{:}D$
rew $\mathit{dat}(\langle d, b\rangle) = d$
$\mathit{empty}(\langle d, b\rangle) = \neg b$
proc $\mathit{Buf}(k{:}\mathbb{N}, \mathit{be}{:}\mathit{BElt}) = \sum_{d:D} r_k(d)\, \mathit{Buf}(k, \langle d, \mathsf{t}\rangle) \triangleleft \mathit{empty}(\mathit{be}) \triangleright \delta\, +$
$s_{k+1}(\mathit{dat}(\mathit{be}))\, \mathit{Buf}(k, \langle \mathit{dat}(\mathit{be}), \mathsf{f}\rangle) \triangleleft \neg \mathit{empty}(\mathit{be}) \triangleright \delta$

Actually, it would be more precise to give separate specifications of buffers for $k = 1, 1 < k < n$ and $k = n$, but for reasons of brevity we only use one equation. Let r_0 abbreviate *read* and s_{n+1} abbreviate *send*.

The buffer process may read a new data element only when it is empty. After a 'read' action the buffer is full. The buffer process may send a data element only when it is not empty. After a 'send' action the buffer is empty.

Note that $r_i(d)$ and $s_i(d)$ $(1 \leq i \leq n)$ actually stand for actions $r(f(i,d))$ and $s(f(i,d))$, respectively, where f is some pairing function. Let $c_i(d)$ denote $c(f(i,d))$ in a similar way.

We further define communications, and sets H and I for encapsulation and abstraction, respectively:

$$\gamma(r,s) \stackrel{\text{def}}{=} c, \quad H \stackrel{\text{def}}{=} \{r,s\}, \quad I \stackrel{\text{def}}{=} \{c\}.$$

Consider the specification of tables in Section 3.1, and let the sorts *BElt* and *BTable* take the place of D and *DTable*, respectively. Instantiate equation (3) in a similar way, and we obtain

$$S(n{:}\mathbb{N}, bt{:}BTable) = Buf(0, get(0,bt)) \triangleleft eq(n,0) \triangleright (Buf(n, get(n,bt)) \parallel S(n-1, bt)).$$

Let $Par'(n,bt)$ abbreviate $\partial_H(Par(n,bt))$. After application of Theorem 3.7 and encapsulation it follows that:

$$\begin{array}{l} Par'(n,bt) = \\ \quad \sum_{d:D} read(d)\, Par'(n, upd(0, \langle d, \mathsf{t}\rangle, bt)) \triangleleft empty(get(0,bt)) \triangleright \delta + \\ \quad send(dat(get(n,bt)))\, Par'(n, upd(n, \langle dat(get(n,dt)), \mathsf{f}\rangle, bt)) \\ \qquad \triangleleft \neg empty(get(n,bt)) \triangleright \delta + \\ \quad \sum_{0 \leq i < n} c_{i+1}(dat(get(i,bt))) \\ \qquad Par'(n, upd(i, \langle dat(get(i,bt)), \mathsf{f}\rangle, upd(i+1, \langle dat(get(i,bt)), \mathsf{t}\rangle, bt))) \\ \qquad\quad \triangleleft \neg empty(get(i,bt)) \wedge empty(get(i+1,bt)) \triangleright \delta. \end{array}$$

Observe the above equation for Par'. The first main summand denotes the reading of data at the external port (numbered 0), and storage of a corresponding buffer element in the first 'cell' of the queue. The second summand stands for the sending of the data from the last cell of the queue, at port number $n+1$. Note that afterwards cell n is empty. The third summand is the most complicated. It specifies the internal activity of the queue. At each possible action c_{i+1} the buffer element in cell number i is moved on to cell number $i+1$.

Next, abstraction is applied to the equation, and all actions c_{i+1} are renamed to τ. It is not hard to see that the resulting equation is still convergent; If process $\tau_I(Par')$ is restricted to perform only internal actions (τ-steps), the process 'converges' to a state where no element in the queue can move closer to the exit of the queue. This situation is captured by the focus condition:

$$FC(i,n,bt) \stackrel{\text{def}}{=} (0 \leq i < n \wedge \neg empty(get(i,bt)) \rightarrow \neg empty(get(i+1,bt))).$$

The following step is to define a queue of size $n+1$ as we want to have it.

sort *Sequence*
func $emS : \rightarrow Sequence$
$in : D \times Sequence \rightarrow Sequence$
$toe : Sequence \rightarrow D$
$untoe : Sequence \rightarrow Sequence$

$$
\begin{array}{ll}
 & size : Sequence \to \mathbb{N} \\
\textbf{var} & d, e{:}D, s{:}Sequence \\
\textbf{rew} & size(emS) = 0 \\
 & size(in(d,s)) = size(s) + 1 \\
 & toe(in(d, emS)) = d \\
 & toe(in(d, in(e,s))) = toe(in(e,s)) \\
 & untoe(in(d, emS)) = emS \\
 & untoe(in(d, in(e,s))) = in(d, untoe(in(e,s))) \\
\textbf{proc} & Q(n{:}\mathbb{N}, s{:}Sequence) = \sum_{d:D} read(d)\, Q(n, in(d,s)) \lhd size(s) \leq n \rhd \delta + \\
 & \qquad\qquad send(toe(s))\, Q(untoe(s)) \lhd size(s) > 0 \rhd \delta
\end{array}
$$

As long as the queue is not full it may read data at port 0, and store it on top of the internal data sequence. As long as the queue is not empty it is able to send the oldest element (*toe*) from the data sequence, via port $n+1$. A 'send' action should lead to removal of the *toe* of the internal data sequence.

The *Conversion Axiom (CA)* will be used to transform the states of process $\tau_I(Par')$ into the states of Q. Function *convert* converts tables with buffer elements to terms of the simpler data type *Sequence*.

$$
\begin{array}{ll}
\textbf{func} & convert : \mathbb{N} \times \mathbb{N} \times BTable \to Sequence \\
\textbf{var} & k, n{:}\mathbb{N}, bt{:}BTable \\
\textbf{rew} & convert(k,n,bt) = if(k \leq n, \\
 & \qquad if(empty(get(k,bt)), \\
 & \qquad\qquad convert(k+1,n,bt), \\
 & \qquad\qquad in(dat(get(k,bt)), convert(k+1,n,bt))), \\
 & \qquad emS) \qquad\qquad CA
\end{array}
$$

Now we are able to state the major conclusion of this example.

Proposition 4.2. *It holds that:*

$$
\begin{array}{l}
\tau_I(Par'(n,bt)) \lhd FC(i,n,bt) \rhd \tau\, \tau_I(Par'(n,bt)) = \\
\qquad Q(n, convert(0,n,bt)) \lhd FC(i,n,bt) \rhd \tau\, Q(n, convert(0,n,bt)).
\end{array}
$$

Proof. (Sketch.) Here the cones and foci technique, described in the previous section, can be applied sucessfully. We have already stated the convergence of the equation for Par' after renaming the c_i to τ (so for $\tau_I(Par')$). Considering the remaining matching criteria (m.c.) we find the following eight proof obligations:

1. (m.c. 2) $0 \leq i < n \wedge \neg empty(get(i,bt)) \wedge empty(get(i+1,bt)) \to$
   $convert(0,n,bt) =$
   $convert(0,n, upd(i, \langle dat(get(i,bt)), \mathsf{f}\rangle, upd(i+1, \langle dat(get(i,bt)), \mathsf{t}\rangle, bt)))$;
2. (m.c. 3) $empty(get(0,bt)) \to size(convert(0,n,bt)) \leq n$;
3. (m.c. 3) $\neg empty(get(n,bt)) \to size(convert(0,n,bt)) > 0$;
4. (m.c. 4) $FC(i,n,bt) \wedge size(convert(0,n,bt)) \leq n \to empty(get(0,bt))$;

5. (m.c. 4) $FC(i,n,bt) \wedge size(convert(0,n,bt)) > 0 \rightarrow \neg empty(get(n,bt))$;

6. (m.c. 5) $\neg empty(get(n,bt)) \rightarrow dat(get(n,bt)) = toe(convert(0,n,bt))$;

7. (m.c. 6) $empty(get(0,bt)) \rightarrow$
$$convert(0,n,upd(0,\langle d,\mathsf{t}\rangle,bt)) = in(d,convert(0,n,bt));$$

8. (m.c. 6) $\neg empty(get(n,bt)) \rightarrow$
$$convert(0,n,upd(n,\langle dat(get(n,dt)),\mathsf{f}\rangle,bt)) = untoe(convert(0,n,bt)).$$

These formulas are proven quite straightforwardly. As an example, and quite arbitrarily chosen, we prove 1 and 4.

1. The formula
$$k \le i \wedge 0 \le i < n \wedge \neg empty(get(i,bt)) \wedge empty(get(i+1,bt)) \rightarrow$$
$$convert(i-k,n,bt) =$$
$$convert(i-k,n,upd(i,\langle dat(get(i,bt)),\mathsf{f}\rangle,upd(i+1,\langle dat(get(i,bt)),\mathsf{t}\rangle,bt)))$$
is easily proven by induction on k using the axioms CA and TA. Formula 4.2.1 is an instance of the above one (take $k=i$);

4. Proof by contradiction. Assume $\neg empty(get(0,bt)) = \mathsf{t}$, and that the premise of the formula holds. By the Focus Condition (induction on i) it follows for all $0 \le i \le n$ that $\neg empty(get(i,bt)) = \mathsf{t}$. By axiom CA it follows easily that the queue must be full, so of size $n+1$. By the assumption that $size(convert(0,n,bt)) \le n$ we have a contradiction.

By the General Equality Theorem (4.1) this theorem is proven. □

Finally, we want to consider the process $\tau_I(Par')$ where, initially, no data is present in the queue. Therefore we define an empty table with $k+1$ entries (all containing elements $\langle d,\mathsf{f}\rangle$) as follows:

func $init : \mathbb{N} \times D \rightarrow BTable$
var $k{:}\mathbb{N}, d{:}D$
rew $init(k,d) = if(k>0, upd(k,\langle d,\mathsf{f}\rangle,init(k-1,d)), upd(0,\langle d,\mathsf{f}\rangle,emB))$

As an instance of Theorem 4.2 we easily obtain:

$$\tau_I(Par'(n,init(n,d))) = Q(n,emS).$$

The $\tau_I(Par')$ process that starts with no data in its table is equal to the queue process Q that starts with an empty sequence.

References

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

[2] J.A. Bergstra, I. Bethke and A. Ponse. Process algebra with combinators. In E. Börger, Y. Gurevich and K. Meinke, editors, *Proceedings CSL'93, Swansea.* LNCS 280, pages 36–65, Springer-Verlag, 1994.

[3] J.A. Bergstra, J.A. Hillebrand and A. Ponse. Grid protocols based on synchronous communication. *Science of Computer Programming*, 29:199–233, 1997.

[4] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In B. Jonsson and J. Parrow, editors, *Proceedings Concur'94*, Uppsala, Sweden. LNCS 836, pages 401–416, Springer-Verlag, 1994.

[5] M.A. Bezem and J.F. Groote. A correctness proof of a One Bit Sliding Window Protocol in μCRL. *The Computer Journal*, 37(4):289–307, 1994.

[6] L.-å. Fredlund, J.F. Groote and H.P. Korver. Formal verification of a Leader Election Protocol in process algebra. *Theoretical Computer Science*, 177:459–486, 1997.

[7] J.F. Groote and H.P. Korver. Correctness proof of the Bakery Protocol in μCRL. In A. Ponse, C. Verhoef and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes.* Workshops in Computing, pages 63–86, Springer-Verlag, 1994.

[8] J.F. Groote, F. Monin and J. Springintveld. A computer checked algebraic verification of a Distributed Summing Protocol. Computer Science Report 97/14, Department of Mathematics and Computer Science. Eindhoven University of Technology, 1997.

[9] 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.* Workshops in Computing, pages 26–62, Springer-Verlag, 1994.

[10] 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, *Proceedings of the International Workshop on Semantics of Specification Languages.* Workshops in Computing, pages 232–251, Springer-Verlag, 1994.

[11] J.F Groote and J. Springintveld. Focus points and convergent process operators. Technical Report 142, Logic Group Preprint Series, Department of Philosophy. Utrecht University, 1995. A draft version appeared in the *Proceedings of the AMAST Workshop on Real-Time Systems; Models and Proofs*. Bordeaux, 1995.

[12] R.J. van Glabbeek and F.W. Vaandrager. Modular specifications in process algebra. *Theoretical Computer Science*, 113(2):294–348, 1993.

[13] H.P. Korver. *Protocol Verification in μCRL.* PhD thesis. University of Amsterdam. 1994.

[14] R.P. Kurshan, M. Merritt, A. Orda and S.R. Sachs. A structural linearization principle for processes. In C. Courcoubetis, editor, *Proceedings of CAV'93.* LNCS 697, pages 491–504, Springer-Verlag, 1993.

[15] H.P. Korver and J. Springintveld. A computer-checked verification of Milner's Scheduler. In *Proceedings of the International Symposium on Theoretical Aspects of Computer Software (TACS'94),* Sendai, Japan. LNCS 789, Springer-Verlag, 1994.

[16] R. Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[17] The μCRL tool set. http://www.cwi.nl/˜mcrl.

[18] A.P. Sistla and S.M. German. Reasoning with many processes. *Proceedings of LICS '87*, Ithaca, New York. Pages 138–152, 1987.

[19] C. Shankland and M.B. van der Zwaag. The Tree Identify Protocol of IEEE 1394 in μCRL. *Formal Aspects of Computing,* 10:509–531, 1998.

[20] B.C. Thompson and J.V. Tucker. Equational Specification of Synchronous Concurrent Algorithms and Architectures (2nd ed.). Report CSR 15-94, University of Wales, Swansea, 1994.