ELSEVIER

The Journal of Logic and Algebraic Programming 49 (2001) 31–60

THE JOURNAL OF LOGIC AND ALGEBRAIC PROGRAMMING

www.elsevier.com/locate/jlap

Focus points and convergent process operators: a proof strategy for protocol verification

Jan Friso Groote [*], Jan Springintveld

Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
Department of Software Technology, CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Received 25 July 2000; accepted 9 April 2001

Abstract

We present a method for efficiently providing algebraic correctness proofs for communication systems. It is described in the setting of μCRL [J.F. Groote, A. Ponse, The syntax and semantics of μCRL, in: A. Ponse, C. Verhoef, S.F.M. van Vlijmen (Eds.), Algebra of Communicating Processes, Workshops in Computing, Springer, Berlin, 1994, pp. 26–62] which is, roughly, ACP [J.C.M. Baeten, W.P. Weijland, Process Algebra, Cambridge Tracts in Theoretical Computer Science, vol. 18, Cambridge University Press, Cambridge 1990, J.A. Bergstra, J.W. Klop, The algebra of recursively defined processes and the algebra of regular processes, in: Proceedings of the 11th ICALP, Antwerp, Lecture Notes in Computer Science, vol. 172, Springer, Berlin, 1984, pp. 82–95] extended with a formal treatment of the interaction between data and processes. The method incorporates assertional methods, such as invariants and simulations, in an algebraic framework, and centers around the idea that the state spaces of distributed systems are structured as a number of cones with focus points. As a result, it reduces a large part of algebraic protocol verification to the checking of a number of elementary facts concerning data parameters occurring in implementation and specification. The resulting method has been applied to various non-trivial case studies of which a number have been verified mechanically with the theorem checker PVS. In this paper the strategy is illustrated by several small examples and one larger example, the Concurrent Alternating Bit Protocol (CABP). © 2001 Elsevier Science Inc. All rights reserved.

Keywords: Communication protocols; Process algebra; Protocol verification; Linear process operators; ACP; μCRL

1. Introduction

One of the main aims of process theory is to be able to formally describe distributed systems and to verify their correctness w.r.t. some specification. In this paper we present

[*]Corresponding author. Tel.: +31-40-247-5003; fax: +31-40-246-8508.
E-mail addresses: j.f.groote@tue.nl (J.F. Groote), jan@jarma.demon.nl (J. Springintveld).

a proof strategy to verify the correctness of such systems in the framework of process algebra. The method has been used (explicitly or implicitly) in a number of non-trivial case studies. We mention a few examples. Bezem and Groote [3] verified a sliding window protocol and Fredlund et al. [8] proved an efficient, but complex leader election protocol in a ring topology correct. In [22], part of the IEEE P1394 high-speed bus protocol [15] is proven correct and in [10] a distributed summation algorithm is verified. We work within the theory μCRL [11], which is, roughly, ACP [1,2] extended with a formal treatment of the interaction between data and processes.

The task we set ourselves can be described as follows. An implementation of a communication protocol can be described as the parallel composition of several components $C_1, \ldots, C_n$. These components can be receivers, senders, timers, channels, etc. They communicate via internal actions (in a set *H*), resulting in internal communications (in a set *I*). The specification that this implementation should satisfy is given by a process *Spec*. Typically, *Spec* defines a one-bit buffer or a bidirectional queue, etc. In our process algebraic framework, satisfying a specification means being equal to it (according to some preferred equality relation). Thus, in μCRL notation, we want to show that

$$\tau_I(\partial_H(C_1 \| \ldots \| C_n)) = \mathit{Spec}.$$

Here, the τ_I-operator hides the communication actions in *I*, while the ∂_H-operator forces the send and read actions in *H* to synchronise; these operators are explained below.

Classical proof strategies for verification in an algebraic style appear among others in [17,18,24]. A fundamental approach towards proving the equation is as follows. First, find a guarded recursive equation *G*, where guarded means that each occurrence of a recursive process variable must be in the scope of an action, not being τ. Then show that both $\tau_I(\partial_H(C_1 \| \ldots \| C_n))$ and *Spec* are solutions of this equation (possibly applying some fairness principle). Usually, *G* is the expanded version of the protocol. Then the desired equality follows from RSP, the principle stating that guarded recursive equations have at most one solution. Actually it suffices that the recursive equation is weakly guarded, or convergent, in the sense that infinite chains of unguarded occurrences of recursive process variables do not exist.

Our strategy can be seen as a considerably refined version of the above strategy. The refinements are based on a particular format for the notation of processes, the so-called *linear process operators*. This format, similar to the UNITY format of [5,7] and to the precondition/effect notation of [16,19], enriches the process algebraic language with a symbolic representation of the (possibly infinite) state space of a process by means of state variables and formulas concerning these variables. Thus it combines the advantages of a compact and easy to manipulate algebraic notation with the advantages of the precondition/effect style.

Instead of using the principle RSP, we reduce the task of proving implementation and specification equal to the existence of a *state mapping*, satisfying certain constraints, the *matching criteria*. A state mapping maps states of the implementation to matching states of the specification. Here, matching means that the same set of external actions can be executed directly. The matching criteria are comparable to the defining clauses of weak refinements [20]. The criteria are formulated as simple formulas over the data parameters and conditions occurring in implementation and specification. Thus we reduce a large part of the correctness of the implementation w.r.t. the specification to a number of mostly trivial facts concerning data parameters and conditions occurring in implementation and specification. This greatly simplifies protocol verifications and makes our approach amenable to mechanical assistance; a number of the above mentioned proofs of verified systems have been checked mechanically with the theorem provers PVS [21] and Coq [6].

The matching criteria embody an important concept, that of a *focus point* (in the literature sometimes called *stable* points). It is often the case that states in the implementation do not match directly with a state of the specification, yet from these states a state can be reached, after some internal computation, that does match directly with a state of the specification. To deal with this, we employ a case distinction between states in which the protocol cannot perform internal actions, the focus points, and non-focus points, where the protocol can perform internal actions. Focus points must match directly with states in the specification. In case the implementation is convergent, a focus point must be reached by performing finitely many internal actions. The set of states from which a focus point can be reached by internal activity is called a *cone*. Under the assumption that there is no unbounded internal activity, every state belongs to some cone. The state mapping maps all states of a cone to the state corresponding to the focus point of the cone.

For distributed systems that only perform bounded internal activity, the proof strategy is formulated as Theorem 3.3. For the case where the implementation can perform unbounded activity, we provide Theorem 4.9. Here one must in addition distinguish between *progressing* and *non-progressing* internal actions in the implementation in order to guarantee convergence. Intuitively, progressing internal steps are those that lead towards focus points, whereas non-progressing internal actions lead away from focus points.

As shown in a number of verifications, the ingredients outlined above appear sufficient for the systematic verification of numerous protocols and distributed systems (see e.g. [3,8]). The main contribution of the present paper is that it explicitly identifies the strategy outlined above, in the form of definitions and theorems. We provide an example of the verification of the Concurrent Alternating Bit Protocol with a correctness proof that consists of 4 amply commented pages. We hope that this example provides some intuition how progressing internal actions, state mappings, and invariants can be identified.

In its present form, our strategy is not complete; in particular the specification is not allowed to contain τ-steps, so these cases cannot be dealt with. Example 5.3 gives a counterexample to our main results in case the specification is allowed to contain τ-steps. We provide an example where a state mapping does not exist, even though implementation and specification are evidently branching bisimilar. A thorough treatment of completeness is deferred to a future paper. Recently, the cones and foci technique has been generalized to timed processes in [25].

Organisation

In Section 2, we present the preliminaries of the theory. In Section 3, we present a general result that formulates sufficient conditions for two processes to be equal in the case where there are no infinite chains of internal action in the implementation. This result is generalized in Section 4 to the verification of systems that do have unbounded internal activity. In Section 5, we illustrate the proof strategy with some positive and negative examples. One of the positive examples is the Concurrent Alternating Bit Protocol. Appendix A contains technical lemmas that are used in the paper. Finally, Appendix B contains the μCRL axioms plus some additional axioms that are used in the verification.

2. Preliminaries

In this section, we present some basic definitions, properties and results that we use in this paper. We apply the proof theory of μCRL as presented in [11], which can be viewed as ACP [1,2] extended with a formal treatment of the interaction between data and processes. Furthermore, an overview of several verification techniques appeared in [14], including a summary of the cones and foci technique.

2.1. A short description of μCRL

The language μCRL is a process algebra comprising data developed in [12]. We do not describe the treatment of data types in μCRL in detail, as we make little use of it in this paper. For our purpose it is sufficient that processes can be parameterised with data. We assume the data sort of booleans **Bool** with constants true T and false F, and the usual operators. Furthermore, we assume for all data types the existence of an equality function *eq* that faithfully reflects equality, and an *if_then_else*-function such that $\mathit{if}(b, t_1, t_2)$ equals t_1 if b equals T and equals t_2 otherwise.

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. (In Section 2.2, we will discuss a useful variant of guarded recursive equations.)

First, there is a constant δ ($\delta \notin \mathsf{Act}$) that cannot perform any action and is henceforth 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 the rest of x or do an action of y and continue to behave as the rest of y.

Interleaving parallelism is modelled by the operator $\|$. The process $x \| y$ 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 synchronise; parameterised actions $a(d)$ and $b(d')$ synchronise to $\gamma(a, b)(d)$, provided $d = d'$ and $\gamma(a, b)$ defined. 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 $\lfloor\!\lfloor$, 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.

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 $\partial_{\{a,b\}}(x \| y)$ 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. A useful feature is offered by the hiding operator τ_I that renames the actions in the set I to τ. By hiding all internal communications of a process only the external actions remain. In this way we can obtain compact descriptions of the external functionality of a set of cooperating processes. A nice example is provided in Theorem 5.4 where the external behaviour of a set of parallel processes modelling the Concurrent Alternating Bit Protocol appears to be the same as that of a simple one place buffer.

Another useful operator is the general renaming ρ_f, where $f : \mathsf{Act} \to \mathsf{Act}$ is a renaming function on actions. If process x can perform an action a, then $\rho_f(x)$ can perform the action $f(a)$.

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 $\_ \lhd \_ \rhd \_$ describes the *then-if-else*. The process $x \lhd b \rhd y$ (where b is a boolean) has the behaviour of x if b is true and the behaviour of y if b is false.

We apply the convention that $\cdot$ binds stronger than Σ, followed by $\_ \lhd \_ \rhd \_$, and $+$ binds weakest. Moreover, $\cdot$ is usually suppressed. Axioms and rules that characterise the operators are given in Appendix B. These axioms equate processes that are (rooted) branching bisimilar. As the results are proven from these axioms and CL-RSP, all results in this paper are valid for branching bisimulation and all weaker equivalence relations that are congruences. Only Lemma 3.4 and Theorem 3.5 are specifically formulated for weak bisimulation.

2.2. *Linear process operators*

We recapitulate some terminology that has been introduced in [3,4]. Especially the notion of a linear process forms the cornerstone for the developments in this paper.

Intuitively, a linear process is a process of the form $X(d{:}D) = \mathit{RHS}$, where d is a parameter of type D and RHS consists of a number of summands of the form

$$\sum_{e:E} a(f(d,e))\, X(g(d,e)) \lhd b(d,e) \rhd \delta.$$

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 $g(d,e)$.

The main feature of linear processes is that for each action there is a most one alternative. This makes it possible to describe them 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 $\mathit{Act} = \{a_1, \ldots, a_n\}$, then $\Sigma_{a\in \mathit{Act}}\, p_a$ denotes $p_{a_1} + p_{a_2} + \cdots + p_{a_n}$. The p_a's are called *summands* of $\Sigma_{a\in \mathit{Act}}\, p_a$. Note that for summation over actions the symbol $\in$ is used (instead of the symbol :).

Formally, we define linear processes by means of linear process operators.

Definition 2.1. Let $\mathit{Act} \subset \mathsf{Act}$ be a finite set of actions, possibly extended with τ. A *linear process operator* (*LPO*) over Act is an expression of the form

$$\Phi = \lambda p.\lambda d{:}D. \sum_{a\in \mathit{Act}} \sum_{e_a:E_a} a(f_a(d,e_a))\, p(g_a(d,e_a)) \lhd b_a(d,e_a) \rhd \delta,$$

i.e., Φ is a function from p and a value of type D to a process, where p is a function from sort D to processes.

LPOs are traditionally defined equationally. Instead of writing, e.g. $\Phi \overset{\text{def}}{=} \lambda p \lambda n{:}\mathit{Nat}.a\; p(n+1)$ one generally writes $p(n{:}\mathit{Nat}) = a.p(n+1)$, see for instance all examples below. We however introduce the operator notation as both views can easily be exchanged, it avoids the implicit style of definition of behaviour that occur in equations, and it is convenient in proofs. Notably, it is useful to define new LPOs based on others, with slightly adapted behaviour, see for instance Definition 4.1. Observe that we mix notation for function application common in general mathematics and lambda calculus. Sometimes we write Φpd meaning apply operator Φ to arguments p and d. Elsewhere we write $p(d)$ to denote the application of p to d.

LPOs are defined having a single data parameter. The LPOs that we consider generally have more than one parameter, but using cartesian products and projection functions, it is easily seen that this is an inessential extension. Often, parameter lists get rather long. Therefore, we use the following notation for updating elements in the list. Let $\vec{d}$ abbreviate the vector $d_1, \ldots, d_n$. A summand of the form

$$\sum_{e_a:E_a} a(f_a(\vec{d}, e_a))\; p(d'_j/d_j) \lhd b_a(\vec{d}, e_a) \rhd \delta$$

in the definition of a process $p(\vec{d}\,)$ abbreviates

$$\sum_{e_a:E_a} a(f_a(\vec{d}, e_a))\; p(d_1, \ldots, d_{j-1}, d'_j, d_{j+1}, \ldots d_n) \lhd b_a(\vec{d}, e_a) \rhd \delta.$$

Here, the parameter d_i is in the recursive call updated to d'_i. This notation is extended in the natural way to multiple updates. If no parameter is updated, we write the summand as $\Sigma_{e_a:E_a} a(f_a(\vec{d}, e_a))\; p \lhd b_a(\vec{d}, e_a) \rhd \delta$.

We give an example of an LPO K which is a channel that reads a pair of a datum from some data type D and a bit. It either delivers this pair correctly, or loses or garbles it. In the last case a checksum error ce is sent. The non-deterministic choice between the three options is modelled by the actions j and j'. If j is chosen, the data are delivered correctly and if j' happens, it is garbled or lost. The state of the channel is modelled by the parameter i_k.

$$\begin{aligned}
\textbf{proc}\quad K(d{:}D, b{:}Bit, i_k{:}Nat) = & \sum_{d':D}\sum_{b':Bit} r(d',b')K(d'/d, b'/b, 2/i_k) \lhd eq(i_k,1) \rhd \delta \\
& + (j'\,K(1/i_k) + j\,K(3/i_k) \\
& + j'\,K(4/i_k)) \lhd eq(i_k,2) \rhd \delta \\
& + s(d,b)\,K(1/i_k) \lhd eq(i_k,3) \rhd \delta \\
& + s(ce)\,K(1/i_k) \lhd eq(i_k,4) \rhd \delta.
\end{aligned}$$

Note that we have deviated from the LPO format in the ‘strict’ sense: in the last three summands there is no summation over a data type E_i, in the second summand j and j' do not carry a parameter (like the τ-action) and the $+$ operator occurs within the scope of the conditional. But it is obvious that the example can be transformed to LPO format, and therefore, we allow ourselves such deviations.

Processes can be defined as solutions for convergent LPOs.

Definition 2.2. A *solution* or *fixed point* of an LPO Φ is a process p, parameterised with a datum of sort D, such that, for all $d{:}D$, $p(d) = \Phi pd$.

We call an LPO convergent if the process it defines cannot perform infinite sequences of τ-actions.

Definition 2.3. An LPO Φ written as in Definition 2.1 is called *convergent* if there is a well-founded ordering $<$ on D such that for all $e_\tau{:}E_\tau$, $d{:}D$ we have that $b_\tau(d, e_\tau)$ implies $g_\tau(d, e_\tau) < d$.

For each LPO Φ, we assume an axiom which postulates that Φ has a canonical solution, which we denote by $\langle\Phi\rangle$. Then, we postulate that every *convergent* LPO has at most one solution. In this way, convergent LPOs define processes. The two principles reflect that we only consider process algebras where every LPO has at least one solution and converging LPOs have precisely one solution.

Thus we assume the following two principles:

- Recursive Definition Principle (L-RDP): For all d of sort D and LPOs Φ over D we have $\langle\Phi\rangle(d) = \Phi\langle\Phi\rangle d$.
- Recursive Specification Principle (CL-RSP): Every convergent linear process operator has at most one fixed point (solution): for all d of sort D and convergent LPOs Φ over D we have $p(d) = \Phi p d \to p = \langle\Phi\rangle$.

Usually, we do not mention $\langle\Phi\rangle$ explicitly and just speak about solutions for Φ.

The following general theorem, taken from [4] on invariants is the basis for our proofs. Roughly, it says that if an LPO is convergent in the part of its state space that satisfies an invariant I, then it has at most one solution in that part of the state space.

Definition 2.4. An *invariant* of an LPO Φ written as in Definition 2.1 is a function $I : D \to \mathbf{Bool}$ such that for all $a \in Act$, $e_a{:}E_a$, and $d{:}D$ we have

$$b_a(d, e_a) \wedge I(d) \to I(g_a(d, e_a)).$$

Theorem 2.5 (Concrete invariant corollary). *Let Φ be an LPO. If, for some invariant I of Φ, the LPO $\lambda p.\lambda d.\Phi p d \lhd I(d) \rhd \delta$ is convergent and for some processes q, q', parameterised by a datum of type D, we have*

$$I(d) \to q(d) = \Phi q d,$$
$$I(d) \to q'(d) = \Phi q' d,$$

then

$$I(d) \to q(d) = q'(d).$$

Remark 2.6. In this paper we will restrict ourselves to processes defined by LPOs. This is not as restrictive as it may seem, as general μCRL processes can effectively be rewritten to equivalent processes in LPO format [13,23].

2.3. Internal actions

We work in the setting of (rooted) branching bisimulation [9], but provide results for (rooted) weak bisimulation too in those cases where they differ. So, we generally use the following two laws.

B1: $x\tau = x$,
B2: $z(\tau(x + y) + x) = z(x + y)$.

We write $x \subseteq y$ if there exists a z such that $x + z = y$. It is easily verified that if $x \subseteq y$ and $y \subseteq x$, then $x = y$. Using this notation, we have the following easy fact.

Lemma 2.7.

$$y \subseteq x \to \tau x = \tau(\tau x + y).$$

Proof. $\tau x = \tau(x + y) \overset{\text{B2}}{=} \tau(\tau(x + y) + y) = \tau(\tau x + y)$. □

We also assume a principle of fair abstraction, in the form of Koomen's Fair Abstraction Rule (KFAR), which expresses that it is not possible to infinitely execute an internal action (the hidden action i in the formulation below); ultimately the alternative must be chosen. The formulation below is the one valid in branching bisimulation:

$$\frac{p(d) = ip(d) + y}{\tau\tau_{\{i\}}(p(d)) = \tau\tau_{\{i\}}(y)}.$$

Here p represents a process that can be parameterised, y represents a process and i represents an action.

3. Sufficient conditions for the equality of LPOs

In this section, we are concerned with proving equality of solutions of LPOs Φ and Ψ. The LPO Φ defines an implementation and the LPO Ψ defines the specification of a system. We assume that τ-steps do not occur in the specification Ψ. Example 5.3 in Section 5 shows that this restriction is necessary. We want to show that after abstraction of internal actions in a set *Int* the solution of Φ is equal to the solution of Ψ. In this section we assume that Φ cannot perform an infinite sequence of internal actions, but in the next section we relax this restriction. It turns out to be convenient to consider Φ where the actions in *Int* are already renamed to τ. Hence, we speak about an LPO Ξ which is Φ where actions in *Int* have been hidden. Note that Ξ is convergent, and hence defines a process. We fix the LPOs Ξ and Ψ as follows (where the actions are taken from a set *Act*):

$$\Xi = \lambda p.\lambda d{:}D_\Xi . \sum_{a \in Act} \sum_{e_a : E_a} a(f_a(d, e_a))p(g_a(d, e_a)) \lhd b_a(d, e_a) \rhd \delta,$$

$$\Psi = \lambda q.\lambda d{:}D_\Psi . \sum_{a \in Act \setminus \{\tau\}} \sum_{e_a : E_a} a(f'_a(d, e_a))\, q(g'_a(d, e_a)) \lhd b'_a(d, e_a) \rhd \delta.$$

The issue that we consider is how to prove the solutions of Ξ and Ψ equal. This is done by means of a *state mapping* $h{:}D_\Xi \to D_\Psi$. The mapping h maps states of the implementation to states of the specification. It explains how the data parameter that encodes states of the specification is constructed out of the data parameter that encodes states of the implementation. In order to prove implementation and specification branching bisimilar, the state mapping should satisfy certain properties, which we call *matching criteria* because they serve to match states and transitions of implementation and specification.

We first describe what a perfect match is for h. This means that in states of the implementation and specification that are h-related, the same set of external actions can be executed directly, with the same data parameter and leading to h-related states. In general, it may be the case that the implementation, say in state d, has to do some internal computation before it can perform the external actions that are possible from $h(d)$ in the specification.

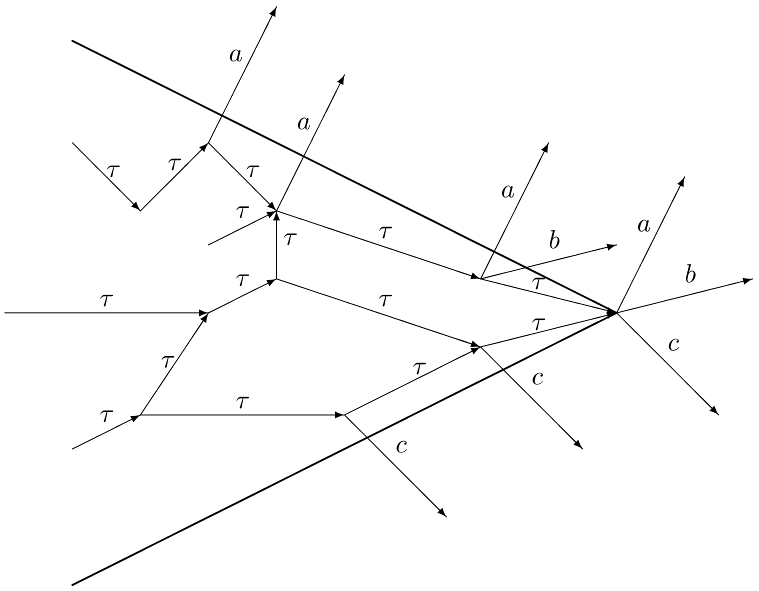

Fig. 1. A cone with its focus point.

In this case, h accomplishes only an "indirect" match: whatever the implementation can do, the specification can do, but not necessarily vice versa. Since the implementation is convergent, it is guaranteed that after finitely many internal steps the internal computation stops and we reach a so-called *focus point*. A focus point is a state in the implementation without outgoing τ-steps. We demand that the focus point that is reached by the implementation should be h-related to $h(d)$ in the specification, and the match should be perfect. The set of states from which a focus point can be reached via internal actions is called the *cone* belonging to this focus point. Now the matching criteria below express that focus points in the state space of the implementation must match perfectly with their h-image in the specification, whereas points in a cone only have to match indirectly.

States without outgoing τ-steps are in the literature also known as *stable states*. We use the terminology *focus points* to stress that in a focus point there is a perfect match between implementation and specification, and in this sense, a focus point is a *goal* of the implementation, and the internal actions in the cone (which are directed to the focus point) are *progressing* towards this goal. The situation is depicted very schematically in Fig. 1. Here the arrows labelled with τ's are internal actions that are all directed towards the focus point.

Note that as we have assumed that Ξ is convergent, each internal step in a cone is indeed directed towards the focus point. In general there may be loops of internal actions, for instance if data must be retransmitted over unreliable channels. Actions that give rise to such loops may be considered *non-progressing* (w.r.t. the focus point). We will deal with them in Section 4.

Before presenting the matching criteria, we give a formal characterisation of focus points of Ξ by means of the *focus condition* $FC_\Xi(d)$, which is true if d is a focus point, and false if not. It simply states that in state d there is no element e of E_τ such that the enabling condition $b_\tau(d, e)$ of the τ-action is satisfied.

Definition 3.1. The *focus condition*, $FC_{\Xi}(d)$, of Ξ is the formula

$$\neg\exists e{:}E_\tau(b_\tau(d,e)) \quad \text{(“in state } d, \tau \text{ is not enabled”).}$$

Now we formulate the criteria. We discuss each criterion directly after the definition. Here, $=$ binds stronger than $\neg$, which binds stronger than $\wedge$ and $\vee$, which in turn bind stronger than $\rightarrow$.

Definition 3.2. Let $h : D_\Xi \to D_\Psi$ be a state mapping. The following criteria referring to Ξ, Ψ and h are called the *matching criteria*. We refer to their conjunction by $C_{\Xi,\Psi,h}(d)$.

$$\forall e_\tau{:}E_\tau\big(b_\tau(d,e_\tau) \to h(d) = h(g_\tau(d,e_\tau))\big), \tag{1}$$

$$\forall a \in Act\backslash\{\tau\}\,\forall e_a{:}E_a\big(b_a(d,e_a) \to b'_a(h(d),e_a)\big), \tag{2}$$

$$\forall a \in Act\backslash\{\tau\}\,\forall e_a{:}E_a\big(FC_\Xi(d) \wedge b'_a(h(d),e_a) \to b_a(d,e_a)\big), \tag{3}$$

$$\forall a \in Act\backslash\{\tau\}\,\forall e_a{:}E_a\big(b_a(d,e_a) \to f_a(d,e_a) = f'_a(h(d),e_a)\big), \tag{4}$$

$$\forall a \in Act\backslash\{\tau\}\,\forall e_a{:}E_a\big(b_a(d,e_a) \to h(g_a(d,e_a)) = g'_a(h(d),e_a)\big). \tag{5}$$

To recapitulate: if in a focus or non-focus point d of the implementation a visible action can be done, then this action must also be possible in $h(d)$ in the specification (criterion (2)). But, conversely, if $h(d)$ in the specification can perform an action, the non-focus point d need not match it directly. As the implementation is convergent (see Theorem 3.3) a focus point will be reached after a finite number of internal steps. Due to criterion (1) this focus point will have the same h-image as d. By criterion (3) this focus point can perform the same actions as $h(d)$. These actions carry the same data parameter (4) and lead to h-related states (5).

Now we come to the main result of this section. Its formulation and proof reflect the discussion above except for two points. First, the theorem is formulated under the condition of an invariant of Ξ. The reason for this is that a specification and an implementation are in general only equivalent for the reachable states in the implementation. A common tool to exclude non-reachable states is an invariant, which is therefore added.

As to the second point, assume that r and q are solutions of Ξ and Ψ, respectively. Let $d{:}D_\Xi$ be given and assume that $I(d)$ and $I(d) \to C_{\Xi,\Psi,h}(d)$ hold. We distinguish two cases. If $FC_\Xi(d)$ holds, so $r(d)$ is in a focus point and cannot perform a τ-action, we prove that $r(d) = q(h(d))$. If $FC_\Xi(d)$ does not hold, then $r(d)$ can perform a τ-action, while $q(h(d))$ cannot (Ψ does not contain τ). So $r(d)$ and $q(h(d))$ cannot be equal. In the setting of branching bisimulation we can in this case only prove $\tau r(d) = \tau q(h(d))$. (In the setting of weak bisimulation this simplifies to $r(d) = \tau q(h(d))$.)

Theorem 3.3 (General equality theorem for branching bisimulation). *Let Ξ, Ψ and h be as above (recall that Ψ does not contain τ-steps). Assume that r and q are solutions of Ξ and Ψ, respectively. If I is an invariant of Ξ, the LPO $\lambda p.\lambda d{:}D.\Xi pd \lhd I(d) \rhd \delta$ is convergent and $\forall d{:}D_\Xi\ (I(d) \to C_{\Xi,\Psi,h}(d))$, then*

$$\forall d{:}D_\Xi\ I(d) \to r(d) \lhd FC_\Xi(d) \rhd \tau r(d) = q(h(d)) \lhd FC_\Xi(d) \rhd \tau q(h(d)).$$

Proof. Define the LPO Ω by

$$\Omega = \lambda r.\lambda d{:}D_\Xi.\Xi rd \lhd FC(d) \rhd \tau\Xi rd.$$

We prove the theorem as an application of the Concrete Invariant Corollary (Theorem 2.5) with Ω as LPO. We verify the conditions of that result.

As the invariant implies that Ξ is convergent, it is straightforward to see that the LPO $\lambda r.\lambda d{:}D_\Xi.\Omega rd \lhd I(d) \rhd \delta$ is convergent too.

Using Lemma A.3 and the fact that r is a solution of Ξ, it is also easy to see that $\lambda d{:}D_\Xi.r(d) \lhd FC(d) \rhd \tau r(d)$ is a solution of Ω.

It is slightly more involved to check that $\lambda d{:}D_\Phi.q(h(d)) \lhd FC(d) \rhd \tau q(h(d))$ is a solution of Ω. After applying Lemma A.3, this boils down to proving the following equation.

$$\begin{aligned}&q(h(d)) \lhd FC(d) \rhd \tau q(h(d))\\&\quad = \Xi\big[\lambda d{:}D_\Xi.q(h(d))\big]d \lhd FC(d) \rhd \tau\Xi\big[\lambda d{:}D_\Xi.q(h(d))\big]d\end{aligned}$$

We distinguish two cases. The first case is where $FC(d)$ holds. We must show that

$$q(h(d)) = \sum_{a\in Act}\sum_{e_a:E_a} a(f_a(d,e_a))\, q(h(g_a(d,e_a))) \lhd b_a(d,e_a) \rhd \delta.$$

We proceed as follows:

$$\begin{aligned}q(h(d)) &= \sum_{a\in Act}\sum_{e_a:E_a} a(f'_a(h(d),e_a))\, q(g'_a(h(d),e_a)) \lhd b'_a(h(d),e_a) \rhd \delta\\&\overset{(2),(3)}{=} \sum_{a\in Act}\sum_{e_a:E_a} a(f'_a(h(d),e_a))\, q(g'_a(h(d),e_a)) \lhd b_a(d,e_a) \rhd \delta\\&\overset{(4),(5)}{=} \sum_{a\in Act}\sum_{e_a:E_a} a(f_a(d,e_a))\, q(h(g_a(d,e_a))) \lhd b_a(d,e_a) \rhd \delta.\end{aligned}$$

The second case is where $FC(d)$ does not hold. Now we must show that

$$\tau q(h(d)) = \tau \sum_{a\in Act}\sum_{e_a:E_a} a(f_a(d,e_a))\, q(h(g_a(d,e_a))) \lhd b_a(d,e_a) \rhd \delta.$$

First note the following Fact:

$$\begin{aligned}q(h(d)) &= \sum_{a\in Act}\sum_{e_a:E_a} a(f'_a(h(d),e_a))\, q(g'_a(h(d),e_a)) \lhd b'_a(h(d),e_a) \rhd \delta\\&\supseteq \sum_{a\in Act\setminus\{\tau\}}\sum_{e_a:E_a} a(f'_a(h(d),e_a))\, q(g'_a(h(d),e_a))\\&\qquad \lhd b'_a(h(d),e_a) \wedge b_a(d,e_a) \rhd \delta\\&\overset{(4),(5)}{=} \sum_{a\in Act\setminus\{\tau\}}\sum_{e_a:E_a} a(f_a(d,e_a))\, q(h(g_a(d,e_a)))\\&\qquad \lhd b'_a(h(d),e_a) \wedge b_a(d,e_a) \rhd \delta\\&\overset{(2)}{=} \sum_{a\in Act\setminus\{\tau\}}\sum_{e_a:E_a} a(f_a(d,e_a))\, q(h(g_a(d,e_a))) \lhd b_a(d,e_a) \rhd \delta.\end{aligned}$$

The theorem now follows from

$$\tau q(h(d)) \overset{\ddagger}{=} \tau\left(\tau q(h(d)) + \sum_{a\in Act\setminus\{\tau\}}\sum_{e_a:E_a} a(f_a(d,e_a))q(h(g_a(d,e_a))) \lhd b_a(d,e_a) \rhd \delta\right)$$

$$\begin{aligned}
&\overset{\bigstar}{=}\tau\Bigg(\sum_{e_\tau:E_\tau}\tau q(h(g_\tau(d,e_\tau)))\lhd b_\tau(d,e_\tau)\rhd\delta\\
&\qquad+\sum_{a\in Act\backslash\{\tau\}}\sum_{e_a:E_a}a(f_a(d,e_a))q(h(g_a(d,e_a)))\lhd b_a(d,e_a)\rhd\delta\Bigg)\\
&=\tau\Bigg(\sum_{a\in Act}\sum_{e_a:E_a}a(f_a(d,e_a))q(h(g_a(d,e_a)))\lhd b_a(d,e_a)\rhd\delta\Bigg).
\end{aligned}$$

At ‡, we have used Lemma 2.7 and the Fact stated above. At ★, we have used Lemma A.2 and matching criterion (1). Recall that, since $\neg FC_\Xi(d)$ holds, there exists an e_τ such that $b_\tau(d,e_\tau)$. For the same reason, $\tau\in Act$; this justifies the last step. □

We can formulate a result similar to Theorem 3.5 in the setting of weak bisimulation semantics, which is axiomatised by the following laws (where $a\neq\tau$).

T1: $x\,\tau = x$,
T2: $\tau\,x=\tau\,x+x$,
T3: $a(\tau\,x+y)=a(\tau\,x+y)+ax$.

First, we prove the following variant of Lemma 2.7.

Lemma 3.4 (Lemma 2.7 for weak bisimulation).

$$y\subseteq x\to\tau x=\tau x+y.$$

Proof. $\tau x\overset{\text{T2}}{=}\tau x+x=\tau x+x+y\overset{\text{T2}}{=}\tau x+y.$ □

Using Lemma 3.4 rather than Lemma 2.7, we can prove the following adaptation of Theorem 3.3.

Theorem 3.5 (General equality theorem for weak bisimulation). *Let Ξ, Ψ and h be as above (recall that Ψ does not contain τ-steps). Assume that r and q are solutions of Ξ and Ψ, respectively. If I is an invariant of Ξ, $\lambda p.\lambda d{:}D.\Xi pd\lhd I(d)\rhd\delta$ is convergent and $\forall d{:}D_\Xi\,(I(d)\to C_{\Xi,\Psi,h}(d))$, then*

$$\forall d{:}D_\Xi\ I(d)\to r(d)=q(h(d))\lhd FC_\Xi(d)\rhd\tau q(h(d)).$$

4. Abstraction and idle loops

The main result of this section, Theorem 4.9, is an adaptation of Theorem 3.3 to the setting where implementations can perform unbounded sequences of internal activity.

Recall that we are concerned with the following situation. We have an implementation, defined by the LPO Φ, and a specification, defined by the LPO Ψ. We want to prove that Φ is equal to Ψ, after abstraction of internal actions in Φ. In the previous section, we have shown how to prove equality of Ψ and Ξ, which is an abstract version of Φ, where internal actions, i.e. actions not in Ψ, are hidden.

Thus our next task is to rename internal actions in Φ in such a way that the resulting LPO Ξ is convergent, i.e. does not contain τ-loops, and such that a state mapping h from Ξ to Ψ, satisfying the matching criteria, can be defined.

In the previous section, we identified τ-steps with internal actions that make progress towards a focus point, and so make progress in the protocol. Following this intuition, we only rename those occurrences of actions to τ that constitute progress in the protocol. Consider for instance the Concurrent Alternating Bit Protocol, which is explained in detail in Section 5, where a sender S repeatedly sends a pair of a datum and an alternating bit b to a receiver R through the channel K of Section 2, until an acknowledgement arrives via channel L. Obviously, losing or garbling the datum in the channel K does not constitute progress in any sense; indeed, these events give rise to an internal loop, since the sender S retransmits the datum. So these transitions are not renamed to τ. Also, the transmission of the datum by the sender is useful only when the receiver has not yet received it, i.e. is still willing to accept data with alternating bit b. Suppose that we have a formula φ that expresses that R will accept data with alternating bit b. Then we split this transmission into two transitions: one where the transmission is renamed to τ and the enabling condition is strengthened by the conjunct φ, and one where the transition is unchanged but the enabling condition is strengthened by the conjunct $\neg\varphi$.

It requires experience to identify progressing internal actions for particular applications; we hope that the examples in Section 5.1 provide sufficient intuition.

We have seen that, when the implementation has unbounded internal behaviour, not all occurrences of all internal actions can be renamed to τ, since this would give rise to a non-convergent LPO Ξ. Hence some occurrences of some internal actions in the implementation remain unchanged. However, in order to apply Theorem 3.3, the specification Ψ and abstracted implementation Ξ should run over the same set of actions, except that Ξ can perform τ-steps. To arrive at this situation, we augment Ψ with "idle" loops: for each internal action j that still occurs in Ξ, we augment Ψ with a j-loop of the form $j\ p(d) \lhd \mathsf{T} \rhd \delta$. As a consequence, the augmented specification is in every state able to do a j-step. In general, the abstracted implementation Ξ is not in every state able to perform a j-step. To remedy this we also add a j-loop to Ξ.

After these preparations, Theorem 3.3 yields that Ξ plus idle loops is equal to Ψ plus idle loops. Now by KFAR, we can abstract from these idle loops to obtain equality of implementation Φ (after abstraction of *all* internal actions) and specification Ψ.

Since the internal actions are eventually all renamed to τ, we may as well rename them first to a single internal action i, and add just a single idle loop (an i-loop) to Ξ and Ψ. This considerably smoothens the presentation.

As opposed to the previous section, the main result of this section, Theorem 4.9, is the same for weak bisimulation and branching bisimulation. In the sequel, we assume that *Ext* (the set of external actions of Φ), *Int* (the set of internal actions of Φ), and $\{\tau\}$ are mutually disjoint and finite sets of actions.

First, we introduce a number of operator transformations that are instrumental in the proof. The operator $i(\Phi)$ is Φ extended with an i-loop; $\rho_{Int}(\Phi)$ is Φ with all actions in *Int* renamed to i; $i_{Int}(\Phi)$ is a composition of the two.

Definition 4.1. Let Φ be a convergent LPO over $Ext \cup Int \cup \{\tau\}$. Let $i \in \mathrm{Act}$ be an action such that $i \notin Ext \cup Int \cup \{\tau\}$. Let ρ_{Int} be a renaming operator renaming the actions in *Int* to i. We define the following operators on LPOs.

$$i(\Phi) \stackrel{\text{def}}{=} \lambda p.\lambda d{:}D_\Phi.\Phi pd + ip(d),$$

$$\rho_{Int}(\Phi) \stackrel{\text{def}}{=} \lambda p.\lambda d{:}D_\Phi.\rho_{Int}(\Phi pd),$$

$$i_{Int}(\Phi) \overset{\text{def}}{=} i(\rho_{Int}(\Phi)).$$

The following theorem gives the relevant properties of these operators. It is proven in Appendix A as Theorem A.4; the proof uses KFAR and CL-RSP.

Theorem 4.2. *Let Φ be a convergent LPO over $Ext \cup Int \cup \{\tau\}$ such that $i \notin Ext \cup Int \cup \{\tau\}$. Assume that p_1 is a solution of Φ, p_2 is a solution of $i(\Phi)$, and p_3 is a solution of $i_{Int}(\Phi)$. Then we have, for all $d{:}D$:*
1. $\tau p_1(d) = \tau \tau_{\{i\}}(p_2(d))$,
2. $\rho_{Int}(p_2(d)) = p_3(d)$ *and*
3. $\tau \tau_{Int}(p_1(d)) = \tau \tau_{\{i\}}(p_3(d))$.

The essential technical concept in this section is a *pre-abstraction* or *partial abstraction* function ξ. If for action a and values d and e_a, $\xi(a)(d, e_a) = \mathsf{T}$, the action a in the summand is replaced by τ, while if $\xi(a)(d, e_a) = \mathsf{F}$, the summand remains unchanged. In this way, the function ξ divides occurrences of internal actions in the implementation into two categories, namely the *progressing* and *non-progressing* internal actions. In this setting, a focus point is not defined in terms of τ-steps, as in the previous section, but in terms of progressing internal actions.

In order to apply Theorem 4.9 below, one must provide not only an invariant and a state mapping h, but also a pre-abstraction.

Definition 4.3. Let Φ be an LPO and let *Int* be a finite set of actions. A *pre-abstraction function* ξ is a mapping that yields for every action $a \in Int$ an expression of sort **Bool**. The *pre-abstraction* Φ_ξ is defined by replacing every summand in Φ of the form

$$\sum_{e_a:E_a} a(f_a(d, e_a))\, p(g_a(d, e_a)) \lhd b_a(d, e_a) \rhd \delta$$

with $a \in Int$ by

$$\sum_{e_a:E_a} (\tau\, p(g_a(d, e_a)) \lhd \xi(a)(d, e_a) \rhd a(f_a(d, e_a))\, p(g_a(d, e_a))) \lhd b_a(d, e_a) \rhd \delta.$$

We extend ξ to all actions by assuming that $\xi(\tau)(d, e_\tau) = \mathsf{T}$ and $\xi(a)(d, e_a) = \mathsf{F}$ for all remaining actions.

Observe that $D_\Phi = D_{\Phi_\xi}$ and that convergence of Φ_ξ implies convergence of Φ.

We redefine the notions *convergent* and *focus point* in a setting where there is a pre-abstraction.

Definition 4.4. Let Φ be an LPO with internal actions *Int* and let ξ be a pre-abstraction function. The LPO Φ is called *convergent w.r.t.* ξ iff there is a well-founded ordering $<$ on D such that for all $a \in Int \cup \{\tau\}$, $d : D$ and all $e_a : E_a$ we have that $b_a(d, e_a)$ and $\xi(a)(d, e_a)$ imply $g_a(d, e_a) < d$. Note that this is equivalent to convergence of Φ_ξ, defined in terms of Φ and ξ.

The difference between Φ and Φ_ξ disappears when the internal actions in *Int* are hidden. This is stated in the next lemma, which is proven as Lemma A.5 in Appendix A.

Lemma 4.5. *Let Φ be an LPO that is convergent w.r.t. a pre-abstraction function ξ. Let p be a solution of Φ and p' be a solution of Φ_ξ. Then*

$$\tau_{Int}(p) = \tau_{Int}(p').$$

Definition 4.6. Let ξ be a pre-abstraction function. The *focus condition* of Φ relative to ξ is defined by

$$FC_{\Phi,Int,\xi}(d) \overset{\text{def}}{=} \forall a \in Int \cup \{\tau\} \forall e_a{:}E_a \neg\big(b_a(d,e_a) \wedge \xi(a)(d,e_a)\big).$$

Note that this is exactly the focus condition of Φ_ξ, defined in terms of Φ and ξ.

In the next definition we define the matching criteria for the case where the implementation can perform unbounded internal activity. After an instrumental technical lemma we formulate the main theorem.

Definition 4.7. Let Φ, Ψ be LPOs, where Φ runs over $Ext \cup Int \cup \{\tau\}$ (*Ext*, *Int* and $\{\tau\}$ mutually disjoint) and Ψ runs over *Ext*. Let $h : D_\Phi \to D_\Psi$ and let ξ be a pre-abstraction function. The following five conditions are called the *matching criteria for idle loops* and their conjunction is denoted by $CI_{\Phi,\Psi,\xi,h}(d)$.

$$\forall a \in Int \cup \{\tau\} \forall e_a{:}E_a \big(b_a(d,e_a) \to h(d) = h(g_a(d,e_a))\big), \qquad (1)$$
$$\forall a \in Ext \forall e_a{:}E_a \big(b_a(d,e_a) \to b'_a(h(d),e_a)\big), \qquad (2)$$
$$\forall a \in Ext \forall e_a{:}E_a \big(FC_{\Phi,Int,\xi}(d) \wedge b'_a(h(d),e_a) \to b_a(d,e_a)\big), \qquad (3)$$
$$\forall a \in Ext \forall e_a{:}E_a \big(b_a(d,e_a) \to f_a(d,e_a) = f'_a(h(d),e_a)\big), \qquad (4)$$
$$\forall a \in Ext \forall e_a{:}E_a \big(b_a(d,e_a) \to h(g_a(d,e_a)) = g'_a(h(d),e_a)\big). \qquad (5)$$

Lemma 4.8. *Let Φ, Ψ, h and ξ as in Definition* 4.7. *We find*

$$CI_{\Phi,\Psi,\xi,h}(d) \to C_{i_{Int}(\Phi_\xi),i(\Psi),h}(d).$$

Proof. Below we show that the conditions in $C_{i_{Int}(\Phi_\xi),i(\Psi),h}(d)$ follow from the conditions in $CI_{\Phi,\Psi,\xi,h}(d)$. In order to see this, we formulate the conditions of $C_{i_{Int}(\Phi_\xi),i(\Psi),h}(d)$ in terms of Φ, Ψ and ξ directly and show how they follow.

1. We must prove

   $$\forall a \in Int \cup \{\tau\} \forall e_a{:}E_a \big(\xi(a)(d,e_a) \wedge b_a(d,e_a) \to h(d) = h(g_a(d,e_a))\big).$$

   (We must consider $a \in Int$ as these are renamed to τ if $\xi(a)(d,e_a)$ holds.) Note that this condition is a direct consequence of condition 2 of $CI_{\Phi,\Psi,\xi,h}(d)$.
2. We get

   $$\forall a \in Int \cup Ext \cup \{i\} \forall e_a{:}E_a \big(b_a(d,e_a) \wedge \neg\xi(a)(d,e_a) \to b'_a(h(d),e_a)\big).$$

   In case $a \in Int$ or a is the new action i, the action a appears as i in $i_{Int}(\Phi_\xi)$. In this case $b'_i(h(d),e_b)$ equals T and the condition trivially holds.
   In case $a \in Ext$, this is exactly condition 3 of $CI_{\Phi,\Psi,\xi,h}(d)$.
3. This condition yields

   $$\forall a \in Int \cup Ext \cup \{i\} \forall e_a{:}E_a$$

$$\big(FC_{\Phi,Int,\xi}(d) \wedge b'_a(h(d), e_a) \to b_a(d, e_a) \wedge \neg\xi(a)(d, e_a)\big).$$

In case $a \in Int \cup \{i\}$, a occurs as i in $i_{Int}(\Phi_\xi)$ and $i_{Int}(\Psi)$. So the conditions $b_i(d, e_i)$ and $b'_i(h(d), e_i)$ are both equal to T. If $\xi(i)(d, e_i) = \mathsf{F}$, we are done; if $\xi(i)(d, e_i) = \mathsf{T}$, the focus condition is false and the theorem follows trivially.
In case $a \in Ext$ we have that $\xi(a)(d, e_a) = \mathsf{F}$ and the theorem follows from condition 4 of $CI_{\Phi,\Psi,\xi,h}(d)$.

4. In this case we get

$$\forall a \in Int \cup Ext \cup \{i\} \forall e_a{:}E_a$$
$$\big(\neg\xi(a)(d, e_a) \wedge b_a(d, e_a) \to f_a(d, e_a) = f'_a(h(d), e_a)\big).$$

In case $a \in Int \cup \{i\}$, a occurs as i in $i_{Int}(\Phi)$ and $i_{Int}(\Psi)$. As i has no parameter, this condition holds trivially.
In case $a \in Ext$ this is exactly condition 5 of $CI_{\Phi,\Psi,\xi,h}(d)$.

5. The last condition is

$$\forall a \in Int \cup Ext \cup \{i\} \forall e_a{:}E_a$$
$$\big(\neg\xi(a)(d, e_a) \wedge b_a(d, e_a) \to h(g_a(d, e_a)) = g'_a(h(d), e_a)\big).$$

In case $a \in Int \cup \{i\}$ the action a appears as i in $i_{Int}(\Phi_\xi)$ and $i_{Int}(\Psi)$. So, g'_i is the identity and we must prove that $h(g_a(d, e_a)) = h(d)$. This follows from condition 2 of $CI_{\Phi,\Psi,\xi,h}(d)$.
In case $a \in Ext$ this is an immediate consequence of condition 6 of $CI_{\Phi,\Psi,\xi,h}(d)$. □

Theorem 4.9 (Equality theorem for idle loops). *Let Φ, Ψ be LPOs, where Φ runs over $Ext \cup Int \cup \{\tau\}$ (Ext, Int and $\{\tau\}$ mutually disjoint) and Ψ runs over Ext. Let $h : D_\Phi \to D_\Psi$ and let ξ be a pre-abstraction function. Let p and q be solutions of Φ and Ψ, respectively.*

If I is an invariant of Φ, $\lambda p.\lambda d{:}D.\Phi pd \lhd I(d) \rhd \delta$ is convergent w.r.t. ξ and $\forall d : D_\Phi(I(d) \to CI_{\Phi,\Psi,\xi,h}(d))$, then

$$\forall d : D_\Phi I(d) \to \tau\tau_{Int}(p(d)) = \tau q(h(d)).$$

Proof. Let p, q, p' and q' be solutions of Φ, Ψ, $i_{Int}(\Phi_\xi)$ and $i_{Int}(\Psi)$, respectively. The following three facts follow straightforwardly from the work done up to now.
1. $\tau\tau_{Int}(p(d)) = \tau\tau_{\{i\}}(p'(d))$ (Theorem 4.2.3),
2. $\tau q(h(d)) = \tau\tau_{\{i\}}(q'(h(d)))$ (Theorem 4.2.1) and
3. $I(d) \to \tau p'(d) = \tau q'(h(d))$ (Theorem 3.3 and Lemma 4.8).

Note that for the third case above we must show that $i_{Int}(\Phi_\xi)$ is convergent. This is an immediate consequence of the fact that $\lambda p.\lambda d{:}D.\Phi pd \lhd I(d) \rhd \delta$ is convergent w.r.t. ξ. The theorem follows straightforwardly by

$$\begin{aligned}\tau\tau_{Int}(p(d)) &\overset{(1)}{=} \tau\tau_{\{i\}}(p'(d))\\ &\overset{(3)}{=} \tau\tau_{\{i\}}(q'(h(d)))\\ &\overset{(2)}{=} \tau q(h(d))\end{aligned}$$ □

5. Examples

In this section we give some examples. We begin with three simple ones, where invariants, progressiveness of internal actions, and convergence hardly play a role. The first example is an easy application of Theorem 4.9. The next example shows that in some cases a state mapping as required by Theorem 3.3 or Theorem 4.9 does not exist, even though the processes in question are evidently branching bisimilar. The third example motivates our restriction to specifications without τ-steps. In Section 5.1, we present a larger example, the Concurrent Alternating Bit Protocol. As an application of Theorem 4.9, we prove the correctness of this protocol. Here, invariants, progressiveness of internal actions and convergence make their appearance.

Example 5.1. The following LPO describes a person who tosses a coin (this event is modeled by the internal action j). When *head* turns up the person performs an external action *out*(*head*), when *tail* turns up the person tosses again. We write *Sides* for the sort consisting of *head* and *tail*.

$$\begin{aligned}\textbf{proc}\quad X(s{:}\mathit{Sides}) = {} & \sum_{s'{:}\mathit{Sides}} jX(s') \lhd eq(s, \mathit{tail}) \rhd \delta \\ & + out(s)X(\mathit{tail}) \lhd eq(s, \mathit{head}) \rhd \delta.\end{aligned}$$

After hiding the internal action j, this process implements the process which does nothing but *out*(*head*)-steps, given by

$$\textbf{proc}\quad Y(s{:}\mathit{Sides}) = out(\mathit{head})Y(s)$$

Here we leave the condition T of the summand implicit. The parameter s is added to Y for convenience. We use Theorem 4.9 to prove that solutions for X and Y are branching bisimilar. More precisely, let p and q be solutions for X and Y, respectively: we prove that for all $s \in \mathit{Sides}$, $\tau\tau_{\{j\}}(p(s)) = \tau q(s)$. Here we take X for Φ, Y for Ψ, $\{j\}$ for *Int* and $\{out\}$ for *Ext*. First we define the ξ-function, which determines when the internal action j is renamed to τ. The coin is tossed when s equals *tail*. When the side that turns up, s', is again *tail*, we have a j-loop (which after renaming would lead to a τ-loop). To exclude this situation, we put $\xi(j) = eq(s', \mathit{head})$. The focus condition $FC_{X,\{j\},\xi}(s)$ is now defined as $\forall s'{:}\mathit{Sides}\, \neg(eq(s, \mathit{tail}) \wedge eq(s', \mathit{head}))$, which is equivalent to $eq(s, \mathit{head})$. As invariant we simply take the always true formula T and we define $h : \mathit{Sides} \to \mathit{Sides}$ by $h(s) = \mathit{head}$.

Spelling out the matching criteria of Definition 4.7, we get the following proof obligations:

1. X is convergent w.r.t. ξ. This is easy: we let the required well-founded ordering on *Sides* be given by: $\mathit{head} < \mathit{tail}$.
2. $eq(s, \mathit{tail}) \to \mathit{head} = \mathit{head}$. This formula is trivially proven.
3. $eq(s, \mathit{head}) \to \mathsf{T}$. Equally trivial.
4. $(FC_{X,\{j\},\xi}(s) \wedge \mathsf{T}) \to eq(s, \mathit{head})$. Easy, since $FC_{X,\{j\},\xi}(s)$ is equivalent to $eq(s, \mathit{head})$.
5. $eq(s, \mathit{head}) \to s = \mathit{head}$. Trivial. Remember that we assume that eq faithfully reflects equality.
6. $eq(s, \mathit{head}) \to \mathit{head} = \mathit{head}$. Trivial.

Example 5.2. Let Y be defined as in Example 5.1. Define a function $\mathit{flip} : \mathit{Sides} \to \mathit{Sides}$ with $\mathit{flip}(\mathit{head}) = \mathit{tail}$ and $\mathit{flip}(\mathit{tail}) = \mathit{head}$ (no other equations hold). Let Z be defined by

$$\mathbf{proc}\quad Z(st{:}Sides) = out(head)Z(\mathit{flip}(st))$$

Processes defined by *Y* and *Z* are evidently strongly bisimilar. However, we cannot give a state mapping $h : Sides \to Sides$ that satisfies the matching criteria. Towards a contradiction, suppose that h exists. By criterion (6), we have $h(s) = \mathit{flip}(h(s))$, which is clearly impossible.

We conjecture that in cases like this, one can always rewrite the implementation and specification in a simple way to (branching) equivalent ones, which can be dealt with by our strategy. (In the present case, just delete the parameter *st* in *Z*.) It remains to make this more precise.

Now we show that the restriction to specifications without τ-steps cannot be dropped. We present a counterexample to this generalisation of Theorem 3.3, which also serves to refute the same generalisation of Theorem 4.9.

Example 5.3. Let *U* be defined by

$$\begin{aligned}\mathbf{proc}\quad U(st{:}Nat) = {} & \tau U(2) \lhd eq(st,1) \rhd \delta \\ & + bU(3) \lhd eq(st,2) \rhd \delta \\ & + cU(st) \lhd eq(st,3) \rhd \delta.\end{aligned}$$

Solutions for this LPO can be written as $\tau\, b\, c^{\omega}$. Next, consider

$$\begin{aligned}\mathbf{proc}\quad V(st{:}Nat) = {} & \tau V(2) \lhd eq(st,1) \rhd \delta \\ & + bV(3) \lhd eq(st,2) \rhd \delta \\ & + \tau V(3) \lhd eq(st,2) \rhd \delta \\ & + cV(st) \lhd eq(st,3) \rhd \delta.\end{aligned}$$

We have that solutions to *U* and *V* are not in general branching (or weakly) bisimilar: the infinite trace c^{ω} is an (infinite) trace of a solution for *V*, but not of a solution for *U*. However, it is easy to show that the conditions of Theorem 3.3 are satisfied, contradicting this result.

We define a state mapping *h* from *U* to *V*, of type $Nat \to Nat$, by

$$h(st) = \begin{cases} 2 & \text{if } eq(st,1), \\ st & \text{otherwise.} \end{cases}$$

The focus condition $FC_U(st)$ is equivalent to $\neg eq(st,1)$. It is easily seen that the matching criteria $C_{U,V,h}$ are satisfied. (For convergence, take the $>$ ordering on *Nat* (restricted to $\{1,2,3\}$) as the required well-founded ordering.)

The question arises whether our strategy can deal with τ-steps in the specification at all. Intuitively, these steps model that the specification internally and invisibly makes choices. In case the implementation is (after abstraction of internal actions) equal to the specification, these choices must also occur in the implementation. Usually, they will be modelled by internal but visible actions. An adaptation of our strategy could be to make the choices in the specification visible by replacing the τ-steps by the corresponding internal actions. Then one might prove this version of the specification equal to the (partially abstracted) implementation. Thereafter, hiding the internal actions in the specification yields the desired result.

5.1. The concurrent alternating bit protocol

In this subsection we prove the correctness of the Concurrent Alternating Bit Protocol (*CABP*), as an application of Theorem 4.9.

5.1.1. Specification

In this section we give the standard description of the Concurrent Alternating Bit Protocol and its specification. The system is built from six components. The overall structure of the *CABP* is depicted in Fig. 2. Information flows clockwise through this picture. The components can perform read ($r_n(\ldots)$) and send actions ($s_n(\ldots)$) to transport data over port n. A read and a send action over port n can synchronise to a communication action ($c_n(\ldots)$) over port n when they are executed simultaneously. In such a case the parameters of the send and read action must match.

We use the sort *Bit* with bits e_0 and e_1 with an inversion function *inv* and the sort *Nat* of natural numbers. We assume an unspecified sort D that contains the data elements to be transferred by the protocol. Besides pairs of a datum and a bit that are transferred, there is also an error message, *ce* (for *checksum error*), indicating that transmission fails.

The channels K and L read data at port 3, resp. port 6. They either deliver the data correctly (via port 4, resp. 7), or lose or garble the data in which case a checksum error *ce* is sent. The non-deterministic choice between the three options is modelled by the actions j and j'. If j is chosen, the data are delivered correctly and if j' happens, they are garbled or lost. The state of the channels is modeled by parameters i_k and i_l.

$$
\begin{aligned}
\textbf{proc}\quad & K(d_k{:}D, b_k{:}Bit, i_k{:}Nat)\\
&\quad = \sum_{d:D}\sum_{b:Bit} r_3(d,b)\, K(d/d_k, b/b_k, 2/i_k) \lhd eq(i_k,1) \rhd \delta\\
&\quad + (j'K(1/i_k) + jK(3/i_k) + j'\,K(4/i_k)) \lhd eq(i_k,2) \rhd \delta\\
&\quad + s_4(d_k,b_k)\, K(1/i_k) \lhd eq(i_k,3) \rhd \delta\\
&\quad + s_4(ce)\, K(1/i_k) \lhd eq(i_k,4) \rhd \delta,
\end{aligned}
$$

$$
\begin{aligned}
& L(b_l{:}Bit, i_l{:}Nat)\\
&\quad = \sum_{b:Bit} r_6(b)\, L(b/b_l, 2/i_l) \lhd eq(i_l,1) \rhd \delta\\
&\quad + \big(j'\,L(1/i_l) + j\,L(3/i_l) + j'\,L(4/i_l)\big) \lhd eq(i_l,2) \rhd \delta\\
&\quad + s_7(b_l)\, L(1/i_l) \lhd eq(i_l,3) \rhd \delta\\
&\quad + s_7(ce)\, L(1/i_l) \lhd eq(i_l,4) \rhd \delta.
\end{aligned}
$$

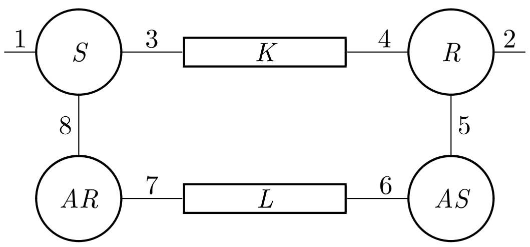

Fig. 2. The structure of the *CABP*.

The sender S reads a datum of sort D at port 1 and repeatedly offers the datum (with a bit attached) at port 3 until it receives an acknowledgement *ac* at port 8 after which the bit-to-be-attached is inverted, and the next datum over port 1 is awaited.

$$\begin{aligned}\textbf{proc}\quad & S(d_s{:}D, b_s{:}Bit, i_s{:}Nat)\\ &= \sum_{d:D} r_1(d)\, S(d/d_s, 2/i_s) \lhd eq(i_s,1) \rhd \delta\\ &\quad \big(s_3(d_s,b_s)\, S + r_8(ac)\, S(inv(b_s)/b_s, 1/i_s)\big) \lhd eq(i_s,2) \rhd \delta.\end{aligned}$$

The receiver R reads a datum at port 4 and if the datum is not a checksum error *ce* and if the bit attached is the expected bit, it sends the datum via port 2 and sends (via port 5) an acknowledgement *ac* to the acknowledgement sender *AS*, after which the bit-to-be-expected is inverted. If the datum is a checksum error or the bit attached is not the expected bit, the datum is ignored.

$$\begin{aligned}\textbf{proc}\quad R(d_r{:}D, b_r{:}Bit, i_r{:}Nat) = {} & \sum_{d:D} r_4(d,b_r)\, R(d/d_r, 2/i_r) \lhd eq(i_r,1) \rhd \delta\\ & + \Big(r_4(ce) + \sum_{d:D} r_4(d, inv(b_r))\Big)\, R \lhd eq(i_r,1) \rhd \delta\\ & + s_2(d_r)\, R(3/i_r) \lhd eq(i_r,2) \rhd \delta\\ & + s_5(ac)\, R(inv(b_r)/b_r, 1/i_r) \lhd eq(i_r,3) \rhd \delta.\end{aligned}$$

The acknowledgement sender *AS* repeatedly sends its acknowledgement bit via port 6, until it reads an acknowledgement *ac* at port 5, after which the acknowledgement bit is inverted.

$$\textbf{proc}\quad AS(b'_r{:}Bit) = r_5(ac) AS(inv(b'_r)) + s_6(b'_r) AS(b'_r).$$

The acknowledgement receiver *AR* reads bits at port 7 and when the bit is the expected acknowledgement bit, it sends via port 8 an acknowledgement *ac* to the sender *S*, after which the bit-to-be-expected is inverted. Checksum errors *ce* or unexpected bits are ignored.

$$\begin{aligned}\textbf{proc}\quad AR(b'_s{:}Bit, i'_s{:}Nat) = {} & r_7(b'_s)\, AR(2/i'_s) \lhd eq(i'_s,1) \rhd \delta\\ & + (r_7(ce) + r_7(inv(b'_s)))\, AR \lhd eq(i'_s,1) \rhd \delta\\ & + s_8(ac)\, AR(inv(b'_s)/b'_s, 1/i'_s) \lhd eq(i'_s,2) \rhd \delta.\end{aligned}$$

The *CABP* is obtained by putting the components in parallel and encapsulating the internal send and read actions at ports $n \in \{3,4,5,6,7,8\}$. Synchronisation between the components is modelled by communication actions at connecting ports.

We put $H = \{s_3, r_3, s_4, r_4, s_5, r_5, s_6, r_6, s_7, r_7, s_8, r_8\}$.

$$\begin{aligned}\textbf{proc}\quad CABP(d{:}D) = \partial_H\big(& S(d,e_0,1) \| AR(e_0,1) \| K(d,e_1,1) \|\\ & L(e_1,1) \| R(d,e_0,1) \| AS(e_1)\big).\end{aligned}$$

The specification of the external behaviour of *CABP* uses the one-datum buffer B, which can read via port 1 if b is true, and deliver via port 2 if b is false.

$$\textbf{proc}\quad B(d{:}D, b{:}\textbf{Bool}) = \sum_{e:D} r_1(e)\, B(e, \mathsf{F}) \lhd b \rhd \delta + s_2(d)\, B(d, \mathsf{T}) \lhd \neg b \rhd \delta.$$

After abstraction of internal actions, the *CABP* should behave as a one-datum buffer, up to initial silent steps. We let $I = \{c_3, c_4, c_5, c_6, c_7, c_8, j, j'\}$. Our goal is to prove the following result.

Theorem 5.4. *For all $d{:}D$ we have*

$$\tau\tau_I(CABP(d)) = \tau B(d, \mathsf{T}).$$

This result will be proven as Theorem 5.10, as an easy consequence of Theorem 4.9, taking a certain expansion *Sys* of *CABP* for Φ, B for Ψ, the set I for *Int*, and $\{r_1, s_2\}$ for *Ext*. In the next section, we determine *Sys*.

5.1.2. Expansion

In this section we expand *CABP* to a linear process term *Sys*. As a preparation, we first group S and AR, respectively, R and AS, together. This has the advantage that we can dispose of the parameters b'_s and b'_r. For $d_s, d_r, d_k : D, b_s, b_r, b_k, b_l : Bit$ and $i_s, i'_s, i_r, i_k, i_l : Nat$, we define:

$$\begin{aligned}
\textbf{proc}\quad & SAR(d_s, b_s, i_s, i'_s) = S(d_s, b_s, i_s) \| AR(b_s, i'_s)\\
& RAS(d_r, b_r, i_r) = R(d_r, b_r, i_r) \| AS(inv(b_r))\\
& Sys\big(d_s, b_s, i_s, i'_s, d_r, b_r, i_r, d_k, b_k, i_k, b_l, i_l\big)\\
& \quad = \partial_H\big(SAR(d_s, b_s, i_s, i'_s) \| K(d_k, b_k, i_k) \| L(b_l, i_l) \| RAS(d_r, b_r, i_r)\big).
\end{aligned}$$

Lemma 5.5. *For all $d{:}D$ we have*

$$CABP(d) = Sys\big(d, e_0, 1, 1, d, e_0, 1, d, e_1, 1, e_1, 1\big).$$

Proof. Direct using the definitions. □

Lemma 5.6. *For all $d_s, d_r, d_k : D, b_s, b_r, b_k, b_l : Bit$ and $i_s, i'_s, i_r, i_k, i_l : Nat$, it holds that*

$$\begin{aligned}
& Sys\big(d_s, b_s, i_s, i'_s, d_r, b_r, i_r, d_k, b_k, i_k, b_l, i_l\big)\\
& \quad = \sum_{d:D} r_1(d) Sys(d/d_s, 2/i_s) \lhd eq(i_s, 1) \rhd \delta\\
& \qquad + c_3(d_s, b_s) Sys(d_s/d_k, b_s/b_k, 2/i_k) \lhd eq(i_s, 2) \land eq(i_k, 1) \rhd \delta\\
& \qquad + c_4(d_k, b_r) Sys(d_k/d_r, 2/i_r, 1/i_k) \lhd eq(i_r, 1) \land eq(b_r, b_k) \land eq(i_k, 3) \rhd \delta\\
& \qquad + c_4(d_k, b_r) Sys(1/i_k) \lhd eq(i_r, 1) \land eq(b_r, inv(b_k)) \land eq(i_k, 3) \rhd \delta\\
& \qquad + c_4(ce) Sys(1/i_k) \lhd eq(i_r, 1) \land eq(i_k, 4) \rhd \delta\\
& \qquad + s_2(d_r) Sys(3/i_r) \lhd eq(i_r, 2) \rhd \delta\\
& \qquad + c_5(ac) Sys(inv(b_r)/b_r, 1/i_r) \lhd eq(i_r, 3) \rhd \delta\\
& \qquad + c_6(inv(b_r)) Sys(inv(b_r)/b_l, 2/i_l) \lhd eq(i_l, 1) \rhd \delta\\
& \qquad + c_7(b_l) Sys(1/i_l, 2/i'_s) \lhd eq(i'_s, 1) \land eq(b_l, b_s) \land eq(i_l, 3) \rhd \delta\\
& \qquad + c_7(b_l) Sys(1/i_l) \lhd eq(i'_s, 1) \land eq(b_l, inv(b_s)) \land eq(i_l, 3) \rhd \delta\\
& \qquad + c_7(ce) Sys(1/i_l) \lhd eq(i'_s, 1) \land eq(i_l, 4) \rhd \delta\\
& \qquad + c_8(ac) Sys(inv(b_s)/b_s, 1/i_s, 1/i'_s) \lhd eq(i_s, 2) \land eq(i'_s, 2) \rhd \delta
\end{aligned}$$

$$
\begin{aligned}
&+\big(j'Sys(1/i_k)+j'Sys(4/i_k)\big) \lhd eq(i_k,2) \rhd \delta\\
&+jSys(3/i_k) \lhd eq(i_k,2) \rhd \delta\\
&+\big(j'Sys(1/i_l)+j'Sys(4/i_l)\big) \lhd eq(i_l,2) \rhd \delta\\
&+jSys(3/i_l) \lhd eq(i_l,2) \rhd \delta.
\end{aligned}
$$

Proof. By straightforward process algebraic calculations. □

Now this expanded version of *Sys* will play the role of Φ as introduced in Section 4. As it would decrease readability, we have chosen not to transform *Sys* to an LPO. We have taken care that all theorems are correctly applied to *Sys*.

5.1.3. *Invariant*

The process *Sys* does not behave as the buffer for all its data states. Actually, there are cases where it can perform two r_1 actions in succession without an intermediate s_2, or two successive s_2 actions without an intermediate r_1. However, such states cannot be reached from the initial state. We formalise this observation by formulating six invariant properties of *Sys*. The first five invariants $I_1, \ldots, I_5$ state what values i_s, i'_s, i_r, i_k, and i_l may have. The last invariant I_6 is less trivial. We first provide the formal definition of the invariant, thereafter we give an informal explanation of I_6.

$$
\begin{aligned}
&I_1 \equiv eq(i_s,1) \vee eq(i_s,2);\\
&I_2 \equiv eq(i'_s,1) \vee eq(i'_s,2);\\
&I_3 \equiv eq(i_k,1) \vee eq(i_k,2) \vee eq(i_k,3) \vee eq(i_k,4);\\
&I_4 \equiv eq(i_r,1) \vee eq(i_r,2) \vee eq(i_r,3);\\
&I_5 \equiv eq(i_l,1) \vee eq(i_l,2) \vee eq(i_l,3) \vee eq(i_l,4);\\
&I_6 \equiv \big(eq(i_s,1) \rightarrow eq(b_s,inv(b_k)) \wedge eq(b_s,b_r) \wedge eq(d_s,d_k)\\
&\qquad \wedge eq(d_s,d_r) \wedge eq(i'_s,1) \wedge eq(i_r,1)\big)\\
&\qquad \wedge \big(eq(b_s,b_k) \rightarrow eq(d_s,d_k)\big)\\
&\qquad \wedge \big(eq(i_r,2) \vee eq(i_r,3) \rightarrow eq(d_s,d_r) \wedge eq(b_s,b_r) \wedge eq(b_s,b_k)\big)\\
&\qquad \wedge \big(eq(b_s,inv(b_r)) \rightarrow eq(d_s,d_r) \wedge eq(b_s,b_k)\big)\\
&\qquad \wedge \big(eq(b_s,b_l) \rightarrow eq(b_s,inv(b_r))\big)\\
&\qquad \wedge \big(eq(i'_s,2) \rightarrow eq(b_s,b_l)\big).
\end{aligned}
$$

The invariant I_6 can be understood in the following way. Every component can be in exactly two modes, which we call *involved* and *unaware*.

If a component is *involved*, it has received correct information about the datum to be transmitted and has the duty to forward this information in the clockwise direction. If a component is *unaware*, it is not (yet) involved in transmitting the datum. In particular the sender S is unaware if there is nothing to transmit. The idea behind the protocol is that initially all components are in the unaware mode. When the sender S reads a datum to be transmitted it gets involved. By transmitting data the components K, R, L and AR become subsequently involved. When AR signals the acknowledgement to S by $s_8(ac)$, it is clear that the datum has correctly been delivered, and all components fall back to the unaware mode. The invariant simply expresses that if a component is in the involved mode, all components in the anti-clockwise direction up to and including the sender S must also be involved. With regard to the components K and R the invariant also expresses the property that if these components are involved, then the data that these contain must be equal to the datum of the sender.

Below we present a table indicating in which case a component is involved, and in case it is involved, what property should hold. It is left to the reader to check that the invariant indeed encodes the intuition explained above. Note that *AS* has been omitted as its parameters do not play a role in *Sys*.

Component	Condition for involvement	Property
S	$eq(i_s, 2)$	
K	$eq(b_s, b_k)$	$eq(d_s, d_k)$
R	$eq(i_r, 2) \vee eq(i_r, 3) \vee eq(b_s, inv(b_r))$	$eq(d_s, d_r)$
L	$eq(b_s, b_l)$	
AR	$eq(i'_s, 2)$	

We write $\vec{d}$ for the vector $d_s, b_s, i_s, i'_s, d_r, b_r, i_r, d_k, b_k, i_k, b_l, i_l$.

Lemma 5.7.

$$I(\vec{d}) = \bigwedge_{j=1}^{6} I_j(\vec{d})$$

is an invariant of Sys.

5.1.4. *Abstraction and focus points*

The Concurrent Alternating Bit Protocol has unbounded internal behaviour that occurs when the channels repeatedly lose data, when acknowledgements are repeatedly being sent by the receiver without being processed by the sender or when the sender repeatedly sends data to the receiver that it has already received. We define a pre-abstraction function to rename all actions in *Int* into τ except those that give rise to loops. So

$$\xi(a)(\vec{d}) = \begin{cases} \mathsf{F} & \text{if } a = j', \\ eq(b_s, b_r) & \text{if } a = c_3, \\ \neg eq(b_s, b_r) & \text{if } a = c_6, \\ \mathsf{T} & \text{for all other } a \in Int. \end{cases}$$

In case $a = j'$ either channel K or L distorts or loses data. In case $a = c_3$ and $\neg eq(b_s, b_r)$ data are being sent by the sender to the receiver that is subsequently ignored by the receiver. And in case $a = c_6$ and $eq(b_s, b_r)$, an acknowledgement sent by the receiver to the sender is ignored by the sender.

We can now derive the focus condition *FC* with respect to ξ. *FC* is the negation of the conditions that enable τ-steps in *Sys*. This results in a rather long formula, which is equivalent to the following formula (assuming that the invariant holds).

Lemma 5.8. *The invariant $I(\vec{d})$ implies that*

$$FC_{Sys,Int,\xi}(\vec{d}) = eq(i'_s, 1) \wedge eq(i_l, 1) \wedge \big((eq(i_s, 1) \wedge eq(i_k, 1)) \\ \vee (eq(i_r, 2) \wedge (eq(i_k, 3) \vee eq(i_k, 4)))\big).$$

Lemma 5.9. *$Sys(\vec{d})$ is convergent w.r.t. ξ.*

Proof. We define a well-founded ordering $\sqsubset$ by means of the function f given below as follows: $\vec{a} \sqsubset \vec{b} \Leftrightarrow f(\vec{a}) < f(\vec{b})$, where $<$ is the usual "less than" ordering on the natural numbers. Since $<$ is well-founded on the natural numbers and—as can easily be checked—f decreases with every internal step of Sys_ξ as above, we see that $\sqsubset$ does the job.

Now we give the function f. For $\alpha \in \{k, l\}$, we let $(x_1, x_2, x_3, x_4)^\alpha$ abbreviate

$$if\big(eq(i_\alpha, 1), x_1, if(eq(i_\alpha, 2), x_2, if(eq(i_\alpha, 3), x_3, x_4))\big).$$

Define $f(d_s, b_s, i_s, i'_s, d_r, b_r, i_r, d_k, b_k, i_k, b_l, i_l)$ by

$$\begin{aligned} &if(eq(i_s, 2), 9, 0) + if(eq(i'_s, 2), 0, 3) + if(eq(i_r, 2), 0, 3) \\ &\quad + if(eq(i_r, 3), 5, 0) + if\big(eq(b_r, b_k), (2, 1, 0, 3)^k, (3, 5, 4, 4)^k\big) \\ &\quad + if\big(eq(b_s, b_l), (2, 1, 0, 3)^l, (3, 5, 4, 4)^l\big). \qquad \square \end{aligned}$$

Theorem 5.10. *For all $d{:}D$ we have*

$$\tau\tau_I(CABP(d)) = \tau B(d, \mathsf{T}).$$

Proof. By Lemma 5.5 it suffices to prove, for all $d{:}D$:

$$\tau\tau_I\big(Sys(d, e_0, 1, 1, d, e_0, 1, d, e_1, 1, e_1, 1)\big) = \tau B(d, \mathsf{T}).$$

Note that the invariant I holds for the parameters of Sys such as displayed. So we can apply Theorem 4.9, taking Sys for Φ, B for Ψ, Sys' for Ξ, the set I for Int, $\{r_1, s_2\}$ for Ext, and I as invariant. It remains to pick an appropriate function h; this function will yield a pair consisting of a datum of type D and a boolean. We choose h to be

$$h(\vec{d}) = \langle d_s, eq(i_s, 1) \vee eq(i_r, 3) \vee \neg eq(b_s, b_r)\rangle.$$

The first component is the datum that is read by the buffer when $eq(i_s, 1)$ and exported when $eq(i_r, 2)$. We can take d_s, because we can show that when action $s_2(d_r)$ happens, $d_s = d_r$.

The second component of the triple is the boolean formula that controls, in terms of the parameters $\vec{d}$ of Sys, whether the buffer is enabled to read (the formula is true) or enabled to write (the formula is false). Typically, Sys is able to read when $eq(i_s, 1)$ as the read action in the sender is enabled. The sender is also enabled to read (after some internal activity) when it is still waiting for an acknowledgement, but the proper acknowledgement is on its way. This case is characterised by $\neg eq(b_s, b_r)$. The same holds when the receiver has delivered a datum, but has not yet informed the acknowledgement handler AS. In this case $eq(i_r, 3)$ holds.

Next, we verify the conditions of Theorem 4.9. We get the following conditions (omitting trivial conditions):

1. Sys is convergent w.r.t. ξ.
2. (a) $eq(i_r, 3) \to \mathsf{T} = eq(i_s, 1) \vee \neg eq(b_s, inv(b_r))$,
   (b) $eq(i_s, 2) \wedge eq(i'_s, 2) \to eq(i_r, 3) \vee \neg eq(b_s, b_r) = \mathsf{T}$.
3. $eq(i_r, 2) \to \neg(eq(i_s, 1) \vee eq(i_r, 3) \vee \neg eq(b_s, b_r))$.
4. (a) $FC_{Sys,Int,\xi}(\vec{d}) \wedge (eq(i_s, 1) \vee eq(i_r, 3) \vee \neg eq(b_s, b_r)) \to eq(i_s, 1)$.
   (b) $FC_{Sys,Int,\xi}(\vec{d}) \wedge \neg(eq(i_s, 1) \vee eq(i_r, 3) \vee \neg eq(b_s, b_r)) \to eq(i_r, 2)$.

5. $eq(i_r, 2) \to d_r = d_s$.
6. $eq(i_s, 1) \to eq(i_r, 3) \vee \neg eq(b_s, b_r) = \mathsf{F}$.

Lemma 5.9 takes care of condition 1. The remaining conditions are easily verified, under the invariant *I*.

Acknowledgments

A preliminary version of this paper was commented by Doeko Bosscher, Dennis Dams, Wan Fokkink, David Griffioen, Henri Korver, Jaco van de Pol, Judi Romijn, Alex Sellink, and Frits Vaandrager, which led to many improvements. Example 5.3 is due to Frits Vaandrager. Many thanks also go to the referees for their detailed and careful commments.

Appendix A: Elementary results

This appendix contains some technical lemmas, which are used in previous sections. We begin with simple properties of the $\_ \lhd \_ \rhd \_$ operator and the $\sum$-operator.

Lemma A.1. *For all processes x, y and (open) terms of sort* **Bool** b, b_1, b_2 *we have*:
1. $x \lhd b \rhd x = x$,
2. $x \lhd b \rhd y = y \lhd \neg b \rhd x$,
3. $x \lhd b \rhd y = x \lhd b \rhd \delta + y \lhd \neg b \rhd \delta$,
4. $x \lhd b_1 \wedge b_2 \rhd \delta = (x \lhd b_1 \rhd \delta) \lhd b_2 \rhd \delta$,
5. $x \lhd b_1 \vee b_2 \rhd \delta = x \lhd b_1 \rhd \delta + x \lhd b_2 \rhd \delta$.

Proof. (1), (2), (3): By induction on b, i.e. by distinguishing the cases where b equals T and where b equals F. (4), (5): by induction on b_1 and b_2. □

Lemma A.2. *If there is some $e{:}D$ such that $b(e)$ holds, then*

$$p = \sum_{d:D} p \lhd b(d) \rhd \delta,$$

where d does not occur free in p.

Proof. Assume $b(e)$ holds.

$$\left(\sum_{d:D} p \lhd b(d) \rhd \delta\right) \supseteq (p \lhd b(e) \rhd \delta) = p = \left(\sum_{d:D} p\right) \supseteq \left(\sum_{d:D} p \lhd b(d) \rhd \delta\right).$$

Note that in the first $\supseteq$-step we use apiom SUM3. In the second $=$-step, we use SUM1. The last step can be seen as follows.

$$\begin{aligned}\sum_{d:D} p &= \sum_{d:D} (p \lhd b(d) \rhd p)\\ &= \sum_{d:D} (p \lhd b(d) \rhd \delta + p \lhd \neg b(d) \rhd \delta)\end{aligned}$$

$$= \sum_{d:D}(p \lhd b(d) \rhd \delta) + \sum_{d:D}(p \lhd \neg b(d) \rhd \delta).$$

We use Lemma A.1.1 at the first step, and Lemma A.1.3 at the second. At the last step SUM4 is used. Note that at the first two steps we also use SUM11. □

The following result is a trivial corollary of τ-law B1.

Lemma A.3. *Let Φ be an LPO. For all processes p and data d:D we have*

$$\Phi pd = \Phi\big[\lambda d.p(d) \lhd b(d) \rhd \tau p(d)\big]d.$$

The last two results concern LPOs extended with idle loops. They are used in Section 4. Remember that we assume that *Ext*, *Int* and $\{\tau\}$ are mutually disjoint and that $i \notin Ext \cup Int \cup \{\tau\}$.

Theorem A.4. *Let Φ be a convergent LPO over $Ext \cup Int \cup \{\tau\}$ such that $i \notin Ext \cup Int \cup \{\tau\}$. Assume that p_1 is a solution of Φ, p_2 is a solution of $i(\Phi)$, and p_3 is a solution of $i_{Int}(\Phi)$. Then we have, for all d:D:*
1. $\tau p_1(d) = \tau\tau_{\{i\}}(p_2(d))$,
2. $\rho_{Int}(p_2(d)) = p_3(d)$ *and*
3. $\tau\tau_{Int}(p_1(d)) = \tau\tau_{\{i\}}(p_3(d))$.

Proof.
1. First we show $\lambda d.\tau p_1(d)$ and $\lambda d.\tau\tau_{\{i\}}(p_2(d))$ to be solutions of

   $$\Psi \overset{\text{def}}{=} \lambda p.\lambda d{:}D_\Phi.\tau\Phi pd.$$

   It is straightforward to see that $\lambda d.\tau p_1(d)$ is a solution of Ψ. We only prove that $\lambda d.\tau\tau_{\{i\}}(p_2(d))$ is a solution of Ψ.
   As p_2 is a solution of $i(\Phi)$ it holds that

   $$p_2(d) = \Phi p_2 d + i p_2(d).$$

   By an application of KFAR we find

   $$\tau\tau_{\{i\}}(p_2(d)) = \tau\tau_{\{i\}}(\Phi p_2 d).$$

   As i does not appear in Φ, we can distribute $\tau_{\{i\}}$ and we find

   $$\tau\tau_{\{i\}}(p_2(d)) = \tau\big(\Phi[\lambda d.(\tau_{\{i\}}(p_2(d)))]d\big).$$

   So, $\lambda d.\tau\tau_{\{i\}}(p_2(d))$ is a solution of Ψ.
   As Φ is convergent, Ψ is convergent. Hence, using the principle CL-RSP we find for all $d{:}D$

   $$\tau p_1(d) = \tau\tau_{\{i\}}(p_2(d)).$$

2. First observe that $i(\rho_{Int}(\Phi))$ and $\rho_{Int}(i(\Phi))$ are syntactically identical operators. So we may assume that p_3 is a solution of $\rho_{Int}(i(\Phi))$. Since p_2 is a solution of $i(\Phi)$, we also have that $\rho_{Int}(p_2(d))$ is a solution of $\rho_{Int}(i(\Phi))$. Since $\rho_{Int}(i(\Phi))$ is convergent, the desired equality follows from CL-RSP.
3. By cases 1 and 2 of this theorem we find

   $$\begin{aligned}&\tau p_1(d) = \tau\tau_{\{i\}}(p_2(d)),\\ &\rho_{Int}(p_2(d)) = p_3(d).\end{aligned} \tag{6}$$

Using the congruence properties we transform the second equation of (6) above into

$$\tau\tau_{\{i\}}(\rho_{Int}(p_2(d))) = \tau\tau_{\{i\}}(p_3(d)).$$

By axioms $R+$ and $T+$ this simplifies to

$$\tau\tau_{Int}(\tau_{\{i\}}(p_2(d))) = \tau\tau_{\{i\}}(p_3(d)).$$

Using the first equation of (6) and the Hiding laws TI, this is reduced to

$$\tau\tau_{Int}(p_1(d)) = \tau\tau_{\{i\}}(p_3(d)),$$

which we had to prove. □

Lemma A.5. *Let Φ be an LPO that is convergent w.r.t. a pre-abstraction function ξ. Let p be a solution of Φ and p' be a solution of Φ_ξ. Then*

$$\tau_{Int}(p) = \tau_{Int}(p').$$

Proof. Consider the LPO Φ_{new} where every summand of Φ of the form

$$\sum_{e_a:E_a} a(f_a(d,e_a))\, p(g_a(d,e_a)) \lhd b_a(d,e_a) \rhd \delta$$

with $a \in Int$ is replaced by

$$\sum_{e_a:E_a} (ip(g_a(d,e_a)) \lhd \xi(a)(d,e_a) \rhd a(f_a(d,e_a))\, p(g_a(d,e_a))) \lhd b_a(d,e_a) \rhd \delta,$$

where i is a fresh action. Assume Φ_{new} has solution p_{new}. Clearly, $\tau_{\{i\}}(p_{\text{new}}) = p'$ as both terms are a solution of Φ_ξ (use Lemma A.1.1). Also $\rho_{Int}(p_{\text{new}}) = \rho_{Int}(p)$ as both terms are solutions of $\rho_{Int}(\Phi)$. Furthermore, $\tau_{\{i\}}(p) = p$ as i does not occur in Φ (so both terms are solutions of Φ).

Using these observations and (at the second and fourth step) axioms $R+$ and $T+$, we derive:

$$\begin{aligned}\tau_{Int}(p) &= \tau_{Int}(\tau_{\{i\}}(p))\\ &= \tau_{\{i\}}(\rho_{Int}(p))\\ &= \tau_{\{i\}}(\rho_{Int}(p_{\text{new}}))\\ &= \tau_{Int}(\tau_{\{i\}}(p_{\text{new}}))\\ &= \tau_{Int}(p'). \qquad \square\end{aligned}$$

Appendix B: Axioms and rules for μCRL

In this section, we present tables containing the axioms for the ACP operators, some axioms for the Sum and the conditional operator, plus some additional axioms that were necessary. The axioms in Table 1 are standard axioms from ACP (see e.g. [1]). The axioms and rule in Table 2 are specifically defined for μCRL and concern the sum operator Σ, the booleans T and F and the conditional. They stem from [12]. The axioms and rule in Table 3 are various axioms from ACP that we require in the proofs (see also [1]). All axioms and rules hold in strong bisimulation, except for the axioms B1 and B2 and the rule KFAR which are typically valid in branching bisimulation.

Table 1
Axioms for the ACP operators

A1	$x+y=y+x$	CM1	$x\\|y=x⌊\\!⌊y+y⌊\\!⌊x+x\|y$
A2	$x+(y+z)=(x+y)+z$	CM2	$c⌊\\!⌊x=c\cdot x$
A3	$x+x=x$	CM3	$c\cdot x⌊\\!⌊y=c\cdot(x\\|y)$
A4	$(x+y)\cdot z=x\cdot z+y\cdot z$	CM4	$(x+y)⌊\\!⌊z=x⌊\\!⌊z+y⌊\\!⌊z$
A5	$(x\cdot y)\cdot z=x\cdot(y\cdot z)$	CM5	$c\cdot x\|d=(c\|d)\cdot x$
A6	$x+\delta=x$	CM6	$c\|d\cdot x=(c\|d)\cdot x$
A7	$\delta\cdot x=\delta$	CM7	$c\cdot x\|d\cdot y=(c\|d)\cdot(x\\|y)$
B1	$x\cdot\tau=x$	CM8	$(x+y)\|z=x\|z+y\|z$
B2	$z(\tau\cdot(x+y)+x)=z(x+y)$	CM9	$x\|(y+z)=x\|y+x\|z$
CD1	$\delta\|x=\delta$	DD	$\partial_H(\delta)=\delta$
CD2	$x\|\delta=\delta$	DT	$\partial_H(\tau)=\tau$
CT1	$\tau\|x=\delta$	D1	$\partial_H(a(d))=a(d)\quad$ if $a\notin H$
CT2	$x\|\tau=\delta$	D2	$\partial_H(a(d))=\delta\quad$ if $a\in H$
		D3	$\partial_H(x+y)=\partial_H(x)+\partial_H(y)$
CF	$a(d)\mid b(e)=\begin{cases}\gamma(a,b)(d) & \text{if } d=e \text{ and } \gamma(a,b) \text{ defined}\\ \delta & \text{otherwise}\end{cases}$	D4	$\partial_H(x\cdot y)=\partial_H(x)\cdot\partial_H(y)$
TID	$\tau_I(\delta)=\delta$	RD	$\rho_R(\delta)=\delta$
TIT	$\tau_I(\tau)=\tau$	RT	$\rho_R(\tau)=\tau$
TI1	$\tau_I(a(d))=a(d)\quad$ if $a\notin I$	R1	$\rho_R(a(d))=R(a)(d)$
TI2	$\tau_I(a(d))=\tau\quad$ if $a\in I$		
TI3	$\tau_I(x+y)=\tau_I(x)+\tau_I(y)$	R3	$\rho_R(x+y)=\rho_R(x)+\rho_R(y)$
TI4	$\tau_I(x\cdot y)=\tau_I(x)\cdot\tau_I(y)$	R4	$\rho_R(x\cdot y)=\rho_R(x)\cdot\rho_R(y)$

Table 2
Axioms for sum and conditional

SUM1	$\Sigma_{d:D}p=p$	d not free in p
SUM2	$\Sigma_{d:D}p=\Sigma_{e:D}(p[e/d])$	e not free in p
SUM3	$\Sigma_{d:D}p=\Sigma_{d:D}p+p[e/d]$	
SUM4	$\Sigma_{d:D}(p_1+p_2)=\Sigma_{d:D}p_1+\Sigma_{d:D}p_2$	
SUM5	$\Sigma_{d:D}(p_1\cdot p_2)=(\Sigma_{d:D}p_1)\cdot p_2$	d not free in p_2
SUM6	$\Sigma_{d:D}(p_1⌊\\!⌊p_2)=(\Sigma_{d:D}p_1⌊\\!⌊p_2$	d not free in p_2
SUM7	$\Sigma_{d:D}(p_1\|p_2)=(\Sigma_{d:D}p_1)\|p_2$	d not free in p_2
SUM8	$\Sigma_{d:D}(\partial_H(p))=\partial_H(\Sigma_{d:D}p)$	
SUM9	$\Sigma_{d:D}(\tau_I(p))=\tau_I(\Sigma_{d:D}p)$	
SUM10	$\Sigma_{d:D}(\rho_R(p))=\rho_R(\Sigma_{d:D}p)$	
SUM11	$\dfrac{\overset{\mathscr{D}}{p_1=p_2}}{\Sigma_{d:D}(p_1)=\Sigma_{d:D}(p_2)}$	d not free in the assumptions of $\mathscr{D}$
BOOL1	$\neg(\mathsf{T}=\mathsf{F})$	
BOOL2	$\neg(b=\mathsf{T})\rightarrow b=\mathsf{F}$	
COND1	$x\lhd\mathsf{T}\rhd y=x$	
COND2	$x\lhd\mathsf{F}\rhd y=y$	

In the tables, D is an arbitrary data type, d and e represent elements of D, x, y, z range over processes, a, b, i are actions, c, d represent either τ, δ or an action $a(d)$, and p, p_1, p_2 are process terms in which the variable d may occur. (Although some names are over-

Table 3
Some extra axioms needed in the verification

KFAR	$p(d) = ip(d) + y \to \tau\tau_{\{i\}}(p(d)) = \tau\tau_{\{i\}}(y)$
$T+$	$\tau_I(\tau_{I'}(x)) = \tau_{I \cup I'}(x)$
$R+$	$\tau_I(\rho_R(x)) = \tau_{I'}(x)$ if $ran(R) \subseteq I$ and $I' = I \cup dom(R)$
SC1	$(x \mathbin{⌊\!⌊} y) \mathbin{⌊\!⌊} z = x \mathbin{⌊\!⌊} (y \| z)$
SC3	$x \mid y = y \mid x$
SC4	$(x \mid y) \mid z = x \mid (y \mid z)$
SC5	$x \mid (y \mathbin{⌊\!⌊} z) = (x \mid y) \mathbin{⌊\!⌊} z$

loaded, the context makes clear what is meant. In Table 2, b also ranges over boolean terms.) Furthermore, R ranges over renaming functions, and I, I' and H range over sets of actions. If $R = \{a_1 \to b_1, \ldots, a_n \to b_n\}$, then $dom(R) = \{a_1, \ldots, a_n\}$ and $ran(R) = \{b_1, \ldots, b_n\}$. Finally, $\mathscr{D}$ in Table 2 ranges over derivations.

Beside these axioms, μCRL features two important principles: RSP, stating that guarded recursive specification have at most one solution, and an induction rule, for inductive reasoning over data types. For more information on μCRL, the reader is referred to [12].

References

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

[2] J.A. Bergstra, J.W. Klop, The algebra of recursively defined processes and the algebra of regular processes, in: Proceedings of the 11th ICALP, Antwerp, Lecture Notes in Computer Science, vol. 172, Springer, Berlin, 1984, pp. 82–95.

[3] M.A. Bezem, J.F. Groote, A correctness proof of a one-bit sliding window protocol in μCRL, Comput. J. 37 (4) (1994) 289–307.

[4] M.A. Bezem, J.F. Groote, Invariants in process algebra with data, in: B. Jonsson, J. Parrow (Eds.), Proceedings of the 5th Conference on Theories of Concurrency, CONCUR '94, Uppsala, Sweden, August 1994, Lecture Notes in Computer Science, vol. 836, Springer, Berlin, 1994, pp. 401–416.

[5] J.J. Brunekreef. Process specification in a UNITY format, in: A. Ponse, C. Verhoef, S.F.M. van Vlijmen (Eds.), Proceedings of the 1st Workshop in the Algebra of Communicating Processes, ACP '94, Utrecht, the Netherlands, July 1994, Workshops in Computing, vol. 458, Springer, Berlin, July 1994, pp. 319–337.

[6] C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, C. Paulin-Mohring, C. Muñoz, C. Murthy, C. Parent, A. Saïbi, B. Werner, The Coq proof assistant reference manual. Version 5.10. Technical report, INRIA-Rocquencourt—CNRS-ENS Lyon, 1995.

[7] K.M. Chandy, J. Misra, Parallel Program Design: A Foundation, Addison-Wesley, Reading, MA, 1988.

[8] L.-Å. Fredlund, J.F. Groote, H. Korver, Formal verification of a leader election protocol in process algebra, Theoret. Comput. Sci. 177 (1997) 459–486.

[9] R.J. van Glabbeek, W.P. Weijland, Branching time and abstraction in bisimulation semantics (extended abstract), in: G.X. Ritter (Ed.), Information Processing 89, 1989, pp. 613–618.

[10] J.F. Groote, F. Monin, J. Springintveld, A computer checked algebraic verification of a distributed summing protocol, Computer Science Report 97/14, Department of Mathematics and Computer Science, Eindhoven University of Technology, 1997. See also J.F. Groote, J. Springintveld, Algebraic verification of a distributed summation algorithm, Technical Report CS-R9640, CWI, Amsterdam, 1996.

[11] J.F. Groote, A. Ponse, Proof theory for μCRL: a language for processes with data, in: D.J. Andrews, J.F. Groote, C.A. Middelburg (Eds.), Proceedings of the International Workshop on Semantics of Specification Languages, Utrecht, The Netherlands, Workshops in Computer Science, Springer, Berlin, 1993, pp. 231–250.

[12] J.F. Groote, A. Ponse, The syntax and semantics of μCRL, in: A. Ponse, C. Verhoef, S.F.M. van Vlijmen (Eds.), Algebra of Communicating Processes, Workshops in Computing, Springer, Berlin, 1994, pp. 26–62.

[13] J.F. Groote, A. Ponse, Y.S. Usenko, Linearization in parallel pCRL, J. Logic and Algebraic Program. 48 (1–2) (2001) 39–70.
[14] J.F. Groote, M. Reniers, Algebraic process verification, in: J.A. Bergstra, A. Ponse, S.A. Smolka (Eds.), Handbook of Process Algebra, Elsevier, Amsterdam, 2001, pp. 1151–1208.
[15] IEEE Standard for a high performance serial bus, International Standard 1394, 1995.
[16] B. Jonsson, Compositional Verification of Distributed Systems, Ph.D. thesis, Department of Computer Systems, Uppsala University, 1987.
[17] C.P.J. Koymans, J.C. Mulder, A modular approach to protocol verification using process algebra, in: J.C.M. Baeten (Ed.), Applications of Process Algebra, Cambridge Tracts in Theoretical Computer Science, vol. 17, Cambridge University Press, Cambridge, 1990, pp. 261–306.
[18] K.G. Larsen, R. Milner, A compositional protocol verification using relativized bisimulation, Inform. Comput. 99 (1992) 80–108.
[19] N.A. Lynch, M.R. Tuttle, Hierarchical correctness proofs for distributed algorithms, in: Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing, 1987, pp. 137–151.
[20] N.A. Lynch, F.W. Vaandrager, Forward and backward simulations. Part I: untimed systems, Inform. Comput. 121 (1995) 214–233.
[21] N. Shankar, S. Owre, J.M. Rushby, The PVS Proof Checker: A Reference Manual, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.
[22] C. Shankland, M.B. van der Zwaag, The tree identify protocol of IEEE 1394 in μCRL, Formal Aspects Comput. 10 (1998) 509–531.
[23] Y.S. Usenko, Linearization of μCRL processes (to appear).
[24] F.W. Vaandrager, Some observations on redundancy in a context, in: J.C.M. Baeten (Ed.), Applications of Process Algebra, Cambridge Tracts in Theoretical Computer Science, vol. 17, Cambridge University Press, Cambridge, 1990, pp. 237–260.
[25] M.B. van der Zwaag, The cones and foci proof technique for timed transition systems, Technical Report SEN-R0038, CWI, Amsterdam, 2000, Accepted for publication in Information Processing Letters.