The Parallel Composition of Uniform Processes with Data

by

J.F. Groote and J. van Wamel

00/06

ISSN 0926-4515

All rights reserved
editors: prof.dr. J.C.M. Baeten
         prof.dr. P.A.J. Hilbers

Reports are available at:
http://www.win.tue.nl/win/cs
The Parallel Composition of Uniform Processes with Data

Jan Friso Groote\textsuperscript{1,2} & Jos van Wamel\textsuperscript{1}

\textsuperscript{1} CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

\textsuperscript{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 $S(n, dt) = P(0, get(0, dt) \& eq(n, 0) \& (P(n, get(n, dt)) \parallel S(n-1, dt))$. 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 $\mu$CRL (\textit{micro} Common Representation Language [9]) this can be described, using recursion and operators for parallelism.

Several benchmark verifications in $\mu$CRL 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:

$$S(n; N) = P(0) \& eq(n, 0) \& (P(n) \parallel S(n-1))$$

(1)

where the function $eq$ is used for an equality test, and the expression $x \& b \& y$ 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 $P(k)$.
1 INTRODUCTION

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 $\mu$CRL 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 $\mu$CRL 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 $\mu$CRL.

Acknowledgements. We thank Wan Fokkink, François Monin and Alban Ponse for their comments.
2 \(\mu\text{CRL}\)

The axiom system \(p\text{CRL}\) (\textit{pico CRL}, which is \(\mu\text{CRL}\) 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 \(\mu\text{CRL}\). Also the concept of linear processes is introduced, as well as the Recursive Specification Principle.

2.1 Axioms for \(p\text{CRL}\)

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 \(\delta\), and silent step by \(\tau\). We use \(a, b, c, \ldots\) to denote elements from either \(AL\) or \(AL \cup \{\tau\}\) (\(AL\tau\)), \(A\) or \(A \cup \{\delta, \tau\}\) (\(A\delta\tau\)).

Table 1 lists the axioms of \(p\text{CRL}\). 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 | \(x + (y + x) = (x + y) + z\) | SUM3 | \(\sum X = \sum X + \sum d\) |
| A3 | \(x + x = x\) | SUM4 | \(\sum_{d:D} (Xd + Yd) = \sum X + \sum Y\) |
| A4 | \((x + y)z = xz + yz\) | SUM5 | \((\sum X) \cdot x = \sum_{d:D} (Xd \cdot x)\) |
| A5 | \((x \cdot y)z = x \cdot (y \cdot z)\) | SUM11 | \((\forall d \in D \ Xd = Yd) \rightarrow \sum X = \sum Y\) |
| A6 | \(x + \delta = x\) | C1 | \(x \wedge t = y\) |
| A7 | \(\delta \cdot x = \delta\) | C2 | \(x \wedge f = y\) |

Table 1: Axioms of \(p\text{CRL}\)

Data types in \(\mu\text{CRL}\) 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 \(\mu\text{CRL}\) we assume sort \textbf{Bool} for booleans.

Sort \textbf{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 \(\_ \leftarrow \_ \rightarrow \_\). For sort \textbf{Bool} we assume connectives \(\neg, \wedge, \vee, \rightarrow\) with straightforward interpretations, and for the construction of proofs we (implicitly) use the proof theory for \(\mu\text{CRL}\) [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 \(\phi\) holds for both \(b = t\) and \(b = f\) then \(\phi\) 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 $\text{Bool2}$). We also have to assume that $t$ and $f$ are different ($\text{Bool1}$), 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 $\mu\text{CRL}$, 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 $\text{SUMx}$ 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 $\alpha$-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 $\text{C1}$ and $\text{C2}$ are used implicitly. As a rule, brackets are omitted from boolean expressions according to the convention that $\neg$ binds stronger than $\lor$ and $\land$, and that these in turn bind stronger than $\rightarrow$.

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

\[ \sum_{d:D} Xd < eq(d, e) \triangleright \delta = X e \quad (SE) \]

**Proof.** See e.g. [7]. \hfill \Box

### 2.2 Parallelism

The axioms of $\mu\text{CRL}$ are the axioms of $p\text{CRL}$, 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 $\partial_H$ and $\tau_I$, and not the general operators for renaming [4].)

For communication we have a binary function $\gamma$, 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 $\text{CF}$. By definition, the function $\gamma$ 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* $\parallel$, the *left merge* $\llbracket$ and the *communication merge* $\llbracket$. The process $p \parallel q$ 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 || q is as p ∥ q, but the first action that is performed comes from p. Process p ∥ q is as p ∥ q, but the first action is a communication between p and q. We will tacitly assume associativity and commutativity of ∥ and ∥, 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 τI are particularly useful for hiding communication actions, by renaming them to τ.

The various operators of Σ(μCRL) are listed in order of decreasing binding strength:

\[ \{a · p, ∥, ||, \} \sum_{d ∊ D} + \]

Brackets are omitted from expressions according to this convention.

| SUM6 | (∑X) ∥ x = ∑_{d ∊ D}(Xd ∥ x) |
| SUM7 | (∑X) | x = ∑_{d ∊ D}(Xd | x) |
| SUM7' | x ∥ (∑X) = ∑_{d ∊ D}(x | Xd) |
| SUM8 | ∂H(∑X) = ∑_{d ∊ D}∂H(Xd) |
| SUM9 | τI(∑X) = ∑_{d ∊ D}τI(Xd) |

| CM1 | x ∥ y = x ∥ y + x + y | CD1 δ | a = δ |
| CM2 | a ∥ x = a ∥ x | CD2 a | δ = δ |
| CM3 | a ∥ x ∥ y = a ∥ (x ∥ y) | CT1 x = δ |
| CM4 | (x + y) ∥ z = x + y | CT2 z | τ = δ |
| CM5 | a ∥ b ∥ x = (a ∥ b) ∥ x |
| CM6 | a ∥ b ∥ x = (a ∥ b) ∥ x | D1 δH(c(d)) = c(d) if c ∉ H |
| CM7 | a ∥ b ∥ y = (a ∥ b) ∥ (y ∥ x) | D2 δH(c(d)) = δ if c ∉ H |
| CM8 | (x + y) ∥ z = x + y | D3 δH(x + y) = δH(x) + δH(y) |
| CM9 | x ∥ (y + z) = x ∥ y + z |
| DD | δH(δ) = δ |
| DT | δH(τ) = τ |
| TID | τI(δ) = δ |
| TIT | τI(τ) = τ |

Table 2: Axioms for parallelism and renaming of μCRL, where a, b ∊ Aδ, and c, c′ ∊ 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) $\Psi$ is an expression of the form:

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

for some finite index set $I$, action labels $a_i \in AL_T$, data types $D_i$ and $D_{a_i}$, functions $f_i : D \to D_i$, $g_i : D \to D_i \to D$ and $c_i : D \to D_i \to \text{Bool}$. $P$ 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 $\Psi$ one can easily derive its associated Linear Process Equation $p(d) = \Psi p d$. The other way, from LPE to LPO, is also easy.

Definition 2.3. A linear process operator $\Psi$ 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 $\tau$-paths from certain states of the process to itself. Cyclic $\tau$-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 $\Psi$ has at least one fixed point, i.e. there exists a $p : D \to P$ 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 $p : D \to P$ and $q : D \to P$ 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:

$$P(k: \mathbb{N}, d: D) = \sum_{i \in I} \sum_{e_i: E_i} a_i(f_i(k, d, e_i)) P(k, g_i(k, d, e_i)) \prec c_i(k, d, e_i) \triangleright \delta$$
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 \( if : Bool \times D \times D \rightarrow D \) reflecting \( if \ - \ then \ - \ else \). In the sequel we also use \( eq : D_a \times D_a \rightarrow Bool \), expressing equality of the elements in \( D_a \) (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).

\[
\text{sort } DTable \\
\text{func} \quad emD : \rightarrow DTable \\
\quad upd : N \times D \times DTable \rightarrow DTable \\
\quad get : N \times DTable \rightarrow D \\
\text{var} \quad n,m:N, d:D, dt:DTable \\
\text{rew} \quad get(n, upd(m, d, dt)) = if(eq(n, m), d, get(n, dt)) \\
\]

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

\[ S(n;N, dt:DTable) = P(0, get(0, dt)) \triangleq eq(n, 0) \triangleright (P(n, get(n, dt)) \parallel S(n - 1, dt)) \]  

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. \( m > n = t \rightarrow S(n, dt) = S(n, upd(m, d, dt)); \)

2. \( k_1 \neq k_2 \rightarrow get(n, upd(k_1, d_1, upd(k_2, d_2, dt))) = get(n, upd(k_2, d_2, upd(k_1, d_1, dt))) \).

**Proof.** The first fact is proven by induction on \( n \). The second follows from axiom TA. \( \square \)

**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 LINEARISATION OF PARALLEL PROCESSES

3. \( P(n+1, \text{get}(n+1, dt)) \)

\[
\| S(n, \text{upd}(k_1, g_{i_1}(k_1, \text{get}(k_1, dt), e_{i_1}), \text{upd}(k_2, g_{i_2}(k_2, \text{get}(k_2, dt), e_{i_2}), dt)) \| = S(n+1, \text{upd}(k_1, g_{i_1}(k_1, \text{get}(k_1, dt), e_{i_1}), \text{upd}(k_2, g_{i_2}(k_2, \text{get}(k_2, dt), e_{i_2}), dt));
\]

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

\[
= S(n+1, \text{upd}(n+1, g_{i_1}(n+1, \text{get}(n+1, dt), e_{i_1}), \text{upd}(k_2, g_{i_2}(k_2, \text{get}(k_2, dt), e_{i_2}), dt)).
\]

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

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 \( \text{upd} \) and \( \text{get} \).

Lemma 3.3 says that process \( \text{Par} \) 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 \( \text{Par} \) may also do some internal action \( \gamma(a_{i_1}, a_{i_2}) \) and exchange data \( (k_1, \text{get}(k_1, dt), e_{i_1}) \) 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 \( \text{Par} \) 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).

\[
\text{Par}(n: N, dt: DTable) = \sum_{i \in I} \sum_{k: N} \sum_{e_i: E_i} a_i(k, \text{get}(k, dt), e_i) \text{Par}(n, \text{upd}(k, g_i(k, \text{get}(k, dt), e_i), dt)) \]

\[
\land c_i(k, \text{get}(k, dt), e_i) \land k \leq n \lor \delta + \\
\sum_{i \in I} \sum_{k_1: N} \sum_{k_2: N} \sum_{e_{i_1}: E_{i_1}} \sum_{e_{i_2}: E_{i_2}} \gamma(a_{i_1}, a_{i_2}) (f_{i_1}(k_1, \text{get}(k_1, dt), e_{i_1})) (f_{i_2}(k_2, \text{get}(k_2, dt), e_{i_2})) \land k_1 > k_2 \land k_1 \leq n \lor \delta.
\]

**Proof.** It is sufficient to show that the equation above holds, with all occurrences of \( \text{Par} \) replaced by \( S \):

\[
S(n, dt) = \sum_{i \in I} \sum_{k: N} \sum_{e_i: E_i} a_i(k, \text{get}(k, dt), e_i) S(n, \text{upd}(k, g_i(k, \text{get}(k, dt), e_i), dt)) \]

\[
\land c_i(k, \text{get}(k, dt), e_i) \land k \leq n \lor \delta + \\
\sum_{i \in I} \sum_{k_1: N} \sum_{k_2: N} \sum_{e_{i_1}: E_{i_1}} \sum_{e_{i_2}: E_{i_2}} \gamma(a_{i_1}, a_{i_2}) (f_{i_1}(k_1, \text{get}(k_1, dt), e_{i_1})) (f_{i_2}(k_2, \text{get}(k_2, dt), e_{i_2})) \land k_1 > k_2 \land k_1 \leq n \lor \delta.
\]

We do this with induction on \( n \).
3 LINEARISATION OF PARALLEL PROCESSES

Base, $n = 0$.

\[ S(0, dt) \]

\[ \text{(1)} P(0, get(0, dt)) \]

\[ \text{(2)} \sum_{i \in I} \sum_{e_i \in E_i} a_i(f_i(0, get(0, dt), e_i))P(0, g_i(0, get(0, dt), e_i)) \]

\[ \text{(3)} \sum_{i \in I} \sum_{e_i \in E_i} a_i(f_i(0, get(0, dt), e_i)) \]

\[ \sum_{i \in I} \sum_{e_i \in E_i} a_i(f_i(0, get(0, dt), e_i))P(0, get(0, upd(0, g_i(0, get(0, dt), e_i), dt))) \]

\[ \sum_{i \in I} \sum_{e_i \in E_i} a_i(f_i(0, get(0, dt), e_i))S(0, upd(0, g_i(0, get(0, dt), e_i), dt)) \]

\[ \sum_{i \in I} \sum_{e_i \in E_i} a_i(f_i(0, get(0, dt), e_i)) \]

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 $\delta$. 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:

\[ S(n + 1, dt) \]

\[ \text{(1)} P(n + 1, get(n + 1, dt)) \]

\[ \text{(2)} \sum_{i \in I} \sum_{e_i \in E_i} a_i(f_i(n + 1, get(n + 1, dt), e_i))P(n + 1, g_i(n + 1, get(n + 1, dt), e_i)) \]

\[ \text{(3)} \sum_{i \in I} \sum_{e_i \in E_i} a_i(f_i(n + 1, get(n + 1, dt), e_i))S(n + 1, upd(n + 1, g_i(n + 1, get(n + 1, dt), e_i), dt)) \]

\[ \sum_{i \in I} \sum_{e_i \in E_i} a_i(f_i(n + 1, get(n + 1, dt), e_i)) \]

We analyse the terms $A, B$ and $C$ separately.

**A.** By equation (2) we have

\[ P(n + 1, get(n + 1, dt)) \parallel S(n, dt) \]

\[ = \sum_{i \in I} \sum_{e_i \in E_i} a_i(f_i(n + 1, get(n + 1, dt), e_i)) \]

\[ (n + 1, g_i(n + 1, get(n + 1, dt), e_i)) \parallel S(n, dt) \]

\[ \text{(3.2.1.4E)} \]

\[ \sum_{i \in I} \sum_{e_i \in E_i} a_i(f_i(n + 1, get(n + 1, dt), e_i)) \]

\[ S(n + 1, upd(n + 1, g_i(n + 1, get(n + 1, dt), e_i), dt)) \]

\[ \sum_{i \in I} \sum_{e_i \in E_i} a_i(f_i(n + 1, get(n + 1, dt), e_i)) \]

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

\[
S(n, dt) \parallel P(n + 1, \text{get}(n + 1, dt)) = \sum_{i \in I} \sum_{e_i : E_i} a_i(f_i(k, \text{get}(k, dt), e_i)) \\
(S(n, \text{upd}(k, g_i(k, \text{get}(k, dt), e_i), dt)) \parallel P(n + 1, \text{get}(n + 1, dt)))
\]

\[
\quad \gamma(a_{i_1}, a_{i_2})(f_{i_1}(k, \text{get}(k, dt), e_{i_1})) \\
\qquad (P(n + 1, g_{i_1}(n + 1, \text{get}(n + 1, dt), e_{i_1})))
\]

\[
\quad \text{eq}(f_{i_1}(n + 1, \text{get}(n + 1, dt), e_{i_1}), f_{i_2}(k, \text{get}(k, dt), e_{i_2})) \wedge \\
\quad k_1 > k_2 \wedge k_1 \leq n + 1 + \delta
\]

\[3.2.\{2,3\} = \sum_{i \in I} \sum_{e_i : E_i} a_i(f_i(k, \text{get}(k, dt), e_i))
\]

\[
S(n + 1, \text{get}(n + 1, dt)) \parallel P(n + 1, \text{get}(n + 1, dt)) \\
\quad \gamma(a_{i_1}, a_{i_2})(f_{i_1}(k, \text{get}(k, dt), e_{i_1})) \\
\qquad (P(n + 1, g_{i_1}(n + 1, \text{get}(n + 1, dt), e_{i_1})))
\]

\[
\quad \text{eq}(f_{i_1}(n + 1, \text{get}(n + 1, dt), e_{i_1}), f_{i_2}(k, \text{get}(k, dt), e_{i_2})) \wedge \\
\quad k_1 > k_2 \wedge k_1 \leq n + 1 + \delta
\]

C. We may again use the induction hypothesis. We use the fact (axiom CP) that the communication of two actions is not a \( \delta \) 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 \( i_2, k_2 \).

\[
P(n + 1, \text{get}(n + 1, dt)) \parallel S(n, dt)
\]

\[
\sum_{i_1 \in I} \sum_{i_2 \in I} \sum_{k_1 : E_i} \sum_{e_{i_1}, e_{i_2}} \gamma(a_{i_1}, a_{i_2})(f_{i_1}(k_1, \text{get}(k_1, dt), e_{i_1}))
\]

\[
\quad (P(n + 1, g_{i_1}(n + 1, \text{get}(n + 1, dt), e_{i_1})))
\]

\[
\quad \text{eq}(f_{i_1}(n + 1, \text{get}(n + 1, dt), e_{i_1}), f_{i_2}(k_2, \text{get}(k_2, dt), e_{i_2})) \wedge \\
\quad k_2 \leq n - 1 + \delta
\]

\[
3.2.4, SE = \sum_{i_1 \in I} \sum_{i_2 \in I} \sum_{k_1 : E_i} \sum_{e_{i_1}, e_{i_2}} \gamma(a_{i_1}, a_{i_2})(f_{i_1}(k_1, \text{get}(k_1, dt), e_{i_1}))
\]

\[
\quad (P(n + 1, g_{i_1}(n + 1, \text{get}(n + 1, dt), e_{i_1})))
\]

\[
\quad \text{eq}(f_{i_1}(n + 1, \text{get}(n + 1, dt), e_{i_1}), f_{i_2}(k_2, \text{get}(k_2, dt), e_{i_2})) \wedge \\
\quad k_1 > k_2 \wedge k_1 = n + 1 + \delta
\]

Now take \( a + b_1 \) and \( b_2 + c \), 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 \( < \subseteq (\mathbb{N} \times D) \times (\mathbb{N} \times D) \), such that if \( c_i(k, d, e_i) = t \) 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:

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

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 \( \text{eq}(k_1, k_2) \land \text{eq}(d_1, d_2) \).

Now consider equation (4). The second main summand of \( \text{Par} \) can never ‘start’ with a \( \tau \)-step, so only the first has to be taken into account. Using \( \prec \) it is straightforward to see that convergence is a fact. \( \square \)

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 \( \text{Par} \) in (4). Using Lemma 3.4 CL-RSP expresses that there is at most one solution for \( \text{Par} \). Consequently, also equation (3) has at most one solution. \( \square \)

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

\[
\text{Par}(n, \text{ upd}(k_1, g_i, (k_1, \text{get}(k_1, dt), e_i)), \text{ upd}(k_2, g_i, \text{get}(k_2, dt), e_i)), dt)) = \\
\text{Par}(n, \text{ upd}(k_2, g_i, \text{get}(k_2, dt), e_i)), \text{ upd}(k_1, g_i, \text{get}(k_1, dt), e_i)), dt)).
\]

Proof. We abbreviate the term \( g_i, (k_1, \text{get}(k_1, dt), e_i) \) by \( d_1 \) and \( g_i, (k_2, \text{get}(k_2, dt), e_i) \) by \( d_2 \). As \( \text{Par}(n, dt) = S(n, dt) \) it suffices to prove

\[
S(n, \text{ upd}(k_1, d_1, \text{ upd}(k_2, d_2, dt))) = S(n, \text{ upd}(k_2, d_2, \text{ upd}(k_1, d_1, dt))).
\]

We prove this fact by induction on \( n \).

Base, \( n = 0 \). We have that

\[
S(0, \text{ upd}(k_1, d_1, \text{ upd}(k_2, d_2, dt))) \overset{(3)}{=} P(0, \text{ get}(0, \text{ upd}(k_1, d_1, \text{ upd}(k_2, d_2, dt)))) \overset{3 \ref{12}}{=} P(0, \text{ get}(0, \text{ upd}(k_2, d_2, \text{ upd}(k_1, d_1, dt)))).
\]

Induction step. Using a similar argument and the induction hypothesis, this part of the proof also follows easily. \( \square \)
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 $\text{Par}$ in the (convergent) equation below; so for all $n: \mathbb{N}$ and $dt:DTable$, $S(n, dt) = \text{Par}(n, dt)$, where the set $I$ and the functions $f_i, g_i$ and $c_i$ are those that occur in equation (2).

$$\text{Par}(n:N, dt:DTable) = \sum_{i \in I} \sum_{k:N} \sum_{e_i:E_i} a_i(f_i(k, get(k, dt), e_i)) \text{Par}(n, upd(k, g_i(k, get(k, dt), e_i), dt))$$

$$\leq c_i(k, get(k, dt), e_i) \land k \leq n \cdot \delta + \sum_{i \in I} \sum_{k_2 \in I \leq i} \sum_{k_1:N} \sum_{e_i:E_i} \sum_{e_i:E_i} \sum_{e_i:E_i} \gamma(a_i, c_i, f_i(k_1, get(k_1, dt), e_i)) \text{Par}(n, upd(k_1, g_i(k_1, get(k_1, dt), e_i), dt))$$

$$\leq c_i(k_1, get(k_1, dt), e_i) \land c_i(k_2, get(k_2, dt), e_i) \land \text{eq}(f_i(k_1, get(k_1, dt), e_i), f_i(k_2, get(k_2, dt), e_i)) \land \neg \text{eq}(k_1, k_2) \land k_1 \leq n \land k_2 \leq n \cdot \delta.$$

**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:

$$P_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2}) = \gamma(a_{i_1}, a_{i_2}, f_i(k_1, get(k_1, dt), e_i)) \text{Par}(n, upd(k_1, g_i(k_1, get(k_1, dt), e_i), dt))$$

$$C_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2}) = c_i(k_1, get(k_1, dt), e_i) \land c_i(k_2, get(k_2, dt), e_i) \land \text{eq}(f_i(k_1, get(k_1, dt), e_i), f_i(k_2, get(k_2, dt), e_i)) \land \neg \text{eq}(k_1, k_2) \land k_1 \leq n \land k_2 \leq n \cdot \delta.$$

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

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

$$\sum_{i_1 \in I} \sum_{i_2 \in I} \sum_{k_1:N} \sum_{k_2:N} \sum_{e_{i_1}:E_{i_1}} \sum_{e_{i_2}:E_{i_2}} P_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2})$$

$$\leq C_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2}) \land k_1 > k_2 \land k_1 \leq n \cdot \delta.$$

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

$$\sum_{i_1 \in I} \sum_{i_2 \in I} \sum_{k_1:N} \sum_{k_2:N} \sum_{e_{i_1}:E_{i_1}} \sum_{e_{i_2}:E_{i_2}} P_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2})$$

$$\leq C_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2}) \land k_1 > k_2 \land \neg \text{eq}(k_1, k_2) \land k_1 \leq n \land k_2 \leq n \cdot \delta +$$

$$\sum_{i_1 \in I} \sum_{i_2 \in I} \sum_{i_2 \geq i_1} \sum_{k_1:N} \sum_{k_2:N} \sum_{e_{i_1}:E_{i_1}} \sum_{e_{i_2}:E_{i_2}} P_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2})$$

$$\leq C_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2}) \land k_1 \geq k_2 \land \neg \text{eq}(k_1, k_2) \land k_1 \leq n \land k_2 \leq n \cdot \delta.$$

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:

$$\sum_{i_1 \in I} \sum_{i_2 \in I} \sum_{i_2 < i_1} \sum_{k_1:N} \sum_{k_2:N} \sum_{e_{i_1}:E_{i_1}} \sum_{e_{i_2}:E_{i_2}} P_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2})$$

$$\leq C_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2}) \land k_1 > k_2 \land \neg \text{eq}(k_1, k_2) \land k_1 \leq n \land k_2 \leq n \cdot \delta +$$

$$\sum_{i_1 \in I} \sum_{i_2 \in I} \sum_{i_2 < i_1} \sum_{k_1:N} \sum_{k_2:N} \sum_{e_{i_1}:E_{i_1}} \sum_{e_{i_2}:E_{i_2}} P_{i_1 i_2}(k_1, k_2, e_{i_1}, e_{i_2})$$

$$\leq C_{i_2 i_1}(k_2, k_1, e_{i_2}, e_{i_1}) \land k_2 \geq k_1 \land \neg \text{eq}(k_2, k_1) \land k_2 \leq n \land k_1 \leq n \cdot \delta.$$
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

$$\sum_{i_1 \in I} \sum_{i_2 \in I \cap i_2 \leq i_1} \sum_{k_1 \in N} \sum_{k_2 \in N} \sum_{e_{i_1}} \sum_{e_{i_2}} P_{i_1i_2}(k_1, k_2, e_{i_1}, e_{i_2})$$

$$\cdot C_{i_1i_2}(k_1, k_2, e_{i_1}, e_{i_2}) \land \neg eq(k_1, k_2) \land k_1 \leq n \land k_2 \leq n \oplus \delta,$$

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 $\tau$-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 ($\tau$-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.
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.

\[ \psi \overset{\text{def}}{=} \lambda p : D_\psi \rightarrow P. \lambda d : D_\psi . \sum_{i \in I} \sum_{e_i \in D_i} a_i(f_i(d, e_i)) p(g_i(d, e_i)) \cdot c_i(d, e_i) \cdot \delta \]

with action labels \( a_i \in AL_L \), and

\[ \phi \overset{\text{def}}{=} \lambda q : D_\phi \rightarrow P. \lambda d : D_\phi . \sum_{i \in I'} \sum_{e_i \in D_i} a_i(f_i'(d, e_i)) q(g_i'(d, e_i)) \cdot c_i'(d, e_i) \cdot \delta \]

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 \( h : D_\psi \rightarrow D_\phi \) 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
   \[ c_i(d, e_i) \rightarrow h(d) = h(g_i(d, e_i)) ; \]
3. If the implementation can do a visible action then the specification can do a similar one. Let \( i \in I' \).
   \[ c_i(d, e_i) \rightarrow c_i'(h(d), e_i) ; \]
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.
   \[ FC_\psi(d) \land c_i'(h(d), e_i) \rightarrow c_i(d, e_i) ; \]
5. The implementation and the specification have the same data parameters on visible actions. Let \( i \in I' \).

\[
c_i(d, e_i) \rightarrow f_i(d, e_i) = f'_i(h(d), e_i);
\]

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 \( i \in I' \).

\[
c_i(d, e_i) \rightarrow h(g_i(d, e_i)) = g'_i(h(d), e_i).
\]

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 \( \Psi \) and \( \Phi \), respectively. It holds that

\[
r(d) \circ FC_{\Psi}(d) \triangleright \tau r(d) = q(h(d)) \circ FC_{\Phi}(d) \triangleright \tau q(h(d)).
\]

### 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** \( \textit{BElt} \)

**func** \( \langle \_, \_ \rangle : D \times \textit{Bool} \rightarrow \textit{BElt} \)

\( \textit{dat} : \textit{BElt} \rightarrow D \)

\( \textit{empty} : \textit{BElt} \rightarrow \textit{Bool} \)

**var** \( b : \textit{Bool}, d : D \)

**rew**

\( \textit{dat}(\langle d, b \rangle) = d \)

\( \textit{empty}(\langle d, b \rangle) = \neg b \)

**proc** \( \textit{Buf}(k : \mathbb{N}, \textit{be} : \textit{BElt}) = \sum_{d, D} r_k(d) \textit{Buf}(k, \langle d, t \rangle) \circ \textit{empty}(\textit{be}) \triangleright \delta + s_{k+1}(\textit{dat}(\textit{be})) \textit{Buf}(k, \langle \textit{dat}(\textit{be}), f \rangle) \circ \neg \textit{empty}(\textit{be}) \triangleright \delta \)

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 \( \textit{read} \) and \( s_{n+1} \) abbreviate \( \textit{send} \).
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:

\[
\gamma(r, s) \triangleq c, \quad H \triangleq \{r, s\}, \quad I \triangleq \{c\}.
\]

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

\[
S(n:N, bt:BTable) = Buf(0, get(0, bt)) \triangleq eq(n, 0) \triangleright (Buf(n, get(n, bt)) \parallel S(n - 1, bt)).
\]

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

\[
Par'(n, bt) = \\
\sum_{d:D} read(d) \; Par'(n, upd(0, (d, t), bt)) \triangleq empty (get(0, bt)) \triangleright \delta + \\
\sum_{i\in\{0,\ldots,n\}} \; c_{i+1}(dat(get(i, bt))) \; Par'(n, upd(i, (dat(get(i, bt)), f), bt)) \triangleq \neg empty (get(n, bt)) \triangleright \delta + \\
\sum_{0 \leq i < n} c_{i+1}(dat(get(i, bt))) \\
\triangleq \neg empty (get(i, bt)) \land empty (get(i + 1, bt)) \triangleright \delta.
\]

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 \( \tau \). It is not hard to see that the resulting equation is still convergent; If process \( \tau_f(Par') \) is restricted to perform only internal actions (\( \tau \)-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:

\[
FC(i, n, bt) \triangleq (0 \leq i < n \land \neg empty (get(i, bt)) \rightarrow \neg empty (get(i + 1, bt))).
\]

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
size : Sequence → N

var d, e : D, s : Sequence

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)))

proc Q(n : N, s : Sequence) = \sum_{d \in D} \text{read}(d) Q(n, in(d, s)) \cdot size(s) \leq n \cdot \delta + 
\text{send}(toe(s)) Q(untoe(s)) \cdot size(s) > 0 \cdot \delta

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.

func convert : N \times N \times BTable → Sequence

var k, n : N, bt : BTable

rew convert(k, n, bt) = if(k \leq n,
  if(empty(get(k, bt)),
    convert(k + 1, n, bt),
    in(dat(get(k, bt)), convert(k + 1, n, bt))),
  emS)

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

Proposition 4.2. It holds that:

\[ \tau_I(Par'(n, bt)) \cdot FC(i, n, bt) \cdot \tau \tau_I(Par'(n, bt)) = \]

\[ Q(n, convert(0, n, bt)) \cdot FC(i, n, bt) \cdot \tau Q(n, convert(0, n, bt)). \]

Proof. (Sketch.) Here the cones and foci technique, described in the previous section, can be applied successfully. We have already stated the convergence of the equation for \( Par' \) after renaming the \( c \) to \( \tau \) (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 \land \neg empty(get(i, bt)) \land empty(get(i + 1, bt)) \rightarrow 
   convert(0, n, bt) = 
   convert(0, n, upd(i, {dat(get(i, bt)), f}, upd(i + 1, {dat(get(i, bt)), t}, bt)));

2. (m.c. 3) empty(get(0, bt)) \rightarrow size(convert(0, n, bt)) \leq n;

3. (m.c. 3) \neg empty(get(n, bt)) \rightarrow size(convert(0, n, bt)) > 0;

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

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

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

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

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

1. The formula

   $k \leq i \land 0 \leq i < n \land \neg empty(get(i, bt)) \land empty(get(i + 1, bt)) \rightarrow$
   
   $convert(i - k, n, bt) =$
   
   $convert(i - k, n, upd(i, (dat(get(i, bt)), f), upd(i + 1, (dat(get(i, bt)), t), bt)))$

   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 $\neg empty(get(0, bt)) = t$, and that the premise of the formula holds. By the Focus Condition (induction on $i$) it follows for all $0 \leq i \leq n$ that $\neg empty(get(i, bt)) = t$. By axiom $CA$ it follows easily that the queue must be full, so of size $n + 1$. By the assumption that $size(convert(0,n, bt)) \leq n$ 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 $(d, f)$) as follows:

```
func init : N x D -> BTable
var k:N, d:D
rew init(k, d) = if(k > 0, upd(k, (d,f), init(k - 1, d)), upd(0, (d,f), 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


REFERENCES


Computing Science Reports

In this series appeared:

96/01 M. Voorhoeve and T. Basten
96/02 P. de Bra and A. Aerts
96/03 W.M.P. van der Aalst
96/05 T. Basten and W.M.P. v.d. Aalst
96/06 W.M.P. van der Aalst and T. Basten
96/07 M. Voorhoeve
96/08 A.T.M. Aerts, P.M.E. De Bra, J.T. de Munk
96/09 F. Dignum, H. Weigand, E. Verharen
96/10 R. Bloo, H. Geuvers
96/11 T. Laan
96/12 F. Kamareddine and T. Laan
96/13 T. Borghuis
96/14 S.H.J. Bos and M.A. Reniers
96/15 M.A. Reniers and J.J. Vereijken
96/17 E. Boiten and P. Hoogenrijk
96/18 P.D.V. van der Stok
96/19 M.A. Reniers
96/20 L. Feijs
96/21 L. Bijlma and R. Nederpelt
96/22 M.C.A. van de Graaf and G.J. Houben
96/23 W.M.P. van der Aalst
96/24 M. Voorhoeve and W. van der Aalst
96/25 M. Vaccarì and R.C. Backhouse
97/02 J. Hoekman and O. v. Roosmalen
97/03 J. Blanco and A. v. Deursen
97/04 J.C.M. Baeten and J.A. Bergstra
97/05 J.C.M. Baeten and J.J. Vereijken
97/06 M. Franssen
97/07 J.C.M. Baeten and J.A. Bergstra
97/08 P. Hoogendijk and R.C. Backhouse

Department of Mathematics and Computing Science
Eindhoven University of Technology

Process Algebra with Autonomous Actions, p. 12.
Multi-User Publishing in the Web: DreSS, A Document Repository Service Station, p. 12.
Life-Cycle Inheritance A Petri-Net-Based Approach, p. 18.
Structural Petri Net Equivalence, p. 16.
AUTOMATH and Pure Type Systems, p. 30.
A Correspondence between Nuprl and the Ramified Theory of Types, p. 12.
Priorean Tense Logics in Modal Pure Type Systems, p. 61
The \( I^2 \) C-bus in Discrete-Time Process Algebra, p. 25.
Completeness in Discrete-Time Process Algebra, p. 139.
Nested collections and polytypism, p. 11.
Real-Time Distributed Concurrency Control Algorithms with mixed time constraints, p. 71.
Static Semantics of Message Sequence Charts, p. 71
Algebraic Specification and Simulation of Lazy Functional Programs in a concurrent Environment, p. 27.
Designing Effective Workflow Management Processes, p. 22.
Structural Characterizations of sound workflow nets, p. 22.
Conservative Adoption of Workflow, p.22
Deriving a systolic regular language recognizer, p. 28
Basic Conditional Process Algebra, p. 20.
Bounded Stacks, Bags and Queues, p. 15.
When do datatypes commute? p. 35.

Distributed real-time systems: a survey of applications and a general design model, p. 31.


Exploring the Process Dimension of Workflow Management, p. 56.

A computer checked algebraic verification of a distributed summation algorithm, p. 28.

λP*: A Pure Type System for First Order Logic with Automated Theorem Proving, p. 35.

On the verification of inter-organizational workflows, p. 23.

Calculating a Round-Robin Scheduler, p. 23.


An analytical method for assessing business processes, p. 29.

Process Algebra in PVS

The Proof-assistant Yarrow, p. 15


Checking verifications of protocols and distributed systems by computer. Extended version of a tutorial at CONCUR'98, p. 27.

Structured Operational Semantics of \( \chi \), p. 27

Diagnosing Workflow Processes using Woflan, p. 44

Final Dialgebras: From Categories to Allegories, p. 26

Process Algebra with Interleaving Probabilistic Parallel Composition, p. 81
99/05 M. Fransen, R.C. Veltkamp and W. Wesselink
Efficient Evaluation of Triangular B-splines, p. 13

99/06 T. Basten and W. v.d. Aalst
Inheritance of Workflows: An Approach to tackling problems related to change, p. 66

99/07 P. Brusilovsky and P. De Bra

99/08 D. Bosnacki, S. Mauw, and T. Willemse

99/09 J. v.d. Pol, J. Hoornan and E. de Jong
Requirements Specification and Analysis of Command and Control Systems

99/10 T.A.C. Willemse
The Analysis of a Conveyor Belt System, a case study in Hybrid Systems and timed μ CRL, p. 44.

99/11 J.C.M. Baeten and C.A. Middelburg
Process Algebra with Timing: Real Time and Discrete Time, p. 50.

99/12 S. Andova

99/13 K.M. van Hee, R.A. van der Toorn, J. van der Woude and P.A.C. Verkoulen
A Framework for Component Based Software Architectures, p. 19

99/14 A. Engels and S. Mauw
Why men (and octopuses) cannot juggle a four ball cascade, p. 10

An algorithm for the asynchronous Write-All problem based on process collision*, p. 11.

99/16 G.J. Houben, P. Lemmens

99/17 T. Basten, W.M.P. v.d. Aalst
Inheritance of Behavior, p.83

99/18 J.C.M. Baeten and T. Basten
Partial-Order Process Algebra (and its Relation to Petri Nets), p. 79

99/19 J.C.M. Baeten and C.A. Middelburg
Real Time Process Algebra with Time-dependent Conditions, p.33.

99/20 Proceedings Conferentie Informatiewetenschap 1999
Centrum voor Wiskunde en Informatica
12 november 1999, p.98 edited by P. de Bra and L. Hardman

00/01 J.C.M. Baeten and J.A. Bergstra
Mode Transfer in process Algebra, p. 14

00/02 J.C.M. Baeten
Process Algebra with Explicit Termination, p. 17.

00/03 S. Mauw and M.A. Reniers
A process algebra for interworkings, p. 63.

00/04 R. Bloor, J. Hoornan and E. de Jong
Semantical Aspects of an Architecture for Distributed Embedded Systems*, p. 47.

00/05 J.F. Groote and M.A. Reniers
Algebraic Process Verification, p. 65.