Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Algebraic verification of a distributed summation algorithm

J.F. Groote and J.G. Springintveld

Computer Science/Department of Software Technology

CS-R9640 1996

Report CS-R9640
ISSN 0169-118X

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

Algebraic Verification of a Distributed Summation Algorithm

Jan Friso Groote

CWI

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

E-mail: `jfg@cwi.nl`

Jan Springintveld

Computing Science Institute, University of Nijmegen

Toernooiveld 1, 6525 ED Nijmegen, The Netherlands

E-mail: `jans@cs.kun.nl`

Abstract

In this note we present an algebraic verification of Segall's Propagation of Information with Feedback (PIF) algorithm. This algorithm serves as a nice benchmark for verification exercises (see [2, 13, 8]). The verification is based on the methodology presented in [7] and demonstrates its applicability to distributed algorithms.

CR Subject Classification (1991): D.2.4 Program Verification; F.3: Logics and Meanings of Programs.
AMS Subject Classification (1991): 68Q60: Specification and verification of programs; 68Q22: Parallel and distributed algorithms.
Keywords & Phrases: Distributed Summing Algorithm, Verification, Process Algebra, μCRL.
Note: The research of the second author is supported by the Netherlands Organization for Scientific Research (NWO) under contract SION 612-33-006.

1 Introduction

The applicability of formal methods for the specification and verification of distributed systems is still a much debated issue. For instance, in [2], Chou claims that there are still no formal methods to reason about distributed systems which are both practical and intuitive. In order to illustrate his opinion he introduces a variant of Segall's PIF (Propagation of Information with Feedback) algorithm [11]. The purpose of this parallel algorithm is to collect the sum of values that are stored by processes which form the nodes of a finite, strongly connected network. The algorithm is indeed an interesting benchmark problem for verification because it is highly parallel and non-deterministic.

In this note we present a verification of a distributed summing algorithm in μCRL, which is a process algebra which allows processes parameterised with data [6, 5]. The correctness of the algorithm is stated as a process equation (Theorem 3.5), the proof of which is a straightforward application of the methodology from [7], which is a combination of algebraic

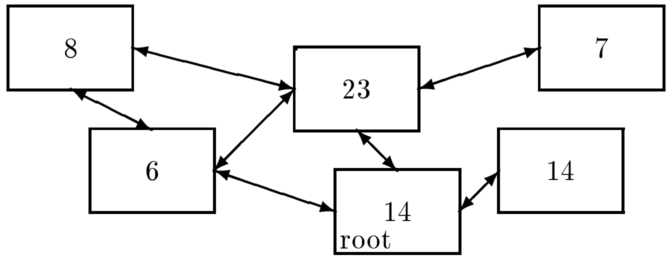

Figure 1: A set of distributed processes

and assertional techniques. In [9] it is shown how proofs using this technique can be proof-checked by computer, but we have not carried out this exercise for the distributed summing protocol.

This paper is organised as follows. The algorithm is described informally in Section 2 and formally in Section 3. In Section 4, a linear process equation for the algorithm is given and it is proven that the resulting process does not admit infinite sequences of internal actions. Section 5 contains a set of invariants that characterise the reachable states of the algorithm. In Section 6, a state mapping is devised that relates configurations of the implementation to corresponding configurations of the specification. We prove that the state mapping is a branching bisimulation between implementation and the specification. Section 7 contains a comparison of our proof with three other verifications of the summation algorithm: the verification done by Vaandrager in the I/O automata model [13], the verification of Chou [2] which uses the notions of causes and events, and the verification of Hesselink [8] which uses the Boyer Moore theorem prover. Finally, Appendix A contains a short overview of the language μCRL and the methodology of [7].

Acknowledgement

Thanks go to Bas Luttik for carefully proof reading and pointing out a confusion in our treatment of the data types.

2 Description

The distributed summing algorithm does the following. Consider a set of processes that are connected via some network of bidirectional links (see e.g. Figure 1). We assume that all processes are connected, i.e. from each process we can reach any other process via one or more links. Each process contains some number, not known to other processes. The algorithm describes how to collect all numbers such that one designated (root) process can output the sum of these numbers. The major difficulty in doing so is to use each value in each process exactly once.

The algorithm is described as the parallel composition of a (finite) number of processes, indexed by natural numbers. Each process works in exactly the same way, except for the root process, which has number 0. This process differs from the other processes in the sense that initially it is already started, and when it has collected all sums of its neighbours, it issues a $\overline{rep}$ message to indicate the total sum to the outside world, instead of a partial sum to a neighbour.

The overall idea behind the algorithm is that a minimal spanning tree over the links between the processes is constructed with as root the process 0. All partial sums are then sent via this spanning tree to the root.

Initially, a process is waiting for a *start* message from a neighbour. After it has received the first start message, the process is considered part of the spanning tree and the process by which it is started is called its *parent*. Thereafter it starts all its neighbours except its parent by a *start* message.

- Those neighbours that were not yet part of the minimal spanning tree will now become part of it with the current process as parent. Eventually, these neighbours will send a partial sum to the current process using an *answer* message.
- Those neighbours that were already part of the spanning tree ignore the start message. Note however that due to symmetry these processes will also send a start message to the current process.

So, a process gets from each neighbour except its parent either a partial sum or a start message. After having received these messages, it adds all received partial sums to its own value, and sends the result as a partial sum to its parent. Eventually, the root process 0 has received all partial sums, and it can report the total sum.

Theorem 3.5 says that this simple scheme is correct, i.e., if each process is connected to the root, processes do not have themselves as neighbours and the neighbour relation is symmetric, then the distributed summing algorithm computes the sum of the values of the individual processes. Note that if any of the stated conditions on the topology does not hold, the algorithm either deadlocks, not yielding a result, or it does not sum up all values.

3 Formal specification

In this section we will formalise the description given above and state the correctness criterion. The algorithm is described as the parallel composition of the algorithms for the individual nodes in the network, which are described generically by means of a linear process equation. For a short introduction to the μCRL syntax of processes, we refer to Appendix A.

For the formal specification, we need the data type **Bool** of the booleans T and F and the usual operators $\wedge$, $\vee$, $\rightarrow$ and $\neg$. We also use natural numbers $\mathbb{N}$ with addition and (cut-off) subtraction.

The data type *nSet* denotes finite sets of natural numbers. For such a set N we let $rem(i, N)$ represent the set N where element i has been removed. The function $size(N)$ yields the number of different elements in the set. We use $\in$ and $\notin$ to test membership of a set.

We also use lists of natural numbers *nList* and lists of sets of natural numbers *SList*. Positions in lists are indexed by natural numbers, starting with index 0. For a list $\boldsymbol{l}$, $\boldsymbol{l}[i]$ is the element at position i of the list. We write $\boldsymbol{l}[i] := t$ for the list $\boldsymbol{l}$ where t has been put at position i. As these data types are fairly standard, we have omitted their specification using abstract data types.

The processes of the network interact via matching actions st, $\overline{st}$ (for *start*), ans, $\overline{ans}$ (for *answer*) and the total sum is communicated using a $\overline{rep}$ (for *report*) action. Although communication is synchronous, we think of the overbarred action as a send activity, and a non-overbarred action as the receiving activity. If an action a synchronises with an action $\overline{a}$, we call the resulting communication $a^\star$. In μCRL we formally declare the actions and communications as follows.

act $st, \overline{st}, st^\star : \mathbb{N} \times \mathbb{N}$ (parameters: destination, source)
$ans, \overline{ans}, ans^\star : \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ (parameters: destination, source, value)
$\overline{rep} : \mathbb{N}$ (parameter: value)

comm $st|\overline{st} = st^\star$
$ans|\overline{ans} = ans^\star$

Definition 3.1 *(Processes)*. Processes P are described by means of six parameters:

- i: the ID-number of the process.
- t: the *total sum* computed so far by the process. Initially, it contains the value that is contributed by process i to the total sum.
- N: a set of *neighbours* to which the process still needs to send a $\overline{st}$ message.
- p: the index of the initiator, or *parent*, of the process. Variable p is also called the parent link of i.
- w: The number of st and ans messages that the process is still *waiting* for.
- s: the *state* the process is in. The process can be in three states, denoted by 0, 1, 2. If s equals 0, the process is in its initial state. If s equals 1, the process is active. If s equals 2, the process has finished and behaves as deadlock.

$$
\begin{aligned}
&P(i,t{:}\mathbb{N}, N{:}nSet, p, w{:}\mathbb{N}, s{:}\mathbb{N}) = \\
&\quad [s=0] \Rightarrow \textstyle\sum_{j:\mathbb{N}} st(i,j)\, P(i,t,rem(j,N),j,size(N){-}1,1) + \\
&\quad \textstyle\sum_{j:\mathbb{N}} [j \in N \wedge s=1] \Rightarrow \overline{st}(j,i)\, P(i,t,rem(j,N),p,w,s) + \\
&\quad \textstyle\sum_{j,m:\mathbb{N}} [s=1] \Rightarrow ans(i,j,m)\, P(i,t+m,N,p,w{-}1,s) + \\
&\quad \textstyle\sum_{j:\mathbb{N}} [s=1] \Rightarrow st(i,j)\, P(i,t,N,p,w{-}1,s) + \\
&\quad [i=0 \wedge N=\emptyset \wedge w=0 \wedge s=1] \Rightarrow \overline{rep}(t)\, P(i,t,N,p,w,2) + \\
&\quad [i \neq 0 \wedge N=\emptyset \wedge w=0 \wedge s=1] \Rightarrow \overline{ans}(p,i,t)\, P(i,t,N,p,w,2)
\end{aligned}
$$

□

In line 1 of P above, process i is in its initial state and an st message is received from some process j, upon which j is stored as the parent and s switches from 0 to 1, indicating that process i has become active. Since it makes no sense to send start messages to one's parent, j is removed from N. The counter w is initialised to the number of neighbours of i, not counting process j. In line 2, a $\overline{st}$ message is sent to a neighbour j, which is thereupon removed from N. In line 3, a sum is received from some process j via an ans message containing the value m, which is added to t, the total sum computed by process i so far. The counter w is decreased. In line 4 a st message is received from neighbour j. The message is ignored, except that the counter w is decreased. In line 5 a $\overline{rep}(t)$ is sent (in case $i = 0$), when process 0 is active, there are no more ans or st messages to be received (formalised by the condition $w = 0$), and a $\overline{st}$ message has been sent to all neighbours (formalised by the condition $N = \emptyset$). The status variable s becomes 2, indicating that process 0 is no longer active. Line 6 is as line 5 but for processes $i \neq 0$; now an $\overline{ans}$ message is sent to parent p, containing the total sum t computed by process i.

Next, we define the parallel composition of $n+1$ copies of the process P. The result can be viewed as a network of processes in the following way. Think of the $n+1$ nodes of the network as items in a list of length $n+1$. The neighbour relation is given by a list $\boldsymbol{n}$ of length $n+1$ of finite sets of natural numbers, with at each position i the set of neighbours of process i. The t-values of the processes are put in a list $\boldsymbol{t}$ of length $n+1$ of natural numbers, with at position i the t-value of process i. Similarly, the lists $\boldsymbol{p}$, $\boldsymbol{w}$, $\boldsymbol{s}$ contain the values of the variables p, w and s of all processes, respectively.

Definition 3.2 *(Parallel composition of processes).*

$$
\begin{aligned}
&\mathit{Impl}(n{:}\mathbb{N}, \boldsymbol{t}{:}\mathit{nList}, \boldsymbol{n}{:}\mathit{SList}, \boldsymbol{p}{:}\mathit{nList}, \boldsymbol{w}{:}\mathit{nList}, \boldsymbol{s}{:}\mathit{nList}) = \\
&\qquad P(0, \boldsymbol{t}[0], \boldsymbol{n}[0], \boldsymbol{p}[0], \boldsymbol{w}[0], \boldsymbol{s}[0]) \triangleleft n = 0 \triangleright \\
&\qquad\quad (P(n, \boldsymbol{t}[n], \boldsymbol{n}[n], \boldsymbol{p}[n], \boldsymbol{w}[n], \boldsymbol{s}[n]) \parallel \mathit{Impl}(n{-}1, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s}))
\end{aligned}
$$

□

Next, we formulate some requirements on the topology of the network.

Definition 3.3 *(Requirements for topology).* We fix a natural number n, denoting the number of non-root processes in the network, a list of natural numbers $\boldsymbol{t}_0$ of length $n+1$, containing the initial t-values of each of the processes, and a list (of length $n+1$) of sets of natural numbers $\boldsymbol{n}_0$, containing for each process the id's its neighbours. We define $\mathit{goodtopology}(n, \boldsymbol{n}_0)$ as the conjunction of the following properties:

- No process has a link to itself: $\forall i \; i \notin \boldsymbol{n}_0[i]$;
- The neighbour relation is symmetric: $\forall i, j \leq n \; i \in \boldsymbol{n}_0[j] \leftrightarrow j \in \boldsymbol{n}_0[i]$;
- Every process $i \leq n$ is connected to process 0:

  for all $i \leq n$ there exist $m \leq n$ and $i = i_0, \ldots, i_m = 0$ such that, for all $0 \leq l < m$, $i_{l+1} \in \boldsymbol{n}_0[i_l]$.

- $\boldsymbol{n}_0$ only contains valid neighbours: $\forall i\ \forall j \leq n\ i \in \boldsymbol{n}_0[j] \rightarrow i \leq n$.

$\square$

Definition 3.4 *(Distributed Summing Algorithm).* The distributed summing algorithm *DSum* is defined as *Impl*, initialised with, apart from n, $\boldsymbol{t}_0$, and $\boldsymbol{n}_0$, the following special values:

- $\boldsymbol{p}_0$, a list of $n+1$ 0's, saying that initially each process considers process 0 as its initiator.
- $\boldsymbol{w}_0$, a list of length $n+1$, with at each position i the size of the set $\boldsymbol{n}_0[i]$. Thus, initially every process expects a message from all its neighbours.
- $\boldsymbol{s}_0$, a list of length $n+1$, with in the first position a 1, to indicate that process 0 is active, and at the remaining n positions a 0, to indicate that all other processes are still sleeping.

We leave it to the reader to devise algebraic specifications of these lists. We put

$$DSum(n, \boldsymbol{t}_0, \boldsymbol{n}_0) = Impl(n, \boldsymbol{t}_0, \boldsymbol{n}_0, \boldsymbol{p}_0, \boldsymbol{w}_0, \boldsymbol{s}_0)$$

$\square$

The theorem below states correctness of the summing algorithm. It says that in a topology as described above, the distributed summing algorithm correctly reports the sum of all values in the processes and halts. The right hand side mentions a function *sum*, which sums up the numbers in a list of natural numbers.

The remainder of this paper is devoted to proving this theorem; it is repeated and proved as Theorem 6.3.

Theorem 3.5.

$$\boxed{goodtopology(n, \boldsymbol{n}_0) \rightarrow\ \tau\, \tau_I \partial_H(DSum(n, \boldsymbol{t}_0, \boldsymbol{n}_0)) = \tau\, \overline{rep}(sum(\boldsymbol{t}_0))\, \delta}$$

where $I = \{st^\star, ans^\star\}$ and $H = \{st, ans, \overline{st}, \overline{ans}\}$. In the trivial case that process 0 has no neighbours, the τ's at the left and right hand side of the equation may be omitted.

4 Linearisation

In Figure 2, we define the process *L-Impl*, which in Lemma 4.1 is stated to be a convergent linearisation of $\tau_I \partial_H(Impl(n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s}))$. The first and second τ-actions originate from hiding the action $st^\star$. The third τ-action comes from hiding $ans^\star$. In the recursive calls of *L-Impl* only the parameters that are changed are displayed.

Lemma 4.1.

1. *L-Impl in Figure 2 is convergent, i.e. does not admit infinite τ-paths.*

$$
\begin{array}{l}
\mathit{L\text{-}Impl}(n{:}\mathbb{N}, \boldsymbol{t}{:}nList, \boldsymbol{n}{:}SList, \boldsymbol{p}, \boldsymbol{w}{:}nList, \boldsymbol{s}{:}nList) = \\
\quad [\boldsymbol{n}[0] = \emptyset \wedge \boldsymbol{w}[0] = 0 \wedge \boldsymbol{s}[0] = 1] \Rightarrow \\
\qquad \overline{rep}(\boldsymbol{t}[0])\ \mathit{L\text{-}Impl}(\boldsymbol{s}[0] := 2) \hfill + \\
\quad \sum_{i,j:\mathbb{N}} [\boldsymbol{s}[i] = 0 \wedge i \in \boldsymbol{n}[j] \wedge \boldsymbol{s}[j] = 1 \wedge i \neq j \wedge i \leq n \wedge j \leq n] \Rightarrow \\
\qquad\qquad \tau\ \mathit{L\text{-}Impl}(\boldsymbol{n}[j] := rem(i, \boldsymbol{n}[j]), \\
\qquad\qquad\qquad \boldsymbol{n}[i] := rem(j, \boldsymbol{n}[i]), \\
\qquad\qquad\qquad \boldsymbol{p}[i] := j, \\
\qquad\qquad\qquad \boldsymbol{w}[i] := size(\boldsymbol{n}[i]) - 1, \\
\qquad\qquad\qquad \boldsymbol{s}[i] := 1) \hfill + \\
\quad \sum_{i,j:\mathbb{N}} [\boldsymbol{s}[i] = 1 \wedge i \in \boldsymbol{n}[j] \wedge \boldsymbol{s}[j] = 1 \wedge i \neq j \wedge i \leq n \wedge j \leq n] \Rightarrow \\
\qquad\qquad \tau\ \mathit{L\text{-}Impl}(\boldsymbol{n}[j] := rem(i, \boldsymbol{n}[j]), \\
\qquad\qquad\qquad \boldsymbol{w}[i] := \boldsymbol{w}[i] - 1) \hfill + \\
\quad \sum_{j:\mathbb{N}} [\boldsymbol{n}[j] = \emptyset \wedge \boldsymbol{w}[j] = 0 \wedge \boldsymbol{s}[j] = 1 \wedge \boldsymbol{s}[\boldsymbol{p}[j]] = 1 \wedge \\
\qquad\quad j \neq 0 \wedge j \neq \boldsymbol{p}[j] \wedge j \leq n \wedge \boldsymbol{p}[j] \leq n] \Rightarrow \\
\qquad\qquad \tau\ \mathit{L\text{-}Impl}(\boldsymbol{t}[\boldsymbol{p}[j]] := \boldsymbol{t}[\boldsymbol{p}[j]] + \boldsymbol{t}[j], \\
\qquad\qquad\qquad \boldsymbol{w}[\boldsymbol{p}[j]] := \boldsymbol{w}[\boldsymbol{p}[j]] - 1, \\
\qquad\qquad\qquad \boldsymbol{s}[j] := 2)
\end{array}
$$

Figure 2: Linearisation of the implementation

2. $\tau_I \partial_H(Impl(n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s})) = L\text{-}Impl(n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s})$.

3. $\tau_I \partial_H(DSum(n, \boldsymbol{t}_0, \boldsymbol{n}_0)) = L\text{-}Impl(n, \boldsymbol{t}_0, \boldsymbol{n}_0, \boldsymbol{p}_0, \boldsymbol{w}_0, \boldsymbol{s}_0)$

Proof.

1. At each τ-step, either a link in $\boldsymbol{n}$ is removed, or a process moves from state 1 to state 2. Hence, the sum of the number of links in $\boldsymbol{n}$ and the number of processes in state 0 or 1 strictly decreases with each τ-step.

2. This follows from Theorem 3.5 in [4] and application of τ_I and ∂_H.

3. By item 2.

⊠

5 Invariants

We provide a number of invariants of which most express that bookkeeping is done properly (see Appendix A for a precise definition of invariants). The most interesting are invariants 11, 12 and 13. The first of these three implies that from each process in state 1 process 0 is reachable in a finite number of steps by iteratively following parent links (i.e. following variable p). As each process has a unique parent, this is an alternative way of saying that the parent links constitute a tree structure with process 0 as root (and a self-loop at the root). Invariant 12 expresses that along each such path all processes are in state 1 too, meaning that they are willing to pass partial results along. Invariant 13 expresses that the total sum in the processes is maintained in the processes that are not in state 2. We will see that at a certain moment all processes, except process 0, are in state 2, which implies that at that moment the total sum is present in process 0.

The invariants mention the functions $Preach$, $starters$, $children$, and $sum_{0,1}$, which are defined first.

Definition 5.1. Let $\boldsymbol{t}$, $\boldsymbol{n}$, $\boldsymbol{p}$, $\boldsymbol{s}$ be as in Definition 3.2.

- The function $Preach(i, j, \boldsymbol{p}, m)$ expresses that from process i process j can be reached by following the parent links in $\boldsymbol{p}$. So $Preach(i, j, \boldsymbol{p}, m)$ holds if there exist $i = i_0, \ldots, i_m = j$ such that, for all $0 \leq l < m$, $\boldsymbol{p}[i_l] = i_{l+1}$.

- $starters(i, \boldsymbol{n})$ is the number of sets L in $\boldsymbol{n}$ such that $i \in L$. Intuitively, $starters(i, \boldsymbol{n})$ is the number of processes that still want to send a $\overline{st}$ message to process i.

- $children(i, \boldsymbol{p}, \boldsymbol{s})$ is the number of processes $j \neq 0$ in the list $\boldsymbol{p}$ such that $\boldsymbol{p}[j] = i$ and $\boldsymbol{s}[j] = 1$. That is, $children(i, \boldsymbol{p}, \boldsymbol{s})$ is the number of active non-root processes that regard process i as their parent.

- $sum_{0,1}(\boldsymbol{t}, \boldsymbol{s})$ is the sum of the $\boldsymbol{t}[i]$-values of the processes i that are not yet finished, i.e. such that $\boldsymbol{s}[i] = 0$ or $\boldsymbol{s}[i] = 1$.

□

Theorem 5.2. *The following are invariants of L-Impl*$(n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s})$. *Here the universal quantification over* i *and* j *is left implicit. The conjunction of the invariants is written as* $Inv(\boldsymbol{n}_0, \boldsymbol{t}_0, n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s})$. *Note that the initial topology* $\boldsymbol{n}_0$ *and the initial distribution of values* $\boldsymbol{t}_0$ *are part of the invariant, although these are not a parameter of L-Impl.*

1. $\boldsymbol{s}[i] \leq 2$.
2. $\boldsymbol{p}[i] \leq n$.
3. $i \in \boldsymbol{n}[j] \rightarrow i \leq n$.
4. $i \notin \boldsymbol{n}[i]$.
5. $\boldsymbol{s}[0] \neq 0$.
6. $\boldsymbol{p}[0] = 0$.
7. $\boldsymbol{s}[i] = 0 \wedge j \in \boldsymbol{n}[i] \rightarrow i \in \boldsymbol{n}[j]$
8. $\boldsymbol{s}[i] = 0 \rightarrow \boldsymbol{n}[i] = \boldsymbol{n}_0[i]$.
9. $\boldsymbol{s}[i] = 2 \rightarrow \boldsymbol{w}[i] = 0 \wedge \boldsymbol{n}[i] = \emptyset$
10. *For every process* i, $\boldsymbol{w}[i]$ *records exactly the number of messages that are to be received. These can either be* st *messages, or* ans *messages:*

    $\boldsymbol{w}[i] = starters(i, \boldsymbol{n}) + children(i, \boldsymbol{p}, \boldsymbol{s})$.
11. *From every process* i *process* 0 *is reachable via parent links in a finite number of steps:* $\exists m \leq n\ Preach(i, 0, \boldsymbol{p}, m)$.
12. *If a process* i *is in state* 1, *then its parent is also in state* 1:

    $\boldsymbol{s}[i] = 1 \rightarrow \boldsymbol{s}[\boldsymbol{p}[i]] = 1$.
13. *As long as no* $\overline{rep}$ *message has been issued by process* 0 *(i.e.* $\boldsymbol{s}[0] \neq 2$*), the total sum (i.e.* $sum(\boldsymbol{t}_0)$*) is present in the processes that are in state* 0 *or* 1:

    $\boldsymbol{s}[0] \neq 2 \rightarrow sum_{0,1}(\boldsymbol{t}, \boldsymbol{s}) = sum(\boldsymbol{t}_0)$

Proof. The invariants 1 to 9 are easily checked (invariant 6 uses invariant 5). The invariants 10 and 12 are proven simultaneously, using invariant 7. The remaining two invariants can be proven on their own. ⊠

6 State mapping, focus point and final lemma

In order to apply the methodology from [7], we specify a linear process *L-Spec* describing the specification.

proc $\quad$ *L-Spec*$(b : \mathbf{Bool}) = [b] \Rightarrow \overline{rep}(sum(\boldsymbol{t}_0))$ *L-Spec*$(\neg b)$

Clearly, $L\text{-}Spec(\boldsymbol{t}) = \overline{rep}(sum(\boldsymbol{t}_0))\delta$.

Furthermore, we provide a *state mapping* h, that specifies how the control variable b of the specification *L-Spec* is constructed out of the parameters $n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s}$ of the implementation *L-Impl*. We put

$$h(n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s}) = (\boldsymbol{s}[0] = 1).$$

The intuition behind this definition is as follows. In a configuration s of *L-Impl* that satisfies $\boldsymbol{s}[0] = 1$, $h(s)$ is T (true), so *L-Spec* can perform the $\overline{rep}$-action, after which it halts. *L-Impl* may not be able to perform a matching $\overline{rep}$ action directly, since the computation of the value to be reported has not yet finished (i.e., $\boldsymbol{n}[0] \neq \emptyset$ or $\boldsymbol{w}[0] \neq 0$). However, using the fact that *L-Impl* is convergent, we see that after a finite number of internal τ-steps a configuration s' is reached where no τ-step is enabled, $\boldsymbol{s}[0]$ is still 1 (h will be invariant under the τ-steps), but also $\boldsymbol{n}[0] = \emptyset$ and $\boldsymbol{w}[0] = 0$. So the $\overline{rep}$-action can be performed (with the correct value), after which *L-Impl* halts. Conversely, it is easy to verify that if in configuration s *L-Impl* can perform the $\overline{rep}$ action, then $\boldsymbol{s}[0] = 1$, so in configuration $h(s)$ the control variable $b = h(s)$ of *L-Spec* has the value T and the specification *L-Spec* can perform the $\overline{rep}$-action (with corresponding value). From these observations it will follow that h is indeed a branching bisimulation function.

We formalise this intuitive argument, using a *focus condition*, which is a formula that characterises the configurations of *L-Impl* in which no τ-step is enabled. (These configurations are so-called *focus points*). Such a formula is extracted from the equation characterising *L-Impl* (see Figure 2) by negating the guards that enable τ-steps in *L-Impl*. As an optimisation, we have put the first two negated guards together, and have restricted the focus condition to configurations satisfying the invariant.

$$\begin{aligned}
&FC(n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s}) = \forall i, j \leq n \\
&\qquad (\boldsymbol{s}[i] = 2 \vee i \notin \boldsymbol{n}[j] \vee \boldsymbol{s}[j] \neq 1 \vee i = j) \wedge \\
&\qquad (\boldsymbol{n}[j] \neq \emptyset \vee \boldsymbol{w}[j] > 0 \vee \boldsymbol{s}[j] \neq 1 \vee \boldsymbol{s}[\boldsymbol{p}[j]] \neq 1 \vee j = 0)
\end{aligned}$$

We distinguish two kinds of focus points of the distributed summing algorithm. One is the set of configurations where the algorithm has reported the sum and is terminated, so $\boldsymbol{s}[0] = 2$. The other one contains the configuration s' mentioned above and is characterised by $\boldsymbol{s}[0] = 1$. At that moment the correct sum should be reported. Items 1 and 2 of the lemma below say that all conditions in the process *L-Impl* for issuing a $\overline{rep}$ action are satisfied; so reporting is possible. Item 3 says that in such a case, all other processes are in state 2. Hence, using invariant 13 we may conclude that the total sum is indeed collected in process 0, i.e. process 0 reports the correct sum.

Lemma 6.1. *$Inv(\boldsymbol{n}_0, \boldsymbol{t}_0, n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s})$ and $\boldsymbol{s}[0] = 1$ together imply*

1. $FC(n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s}) \wedge \boldsymbol{s}[i] = 1 \rightarrow \boldsymbol{n}[i] = \emptyset$
2. $FC(n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s}) \rightarrow \boldsymbol{w}[0] = 0$.
3. $goodtopology(n, \boldsymbol{n}_0) \wedge \boldsymbol{w}[0] = 0 \wedge i \neq 0 \rightarrow \boldsymbol{s}[i] = 2$.

Proof.

1. Towards a contradiction, assume there exists a process i such that $\boldsymbol{s}[i]=1$ and $\boldsymbol{n}[i]\neq\emptyset$, say $j\in\boldsymbol{n}[i]$. By invariant 4 we have $j\neq i$. By the first part of $FC(n,\boldsymbol{t},\boldsymbol{n},\boldsymbol{p},\boldsymbol{w},\boldsymbol{s})$, $\boldsymbol{s}[j]=2$. By invariant 9, $\boldsymbol{w}[j]=0$, contradicting invariant 10 (remember that $j\in\boldsymbol{n}[i]$).

2. In order to derive a contradiction, assume that $\boldsymbol{w}[0]>0$. For arbitrary m, we construct a sequence of $m+1$ processes $0=i_0,i_1,\ldots,i_m$ such that for all $0\leq l\leq m$, we have $\boldsymbol{s}[i_l]=1$, $\boldsymbol{w}[i_l]>0$, $\boldsymbol{p}[i_{l+1}]=i_l$, and if $l\neq 0$, $i_l\neq 0$. Clearly, if $m>n$, this contradicts invariant 11, and the fact that each process has a unique parent link.

   Let a process i_l be given such that $\boldsymbol{w}[i_l]>0$ and $\boldsymbol{s}[i_l]=1$. According to invariant 10 at least one of the following should hold.

   - There exists some i such that $i_l\in\boldsymbol{n}[i]$. By invariant 4, $i_l\neq i$. By the first part of $FC(n,\boldsymbol{t},\boldsymbol{n},\boldsymbol{p},\boldsymbol{w},\boldsymbol{s})$ it follows that $\boldsymbol{s}[i]\neq 1$. So, either $\boldsymbol{s}[i]=2$, but this leads to a contradiction using invariant 9 (remember that $\boldsymbol{n}[i]\neq\emptyset$). Or, $\boldsymbol{s}[i]=0$. By invariant 7, $i\in\boldsymbol{n}[i_l]$. So, by $FC(n,\boldsymbol{t},\boldsymbol{n},\boldsymbol{p},\boldsymbol{w},\boldsymbol{s})$, $\boldsymbol{s}[i_l]\neq 1$. Contradiction.
   - Or there is some i such that $\boldsymbol{p}[i]=i_l$, $i\neq 0$ and $\boldsymbol{s}[i]=1$. By the second part of $FC(n,\boldsymbol{t},\boldsymbol{n},\boldsymbol{p},\boldsymbol{w},\boldsymbol{s})$, we have $\boldsymbol{w}[i]>0\vee\boldsymbol{n}[i]\neq\emptyset$. By item 1 of this lemma, $\boldsymbol{n}[i]=\emptyset$. So $\boldsymbol{w}[i]>0$. We can take $i_{l+1}=i$.

3. First, assume there is some process $i\neq 0$ such that $\boldsymbol{s}[i]=1$. Using invariants 10, 12 and 11, it follows that there is a sequence of processes $i=i_0,\ldots,i_m=0$ such that, for all $0\leq l<m$, $\boldsymbol{p}[i_l]=i_{l+1}$, $\boldsymbol{s}[i_l]=1$ and $\boldsymbol{w}[i_{l+1}]>0$. In particular $\boldsymbol{w}[0]>0$ contradicting an assumption.

   So, assume that there is no process $i\neq 0$ such that $\boldsymbol{s}[i]=1$, but there is some process $i\neq 0$ such that $\boldsymbol{s}[i]=0$. From the topology requirement it follows that there is a sequence $i=i_0,\ldots,i_m=0$ such that for all $0\leq l<m$, $i_{l+1}\in\boldsymbol{n}_0[i_l]$. We show that $\boldsymbol{s}[i_l]=0$ for all l, $0\leq l\leq m$. This contradicts the assumption that $\boldsymbol{s}[0]=1$.

   Note that by assumption $\boldsymbol{s}[i_0]=0$. So let i_l such that $\boldsymbol{s}[i_l]=0$. By invariant 8, it follows that $i_{l+1}\in\boldsymbol{n}[i_l]$. By invariant 10, $\boldsymbol{w}[i_{l+1}]>0$, so $i_{l+1}\neq 0$ and, by invariant 9, $\boldsymbol{s}[i_{l+1}]\neq 2$. As we have excluded that process i_{l+1} is in state 1, it must hold that $\boldsymbol{s}[i_{l+1}]=0$, as required.

⊠

Below we copy the General Equality Theorem (see Theorem A.3) instantiated for the distributed summing algorithm. It says that, given the invariant, implementation *L-Impl* and specification *L-Spec* are equivalent (with or without a preceding τ-step, depending on whether the focus condition holds). Its proof requires that 6 groups of requirements, the so-called *matching criteria*, are checked. Given Lemma 6.1 this is completely straightforward.

Lemma 6.2. *Assume* $goodtopology(n,\boldsymbol{n}_0)$.

$$
\begin{array}{l}
Inv(\boldsymbol{n}_0,\boldsymbol{t}_0,n,\boldsymbol{t},\boldsymbol{n},\boldsymbol{p},\boldsymbol{w},\boldsymbol{s})\rightarrow\\
\qquad\begin{array}{lcl}
L\text{-}Impl(n,\boldsymbol{t},\boldsymbol{n},\boldsymbol{p},\boldsymbol{w},\boldsymbol{s}) & \lhd FC(n,\boldsymbol{t},\boldsymbol{n},\boldsymbol{p},\boldsymbol{w},\boldsymbol{s})\rhd & \tau\, L\text{-}Impl(n,\boldsymbol{t},\boldsymbol{n},\boldsymbol{p},\boldsymbol{w},\boldsymbol{s})\\
 & = & \\
L\text{-}Spec(\boldsymbol{s}[0]=1) & \lhd FC(n,\boldsymbol{t},\boldsymbol{n},\boldsymbol{p},\boldsymbol{w},\boldsymbol{s})\rhd & \tau\, L\text{-}Spec(\boldsymbol{s}[0]=1)
\end{array}
\end{array}
$$

Proof. According to [7] it suffices to check that the following instances of the matching criteria are implied by the invariant.

1. By Lemma 4.1.1 *L-Impl* is convergent.

2. The following three requirements ensure that the state mapping h is invariant under τ-steps of *L-Impl*.

   (a) $\boldsymbol{s}[i] = 0 \wedge i \in \boldsymbol{n}[j] \wedge \boldsymbol{s}[j] = 1 \wedge i \neq j \wedge i \leq n \wedge j \leq n$ implies $\boldsymbol{s}[0] = (\boldsymbol{s}[i] := 1)[0]$ (note that $(\boldsymbol{s}[i] := 1)[0]$ is the first element of $\boldsymbol{s}$ where the i^{th} element has been replaced by 1).
   We distinguish two cases. If $i \neq 0$, the condition trivially holds because in that case $(\boldsymbol{s}[i] := 1)[0] = \boldsymbol{s}[0]$. If $i = 0$, one conjunct of the precondition says $\boldsymbol{s}[0] = 0$. This contradicts invariant 5.

   (b) $\boldsymbol{s}[i] = 1 \wedge i \in \boldsymbol{n}[j] \wedge \boldsymbol{s}[j] = 1 \wedge i \neq j \wedge i \leq n \wedge j \leq n$ implies $\boldsymbol{s}[0] = \boldsymbol{s}[0]$.
   This requirement clearly holds.

   (c) $\boldsymbol{n}[j] = \emptyset \wedge \boldsymbol{w}[j] = 0 \wedge \boldsymbol{s}[j] = 1 \wedge \boldsymbol{s}[\boldsymbol{p}[j]] = 1 \wedge j \neq 0 \wedge j \neq \boldsymbol{p}[j] \wedge j \leq n \wedge \boldsymbol{p}[j] \leq n$ implies $\boldsymbol{s}[0] = (\boldsymbol{s}[j] := 2)[0]$.
   This requirement is also trivially valid, because the assumption explicitly says $j \neq 0$. Hence, $(\boldsymbol{s}[j] := 2)[0] = \boldsymbol{s}[0]$.

3. Next, we verify that when the $\overline{rep}$ action is enabled in *L-Impl*, it is enabled in *L-Spec*: $\boldsymbol{n}[0] = \emptyset \wedge \boldsymbol{w}[0] = 0 \wedge \boldsymbol{s}[0] = 1$ implies $\boldsymbol{s}[0] = 1$. This is obviously true.

4. We must show that if *L-Impl* is in a focus point (no internal actions enabled) and *L-Spec* can perform a $\overline{rep}$-action, *L-Impl* can also perform the $\overline{rep}$ action:
   $FC(n, \boldsymbol{t}, \boldsymbol{n}, \boldsymbol{p}, \boldsymbol{w}, \boldsymbol{s}) \wedge \boldsymbol{s}[0] = 1$ implies $\boldsymbol{n}[0] = \emptyset \wedge \boldsymbol{w}[0] = 0 \wedge \boldsymbol{s}[0] = 1$. This is a direct consequence of Lemma 6.1.2 and Lemma 6.1.1.

5. We must show that if the $\overline{rep}$ action is enabled in *L-Impl* then the reported sum is equal to the sum reported in *L-Spec*: $\boldsymbol{n}[0] = \emptyset \wedge \boldsymbol{w}[0] = 0 \wedge \boldsymbol{s}[0] = 1$ implies $\boldsymbol{t}[0] = sum(\boldsymbol{t}_0)$. By invariant 13 we have $sum(\boldsymbol{t}_0) = sum_{0,1}(\boldsymbol{t}, \boldsymbol{s})$. By definition, $sum_{0,1}(\boldsymbol{t}, \boldsymbol{s})$ contains the sum of the $\boldsymbol{t}[i]$ values of all processes i that are not in state 2. By Lemma 6.1.3, only process 0 is not in state 2. Hence $sum(\boldsymbol{t}_0) = sum_{0,1}(\boldsymbol{t}, \boldsymbol{s}) = \boldsymbol{t}[0]$.

6. Finally, we have to show that the h-mapping commutes with the $\overline{rep}$ action, i.e. $(\boldsymbol{s}[0] := 2)[0] \neq 1$. This is easily seen to hold.

$\boxtimes$

Theorem 6.3.

$$\boxed{goodtopology(n, \boldsymbol{n}_0) \rightarrow \tau\, \tau_I \partial_H(DSum(n, \boldsymbol{t}_0, \boldsymbol{n}_0)) = \tau\, \overline{rep}(sum(\boldsymbol{t}_0))\, \delta}$$

where $I = \{st^\star, ans^\star\}$ *and* $H = \{st, ans, \overline{st}, \overline{ans}\}$. *In the trivial case that process 0 has no neighbours, the τ's at the left and right side of the equation may be omitted.*

Proof. Apply Lemma 6.2 with $\boldsymbol{t}_0$ substituted for $\boldsymbol{t}$, $\boldsymbol{n}_0$ for $\boldsymbol{n}$, $\boldsymbol{p}_0$ for $\boldsymbol{p}$, $\boldsymbol{w}_0$ for $\boldsymbol{w}$ and $\boldsymbol{s}_0$ for $\boldsymbol{s}$. This substitution reduces the invariant to $\top$. Furthermore, reduction of the term $FC(n, \boldsymbol{t}_0, \boldsymbol{n}_0, \boldsymbol{p}_0, \boldsymbol{w}_0, \boldsymbol{s}_0)$ leads to $\forall i\ i \notin \boldsymbol{n}_0[0]$. Thus we have

$$\begin{array}{rcl} \textit{L-Impl}(n, \boldsymbol{t}_0, \boldsymbol{n}_0, \boldsymbol{p}_0, \boldsymbol{w}_0, \boldsymbol{s}_0) & \lhd \forall i\ i \notin \boldsymbol{n}_0[0] \rhd & \tau\, \textit{L-Impl}(n, \boldsymbol{t}_0, \boldsymbol{n}_0, \boldsymbol{p}_0, \boldsymbol{w}_0, \boldsymbol{s}_0) \\ & = & \\ \textit{L-Spec}(\top) & \lhd \forall i\ i \notin \boldsymbol{n}_0[0] \rhd & \tau\, \textit{L-Spec}(\top). \end{array}$$

Hence we can conclude

$$\tau\, \textit{L-Impl}(n, \boldsymbol{t}_0, \boldsymbol{n}_0, \boldsymbol{p}_0, \boldsymbol{w}_0, \boldsymbol{s}_0) = \tau\, \textit{L-Spec}(\top)$$

by adding an initial τ if appropriate. We can conclude the stronger

$$\textit{L-Impl}(n, \boldsymbol{t}_0, \boldsymbol{n}_0, \boldsymbol{p}_0, \boldsymbol{w}_0, \boldsymbol{s}_0) = \textit{L-Spec}(\top)$$

in case $\forall i\ i \notin \boldsymbol{n}_0[0]$, i.e. in case process 0 has no neighbours.

By Lemma 4.1.3, we have $\tau_I \partial_H(\textit{DSum}(n, \boldsymbol{t}_0, \boldsymbol{n}_0)) = \textit{L-Impl}(n, \boldsymbol{t}_0, \boldsymbol{n}_0, \boldsymbol{p}_0, \boldsymbol{w}_0, \boldsymbol{s}_0)$. We also have seen that $\textit{L-Spec}(\top) = \overline{rep}(sum(\boldsymbol{t}_0))\,\delta$. The theorem follows. ⊠

7 Comparison

Our appraisal of the applicability of formal techniques for reasoning about distributed algorithms differs strongly from Chou's. We feel that proof techniques from the area of formal methods are sufficiently mature to prove the correctness of protocols of at least the complexity of a distributed summing algorithm. We are convinced that the reader – after having read, digested and understood the correctness proof – will agree that it is straightforward and not at all more complex than necessary.

There are as far as we know three other formal proofs of the distributed summing algorithm. In [13] Vaandrager proves the summing algorithm correct in the setting of I/O automata. His description of the algorithm, which is best compared to the linearisation of the algorithm in Figure 2, differs from ours in two aspects. First, in his set-up processes communicate asynchronously by means of queues, whereas we let processes communicate using synchronous interaction. The second difference is that in [13] when a process reads a $\overline{st}$ message from its input queue, $\overline{st}$ messages are put simultaneously in all outgoing queues, whereas in our setting sending these messages happens in an interleaved way.

The structure of Vaandrager's proof is the following. First, some invariants are proven. Using these, a relation is defined between implementation and specification that is proven to be a refinement. From this it may be concluded that the trace set of the implementation is included in the trace set of the specification. As trace inclusion does not imply deadlock-freeness, this fact is proven separately.

There are two major differences between both proofs. In [13] history and prophecy variables are employed which are not present in our paper. It is remarked in [13] that it should be possible to give the proof without such auxiliary variables, but that they have been included to illustrate their use. Secondly, although the refinement that is presented is very much like our state mapping h, we establish branching bisimulation between specification and the algorithm, whereas using the refinement, only a weaker fact, namely trace inclusion is shown.

Therefore, we do not have to show deadlock freeness separately, as branching bisimulation preserves deadlock freeness.

It is also important to note the similarities between both proofs. The overall structure of the proofs is the same, as are the essential arguments. Actually, it would not be very hard to upgrade the proofs of trace inclusion and deadlock freeness in [13] to imply a result such as ours.

The description of the algorithm by Chou [2] closely resembles the description of [13]. Chou's proof sets out with defining three modal properties together stating that the algorithm will deliver the total sum exactly once. First, it is argued that proving the modal properties directly on the description of the distributed summing algorithm is too complicated. Then a more abstract version of the algorithm is defined in terms of causes and events, the state space of which can be characterised by simple invariants. The abstract version is related to the original one by means of a simulation relation and a 'joint invariant'. It is shown that translated versions of the modal correctness properties hold for the abstract version. Using the simulation relation and the joint invariant it is shown that validity of the original correctness properties can be derived for the original algorithm. Chou's proof thus is similar to Vaandrager's proof except that correctness is stated by means of modal properties instead of by a specification automaton, and the abstract version is defined in terms of causes and events.

We remark that our proof method is purely syntactical and axiomatic, while the proofs in [2, 13] have a semantical nature. This is not very visible in this paper, as we have for readability omitted all syntactic definitions of data types and employ the General Equality Theorem from [7] whose proof is syntactical but which has a semantic flavour. We feel that our method shares the advantages of semantical reasoning, while its axiomatic nature allows a complete, computer-checked formalisation.

A third proof of essentially the same description of the protocol as the one of Chou and Vaandrager is given by Hesselink [8]. He describes the protocol using LISP functions that are triggered by data in input queues and atomically put data in all output queues of a process. In order to model non-deterministic behaviour, Hesselink introduces an oracle. He then proves that the protocol terminates and that if terminated the total sum is collected in the root. These observations exactly match with proof steps one and five of Lemma 6.2. Hesselink uses the Boyer-Moore theorem prover to verify the correctness of his proofs.

A Short description of the syntax

The language μCRL is a formalism (with proof theory) for process algebra comprising data [6, 5]. In this section we give a brief overview of the μCRL syntax for processes and restate the General Equality Theorem of [7], which is the basis of the correctness proof in this paper. In order to do the latter we have to define the format for linear process equations.

A.1 Overview of syntax

Starting from a set Act of actions that can be parameterised with data, processes are defined by means of guarded recursive equations and the following operators.

First, there is a constant δ ($\delta \notin \mathsf{Act}$) that cannot perform any action and is called deadlock or inaction.

Next, there are the sequential composition operator $\cdot$ and the alternative composition operator $+$. The process $x \cdot y$ first behaves as x and if x successfully terminates continues to behave as y. The process $x + y$ can either do an action of x and continue to behave as x or do an action of y and continue to behave as y.

Interleaving parallelism is modeled by the operator $\|$. The process $x \parallel y$ is the result of interleaving actions of x and y, except that actions from x and y may also synchronise to a communication action, when this is explicitly allowed by a communication function. This is a partial, commutative and associative function $\gamma : \mathsf{Act} \times \mathsf{Act} \to \mathsf{Act}$ that describes how actions can communicate; parameterised actions $a(d)$ and $b(d')$ communicate to $\gamma(a,b)(d)$, provided $d = d'$. A specification of a process typically contains a specification of a communication function.

In order to axiomatise the parallel operator there are two auxiliary parallel operators. First, the left merge $⌊\!⌊$, which behaves as the parallel operator, except that the first step must come from the process at the left. Secondly, the communication merge $|$ which also behaves as the parallel operator, except that the first step is a communication between both arguments, as specified by the communication function γ. We often write $a \mid b = c$ for $\gamma(a,b) = c$.

To enforce that actions in processes x and y synchronise, we can prevent actions from happening on their own, using the encapsulation operator ∂_H. The process $\partial_H(x)$ can perform all actions of x except that actions in the set H are blocked. So, assuming $\gamma(a,b) = c$, in $\partial_{\{a,b\}}(x \parallel y)$ the actions a and b are forced to synchronise to c.

We assume the existence of a special action τ ($\tau \notin \mathsf{Act}$) that is internal and cannot be directly observed. The hiding operator τ_I renames the actions in the set I to τ. By hiding all internal communications of a process only the external actions remain.

The following two operators combine data with processes. The sum operator $\Sigma_{d:D} p(d)$ describes the process that can execute the process $p(d)$ for some value d selected from the sort D. The conditional operator $\_\triangleleft\_\triangleright\_$ describes the *then-if-else*. The process $x \triangleleft b \triangleright y$ (where b is a boolean) has the behaviour of x if b is true and the behaviour of y if b is false. When the right hand side trivialises, i.e. y equals δ, we write $[b] \Rightarrow x$.

We apply the convention that $\cdot$ binds stronger than Σ, followed by $\_\triangleleft\_\triangleright\_$, the parallel operators, and $+$ binds weakest. Moreover, $\cdot$ is usually suppressed.

We work in the setting of branching bisimulation [12], which is a refinement of weak bisimulation [10].

Axioms for the operators can be found, e.g., in [6].

A.2 Linear process equations

The process equations for process P in Definition 3.1 and for *L-Impl* in Figure 2 are (essentially) written in the format of *linear process equations* (LPEs). A linear process equation is of the form $X(d{:}D) = \mathit{RHS}$, where d is a parameter of type D and RHS consists of an alternative composition of a number of summands of the form

$$\sum_{e:E} [b(d,e)] \Rightarrow a(f(d,e))\, X(g(d,e))$$

Such a summand means that if for some e of type E the guard $b(d,e)$ is satisfied, the action a can be performed with parameter $f(d,e)$, followed by a recursive call of X with new value $g(d,e)$. Now the main feature of LPEs is that for each action there is a most one summand in the alternative composition[1]. This makes it possible to describe LPEs by means of a finite set Act of actions as indices, giving for each action a the set E_a over which summation takes place, the guard b_a that enables the action, the function f_a that determines the data parameter of the action and the function g_a that determines the value of the recursive call.

In the next definition the symbol Σ, used for summation over data types, is also used to describe an alternative composition over a finite set of actions. If $Act = \{a_1, \ldots, a_n\}$, then $\Sigma_{a \in Act}\, p_a$ denotes $p_{a_1} + p_{a_2} + \cdots + p_{a_n}$. Note that for summation over actions the symbol $\in$ is used (instead of the symbol :).

Definition A.1. Let $Act \subseteq \mathsf{Act} \cup \{\tau\}$ be a finite set of actions, and let D be a data type. A *linear process equation* (*LPE*) over Act and D is an equation of the form

$$X(d:D) = \sum_{a \in Act} \sum_{e:E_a} [b_a(d,e)] \Rightarrow a(f_a(d,e))\, X(g_a(d,e)).$$

for some data types E_a, D_a, and functions $f_a : D \to E_a \to D_a$, $g_a : D \to E_a \to D$, $b_a : D \to E_a \to \mathbf{Bool}$. (We assume that τ has no parameter.) □

The process equations for process P in Definition 3.1 and for *L-Impl* in Figure 2 do not directly fit in the LPE format; consult [7] to verify that the deviations are harmless.

Definition A.2. An LPE X written as in Definition A.1 is called *convergent* if it does not admit infinite τ-paths, i.e., there is a well-founded ordering $<$ on D such that for all $e : E_\tau$ and $d : D$ we have that $b_\tau(d,e)$ implies $g_\tau(d,e) < d$.

An *invariant* of an LPE X written as in Definition A.1 is a function $I : D \to \mathbf{Bool}$ such that for all $a \in Act$, $e : E_a$, and $d : D$ we have $b_a(d,e) \wedge I(d) \to I(g_a(d,e))$. □

For each LPE X, we assume an axiom which postulates that X has a solution, and an axiom that postulates that every *convergent* LPE has at most one solution. In this way, convergent LPEs define processes. The two principles reflect that we only consider process algebras where every LPE has at least one solution and converging LPEs have precisely one solution.

A.3 General Equality Theorem

Theorem A.3 *(General Equality Theorem from [7]). Let X and Y be LPEs given as follows:*

$$X(d:D_X) = \sum_{a \in Act} \sum_{e:E_a} [b_a(d,e)] \Rightarrow a(f_a(d,e))\, X(g_a(d,e))$$

$$Y(d:D_Y) = \sum_{a \in Act \setminus \{\tau\}} \sum_{e:E_a} [b'_a(d,e)] \Rightarrow a(f'_a(d,e))\, Y(g'_a(d,e))$$

Let FC_X be a formula over $d{:}D_X$ describing exactly the states of X from which no τ-action is enabled (i.e. equivalent to $\neg \exists x{:}E_\tau\, b_\tau(d,x)$). Assume that r and q are solutions of X and

[1]The LPEs described here, are called *deterministic* in [7].

Y, *respectively. Suppose* I *is an invariant of* X *and, for all* $d : D_X$, $I(d)$ *implies the following set of* matching criteria.

$$X \text{ is convergent} \tag{1}$$

$$\forall e{:}E_\tau(b_\tau(d,e) \to h(d) = h(g_\tau(d,e))) \tag{2}$$

$$\forall a \in Act \setminus \{\tau\}\forall e{:}E_a\,(b_a(d,e) \to b'_a(h(d),e)) \tag{3}$$

$$\forall a \in Act \setminus \{\tau\}\forall e{:}E_a\,(FC_X(d) \wedge b'_a(h(d),e) \to b_a(d,e)) \tag{4}$$

$$\forall a \in Act \setminus \{\tau\}\,\forall e{:}E_a\,(b_a(d,e) \to f_a(d,e) = f'_a(h(d),e)) \tag{5}$$

$$\forall a \in Act \setminus \{\tau\}\,\forall e{:}E_a\,(b_a(d,e) \to h(g_a(d,e)) = g'_a(h(d),e)) \tag{6}$$

Then

$$\forall d{:}D_X\; I(d) \to r(d) \triangleleft FC_X(d) \triangleright \tau\, r(d) = q(h(d)) \triangleleft FC_X(d) \triangleright \tau\, q(h(d)).$$

References

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

[2] C.-T. Chou. Practical use of the notions of events and causality in reasoning about distributed algorithms. CS Report #940035, UCLA, October 1994.

[3] E.W. Dijkstra and C.S. Scholtens. Termination detection for diffusing computations. *Inf. Processing Letters*, 11(1):1-4, 1980.

[4] J.F. Groote. A note on n similar processes. Technical report CS-R9626, Department of Software Technology, CWI, Amsterdam, 1996, June 1996.

[5] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef and S.F.M. van Vlijmen, eds, *Algebra of Communicating Processes*, Workshops in Computing, pp. 26–62, 1994.

[6] J.F. Groote and A. Ponse. Proof theory for μCRL: a language for processes with data. In Andrews et al. *Proceedings of the International Workshop on Semantics of Specification Languages.* Workshops in Computing, pages 231–250. Springer Verlag, 1994.

[7] J.F. Groote and J. Springintveld. Focus points and convergent process operators. A proof strategy for protocol verification. Technical Report 142, Logic Group Preprint Series, Utrecht University, 1995. This report also appeared as Technical Report CS-R9566, Centrum voor Wiskunde en Informatica, 1995

[8] W.H. Hesselink. A mechanical proof of Segall's PIF algorithm. Computing Science Report CS-R9604, Department of Computing Science, University of Groningen, 1996.

[9] H. Korver and A. Sellink. On automating process algebra proofs. Technical Report 154, Logic Group Preprint Series, Utrecht University, 1996. To appear in proceedings of ISCIS'96, Antalya, Turkey, 1996.

[10] R. Milner. *Communication and Concurrency.* Prentice Hall, London, 1989.

[11] S. Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29(2):23-35, 1983.

[12] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics (extended abstract). In G.X. Ritter, editor, *Information Processing 89*, pages 613–618, 1989.

[13] F.W. Vaandrager. Verification of a distributed summation algorithm. In I. Lee and S.A. Smolka, editors, *Proceedings CONCUR95*, pages 190–203, LNCS 962, Springer-Verlag, 1995.