Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Focus points and convergent process operators

J.F. Groote and J.G. Springintveld

Computer Science/Department of Software Technology

CS-R9566 1995

Report CS-R9566
ISSN 0169-118X

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

Focus Points and Convergent Process Operators

A Proof Strategy for Protocol Verification

Jan Friso Groote
Department of Philosophy, Utrecht University
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands
jfg@phil.ruu.nl

Jan Springintveld
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
js@cwi.nl

Abstract

We present a strategy for finding algebraic correctness proofs for communication systems. It is described in the setting of μCRL [11], which is, roughly, ACP [2, 3] extended with a formal treatment of the interaction between data and processes.

The strategy has already been applied successfully in [4] and [10], but was not explicitly identified as such. Moreover, the protocols that were verified in these papers were rather complex, so that the general picture was obscured by the amount of details. In this paper, the proof strategy is materialised in the form of definitions and theorems. These results reduce a large part of protocol verification to a number of trivial facts concerning data parameters occurring in implementation and specification. This greatly simplifies protocol verifications and makes our approach amenable to mechanical assistance; experiments in this direction seem promising.

The strategy is illustrated by several small examples and one larger example, the Concurrent Alternating Bit Protocol (CABP). Although simple, this protocol contains a large amount of internal parallelism, so that all relevant issues make their appearance.

AMS Subject Classification (1991): 68M10, 68Q10, 68Q22, 68Q60, 68Q65, 68Q70, 68Q75.
CR Subject Classification (1991): C.2.2, D.2.4, F.1.2., F.3.1.
Keywords & Phrases: Communication protocols, process algebra, protocol verification, linear process operators, ACP, μCRL.
Note: The second author is supported by the Netherlands Computer Science Research Foundation (SION) with financial support of the Netherlands Organisation for Scientific Research (NWO). A preliminary version of this paper appeared in *Models and Proofs, proceedings of AMAST workshop on Real-Time systems and Opération Inter-PRC "Modèles et Preuves"*, Bordeaux, 1995.

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 focus on communication protocols and present a proof strategy to verify the correctness of such protocols in

the framework of process algebra. This strategy has implicitly been used in [4] and [10] as well as in a number of unpublished verifications. It appeared to structure and simplify the proofs considerably. In this paper we explicitly present the strategy. We work in the theory μCRL [11], which is, roughly, ACP [2, 3] 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

$$\tau_I(\partial_H(C_1 \parallel \ldots \parallel C_n)) = Spec.$$

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 will be explained below.

In simple cases, the equation can be proved 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 $\tau_I(\partial_H(C_1 \parallel \ldots \parallel C_n))$ 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 there exist no infinite chains of unguarded occurrences of recursive process variables.

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 [8] and to the precondition/effect notation of [13] and [18], 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 the principle RSP, we use the Concrete Invariant Corollary (taken from [5]) that says that if G is convergent and the processes $\tau_I(\partial_H(C_1 \parallel \ldots \parallel C_n))$ and *Spec* are solutions of G under the assumption of some invariant, then the two processes are equal in all states satisfying the invariant. Since the invariant supposedly holds for the initial state, we are done. We obtain G from the expanded version of the implementation by carefully renaming internal actions to the silent step τ so that the result is convergent.

Exploiting the symbolic representation of state spaces, we reduce the task of proving implementation and specification solutions for G 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 [19]. 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; currently, our approach is being implemented in the proof-assistant Coq [9, 15].

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. [4, 10]). 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 counter example to our main results in case the specification is allowed to contain τ-steps. We will also give 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. Another future topic will be to exploit possible connections with the theory of simulations.

Related work. We have incorporated several well-known and useful concepts such as precondition/effect notation, invariants and simulations in an algebraic framework, leading to a powerful methodology. The linear process format is similar to the UNITY format of [8] and to the precondition/effect notation of [13] and [18]. Our state mappings are comparable to weak refinements. For a comprehensive treatment of refinements and other simulation relations, see [19]. Invariants are omnipresent in computer science. Proof strategies for protocol verification in an algebraic style appear among others in [16, 17, 21].

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 specialised in Section 4 to the verification of communication protocols 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.

Acknowledgements. A preliminary version of this paper was read by Doeko Bosscher, Dennis Dams, Wan Fokkink, David Griffioen, Henri Korver, Jaco van de Pol, Judi Romijn, Alex Sellink, and Frits Vaandrager. Their comments and subsequent discussions lead to many improvements. Example 5.3 is due to Frits Vaandrager.

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 [11], which is, roughly, ACP [2, 3] extended with a formal treatment of the interaction between data and processes.

2.1 A short description of μCRL

The language μCRL is a process algebra comprising data [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 $if(b, t_1, t_2)$ 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 Subsection 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 x or do an action of y and continue to behave as y.

Interleaving parallelism is modeled 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 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 $\between$, 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 $\partial_{\{a,b\}}(x \parallel y)$ 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. 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 that characterise the operators are given in Appendix B.

2.2 Linear process operators

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

Definition 2.1. A *linear process operator* (*LPO*) over data type D is an expression of the form

$$\Phi = \lambda p.\lambda d{:}D.\sum_{i\in I}\sum_{e_i:E_i} c_i(f_i(d,e_i))\cdot p(g_i(d,e_i)) \lhd b_i(d,e_i) \rhd \delta$$

for some finite index set I, actions $c_i \in \mathsf{Act} \cup \{\tau\}$, data types E_i, D_i, and functions $f_i : D \to E_i \to D_i$, $g_i : D \to E_i \to D$, $b_i : D \to E_i \to \mathbf{Bool}$. (We assume that τ has no parameter.) □

We will give an example below. Note that the bound variable p ranges over processes parameterised with a datum of sort D. When writing $I = \{1,\ldots,n\}$, we use a meta-sum notation $\Sigma_{i\in I} p_i$ for $p_1 + p_2 + \cdots + p_n$; the p_i's are called *summands* of $\Sigma_{i\in I} p_i$.

In [4] an LPO is defined as having also summands that allow termination. We have omitted these here, because they hardly occur in actual specifications and obscure the presentation of the theory. Moreover, it is not hard to add them if so required.

LPOs are defined having a single data parameter. The LPOs that we will 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 $\Sigma_{e_i:E_i} c_i(f_i(\vec{d}, e_i))\, p(d_i'/d_i) \lhd b_i(\vec{d}, e_i) \rhd \delta$ in the definition of a process $p(\vec{d})$ abbreviates $\Sigma_{e_i:E_i} c_i(f_i(\vec{d}, e_i))\, p(d_1, \ldots, d_{i-1}, d_i', d_{i+1}, \ldots d_n) \lhd b_i(\vec{d}, e_i) \rhd \delta$. 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 $\Sigma_{e_i:E_i} c_i(f_i(\vec{d}, e_i))\, p \lhd b_i(\vec{d}, e_i) \rhd \delta$.

LPOs are often defined equationally. We give an example of an LPO K which is a channel that reads frames consisting of a datum from some data type D and an alternating bit. It either delivers the frame 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 modeled by the actions j and j'. If j is chosen the frame is delivered correctly and if j' happens it is garbled or lost. The state of the channel is modeled by the parameter i_k.

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

Note that we have deviated from the pure LPO format: 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. But, using axiom SUM1 from Appendix B, we can always add a dummy summation over some data type. Also, it is possible to give j and j' some dummy argument. Finally, using axiom SUM4, the $\sum$-operator can be distributed over the $+$. In the sequel we will allow ourselves these 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 $d : D$, $p(d) = \Phi pd$. □

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 $i \in I$ with $c_i = \tau$ and for all $e_i : E_i$, $d : D$ we have that $b_i(d, e_i)$ implies $g_i(d, e_i) < d$. □

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 solution and converging LPOs have precisely one solution.

Definition 2.4. We assume the following two principles:

- L-RDP : For all d of sort D and LPOs Φ over D we have $\langle\Phi\rangle(d) = \Phi\langle\Phi\rangle d$

- 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 $p(d) = \Phi pd \rightarrow p = \langle \Phi \rangle$.

□

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

The following general theorem, taken from [5], 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 2.5. An *invariant* of an LPO Φ written as in Definition 2.1 is a function $I : D \rightarrow \mathbf{Bool}$ such that for for all $i \in I$, $e_i : E_i$, and $d : D$ we have:

$$b_i(d, e_i) \wedge I(d) \rightarrow I(g_i(d, e_i)).$$

□

Theorem 2.6 *(Concrete Invariant Corollary [5]). Let Φ be an LPO. If, for some invariant I of Φ, the LPO $\lambda p.\lambda d.\Phi pd \lhd I(d) \rhd \delta$ is convergent and for some processes q, q', parameterised by a datum of type D, we have*

$$I(d) \rightarrow q(d) = \Phi qd,$$
$$I(d) \rightarrow q'(d) = \Phi q'd,$$

then

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

To develop the theory it is convenient to work with a particular form of LPOs, which we call *deterministic*. Deterministic LPOs contain, for each action a, at most one summand starting with a. Thus deterministic LPOs can be defined by summation over a finite set of actions instead of over a general finite index set I.

Definition 2.7. Let $Act \subset \mathsf{Act}$ be a finite set of actions, possibly extended with τ. A *deterministic linear process operator* (*D-LPO*) over Act is an expression of the form

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

□

The following theorem states that it is no restriction to assume that LPOs are deterministic.

Theorem 2.8.

1. *Every convergent LPO Φ can be rewritten to a D-LPO Φ' with the same solution, provided every occurrence of an action a in Φ has a parameter of a unique type D_a.*

2. *Consider convergent D-LPOs Φ, Ψ such that action a occurs both in Φ and in Ψ (with parameters of the same data type). There exist convergent D-LPOs Φ', Ψ' having the same solutions as Φ, Ψ, respectively, such that a occurs in Φ' and Ψ' in summands with summation over the same sort E_a.*

This result is proved as Theorem A.4 in Appendix A. Here we just give an example. The two summands $s(\langle d, b\rangle)\, K(1/i_k) \lhd eq(i_k, 3) \rhd \delta$ and $s(ce)\, K(1/i_k) \lhd eq(i_k, 4) \rhd \delta$ of the channel K can be grouped together as

$$s(if(eq(i_k, 3), \langle d, b\rangle, ce))\, K(1/i_k) \lhd eq(i_k, 3) \vee eq(i_k, 4) \rhd \delta.$$

Here we assume that ce is of the same sort as the pair $\langle d, b\rangle$.

We end this subsection by remarking that, due to the symbolic representation of state spaces, the parallel composition of LPOs can be computed very easily. This property is well-known for similar formats. For LPOs, the precise formulation is given by Lemma A.3 from Appendix A. Currently, a tool set for linear processes, which handles expansion and many other operations, is being built using the ASF-SDF meta-environment [6, 14].

2.3 Internal actions

We work in the setting of branching bisimulation [22], but provide results for 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 2.9.

$$y \subseteq x \rightarrow \tau x = \tau(\tau x + y)$$

Proof. $\tau x = \tau(x + y) \stackrel{\text{B2}}{=} \tau(\tau(x + y) + y) = \tau(\tau x + y)$. ⊠

We also assume a principle of fair abstraction, in the form of Koomen's Fair Abstraction Rule (KFAR). The formulation below is the one valid in branching bisimulation:

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

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. 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. They are inspired by numerous case studies in protocol verification, and reduce complex calculations to a few straightforward checks.

In order to understand the matching criteria we first introduce an important concept, called a focus point. A focus point is a state in the implementation without outgoing τ-steps. Focus points are characterised by the *focus condition* $FC_\Xi(d)$, which is true if d is a focus point, and false if not.

Definition 3.1. The *focus condition* $FC_\Xi(d)$ of Ξ is the formula $\neg\exists e_\tau{:}E_\tau\ (b_\tau(d,e_\tau))$. □

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 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. Here, a direct match means that the same set of external actions can be executed directly (requirement (3) and (4) below), with the same data parameter (requirement (5)) and leading to h-related states (requirement (6)). If in non-focus points a visible action can be done, then this action must also be possible in the specification (requirement (3) below). But if an h-image in the specification of a non-focus point s in the implementation can perform an action, the non-focus point s need not match it directly. As Ξ is convergent a focus point will be reached after a finite number of internal steps. Due to condition (2) this focus point will have the same h-image as s, and can therefore perform the same actions. So, it is guaranteed that s can eventually mimic the step of its h-image.

The situation is depicted very schematically in Figure 1. Here the dashed arrows are internal actions (τ-steps) that are all directed towards the focus point. Since in a focus point there is a perfect match between implementation and specification, we can say that 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. Note that as we have assumed that Ξ is convergent, each internal step in Figure 1 is directed towards the focus point. This is a real restriction, as in general there may be loops of internal actions, for instance if data

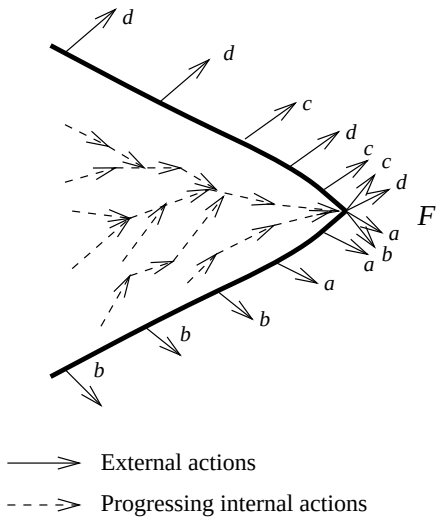

Figure 1: A cone and a focus point

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.

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

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

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

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

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

$$\forall a \in Act \setminus \{\tau\} \forall e_a{:}E_a\,(FC_\Xi(d) \wedge b'_a(h(d),e_a) \rightarrow b_a(d,e_a)) \tag{4}$$

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

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

□

Criterion (1) says that Ξ must be convergent. In effect this does not say anything else than that in a cone every internal action τ constitutes progress towards a focus point.

Criterion (2) says that if in a state d in the implementation an internal step can be done (i.e. $b_\tau(d,e_\tau)$ is valid) then this internal step is not observable. This is described by saying that both states relate to the same state in the specification.

Criterion (3) says that when the implementation can perform an external step, then the corresponding point in the specification must also be able to perform this step. Note that

in general, the converse need not hold. If the specification can perform an a-action in a certain state e, then it is only necessary that in every state d of the implementation such that $h(d) = e$ an a-step can be done *after some internal actions*.

This is guaranteed by criterion (4). It says that in a focus point of the implementation, an action a in the implementation can be performed if it is enabled in the specification.

Criteria (5) and (6) express that corresponding external actions carry the same data parameter (modulo h) and lead to corresponding states.

Assume that r and q are solutions of Ξ and Ψ, respectively. Using the matching criteria, we would like to prove that, for all $d{:}D$, $C_{\Xi,\Psi,h}(d)$ implies $r(d) = q(h(d))$.

In fact we prove a more complicated result. This has two reasons. The first one is that the statement above is not generally true. Consider the case where d is a non-focus point of Ξ. In this case, $r(d)$ can perform a τ-step. Since q cannot perform τ-steps, $r(d)$ cannot be equal to $q(h(d))$. Therefore, in the setting of branching bisimulation we can for non-focus points d only prove $\tau\, r(d) = \tau\, q(h(d))$. (In the setting of weak bisimulation this simplifies to $r(d) = \tau\, q(h(d))$.)

The second reason why we need a more complicated result is of a very general nature. 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. Therefore we have added an invariant to the theorem below.

Theorem 3.3 *(General Equality Theorem). 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 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:

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

We prove the theorem as an application of the Concrete Invariant Corollary (Theorem 2.6) 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 $\lambda r.\lambda d{:}D_\Xi = \Omega r d \lhd I(d) \rhd \delta$ is convergent too.

Using Lemma A.5 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.5, this boils down to proving the following equation.

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

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:

$$
\begin{array}{l}
q(h(d)) = \\
\sum_{a\in Act}\sum_{e_a:E_a} a(f'_a(h(d),e_a))\,q(g'_a(h(d),e_a)) \lhd b'_a(h(d),e_a) \rhd \delta \overset{(3),(4)}{=} \\
\sum_{a\in Act}\sum_{e_a:E_a} a(f'_a(h(d),e_a))\,q(g'_a(h(d),e_a)) \lhd b_a(d,e_a) \rhd \delta \overset{(5),(6)}{=} \\
\sum_{a\in Act}\sum_{e_a:E_a} a(f_a(d,e_a))\,q(h(g_a(d,e_a))) \lhd b_a(d,e_a) \rhd \delta
\end{array}
$$

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:

$$
\begin{array}{l}
q(h(d)) = \\
\sum_{a\in Act}\sum_{e_a:E_a} a(f'_a(h(d),e_a))\,q(g'_a(h(d),e_a)) \lhd b'_a(h(d),e_a) \rhd \delta \supseteq \\
\sum_{a\in Act\setminus\{\tau\}}\sum_{e_a:E_a} a(f'_a(h(d),e_a))\,q(g'_a(h(d),e_a)) \lhd b'_a(h(d),e_a) \wedge b_a(d,e_a) \rhd \delta \overset{(5),(6)}{=} \\
\sum_{a\in Act\setminus\{\tau\}}\sum_{e_a:E_a} a(f_a(d,e_a))\,q(h(g_a(d,e_a))) \lhd b'_a(h(d),e_a) \wedge b_a(d,e_a) \rhd \delta \overset{(3)}{=} \\
\sum_{a\in Act\setminus\{\tau\}}\sum_{e_a:E_a} a(f_a(d,e_a))\,q(h(g_a(d,e_a))) \lhd b_a(d,e_a) \rhd \delta.
\end{array}
$$

The theorem now follows by:

$$
\begin{array}{l}
\tau\,q(h(d)) \overset{\ddagger}{=} \\
\tau\,(\tau\,q(h(d)) + \\
\quad \sum_{a\in Act\setminus\{\tau\}}\sum_{e_a:E_a} a(f_a(d,e_a))\,q(h(g_a(d,e_a))) \lhd b_a(d,e_a) \rhd \delta) \overset{\star}{=} \\
\tau\,(\sum_{e_\tau:E_\tau} \tau\,q(h(g_\tau(d,e_\tau))) \lhd b_\tau(d,e_\tau) \rhd \delta + \\
\quad \sum_{a\in Act\setminus\{\tau\}}\sum_{e_a:E_a} a(f_a(d,e_a))\,q(h(g_a(d,e_a))) \lhd b_a(d,e_a) \rhd \delta) = \\
\tau\,(\sum_{a\in Act}\sum_{e_a:E_a} a(f_a(d,e_a))\,q(h(g_a(d,e_a))) \lhd b_a(d,e_a) \rhd \delta)
\end{array}
$$

At $\ddagger$, we have used Lemma 2.9 and the Fact stated above. At $\star$, we have used Lemma A.2 and matching criterion (2). 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 similar result 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 2.9.

Lemma 3.4 *(Lemma 2.9 for weak bisimulation).*

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

Proof. $\tau x \stackrel{\mathrm{T2}}{=} \tau x + x = \tau x + x + y \stackrel{\mathrm{T2}}{=} \tau x + y$. ⊠

Using Lemma 3.4 rather than Lemma 2.9, 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 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 that constitute progress in the protocol. Consider for instance the Concurrent Alternating Bit Protocol of Section 5, where a sender S repeatedly sends a datum with an alternating bit b attached to 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 Subsection 5.1 provide enough 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 combination of the two.

Definition 4.1. Let Φ be a convergent LPO over $Ext \cup Int \cup \{\tau\}$. Let $i \in \mathsf{Act}$ 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.

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

□

The following theorem gives the relevant properties of these operators. It is proved in Appendix A as Theorem A.6; 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. 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 a D-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. □

Note that if $\xi(a)(d,e_a) = \mathsf{T}$, the action a in the summand is replaced by τ, while if $\xi(a)(d,e_a) = \mathsf{F}$, the summand remains unchanged. In the remaining case, a-transitions are divided into progressing ones (renamed to τ) and non-progressing ones. 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.7 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:

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

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 D-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 6 conditions are called the *matching criteria for idle loops* and their conjunction is denoted by $CI_{\Phi,\Psi,\xi,h}(d)$.

$$\Phi \text{ is convergent w.r.t. } \xi \tag{1}$$

$$\forall a \in Int \cup \{\tau\}\, \forall e_a{:}E_a\, (b_a(d,e_a) \to h(d) = h(g_a(d,e_a))) \tag{2}$$

$$\forall a \in Ext\, \forall e_a{:}E_a (b_a(d,e_a) \to b'_a(h(d),e_a)) \tag{3}$$

$$\forall a \in Ext\, \forall e_a{:}E_a\, (FC_{\Phi,Int,\xi}(d) \wedge b'_a(h(d),e_a) \to b_a(d,e_a)) \tag{4}$$

$$\forall a \in Ext\, \forall e_a{:}E_a\, (b_a(d,e_a) \to f_a(d,e_a) = f'_a(h(d),e_a)) \tag{5}$$

$$\forall a \in Ext\, \forall e_a{:}E_a\, (b_a(d,e_a) \to h(g_a(d,e_a)) = g'_a(h(d),e_a)) \tag{6}$$

□

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 show that $i_{Int}(\Phi_\xi)$ is convergent. This is an immediate consequence of the fact that Φ is convergent w.r.t. ξ.

2. We must prove $\forall a \in Int \cup \{\tau\}\, \forall e_a{:}E_a\, (\xi(a)(d,e_a) \wedge b_a(d,e_a) \to h(d) = h(g_a(d,e_a))$. (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 $CI_{\Phi,\Psi,\xi,h}(d)$.

3. We get

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

   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 $CI_{\Phi,\Psi,\xi,h}(d)$.

4. This condition yields

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

   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 $CI_{\Phi,\Psi,\xi,h}(d)$.

5. In this case we get $\forall a \in Int \cup Ext \cup \{i\}\, \forall e_a{:}E_a\, (\neg\xi(a)(d,e_a) \wedge b_a(d,e_a) \rightarrow f_a(d,e_a) = f'_a(h(d),e_a))$.

   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 $CI_{\Phi,\Psi,\xi,h}(d)$.

6. The last condition is $\forall a \in Int \cup Ext \cup \{i\}\, \forall e_a{:}E_a\, (\neg\xi(a)(d,e_a) \wedge b_a(d,e_a) \rightarrow h(g_a(d,e_a)) = g'_a(h(d),e_a))$.

   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 $CI_{\Phi,\Psi,\xi,h}(d)$.

$\boxtimes$

Theorem 4.9 *(Equality theorem for idle loops). Let Φ, Ψ be D-LPOs, where Φ runs over $Ext \cup Int \cup \{\tau\}$ (Ext, Int and $\{\tau\}$ mutually disjoint) and Ψ runs over Ext. Let $h : D_\Phi \rightarrow D_\Psi$ 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 and $\forall d : D_\Phi\, (I(d) \rightarrow CI_{\Phi,\Psi,\xi,h}(d))$, then

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

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. $I(d) \rightarrow \tau\, p'(d) = \tau\, q'(h(d))$ (Theorem 3.3 and Lemma 4.8).

The theorem follows straightforwardly by

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

$\boxtimes$

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 Subsection 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*.

proc $X(s{:}Sides) = \sum_{s':Sides} j\, X(s') \lhd eq(s, tail) \rhd \delta + out(s)\, X(tail) \lhd eq(s, head) \rhd \delta$

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

proc $Y(s{:}Sides) = out(head)\, Y(s)$

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 $s \in Sides$, $\tau\, \tau_{\{j\}}(p(s)) = \tau\, q(s)$. 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 $\xi(j) = eq(s', head)$. The focus condition $FC_{X,\{j\},\xi}(s)$ is now defined as $\forall s'{:}Sides\ \neg(eq(s, tail) \wedge eq(s', head))$, which is equivalent to $eq(s, head)$. As invariant we simply take the always true formula T and we define $h : Sides \to Sides$ by $h(s) = head$.

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: $head < tail$.
2. $eq(s, tail) \to head = head$. This formula is trivially proved.
3. $eq(s, head) \to \mathsf{T}$. Equally trivial.
4. $(FC_{X,\{j\},\xi}(s) \wedge \mathsf{T}) \to eq(s, head)$. Easy, since $FC_{X,\{j\},\xi}(s)$ is equivalent to $eq(s, head)$.
5. $eq(s, head) \to s = head$. Trivial. Remember that we assume that eq faithfully reflects equality.
6. $eq(s, head) \to head = head$. Trivial.

(End example.)

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

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

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. *(End example.)*

Now we show that the restriction to specifications without τ-steps cannot be dropped. We present a counter example 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

proc $U(st{:}Nat) =$
$\quad \tau\, U(2) \lhd eq(st,1) \rhd \delta +$
$\quad b\, U(3) \lhd eq(st,2) \rhd \delta +$
$\quad c\, U(st) \lhd eq(st,3) \rhd \delta$

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

proc $V(st{:}Nat) =$
$\quad \tau\, V(2) \lhd eq(st,1) \rhd \delta +$
$\quad b\, V(3) \lhd eq(st,2) \rhd \delta +$
$\quad \tau\, V(3) \lhd eq(st,2) \rhd \delta +$
$\quad c\, V(st) \lhd eq(st,3) \rhd \delta$

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 modeled 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. *(End example.)*

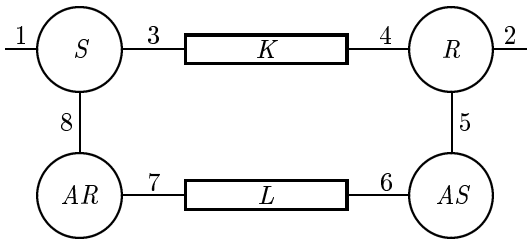

Figure 2: The structure of the *CABP*

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 Figure 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. The sort *Frame* consists of pairs $\langle d, b\rangle$ with $d : D$ and $b : Bit$ (b models the alternating bit). This sort also contains two error messages, *ce* (for *checksum error*) and *ae* (for *acknowledgement error*).

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 the last case a checksum error *ce* (resp., acknowledgement error *ae*)) is sent. The non-deterministic choice between the three options is modeled 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.

proc $K(d_k{:}D, b_k{:}Bit, i_k{:}Nat) =$
$\sum_{d:D} \sum_{b:Bit} r_3(\langle d, b\rangle)\, K(d/d_k, b/b_k, 2/i_k) \lhd eq(i_k, 1) \rhd \delta +$
$(j'\, K(1/i_k) + j\, K(3/i_k) + j'\, K(4/i_k)) \lhd eq(i_k, 2) \rhd \delta +$
$s_4(\langle d_k, b_k\rangle)\, K(1/i_k) \lhd eq(i_k, 3) \rhd \delta +$
$s_4(ce)\, K(1/i_k) \lhd eq(i_k, 4) \rhd \delta$

$L(b_l{:}Bit, i_l{:}Nat) =$
$\sum_{b:Bit} r_6(b)\, L(b/b_l, 2/i_l) \lhd eq(i_l, 1) \rhd \delta +$

$(j'\,L(1/i_l) + j\,L(3/i_l) + j'\,L(4/i_l)) \lhd eq(i_l, 2) \rhd \delta +$
$s_7(b_l)\,L(1/i_l) \lhd eq(i_l, 3) \rhd \delta +$
$s_7(ae)\,L(1/i_l) \lhd eq(i_l, 4) \rhd \delta$

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.

proc $S(d_s{:}D, b_s{:}Bit, i_s{:}Nat) =$
$\sum_{d:D} r_1(d)\,S(d/d_s, 2/i_s) \lhd eq(i_s, 1) \rhd \delta$
$(s_3(\langle d_s, b_s\rangle)\,S + r_8(ac)\,S(inv(b_s)/b_s, 1/i_s)) \lhd eq(i_s, 2) \rhd \delta$

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.

proc $R(d_r{:}D, b_r{:}Bit, i_r{:}Nat) =$
$\sum_{d:D} r_4(\langle d, b_r\rangle)\,R(d/d_r, 2/i_r) \lhd eq(i_r, 1) \rhd \delta +$
$(r_4(ce) + \sum_{d:D} r_4(\langle d, inv(b_r)\rangle))\,R \lhd eq(i_r, 1) \rhd \delta +$
$s_2(d_r)\,R(3/i_r) \lhd eq(i_r, 2) \rhd \delta +$
$s_5(ac)\,R(inv(b_r)/b_r, 1/i_r) \lhd eq(i_r, 3) \rhd \delta$

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.

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

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. Acknowledgements errors ae or unexpected bits are ignored.

proc $AR(b'_s{:}Bit, i'_s{:}Nat) =$
$r_7(b'_s)\,AR(2/i'_s) \lhd eq(i'_s, 1) \rhd \delta +$
$(r_7(ae) + r_7(inv(b'_s)))\,AR \lhd eq(i'_s, 1) \rhd \delta +$
$s_8(ac)\,AR(inv(b'_s)/b'_s, 1/i'_s) \lhd eq(i'_s, 2) \rhd \delta$

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 modeled by communication actions at connecting ports.
We put $H = \{s_3, r_3, s_4, r_4, s_5, r_5, s_6, r_6, s_7, r_7, s_8, r_8\}$.

proc $CABP(d{:}D) =$
$\partial_H(S(d, e_0, 1) \parallel AR(e_0, 1) \parallel K(d, e_1, 1) \parallel L(e_1, 1) \parallel R(d, e_0, 1) \parallel AS(e_1))$

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.

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

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 proved 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:

proc $SAR(d_s, b_s, i_s, i'_s) = S(d_s, b_s, i_s) \parallel AR(b_s, i'_s)$

$RAS(d_r, b_r, i_r) = R(d_r, b_r, i_r) \parallel AS(inv(b_r))$

$Sys(d_s, b_s, i_s, i'_s, d_r, b_r, i_r, d_k, b_k, i_k, b_l, i_l) =$
$\partial_H(SAR(d_s, b_s, i_s, i'_s) \parallel K(d_k, b_k, i_k) \parallel L(b_l, i_l) \parallel RAS(d_r, b_r, i_r))$

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

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

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*

$Sys(d_s, b_s, i_s, i'_s, d_r, b_r, i_r, d_k, b_k, i_k, b_l, i_l) =$
$\sum_{d:D} r_1(d)\, Sys(d/d_s, 2/i_s) \lhd eq(i_s, 1) \rhd \delta +$
$c_3(\langle d_s, b_s\rangle)\, Sys(d_s/d_k, b_s/b_k, 2/i_k) \lhd eq(i_s, 2) \wedge eq(i_k, 1) \rhd \delta +$
$c_4(\langle d_k, b_r\rangle)\, Sys(d_k/d_r, 2/i_r, 1/i_k) \lhd eq(i_r, 1) \wedge eq(b_r, b_k) \wedge eq(i_k, 3) \rhd \delta +$
$c_4(\langle d_k, b_r\rangle)\, Sys(1/i_k) \lhd eq(i_r, 1) \wedge eq(b_r, inv(b_k)) \wedge eq(i_k, 3) \rhd \delta +$
$c_4(ce)\, Sys(1/i_k) \lhd eq(i_r, 1) \wedge eq(i_k, 4) \rhd \delta +$
$s_2(d_r)\, Sys(3/i_r) \lhd eq(i_r, 2) \rhd \delta +$

$$
\begin{aligned}
&c_5(ac)\,Sys(inv(b_r)/b_r, 1/i_r) \lhd eq(i_r, 3) \rhd \delta+\\
&c_6(inv(b_r))\,Sys(inv(b_r)/b_l, 2/i_l) \lhd eq(i_l, 1) \rhd \delta+\\
&c_7(b_l)\,Sys(1/i_l, 2/i'_s) \lhd eq(i'_s, 1) \wedge eq(b_l, b_s) \wedge eq(i_l, 3) \rhd \delta+\\
&c_7(b_l)\,Sys(1/i_l) \lhd eq(i'_s, 1) \wedge eq(b_l, inv(b_s)) \wedge eq(i_l, 3) \rhd \delta+\\
&c_7(ae)\,Sys(1/i_l) \lhd eq(i'_s, 1) \wedge eq(i_l, 4) \rhd \delta+\\
&c_8(ac)\,Sys(inv(b_s)/b_s, 1/i_s, 1/i'_s) \lhd eq(i_s, 2) \wedge eq(i'_s, 2) \rhd \delta+\\
&(j'\,Sys(1/i_k) + j\,Sys(3/i_k) + j'\,Sys(4/i_k)) \lhd eq(i_k, 2) \rhd \delta+\\
&(j'\,Sys(1/i_l) + j\,Sys(3/i_l) + j'\,Sys(4/i_l)) \lhd eq(i_l, 2) \rhd \delta
\end{aligned}
$$

Proof. By straightforward process algebraic calculations, using Lemma A.3 and the auxiliary definitions given above. ⊠

Now this expanded version of Sys will play the role of Φ as introduced in section 4. Note however, that this LPO is not deterministic in the sense of Definition 2.7. As it would decrease readability, we have chosen not to transform Sys to a D-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 an r_1 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.

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

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 is 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*

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

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

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

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

$$if(eq(i_s,2),9,0) + if(eq(i'_s,2),0,3) + if(eq(i_r,2),0,3) + if(eq(i_r,3),5,0) +$$
$$if(eq(b_r,b_k),(2,1,0,3)^k,(3,5,4,4)^k) +$$
$$if(eq(b_s,b_l),(2,1,0,3)^l,(3,5,4,4)^l).$$

⊠

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:

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

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. ⊠

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. $x \lhd b_1 \wedge b_2 \rhd \delta = (x \lhd b_1 \rhd \delta) \lhd b \rhd \delta$
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*

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

Proof. Assume $b(e)$ holds.

$$(\sum_{d:D} x \lhd b(d) \rhd \delta) \sqsupseteq (x \lhd b(e) \rhd \delta) = x = (\sum_{d:D} x) \sqsupseteq (\sum_{d:D} x \lhd b(d) \rhd \delta).$$

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

$$\begin{array}{l}\sum_{d:D} x = \\ \sum_{d:D}(x \lhd b(d) \rhd x) = \\ \sum_{d:D}(x \lhd b(d) \rhd \delta + x \lhd \neg b(d) \rhd \delta) = \\ \sum_{d:D}(x \lhd b(d) \rhd \delta) + \sum_{d:D}(x \lhd \neg b(d) \rhd \delta)\end{array}$$

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

LPOs do not blow up when put in parallel. This is the content of the next lemma, taken from [7].

Lemma A.3. *Let*

$$\Phi = \lambda p.\lambda d.\Sigma_{i\in I}\Sigma_{e_i:E_i} c_i(f_i(d,e_i))\, p(g_i(d,e_i)) \lhd b_i(d,e_i) \rhd \delta \text{ and}$$
$$\Psi = \lambda p.\lambda d\Sigma_{i\in I'}\Sigma_{e'_i:E'_i} c'_i(f'_i(d,e'_i))\, p(g'_i(d,e'_i)) \lhd b'_i(d,e'_i) \rhd \delta$$

be convergent LPOs with solutions p and q. Then the parallel composition of p and q, $p \parallel q$, is the solution of the following convergent LPO.

$$\begin{array}{ll}\lambda p.\lambda\langle d,d'\rangle{:}D\times D' . & \Sigma_{i\in I}\Sigma_{e_i:E_i} c_i(f_i(d,e_i))\, p(g_i(d,e_i),d') \lhd b_i(d,e_i) \rhd \delta + \\ & \Sigma_{i\in I'}\Sigma_{e'_i:E'_i} c'_i(f'_i(d',e'_i))\, p(d,g'_i(d',e'_i)) \lhd b'_i(d',e'_i) \rhd \delta + \\ & \Sigma_{i\in I}\Sigma_{i'\in I'}\Sigma_{e_i:E_i}\Sigma_{e'_i:E'_i} \\ & \quad (c_i(f_i(d,e_i)) \mid c'_i(f'_i(d',e'_i)))\, p(g_i(d,e_i),g'_i(d',e'_i)) \lhd b_i(d,e_i) \wedge b'_i(d',e'_i) \rhd \delta\end{array}$$

Note that a summand of the last form is only present when $(c_i(f_i(d,e_i)) \mid c'_i(f'_i(d',e'_i)))$ is defined.

Next, we give a proof of the fact that linear process operators (LPOs) can be rewritten to equivalent deterministic linear process operators (D-LPOs).

Theorem A.4.

1. *Every convergent LPO Φ can be rewritten to a D-LPO Φ' with the same solution, provided every occurrence of an action a in Φ has a parameter of a unique type D_a.*
2. *Consider convergent D-LPOs Φ, Ψ such that action a occurs both in Φ and in Ψ (with parameters of the same data type). There exist convergent D-LPOs Φ', Ψ' having the same solutions as Φ, Ψ, respectively, such that a occurs in Φ' and Ψ' in summands with summation over the same sort E_a.*

Proof.

1. We define Φ' as the result of iterating the following procedure. Let action a occur more than once in Φ. We define $\mathcal{E} = \{\sum_{e_i:E_i} a(f_i(d,e_i))\, p(g_i(d,e_i)) \lhd b_i(d,e_i) \rhd \delta \mid 1 \leq i \leq n\}$ as the set of summands in Φ with action a (we have $n \geq 2$).

   First we treat a simple case, where the formulas $b_i(d,e_i)$ are mutually exclusive (i.e. for no i,j such that $i \neq j$, the formula $b_i(d,e_i) \wedge b_j(d,e_j)$ is satisfiable). Define $E \equiv E_1 \times \cdots \times E_n$. For $1 \leq i \leq n$ and $e \in E$, we let $\pi_i(e)$ denote the i^{th} projection of e (yielding a term of sort E_i). Using E and the projection functions, we represent the summands in $\mathcal{E}$ by the following summand in Φ':

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

   Here, $b : D \to E \to \mathbf{Bool}$ is given by

   $$b(d,e) = b_1(d,\pi_1(e)) \vee \cdots \vee b_n(d,\pi_n(e))$$

   and $f : D \to E \to D_a$ is defined by

   $$f(d,e) = \\ \mathit{if}(b_1(d,\pi_1(e)), f_1(d,\pi_1(e)), \mathit{if}(b_2(d,\pi_2(e)), f_2(d,\pi_2(e)), \ldots, f_n(d,\pi_n(e))\cdots)$$

   Similarly, we define $g : D \to E \to D$ from the g_i functions. It is easy to check that Φ' has the same solution as Φ.

   In general we cannot assume that the formulas $b_i(d,e_i)$ are mutually exclusive. So we add an extra summation over vectors of booleans to model the non-deterministic choice between any of the alternatives.

   Define

   $$E \equiv E_1 \times \cdots \times E_n \times \underbrace{\mathbf{Bool} \times \cdots \times \mathbf{Bool}}_{n-1 \text{ times}}$$

   For $1 \leq i \leq n$ and $e \in E$, we let $\pi_i(e)$ denote the i^{th} projection of e (yielding a term of sort E_i), and, for $1 \leq i \leq n-1$, $\phi_i(e)$ denotes the $(n+i)^{th}$ projection of e (yielding a term of sort **Bool**). We define the summand in Φ' as before, but with different f and g functions. Write $b'_i(d,\pi_i(e))$ for $b_i(d,\pi_i(e)) \wedge \phi_i(e)$. Now we define $f : D \to E \to D_a$ by

   $$f(d,e) = \\ \mathit{if}(b'_1(d,\pi_1(e)), f_1(d,\pi_1(e)), \mathit{if}(b'_2(d,\pi_2(e)), f_2(d,\pi_2(e)), \ldots, f_n(d,\pi_n(e))\cdots)$$

   Similarly, $g : D \to E \to D$ is defined from the g_i functions. Again, it is easy to check that Φ' has the same solution as Φ.

2. By a coding trick as in (1), we obtain that summands in Φ and Ψ with action a have summation over the same data type.

$\boxtimes$

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

Lemma A.5. *Let Φ be an LPO. For all processes p and data $d : D$ we have*

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

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.6. *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

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

   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:

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

   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 case 1 and 2 of this theorem we find:

$$\begin{array}{l} \tau\, p_1(d) = \tau\, \tau_{\{i\}}(p_2(d)) \\ \rho_{Int}(p_2(d)) = p_3(d) \end{array} \tag{7}$$

Using the congruence properties we transform the second equation of (7) 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 (7) 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.7. *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^ξ where every summand 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

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

where i is a fresh action. Assume Φ^ξ has solution p^ξ. Clearly, $\tau_{\{i\}}(p^\xi) = p'$ as both terms are a solution of Φ_ξ (use Lemma A.1.1). Also $\rho_{Int}(p^\xi) = \rho_{Int}(p)$ 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:

$$\begin{array}{lcl} \tau_{Int}(p) & = & \tau_{Int}(\tau_{\{i\}}(p)) \\ & = & \tau_{\{i\}}(\rho_{Int}(p)) \\ & = & \tau_{\{i\}}(\rho_{Int}(p^\xi)) \\ & = & \tau_{Int}(\tau_{\{i\}}(p^\xi)) \\ & = & \tau_{Int}(p') \end{array}$$

⊠

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. In the tables, D is an arbitrary data type, d represents an element 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 overloaded, 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, $\mathcal{D}$ 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 [11].

References

[1] J.C.M. Baeten. *Applications of Process Algebra*, volume 17 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1990.

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

[3] J.A. Bergstra and J.W. Klop. The algebra of recursively defined processes and the algebra of regular processes. In *Proceedings of the* 11^{th} *ICALP,* Antwerp, volume 172 of *Lecture Notes in Computer Science*, pages 82–95. Springer-Verlag, 1984.

[4] M.A. Bezem and J.F. Groote. A correctness proof of a one-bit sliding window protocol in μCRL. *The Computer Journal*, 37(4):289–307, 1994.

[5] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In B. Jonsson and J. Parrow, editors, *Proceedings of the* 5^{th} *Conference on Theories of Concurrency, CONCUR '94,* Uppsala, Sweden, August 1994, volume 836 of *Lecture Notes in Computer Science*, pages 401–416. Springer-Verlag, 1994.

[6] D.J. Bosscher and A. Ponse. Translating a process algebra with symbolic data values to linear format. In U.H. Engberg, K.G. Larsen, and A.S. Skou, editors, Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pages 119–130. BRICS Notes Series NS-95-2, May 1995.

[7] J.J. Brunekreef. Process specification in a UNITY format. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Proceedings of the* 1^{st} *Workshop in the Algebra of Communicating Processes, ACP '94,* Utrecht, the Netherlands, July 1994, volume 458 of *Workshops in Computing, Springer-Verlag*, pages 319–337. Springer-Verlag, July 1994.

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

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

[10] L.-Å. Fredlund, J.F. Groote, and H.P. Korver. Formal verification of a leader election protocol in process algebra. Technical Report R95:01, SICS, 1995.

[11] J.F. Groote and A. Ponse. Proof theory for μCRL: a language for processes with data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Proceedings of the International Workshop on Semantics of Specification Languages,* Utrecht, The Netherlands, pages 231–250. Workshops in Computer Science, Springer-Verlag, 1993.

[12] 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. Springer Verlag, 1994.

[13] B. Jonsson. *Compositional Verification of Distributed Systems.* PhD thesis, Department of Computer Systems, Uppsala University, 1987.

[14] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[15] H. Korver. Personal communication, 1995.

[16] C.P.J. Koymans and J.C. Mulder. A modular approach to protocol verification using process algebra. In Baeten [1], pages 261–306.

[17] K.G. Larsen and R. Milner. A compositional protocol verification using relativized bisimulation. *Information and Computation*, 99:80–108, 1992.

[18] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In: *Proceedings of the 6^{th} Annual ACM Symposium on Principles of Distributed Computing*, pages 137-151, 1987.

[19] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations. Part I: untimed systems. In: *Information and Computation*, 121:214–233, 1995.

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

[21] F.W. Vaandrager. Some observations on redundancy in a context. In Baeten [1], pages 237–260.

[22] 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.

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

Table 1: Axioms for the ACP operators

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	$\Sigma_{d:D} p = \Sigma_{d:D} p + p(d)$	
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	$\Sigma_{d:D}(p_1 \between p_2) = (\Sigma_{d:D} p_1) \between p_2$	d not free in p_2
SUM7	$\Sigma_{d:D}(p_1 \mid p_2) = (\Sigma_{d:D} p_1) \mid p_2$	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	$\dfrac{\mathcal{D}}{\Sigma_{d:D}(p_1) = \Sigma_{d:D}(p_2)}$	d not free in the assumptions of $\mathcal{D}$
BOOL1	$\neg(\mathsf{T} = \mathsf{F})$	
BOOL2	$\neg(b = \mathsf{T}) \to b = \mathsf{F}$	
COND1	$x \lhd \mathsf{T} \rhd y = x$	
COND2	$x \lhd \mathsf{F} \rhd y = y$	

Table 2: Axioms for Sum and Conditional

KFAR	$p(d) = i\,p(d) + y \to \tau\,\tau_{\{i\}}(p(d)) = \tau\,\tau_{\{i\}}(y)$	
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	$(x \between y) \between z = x \between (y \parallel z)$	
SC3	$x \mid y = y \mid x$	
SC4	$(x \mid y) \mid z = x \mid (y \mid z)$	
SC5	$x \mid (y \between z) = (x \mid y) \between z$	

Table 3: Some extra axioms needed in the verification