On Confluence in the $\pi$-Calculus

Anna Philippou and David Walker
Department of Computer Science, University of Warwick
Coventry CV4 7AL, U.K.

Abstract. An account of the basic theory of confluence in the $\pi$-calculus is presented, techniques for showing confluence of mobile systems are given, and the utility of some of the theory presented is illustrated via an analysis of a distributed algorithm.

1 Introduction

Confluence arises in a variety of forms in computation theory. It was first studied in the context of concurrent systems by Milner in [6]. Its essence, to quote [7], is that “of any two possible actions, the occurrence of one will never preclude the other”. As shown in the works cited, for pure CCS agents confluence implies determinacy and semantic-invariance under silent actions, and is preserved by several important system-building operations. These facts make it possible to guarantee by construction that certain systems are confluent and to exploit this fact fruitfully when analysing their behaviours. A more general study was made in [1] which in particular clarified the relationships among various notions of confluence and semantic-invariance under silent actions, and illustrated the utility of the ideas for state-space reduction and protocol analysis; see also [1] for further references. Confluence of value-passing CCS agents was studied first in [18] and later in [22] where consideration was given to conditions under which confluent systems result from combinations of ‘semi-confluent’ agents and the ideas were utilized to show determinacy of programs in a fragment of a concurrent imperative programming language.

The elaboration of techniques for reasoning about mobile systems expressed in the $\pi$-calculus [9] and variants of it has involved extension of established methods and development of new concepts specific to the richer setting. Stemming from [8] there have been several works on disciplines of name-use respected by agents, sometimes expressed via type systems; see for instance [2, 15, 23, 25, 20, 16]. Such disciplines contribute much to the effectiveness of $\pi$-calculi as descriptive formalisms and analytical tools. One promising strand of development concerns varieties of confluence. These have been used in showing determinacy of systems prescribed by concurrent object-oriented programs [13], in justifying optimizations in the Pict compiler [3, 17], and in proving the soundness of transformation rules for concurrent object-oriented programs [4, 14]. The aims of this paper are to give an account of the basic theory of confluence in the $\pi$-calculus, to develop techniques for showing that mobile systems are confluent, and to illustrate the utility of some of the theory presented via an analysis of a
distributed algorithm. The extension of the theory from pure and simple value-passing agents to mobile agents is at some places fairly straightforward: we then proceed quickly, drawing attention only to significant points. Due to the richness of name-passing, however, techniques for showing mobile systems to be confluent are more involved. This paper contains a sample of results obtained on this topic in the first author’s thesis [12]. Independently, Uwe Nestmann in his thesis [11] has developed a static type system concerned with sharing of ports (polarized names) by mobile agents and shown that well-typed agents are confluent.

A summary of the paper follows. Preliminary material is collected in the next section, while in section 3 the basic definitions and results on confluence in the \( \pi \)-calculus are given. Section 4 is concerned with techniques for showing that complex systems are confluent. The final section is devoted to an illustration of the utility of some of the theory presented: an analysis of a distributed algorithm. Due to lack of space all proofs are omitted; see [12] for a detailed technical account.

We are grateful to an anonymous referee for helpful comments.

2 Preliminaries

In this section we recall briefly background material on the (polyadic) \( \pi \)-calculus [9, 8]. For undefined terms and explanation we refer to these papers.

We assume an infinite set \( N \) of names, ranged over by lower-case letters, a partition \( S \) of \( N \) into a set of infinite (subject) sorts, and a sorting \( \lambda : S \rightarrow S^* \). For \( S \in S \), \( \lambda(S) \) is the object sort associated with \( S \). The agents are the expressions given as follows which respect the sorting \( \lambda \):

\[
P ::= 0 \mid \pi P \mid P + Q \mid P \mid Q \mid (\nu y)P \mid A(\overline{y}).
\]

Here \( \pi \) ranges over the prefixes \( \tau \), \( x(\overline{y}) \) and \( \overline{x}(\overline{y}) \), in the latter two of which \( x \) is the subject and the tuple \( \overline{y} \) is the object. In a prefix \( x(\overline{y}) \) the occurrences of the pairwise-distinct names \( \overline{y} \) are binding; the occurrence of \( y \) in \( (\nu y) \) is also binding.

We write \( \text{fn}(P) \) (resp. \( \text{bn}(P) \)) for the set of free (resp. bound) names of \( P \), and \( n(P) \) for the set of all names occurring in \( P \). We write also \( \text{fn}_S(P) \) for the free names of \( P \) of sort \( S \). Each agent constant \( A \) has a defining equation \( A(\overline{x}) \overset{d}{=} P \) where \( \text{fn}(P) \subseteq \overline{x} \) and \( \overline{x} \) are pairwise distinct. We regard as identical agents which differ only by change of bound names. We write \( \equiv \) for structural congruence of agents. A substitution is a sort-respecting mapping from \( N \) to \( N \). We write \( P\sigma \) for the agent obtained from \( P \) by applying the substitution \( \sigma \). We write \( \{\overline{y}/\overline{x}\} \) for the substitution which maps each component of \( \overline{x} \) to the corresponding component of \( \overline{y} \) and is otherwise the identity.

Here we give the behaviour of agents by the early transition rules [10, 19]. In this system there are three kinds of action: input actions of the form \( x(\overline{y}) \); output actions of the form \( (\nu \overline{x})\overline{x}(\overline{y}) \), where the set \( \overline{x} \) of bound names of the action (which is omitted when it is empty) satisfies \( \overline{x} \subseteq \overline{y} \); and the silent action \( \tau \) representing communication between agents. We write \( \text{bn}(\alpha) \) for the set of
bound names of the output $\alpha$ and set $bn(\alpha) = \emptyset$ if $\alpha$ is an input or $\tau$. We write $\text{Act}$ for the set of actions. The subject/object terminology carries over from prefixes to visible actions. The transition rules are as follows where $n(\alpha)$ is the set of names occurring in the action $\alpha$. The third, fourth and fifth have symmetric forms.

1. $\pi(x(y)). P \xrightarrow{\pi(x(y))} P[\bar{z}/y]$ if the sorts of the components of $\bar{y}$ and $\bar{z}$ agree.
2. $\pi. P \xrightarrow{\pi} P$ if $\pi$ is $\tau$ or $\bar{x}(\bar{y})$.
3. If $P \xrightarrow{\alpha} P'$ then $P + Q \xrightarrow{\alpha} P'$.
4. If $P \xrightarrow{\alpha} P'$ then $P \mid Q \xrightarrow{\alpha} P' \mid Q$ if $bn(\alpha) \cap fn(Q) = \emptyset$.
5. If $P \xrightarrow{(\nu \bar{x})\bar{x}(\bar{y})} P'$ and $Q \xrightarrow{x(\bar{y})} Q'$ then $P \mid Q \xrightarrow{\tau} (\nu \bar{z})(P' \mid Q')$ if $\bar{z} \cap fn(Q) = \emptyset$.
6. If $P \xrightarrow{\alpha} P'$ and $y \notin n(\alpha)$ then $(\nu y)P \xrightarrow{\alpha} (\nu y)P'$.
7. If $P \xrightarrow{(\nu \bar{x})\bar{x}(\bar{y})} P'$ and $w \in \bar{y} - (\bar{z} \cup \{x\})$ then $(\nu w)P \xrightarrow{(\nu \bar{w})\bar{x}(\bar{y})} P'$.
8. If $P[\bar{y}/\bar{x}] \xrightarrow{\alpha} P'$ and $A(\bar{x}) \overset{\text{def}}{=} P$ then $A(\bar{y}) \xrightarrow{\alpha} P'$.

We write $\xRightarrow{}$ for the reflexive and transitive closure of $\xrightarrow{\tau}$, $\xRightarrow{}$ for the composition $\xRightarrow{} \xrightarrow{\alpha} \xRightarrow{}$, and $\xRightarrow{\alpha}$ for $\xRightarrow{}$ if $\alpha = \tau$ and $\xRightarrow{}$ otherwise. We further write $P \xRightarrow{\alpha} Q$ if $P \xrightarrow{\alpha} Q$, or $\alpha = \tau$ and $P \equiv Q$.

We often tacitly assume that bound names of actions are fresh. (Early) bisimilarity is the largest symmetric relation $\approx$ such that if $P \approx Q$ and $P \xrightarrow{\alpha} P'$, for some $Q'$, $Q \xRightarrow{\alpha} Q'$ and $P' \approx Q'$. Branching bisimilarity is the largest symmetric relation $\simeq$ such that if $P \simeq Q$ and $P \xrightarrow{\alpha} P'$, then either $\alpha = \tau$ and $P' \simeq Q$, or for some $Q', Q''$, $Q \xRightarrow{} Q' \xrightarrow{\alpha} Q'$, $P \simeq Q''$ and $P' \simeq Q'$. The standard notations for these relations have a dot to differentiate them from the congruences defined as bisimilarity under all substitutions. Since we do not consider the latter here we use the less cumbersome symbols. Finally, an agent $P$ diverges, written $P \uparrow$, if $P$ can perform an infinite sequence of $\tau$-actions; otherwise $P$ converges, $P \downarrow$; and $P$ is fully convergent if for each derivative $P'$ of $P$, $P' \downarrow$.

3 Confluence

In [7] confluence for pure CCS agents was defined using bisimilarity, and it was shown that a wide range of behavioural equivalences coincide on confluent agents. In developing a theory of confluence for the $\pi$-calculus we choose here to base it on early bisimilarity. The connections between this treatment and the various other possibilities are straightforward. In our view, in applications of the theory there is likely to be little substantial difference between the variants. With this choice ‘determinacy’ can be defined as it can for pure CCS agents.

Definition 1. $P$ is determinate if for each derivative $Q$ of $P$ and action $\alpha$, if $Q \xrightarrow{\alpha} Q'$ and $Q \xRightarrow{\alpha} Q''$ then $Q' \approx Q''$. \qed
Note for instance that \( P \equiv a(x). (x(y). \overline{a}(y). \mathbf{0} + b(y). \mathbf{0}) \) is not determinate if \( x, b \) have the same sort as \( P \overset{a(b)}{\longrightarrow} Q \equiv b(y). \overline{a}(y). \mathbf{0} + b(y). \mathbf{0} \) and \( Q \) has non-bisimilar \( b(y) \)-derivatives. As in pure CCS, an agent bisimilar to a determinate agent is determinate, and determinate agents are bisimilar if they may perform the same sequences of visible actions. The following lemma summarizes conditions under which determinacy is preserved by operators. In the last part, \( \text{sort}(M) \) is the set of sorts of the names in \( M \).

**Lemma 2.**

1. If \( P \) is determinate so are \( \tau \cdot P, \overline{e}(y). P \) and \( (\nu y)P \).

2. If \( P \) is determinate and for each \( y \in \overline{y} \), if \( y \) is of sort \( S \) then \( \text{fn}_S(P) \subseteq \{y\} \), then \( a(y). P \) is determinate.

3. If each \( \pi_i : P_i \) is determinate, no \( \pi_i \) is \( \tau \) and no two of the \( \pi_i \) are inputs or outputs with the same subject, then \( \sum_i \pi_i : P_i \) is determinate.

4. If \( P_1, P_2 \) are determinate, \( \text{fn}(P_1) \cap \text{fn}(P_2) = \emptyset \), \( \text{sort}([\text{bn}(P_1)]) \cap \text{sort}(\text{bn}(P_2)) = \emptyset \) and \( \text{sort}([\text{bn}(P_2)]) \cap \text{sort}(\text{bn}(P_1)) = \emptyset \), then \( P_1 \mid P_2 \) is determinate.

The condition in (2) cannot be dropped: consider \( R \equiv x(y). g(y). \mathbf{0} + b(y). \mathbf{0} \) where \( x, b \) have the same sort. Clearly \( R \) is determinate but \( P \equiv a(x). R \) above is not as \( R \{ b/x \} \) is not. Note, however, that if \( x, b \) were of different sorts, \( P \) would be determinate. Using sorts to make distinctions among names in this way is often helpful in applications of the calculus. Similarly, the condition in (4) cannot be dropped: as in CCS, \( P_1, P_2 \) cannot share free names (consider \( a. \mathbf{0} \mid \overline{a}. \mathbf{0} \)), but in addition in the mobile setting more must be said as that property need not be preserved under transition; for instance if \( P \equiv w(z). x(z). b(y). \mathbf{0}, Q \equiv a(y). \overline{c}. \mathbf{0} \) and \( a, z \) are of the same sort, then \( P \mid Q \overset{a(z)}{\longrightarrow} R \equiv a(x). b(y). \mathbf{0} \) or \( a(y). \overline{c}. \mathbf{0} \) and \( R \) is not determinate. The condition in (4) ensures that a bound name of one component cannot be instantiated with a name free in the other.

A pure CCS agent \( P \) is confluent if for each derivative \( Q \) of it and distinct \( \alpha, \beta \), (i) if \( Q \overset{\alpha}{\longrightarrow} Q_1 \) and \( Q \overset{\beta}{\longrightarrow} Q_2 \), then \( Q_1 \Rightarrow Q'_1 \) and \( Q_2 \Rightarrow Q'_2 \approx Q'_1 \), and (ii) if \( Q \overset{\alpha}{\longrightarrow} Q_1 \) and \( Q \overset{\beta}{\longrightarrow} Q_2 \), then \( Q_1 \overset{\beta}{\longrightarrow} Q'_1 \) and \( Q_2 \overset{\alpha}{\longrightarrow} Q'_2 \approx Q'_1 \). For value-passing CCS agents the definition must be refined to account for different inputs with the same subject [18, 22]. This holds also for mobile agents with the additional point that data received are names which may be used for interaction: consider \( P \overset{a}{\equiv} a(x). \overline{e}(y). \mathbf{0} \) which one would expect to be determinate and the transitions \( P \overset{a(b)}{\longrightarrow} b(y). \mathbf{0} \) and \( P \overset{a(c)}{\longrightarrow} c(y). \mathbf{0} \). In the \( \pi \)-calculus a further consideration arises: consider \( P \overset{\nu z}{\equiv} (\nu z)(\overline{a}(z). \mathbf{0} \mid \overline{b}(z). \mathbf{0}) \) and its transitions \( P \overset{\nu z}{\longrightarrow} P_1 \equiv \overline{b}(z). \mathbf{0} \) and \( P \overset{\nu z}{\longrightarrow} P_2 \equiv \overline{a}(z). \mathbf{0} \). Note that \( P_1 \) has no \( (\nu z) \overline{b}(z) \)-transition, and dually for \( P_2 \). In our view \( P \) should none the less be regarded as confluent. To give the definition we introduce two pieces of notation.

**Notation 3** We write \( \alpha \bowtie \beta \) if \( \alpha \) and \( \beta \) are different actions and are not both
inputs with the same subject. The weight $\alpha|\beta$ of action $\alpha$ over action $\beta$ is $\alpha$ except if $\alpha = (\nu \tilde{z})\tilde{a}(\tilde{y})$ when it is $(\nu \tilde{z} - \text{bn}(\beta))\tilde{a}(\tilde{y})$.

Thus for instance, $(\nu y z)\tilde{a}(y, z) \parallel (\nu z)\tilde{b}(x, z)$ is $(\nu y)\tilde{a}(y, z)$. We then have:

**Definition 4.** An agent $P$ is confluent if for each derivative $Q$ of $P$ and $\alpha, \beta$ with $\alpha \bowtie \beta$, (i) if $Q \xrightarrow{\alpha} Q_1$ and $Q \xrightarrow{\tilde{\alpha}} Q_2$, then $Q_1 \xrightarrow{\beta} Q_1'$ and $Q_2 \xrightarrow{\tilde{\beta}} Q_2'$, and (ii) if $Q \xrightarrow{\alpha} Q_1$ and $Q \xrightarrow{\tilde{\alpha}} Q_2$, then $Q_1 \xrightarrow{\beta \tilde{\alpha}} Q_1'$ and $Q_2 \xrightarrow{\alpha \tilde{\beta}} Q_2'$.

Thus for instance $P \equiv (\nu z)(\tilde{a}(z) \cdot 0 \parallel \tilde{b}(z) \cdot 0)$ above is confluent as after $P \xrightarrow{(\nu z)\tilde{a}(z)} P_1 \equiv \tilde{b}(z) \cdot 0$ and $P \xrightarrow{(\nu z)\tilde{a}(z)} P_2 \equiv \tilde{a}(z) \cdot 0$ we have $P_1 \xrightarrow{\tilde{b}(z)} 0$ and $P_2 \xrightarrow{\tilde{a}(z)} 0$.

It is easy to see that an agent bisimilar to a confluent agent is itself confluent. An agent $P$ is $\tau$-inert if for each derivative $Q$ of $P$, if $Q \xrightarrow{\tau} Q'$ then $Q' \approx Q$. By a generalization of the argument from the CCS case we have:

**Lemma 5.** If $P$ is confluent then $P$ is $\tau$-inert.

The following result is a useful characterization of confluence in which only single transitions need be considered. It holds only for fully convergent agents. In [1] it was observed that for fully convergent ("$\tau$-well founded") agents, $\tau$-inertness implies confluence. A similar observation is included here.

**Lemma 6.** Suppose $P$ is fully convergent. Then $P$ is confluent iff $P$ is $\tau$-inert and for each derivative $Q$ of $P$ and $\alpha, \beta$ with $\alpha \bowtie \beta$, (i) if $Q \xrightarrow{\alpha} Q_1$ and $Q \xrightarrow{\tilde{\alpha}} Q_2$ then $Q_1 \approx Q_2$, and (ii) if $Q \xrightarrow{\alpha} Q_1$ and $Q \xrightarrow{\tilde{\alpha}} Q_2$, then $Q_1 \xrightarrow{\beta \tilde{\alpha}} Q_1'$ and $Q_2 \xrightarrow{\alpha \tilde{\beta}} Q_2'$.

The proof shows that if $P$ is fully convergent and $\tau$-inert and satisfies (i), then $P$ is determinate. The assumption that $P$ is fully convergent cannot be dropped: consider $P \overset{\text{def}}{=} a \cdot b \cdot 0 \parallel \tau \cdot (a \cdot 0 \parallel \tau \cdot P)$. It is easy to see that $P$ is $\tau$-inert and that all of its derivatives satisfy (i) and (ii). However, $P$ is not determinate.

We record the analogues for confluence of the earlier results on preservation of determinacy by operators.

**Lemma 7.**

1. If $P$ is confluent so are $\tau \cdot P$, $\tau \cdot \tilde{y} \cdot P$ and $(\nu y)P$.
2. If $P$ is confluent and for each $y \in \tilde{y}$, if $y$ is of sort $S$ then $\text{fn}(P) \subseteq \{y\}$, then $a(\tilde{y}) \cdot P$ is confluent.
3. If $P_1, P_2$ are confluent, $\text{fn}(P_1) \cap \text{fn}(P_2) = \emptyset$, $\text{sort}(\text{bn}(P_1)) \cap \text{sort}(\text{bn}(P_2)) = \emptyset$ and $\text{sort}(\text{bn}(P_2)) \cap \text{sort}(\text{fn}(P_1)) = \emptyset$, then $P_1 \parallel P_2$ is confluent.

Of course here the guarded summation clause is missing.

In the following section we will consider further techniques for showing systems to be confluent. Before doing so we consider a variant of confluence based on branching bisimilarity.
Definition 8. $P$ is $\simeq$-confluent if for each derivative $Q$ of $P$ and $\alpha, \beta$ with $\alpha \gg \beta$, (i) if $Q \xrightarrow{\alpha} Q_1$ and $Q \xrightarrow{\beta} Q_2$, then $Q_1 \Rightarrow Q'_1$ and $Q_2 \Rightarrow Q'_2 \simeq Q'_1$, and (ii) if $Q \xrightarrow{\beta} Q_1$ and $Q \xrightarrow{\alpha} Q_2$, then $Q_1 \Rightarrow Q'_1$ and $Q_2 \Rightarrow Q'_2 \simeq Q'_1$.

The following observations were made in [4]. Confluence (for non-mobile labelled transition systems) based on branching bisimilarity was also considered in [1] and observations similar to some of these made. An agent $P$ is $\tau_{\simeq}$-inert if for each derivative $Q$ of $P$, if $Q \xrightarrow{\tau} Q'$ then $Q' \simeq Q$.

Lemma 9.

1. If $P$ is $\simeq$-confluent then $P$ is $\tau_{\simeq}$-inert.
2. If $P, Q$ are $\tau$-inert and $P \simeq Q$ then $P \simeq Q$.
3. $P$ is $\tau_{\simeq}$-inert iff $P$ is $\tau$-inert.
4. $P$ is confluent iff $P$ is $\simeq$-confluent.

In contrast to these coincidences, to obtain a satisfactory notion of 'partial' confluence which is not $\tau$-inert it is essential to base the theory on branching bisimilarity rather than bisimilarity; see [4].

4 Confluence by construction

A main motivation in [7] for studying confluence was to find an interesting property implying determinacy which can be guaranteed to hold simply by confining the use of combinators in building systems. Work elaborating this view and showing its fruitfulness has been described in the Introduction. Here the emphasis is on sample results of this kind in the richer setting of name-passing. The approach is complementary to development of static type systems as in [11, 20].

A useful definition: an agent $P$ is o-determinate if for each derivative $Q$ of $P$, there are not two distinct output actions $\alpha, \beta$ with the same subject such that $Q \xrightarrow{\alpha}$ and $Q \xrightarrow{\beta}$. The first result gives conditions under which a combination of confluent agents is confluent.

Theorem 10. Suppose $P \equiv (\nu z)(P_1 | \ldots | P_n)$ where each $P_i$ is confluent and o-determinate. Suppose that for each derivative $P' \equiv (\nu z')(P'_1 | \ldots | P'_n)$ of $P$, no name occurs free in more than two components of $P'$, and a free name of $P'$ occurs in exactly one component of $P'$. Then $P$ is confluent.

Note that in this theorem it is not possible to replace 'confluent' by 'determinate': consider $(\nu a)(\overline{a} \cdot 0 | (a \cdot 0 + b \cdot 0))$.

It is often the case that although the components of a system are not themselves confluent, the constraints they place upon one another's behaviour ensure that the system itself is confluent. The second theorem is an instance of this idea. To state it we need some definitions. We refer to a set of agents closed under derivation as a system. For $S \subseteq S$ we say a system is $S$-closed if none of its agents may perform an input or an output via an $S$-name.
Definition 11. Suppose $S$ and $\tilde{S} = S_1 \ldots S_n$ are distinct sorts and the sorting $\lambda$ is such that $\lambda(S) = (\tilde{S})$ and no $S_i$ occurs in any other $\lambda(S')$. A system $\mathcal{P}$ is $S, \tilde{S}$-sensitive if there is a partition $\{\mathcal{P}^p \mid \tilde{p} \text{ a finite subset of } S_1 \times \ldots \times S_n\}$ of $\mathcal{P}$ such that:

1. if $P \in \mathcal{P}^\tilde{p}$ and $P \xrightarrow{\alpha} P'$ where $\alpha$ is not an input or output via an $S$-name or an input via an $S_i$-name, then $P' \in \mathcal{P}^\tilde{p}$;
2. if $P \in \mathcal{P}^\tilde{p}$ and $P \xrightarrow{\alpha} P'$ where $\alpha$ is an output via an $S$-name, then $\alpha = (\nu \tilde{z}) \exists (\tilde{z}) P' \in \mathcal{P}^\tilde{p} \cup \{\tilde{z}\}$;
3. if $P \in \mathcal{P}^\tilde{p}$ and $P \xrightarrow{\alpha} P'$ where $\alpha = x(z_1, \ldots, z_n)$ with $x : S$, then at most one of the $z_i$ occurs free in $P'$;
4. if $P \in \mathcal{P}^\tilde{p}$ and $P \xrightarrow{\alpha} P'$ where $\alpha$ is an input via an $S_i$-name, then there is $\tilde{z} = (z_1, \ldots, z_n) \in \tilde{p}$ such that the subject of $\alpha$ is $z_i$ and $P' \in \mathcal{P}^\tilde{p} \setminus \{\tilde{z}\}$.

Further, $\mathcal{P}$ is $S, \tilde{S}$-confluent if it is $S, \tilde{S}$-sensitive and whenever $P \in \mathcal{P}^\tilde{p}$, $P \xrightarrow{\alpha} P_1$ and $P \xrightarrow{\beta} P_2$, then unless for some $(z_1, \ldots, z_n) \in \tilde{p}$, $\alpha$ and $\beta$ are inputs via distinct $z_i$ and $z_j$, $P_1 \xrightarrow{\beta} P'_1$ and $P_2 \xrightarrow{\alpha} P'_2 \approx P'_1$.

We then have:

Theorem 12. Suppose $P \equiv (\nu \tilde{z})(P_1 | \ldots | P_n)$ and $\mathcal{P} = \{Q \mid Q$ is a derivative of a $P_i\}$ is $S$-closed and $S, \tilde{S}$-confluent with partition $\{\mathcal{P}^p\}_{p \in \tilde{p}}$. Suppose each $P_i \in \mathcal{P}^0$ and is o-determinate. Suppose that for each derivative $P' \equiv (\nu \tilde{z}')(P'_1 | \ldots | P'_n)$ of $P$, no name occurs free in more than two components of $P'$, and a free name of $P'$ occurs in exactly one component of $P'$. Then $P$ is confluent.

In closing this section we mention that related results of a synthetic nature can also be obtained for useful varieties of 'partial confluence' as described in the Introduction, and that static type systems as in for instance the papers cited earlier complement them effectively.

5 An application

The aim of this section is to illustrate the utility of some of the theory presented via an analysis of a distributed algorithm. It is a variant of the Propagation of Information with Feedback protocol of [21] studied in [24]. Consider a network of $m$ processes connected by communication links, where the graph having the processes as nodes and the links as edges is connected. Each process stores an integer, its value. A distinguished process, the root, conducts the interaction between the network and its environment. The intended behaviour of the algorithm is that on receiving a request from the environment, the root should emit to it the value of the network, i.e. the sum of the values of the $m$ processes. We proceed to give and explain the process-calculus description of the algorithm.
We use the following sorts: E, T, D, I, O. The sorting λ is as follows: λ(E) = (T, D), λ(T) = (int), λ(D) = (), λ(I) = (O), λ(O) = (int). Here int is the type of integers; we allow simple arithmetic expressions in the descriptions – the foregoing theory extends easily to accommodate this. It is intended that each process passes from its initial quiescent state through some active states to a final inactive state. The behaviour of a non-root process is described as follows, where Q represents the quiescent state, A the active states, I the inactive state, and ε is the empty tuple.

\[
Q(\bar{e}, v) \stackrel{\text{def}}{=} \sum_{e \in \bar{e}} e(t, d). A(t, \bar{e} - e, \bar{e} - e, e, e, v)
\]

\[
A(t, e, e, e, e, v) \stackrel{\text{def}}{=} \bar{e}(v). I
\]

\[
I \stackrel{\text{def}}{=} 0
\]

\[
A(t, \bar{s}, \bar{r}, \bar{d}, \bar{p}, v) \stackrel{\text{def}}{=} \sum_{e \in \bar{e}} (\nu t' d) \bar{e}(t', d). A(t, \bar{s} - e, \bar{r}, \bar{d}, \bar{p}(t', d), v)
\]

\[
+ \sum_{e \in \bar{e}} e(t', d). A(t, \bar{s}, \bar{r} - e, \bar{d}d, \bar{p}, v)
\]

\[
+ \sum_{d \in \bar{d}} A(t, \bar{s}, \bar{r}, \bar{d} - d, \bar{p}, v)
\]

\[
+ \sum_{(t', d) \in \bar{p}} (t'(v')). A(t, \bar{s}, \bar{r}, \bar{d}, \bar{p} - (t', d), v + v')
\]

\[
+ d. A(t, \bar{s}, \bar{r}, \bar{d}, \bar{p} - (t', d), v)).
\]

In \(Q(\bar{e}, v)\), \(v\) is the value of the process and the names \(\bar{e}\) of sort E represent the edges incident on it in the network. In the quiescent state the agent may receive via any such name a pair of names, \(t\) of sort T and \(d\) of sort D. It discards \(d\) and undertakes to send an integer along \(t\) which it does when it has all but completed its activity (second and third clauses). That activity is described in the fourth clause: \(A(t, \bar{s}, \bar{r}, \bar{d}, \bar{p}, v)\) represents the state in which the process is storing \(v\), has yet to send data along each E-name in \(\bar{s}\), has yet to receive data along each E-name in \(\bar{e}\), has yet to send a signal along each D-name in \(\bar{d}\), and for each T-name, D-name pair in \(\bar{p}\), has yet to receive either an integer along the T-name or a signal along the D-name.

The behaviour of the root is given as follows:

\[
Q_0(in, \bar{e}, v) \stackrel{\text{def}}{=} in(out). A_0(out, \bar{e} - e, \bar{e} - e, e, e, v)
\]

\[
A_0(out, e, e, e, e, v) \stackrel{\text{def}}{=} out(v). I_0
\]

\[
I_0 \stackrel{\text{def}}{=} 0
\]

\[
A_0(out, \bar{s}, \bar{r}, \bar{d}, \bar{p}, v) \stackrel{\text{def}}{=} ...
\]

where the fourth clause is as for \(A\) but with ‘\(A_0\)’ in place of ‘\(A\)’ and ‘out’ in place of ‘\(t'\). Thus the root behaves similarly to the other nodes except that it is activated by receiving along the name in of sort I a name of sort O via which it undertakes to send the network’s value. The network is represented by

\[
P_0 \stackrel{\text{def}}{=} (\nu \bar{e})(Q_0(in, \bar{e}_0, v_0) \mid \prod_{1 \leq i < m} Q(\bar{e}_i, v_i))
\]

where \(\bar{e}\) are the E-names representing all the edges and for each \(i\), \(\bar{e}_i\) those incident on the \(i\th\) process. We will prove the following correctness result:
Theorem 13. \( P_0 \approx \text{in(out).out(v).0} \), where \( v = \sum_{i=0}^{m-1} v_i \).

The algorithm may be thought of as consisting of two phases. In the first a spanning tree for the network is established, and in the second each non-root process passes to its parent the sum of the values stored in its descendants, and the root then emits to the environment the network's value. The sending by \( A_0 \) or \( A \) along a name \( e \) of a pair \( t', d \) of fresh names is an invitation to the receiver either to become a child of the sender and to undertake to send it an integer along \( t' \), or, if the receiver is already active (and so has a parent), to decline to do so by sending a signal via \( d \). A process sends an integer to its parent only when it has determined the sum of the values of its descendants.

First we give a characterization of derivatives of \( P_0 \). For \( S \in S \), in an agent of the form \((v \overline{\exists} P_i, Z_i, i, Z_i, \overline{\exists} P_i)\), we say there is an \( S \)-path between components \( Z \) and \( Z' \) if there are \( S \)-names \( x_1, \ldots, x_p \) such that \( x_i \in \text{fn}(Z_i, Z_{i+1}) \) for each \( i \), \( Z' = Z_1 \) and \( Z' \equiv Z_{p+1} \).

Lemma 14. If \( P_0 \xrightarrow{w} P \) where \( w \in \text{Act}^* \) then \( P \equiv (v \overline{\exists} \tilde{d})(R \mid \Pi_{1 \leq i < m} N_i) \) where:

- (a) \( \text{fn}(P) = \{\text{in}, \text{out}\} \) or \( \emptyset \), and in and out may occur only in \( R \) (the derivative of the root \( Q_0 \));
- (b) no name occurs free in more than two components of \( P \);
- (c) if a \( T \)-name occurs free in a component of \( P \), there is a unique \( T \)-path between that component and \( R \);
- (d) the sum of the integers stored in the components which are quiescent or active is the network's value.

Some useful notation: \( P_1 \equiv (v \overline{\exists} (A_0 \text{out,} \tilde{e}_0, \overline{\exists} \tilde{e}_0, e, e, v_0) \mid \Pi_{1 \leq i < m} Q(e_i, v_i)), P_\psi \equiv (v \overline{\exists} (A_0 \text{out,} e, e, e, e, v) \mid \Pi_{1 \leq i < m} I), \) and \( P_\omega \equiv (v \overline{\exists} (J_0 \mid \Pi_{1 \leq i < m} I) \). We will later show that \( P_\psi \) and \( P_\omega \) are derivatives of \( P_0 \). We use \( P \) to range over derivatives of \( P_0 \). Key in proving the theorem will be the agents of the form

\[
Q'(e, \tilde{e}, v) \overset{\text{def}}{=} e(t, d). A(t, \tilde{e} - e, \tilde{e} - e, e, e, v)
\]

where \( e \in \tilde{e} \). \( Q' \) is similar to \( Q \) except that it may be activated only by an interaction along the specific name \( e \). Note that, where \( \sim \) is strong bisimilarity,

\[
Q(\tilde{e}, v) \sim \sum_{e \in \tilde{e}} Q'(e, \tilde{e}, v).
\]

Let \( T \) be the set of agents of the form

\[
T_0 \overset{\text{def}}{=} (v \overline{\exists} (Q_0 \text{in,} \tilde{e}_0, v_0) \mid \Pi_{1 \leq i < m} Q'(e_i, \tilde{e}_i, v_i))
\]

where \( e_1, \ldots, e_{m-1} \) represent a spanning tree of the graph, with \( e_i \in \tilde{e}_i \) for each \( i \). Note that such a \( T_0 \) differs from \( P_0 \) just in having \( Q' \) where \( P_0 \) has \( Q \); the edge via which each non-root node will receive its first communication is determined; intuitively, \( T_0 \) represents the fragment of the behaviour of \( P_0 \) in which the spanning tree is given by those edges. Let \( T_0 \in T \). Directly from (1) and Lemma 14 we have:

Corollary 15. If \( T_0 \xrightarrow{w} T \) where \( w \in \text{Act}^* \) then \( T \equiv (v \overline{\exists} \tilde{d})(R \mid \Pi_{1 \leq i < m} N'_i) \) where (a)–(d) as in Lemma 14 (with 'T' for 'P') hold.
Some useful notation: $T_1 \equiv (\nu \bar{e})(A_0(\text{out}, \bar{e}_0, \bar{e}, \bar{e}, v_0) \mid \Pi_{1 \leq i < m} Q'(e_i, \bar{e}_i, v_i))$, $T_\psi \equiv (\nu \bar{e})(A_0(\text{out}, e, \bar{e}, e, v) \mid \Pi_{1 \leq i < m} I)$ and $T_\omega \equiv (\nu \bar{e})(I_0 \mid \Pi_{1 \leq i < m} I)$. We use $T$ to range over derivatives $T_0$. We analyse $T_0$, noting first that it has a specific behaviour:

**Lemma 16.** $T_0 \xrightarrow{\text{in(out)}} T_1 \xrightarrow{\text{out}(v)} T_\psi \xrightarrow{} T_\omega$. □

We now have the key observation whose proof appeals Theorem 12.

**Lemma 17.** $T_0$ is confluent. □

Note that from these two results we have:

**Corollary 18.** $T_0 \approx \text{in(out)} \cdot \text{out}(v) \cdot 0$. □

Having used confluence to analyse the behaviour of $T_0$ we now relate it to that of $P_0$. We say $P$ and $T$ are *similar* if they differ only in that where $P$ has a quiescent component $Q$, $T$ has a quiescent component $Q'$.

**Lemma 19.** $\{\\langle T, P \rangle \mid P$ and $T$ are similar$\}$ is a strong simulation. □

By Lemma 16 and 19 we have that $P_0 \xrightarrow{\text{in(out)}} P_1 \xrightarrow{\text{out}(v)} P_\psi \xrightarrow{} P_\omega$. We say that $T_0$ is *compatible* with a computation $P_0 \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_r} P_r$ if for each $i$, if $\alpha_i$ is $\tau$ and arises from complementary actions $(\nu t', d)\bar{e}(t', d)$, $e(t', d)$ where the second is performed by a quiescent component $Q(\bar{e}_j, v_j)$, then in $T_0$ that component is $Q'(e, \bar{e}_j, v_j)$; i.e. the E-names used to activate components in the computation are those via which the $Q'$-components of $T_0$ may be activated.

**Lemma 20.** If $P_0 \xrightarrow{w} P$ then for any $T_0$ compatible with the computation, $T_0 \xrightarrow{w} T$ with $P$ and $T$ similar. □

We can now prove the theorem. Since $P_0 \approx \text{in(out)} \cdot P_1$ it suffices to show that $P_1 \approx \text{out}(v) \cdot 0$. We have seen that $P_1 \approx \text{out}(v) \cdot P_\omega \approx 0$. Choose one such computation and, by Lemma 14, choose $T_0$ compatible with it. Then not $(P_1 \xrightarrow{\alpha})$ with $\alpha \neq \text{out}(v)$ as otherwise by Lemma 20, $(T_1 \xrightarrow{\alpha})$, contradicting Lemma 18. Finally, and for the same reason, not $(P_1 \xrightarrow{\alpha} \not\xrightarrow{})$. □

We conclude by briefly comparing this analysis with that in [24]. The latter uses a static I/O-automaton model [5] of the algorithm and establishes that the fair traces of the automaton representing it are included in those of an automaton akin to the agent in(out).out$(v)$. 0. In our view name-passing and careful use of sorts allow a very direct and perspicuous description of the algorithm's behaviour: the construction and use of the spanning tree are manifest in the description. Moreover, the use of reasoning techniques involving name-passing aids the analysis, and the proof illustrates the idea that when studying the behaviour of a confluent system it may suffice to examine in detail only a (small) part of it. Finally, here the correctness criterion is bisimilarity, rather than inclusion of fair traces.
References