Verification Techniques for Distributed Algorithms

Anna Philippou and George Michael

Department of Computer Science, University of Cyprus,
Kallipoleos Street, 1678 Nicosia, Cyprus
{annap, gmichael}@cs.ucy.ac.cy

Abstract. A value-passing, asynchronous process calculus and its associated theory of confluence are considered as a basis for establishing the correctness of distributed algorithms. In particular, we present an asynchronous version of value-passing CCS and we develop its theory of confluence. We show techniques for demonstrating confluence of complex processes in a compositional manner and we study properties of confluent systems that can prove useful for their verification. These results give rise to a methodology for system verification which we illustrate by proving the correctness of two distributed leader-election algorithms.

1 Introduction

Distributed systems present today one of the most challenging areas of research in computer science. Their high complexity and dynamic nature and features such as concurrency and unbounded nondeterminism, render their construction, description and analysis a challenging task. The development of formal frameworks for describing and associated methodologies for reasoning about distributed systems has been an active area of research for the last few decades and is becoming increasingly important as a consequence of the great success of worldwide networking and the vision of ubiquitous computing.

Process calculi, otherwise referred to as *process algebras*, such as CCS [8], the π-calculus [10], and others, are a well-established class of modeling and analysis formalisms for concurrent and distributed systems. They can be considered as high-level description languages consisting of a number of operators for building processes including constructs for defining recursive behaviors. They are accompanied by semantic theories which give precise meaning to processes, translating each process into a mathematical object on which rigorous analysis can be performed. In addition, they are associated with axiom systems which prescribe the relations between the various constructs and can be used to reason algebraically about processes. During the last two decades, they have been extensively studied and they have proved quite successful in the modeling and reasoning about system correctness. They have been extended for modeling a variety of aspects of process behavior including mobility, distribution, value-passing and asynchronous communication.

A. Shvartsman (Ed.): OPODIS 2006, LNCS 4305, pp. 172–186, 2006.

Confluence arises in a variety of forms in computation theory. It was first studied in the context of concurrent systems by Milner in [8]. Its essence, to quote [9], is that "of any two possible actions, the occurrence of one will never preclude the other". As shown in the mentioned papers, for pure CCS processes confluence implies determinacy and semantic-invariance under internal computation, and it is preserved by several system-building operators. These facts make it possible to reason compositionally that a system is confluent and to exploit this fact while reasoning about its behavior. Work on the study of confluence in concurrent systems was subsequently carried out in various directions. In [2] various notions of confluence were studied and the utility of the ideas was illustrated for state-space reduction and protocol analysis. Furthermore, the theory of confluence was developed for value-passing CCS in [16,18]. In the context of mobile process calculi, such as the π-calculus and extensions of it, notions of confluence and *partial* confluence were studied and employed in a variety of contexts including [5,11,13,14,15].

The aims of this paper is to study the notion of confluence in an asynchronous, value-passing process calculus and to develop useful techniques for showing that complex asynchronous processes are confluent. Based on this theory, we develop a methodology for the analysis of distributed algorithms. We illustrate its utility via the verification of two distributed leader-election algorithms.

The extension of the theory of confluence to the asynchronous setting is at most places pretty straightforward. Thus, due to the lack of space, we omit the proofs of the results and draw the attention only to significant points pertaining to the ways in which the presence of asynchrony admits a larger body of compositional results. We then take a step from the traditional definition of confluence in value-passing process calculi, and we introduce a new treatment of input actions, motivated by the notion of "input-enabledness" often present in distributed systems. Our main result shows that the resulting notion allows an interesting class of complex processes to be shown as confluent by construction. These compositional results and other confluence properties are exploited in the verification of both of the algorithms we consider. We illustrate the application of the theory for the algorithm verification in detail.

The remainder of the paper is structured as follows. In the next section we present our asynchronous extension of value-passing CCS while, in Sect. 3, the theory of confluence is developed. Section 4 contains application of our verification methodology for establishing the correctness of two distributed leader-election algorithms and Sect. 5 concludes the paper.

2 The Calculus

In this section we present the CCS_v process calculus, an amalgamation of value-passing CCS [8,18], with features of asynchronous communication from the asynchronous π-calculus [1,4] and a kind of conditional agents.

We begin by describing the basic entities of the calculus. We assume a set of *constants*, ranged over by u, v, including the positive integers and the boolean

values and a set of functions, ranged over by f, operating on these constants. Moreover, we assume a set of *variables* ranged over by x, y. Then, the set of *terms* of CCS_v, ranged over by e, is given by (1) the set of constants, (2) the set of variables, and (3) function applications of the form $f(e_1, \ldots e_n)$, where the e_i are terms. We say that a term is *closed* if it contains no variables. The evaluation relation $\twoheadrightarrow$ for closed terms is defined in the expected manner. We write $\tilde{r}$ for a tuple of syntactic entities $r_1, \ldots, r_n$.

Moreover, we assume a set of *channels*, $\mathcal{L}$, ranged over by a, b. Channels provide the basic communication and synchronization mechanisms in the language. A channel a can be used in *input position*, denoted by a, and in o*utput position*, denoted by $\overline{a}$. This gives rise to the set of *actions* Act of the calculus, ranged over by α, β, containing

- the set of *input actions* which have the form $a(\tilde{v})$ representing the input along channel a of a tuple $\tilde{v}$,
- the set of *output actions* which have the form $\overline{a}(\tilde{v})$ representing the output along channel a of a tuple $\tilde{v}$, and
- the *internal action* τ, which arises when an input action and an output action along the same channel are executed in parallel.

For simplicity, we write a and $\overline{a}$ for $a(\langle\rangle)$ and $\overline{a}(\langle\rangle)$, where $\langle\rangle$ is the empty tuple. We say that an input and an output action on the same channel are *complementary* actions, and, for a non-internal action α, we denote by $\ell(\alpha)$ the channel of α. Finally, we assume a set of process constants $\mathcal{C}$, ranged over by C.

The syntax of CCS_v processes is given by the following BNF definition:

$$P ::= \mathbf{0} \;\mid\; \overline{a}(\tilde{v}) \;\mid\; \sum_{i \in I} a_i(\tilde{x_i}).P_i \;\mid\; P_1 \parallel P_2 \;\mid\; P \backslash L$$
$$\mid\; \mathsf{cond}(e_1 \rhd P_1, \ldots, e_n \rhd P_n) \;\mid\; C\langle \tilde{v} \rangle$$

Process $\mathbf{0}$ represents the inactive process. Process $\overline{a}(\tilde{v})$ represents the asynchronous output process. It can output the tuple $\tilde{v}$ along channel a. Process $\sum_{i \in I} a_i(\tilde{x_i}).P_i$ represents the nondeterministic choice between the set of processes $a_i(\tilde{x_i}).P_i$, $i \in I$. It may initially execute any of the actions $a_i(\tilde{x_i})$ and then evolve into the corresponding continuation process P_i. It is worth noting that nondeterministic choice is only defined with respect to input-prefixed processes. This follows the intuition adopted in [1,4] that, if an output action is enabled, it should be performed and not precluded from arising by nondeterministic choice. Further, note that, unlike the summands $a_i(\tilde{x_i}).P_i$, the asynchronous output process does not have a continuation process due to the intuition that an output action is simply emitted into the environment and execution of the emitting process should continue irrespectively of when the output is consumed.

Process $P \| Q$ describes the concurrent composition of P and Q: the component processes may proceed independently or interact with one another while executing complementary actions. The conditional process $\mathsf{cond}(e_1 \rhd P_1, \ldots, e_n \rhd P_n)$ offers a conditional choice between a set of processes: assuming that all e_i are closed terms, it behaves as P_i, where i is the smallest integer for which $e_i \twoheadrightarrow \mathsf{true}$.

In $P\backslash F$, where $F \subseteq \mathcal{L}$, the scope of channels in F is restricted to process P: components of P may use these channels to interact with one another but not with P's environment. This gives rise to the free and bound channels of a process. We write $\mathsf{fc}(P)$ for the free channels of P, and $\mathsf{fic}(P)$, $\mathsf{foc}(P)$ for the free channels of P which are used in input and output position within P, respectively.

Finally, process constants provide a mechanism for including recursion in the process calculus. We assume that each constant C has an associated definition of the form $C\langle\tilde{x}\rangle \stackrel{\text{def}}{=} P$, where process P may contain occurrences of C, as well as other process constants.

Each operator is given precise meaning via a set of rules which, given a process P, prescribe the possible transitions of P, where a transition of P has the form $P \xrightarrow{\alpha} P'$, specifying that process P can perform action α and evolve into process P'. The rules themselves have the form

$$\frac{T_1, \ldots, T_n}{T} \quad \phi$$

which is interpreted as follows: if transitions $T_1, \ldots, T_n$, can be derived, and condition ϕ holds, then we may conclude transition T. The semantics of the CCS_v operators are given in Table 1.

Table 1. The operational semantics

(Sum)	$\sum_{i\in I} a_i(\tilde{x_i}).P_i \xrightarrow{a_i(\tilde{v}_i)} P_i\{\tilde{v}_i/\tilde{x_i}\}$	(Out)	$\overline{a}(\tilde{v}) \xrightarrow{\overline{a}(\tilde{v})} \mathbf{0}$
(Par1)	$\dfrac{P_1 \xrightarrow{\alpha} P_1'}{P_1 \| P_2 \xrightarrow{\alpha} P_1' \| P_2}$	(Par2)	$\dfrac{P_2 \xrightarrow{\alpha} P_2'}{P_1 \| P_2 \xrightarrow{\alpha} P_1 \| P_2'}$
(Par3)	$\dfrac{P_1 \xrightarrow{a(\tilde{v})} P_1',\ P_2 \xrightarrow{\overline{a}(\tilde{v})} P_2'}{P_1 \| P_2 \xrightarrow{\tau} P_1' \| P_2'}$	(Res)	$\dfrac{P \xrightarrow{\alpha} P',\ \ell(\alpha), \notin F}{P\backslash F \xrightarrow{\alpha} P'\backslash F}$
(Cond)	$\dfrac{P_i \xrightarrow{\alpha} P_i'}{\mathsf{cond}(e_1 \rhd P_1, \ldots, e_n \rhd P_n) \xrightarrow{\alpha} P_i}$	$e_i \twoheadrightarrow \mathsf{true}, \forall j < i, e_j \twoheadrightarrow \mathsf{false}$	
(Const)	$\dfrac{P\{\tilde{v}/\tilde{x}\} \xrightarrow{\alpha} P'}{C(\tilde{v}) \xrightarrow{\alpha} P}$	$C(\tilde{x}) \stackrel{\text{def}}{=} P$	

We discuss some of the rules below:

– (Sum). This axiom employs the notion of *substitution*, a partial function from variables to values. We write $\{\tilde{v}/\tilde{x}\}$ for the substitution that maps variables $\tilde{x}$ to values $\tilde{v}$. Thus, for all $i \in I$, the input-prefixed summation process can receive a tuple of values $\tilde{v}_i$ along channel a_i, and then continue as process P_i, with the occurrences of the variables $\tilde{x}_i$ in P_i substituted by values $\tilde{v}_i$. For example: $a(x,y).\overline{b}(x) + c(z).\mathbf{0} \xrightarrow{a(2,5)} \overline{b}(2)$ and $a(x,y).\overline{b}(x) + c(z).\mathbf{0} \xrightarrow{c(1)} \mathbf{0}$.

- (Par1). This axiom (and its symmetric version (Par2)) expresses that a component in a parallel composition of processes may execute actions independently. For example, since $\overline{a}(3) \xrightarrow{\overline{a}(3)} \mathbf{0}$, $\overline{a}(3) \parallel a(v).\overline{b}(c) \xrightarrow{\overline{a}(3)} \mathbf{0} \parallel a(v).\overline{b}(c)$.
- (Par3). This axiom expresses that two parallel processes executing complementary actions may synchronize with each other producing the internal action τ: $\overline{a}(3) \parallel a(v).\overline{b}(v) \xrightarrow{\tau} \overline{b}(3)$.
- (Cond). This axiom formalizes the behavior of the conditional operator. An example of the rule follows: $\mathsf{cond}(2 = 3 \rhd \overline{b}(3), \mathsf{true} \rhd \overline{c}(4)) \xrightarrow{\overline{c}(4)} \mathbf{0}$.
- (Const). This axiom stipulates that, given a process constant and its associated definition $C\langle \tilde{x} \rangle \stackrel{\text{def}}{=} P$, its instantiation $C\langle \tilde{v} \rangle$ behaves as process P with variables $\tilde{x}$ substituted by $\tilde{v}$. For example, if $C\langle x, y \rangle \stackrel{\text{def}}{=} \mathsf{cond}(x = y \rhd \overline{b}(x), \mathsf{true} \rhd \overline{c}(y))$, then $C(2,2) \xrightarrow{\overline{b}(2)} \mathbf{0}$.

An additional form of process expression derivable from our syntax and used in the sequel is the following: $!\,P \stackrel{\text{def}}{=} P \parallel\ !\,P$, usually referred to as the *replicator* process, represents an unbounded number of copies of P running in parallel.

We recall some useful definitions. We say that Q is a *derivative* of P, if there are $\alpha_1, \ldots, \alpha_n \in Act$, $n \geq 0$, such that $P \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} Q$. Moreover, given $\alpha \in Act$ we write $\Longrightarrow$ for the reflexive and transitive closure of $\xrightarrow{\tau}$, $\stackrel{\alpha}{\Longrightarrow}$ for the composition $\Longrightarrow\xrightarrow{\alpha}\Longrightarrow$, and $\stackrel{\hat{\alpha}}{\Longrightarrow}$ for $\Longrightarrow$ if $\alpha = \tau$ and $\stackrel{\alpha}{\Longrightarrow}$ otherwise.

We conclude this section by presenting a notion of process equivalence in the calculus. Observational equivalence is based on the idea that two equivalent systems exhibit the same behavior at their interfaces with the environment. This requirement was captured formally through the notion of *bisimulation* [8,12]. Bisimulation is a binary relation on states of systems. Two processes are bisimilar if, for each step of one, there is a matching (possibly multiple) step of the other, leading to bisimilar states. Below, we introduce a well-known such relation on which we base our study.

Definition 1. *Bisimilarity* is the largest symmetric relation, denoted by $\approx$, such that, if $P \approx Q$ and $P \xrightarrow{\alpha} P'$, there exists Q' such that $Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q'$ and $P' \approx Q'$.

Note that bisimilarity abstracts away from internal computation by focusing on *weak* transitions, that is, transitions of the form $\stackrel{\hat{a}}{\Longrightarrow}$ and requires that bisimilar systems can match each other's *observable* behavior. We also point out that, while two bisimilar processes have the same traces, the opposite does not hold.

Bisimulation relations have been studied widely in the literature. They have been used to establish system correctness by modeling a system and its specification as two process-calculus processes and discovering a bisimulation that relates them. The theory of bisimulation relations has been developed into two directions. On one hand, axiom systems have been developed for establishing algebraically the equivalence of processes. On the other hand, proof techniques that ease the task of showing two processes to be equivalent have been proposed. The results presented in the next section belong to the latter type.

3 Confluence

In [8,9], Milner introduced and studied a precise notion of *determinacy* of CCS processes. The same notion carries over straightforwardly to the CCS_v-calculus:

Definition 2. P is *determinate* if, for every derivative Q of P and for all $\alpha \in Act$, whenever $Q \xrightarrow{\alpha} Q'$ and $Q \stackrel{\hat{\alpha}}{\Longrightarrow} Q''$, then $Q' \approx Q''$.

This definition makes precise the requirement that, when an experiment is conducted on a process, it should always lead to the same state up to bisimulation. As in pure CCS, a CCS_v process bisimilar to a determinate process is determinate, and determinate processes are bisimilar if they may perform the same sequence of visible actions. The following lemma summarizes conditions under which determinacy is preserved by the CCS_v operators.

Lemma 1

1. $\mathbf{0}$ and $\overline{a}(\tilde{v})$ are determinate processes.
2. If P is determinate so is $P\backslash F$.
3. If, for all $i \in I$, each P_i is determinate and the a_i are distinct channels, $\sum_{i\in I} a_i(\tilde{x}_i).\, P_i$ is also determinate.
4. If, for all $i \in I$, each P_i is determinate so is $\mathsf{cond}(e_1 \rhd P_1, \ldots, e_n \rhd P_n)$.
5. If P_1 and P_2 are determinate and $\mathsf{fic}(P_1) \cap \mathsf{fc}(P_2) = \emptyset$, $\mathsf{fic}(P_2) \cap \mathsf{fc}(P_1) = \emptyset$, then $P_1 \| P_2$ is also determinate.

Proof: The interesting case is Clause (5). The proof consists of a case analysis on all pairs of actions that can be taken from a derivative of $P_1 \| P_2$ and it takes advantage of the asynchronous-output mechanism. Intuitively, since output actions have no continuation, if two identical outputs are concurrently enabled within a system, it does not matter which one is fired first. □

Note that, in the case of parallel composition, previous results in CCS and the π-calculus apply a stronger side-condition than the one of Clause (5) above. Namely, these side-conditions require that the parallel components P_1 and P_2 have no channels in common. Here, however, the asynchronous nature of output actions allows us to weaken the condition as shown.

According to the definition of [9], a CCS process P is *confluent* if it is determinate and, for each of its derivatives Q and distinct actions α, β, given the transitions to Q_1 and Q_2, the following diagram can be completed.

$$\begin{array}{ccc} Q & \xrightarrow{\alpha} & Q_1 \\ {\scriptstyle\beta}\,\Downarrow & & \Downarrow\,{\scriptstyle\hat{\beta}} \\ Q_2 & \stackrel{\hat{\alpha}}{\Longrightarrow} & Q_2' \sim Q_1' \end{array}$$

Let P be the CCS_v-calculus process $P \stackrel{\text{def}}{=} a(x).\, \overline{b}(x).\, \mathbf{0}$ and consider the transitions $P \xrightarrow{a(2)} \overline{b}(2).\, \mathbf{0}$ and $P \xrightarrow{a(3)} \overline{b}(3).\, \mathbf{0}$. Clearly, the two transitions cannot be 'brought together' in order to complete the diagram above. Despite this fact, it appears natural to classify P as a confluent process. Indeed, investigation of

confluence in the context of value-passing calculi resulted in extending the CCS definition above to take account of substitution of values [16,18]. The definitions highlight the asymmetry between input and output actions by considering them separately. Here we express this separation as follows:

Definition 3. A CCS_v process P is *confluent* if it is determinate and, for each of its derivatives Q and distinct actions α, β, where α and β are not input actions on the same channel, if $Q \xrightarrow{\alpha} Q_1$ and $Q \xrightarrow{\beta} Q_2$ then, there are Q'_1 and Q'_2 such that $Q_2 \stackrel{\hat{\alpha}}{\Longrightarrow} Q'_2$, $Q_1 \stackrel{\hat{\beta}}{\Longrightarrow} Q'_1$ and $Q'_1 \approx Q'_2$. □

We may see that bisimilarity preserves confluence. Furthermore, confluent processes possess an interesting property regarding internal actions. We define a process P to be τ-*inert* if, for each derivative Q of P, if $Q \xrightarrow{\tau} Q'$, then $Q \approx Q'$. By a generalization of the proof in CCS, we obtain:

Lemma 2. If P is confluent then P is τ-inert.

As observed in [2], τ-inertness implies confluence for a certain class of processes. An analogue result also holds in our setting, as stated below.

Lemma 3. Suppose P is a fully convergent process. Then P is confluent *iff* P is τ-inert and for all derivatives Q of P

1. if $\alpha \in Act$ and $P \xrightarrow{\alpha} P_1$, $P \xrightarrow{\alpha} P_2$, then $P_1 \approx P_2$, and
2. if α, β are distinct actions and are not input actions on the same channel, if $Q \xrightarrow{\alpha} Q_1$ and $Q \xrightarrow{\beta} Q_2$ then, there are Q'_1 and Q'_2 such that $Q_2 \stackrel{\hat{\alpha}}{\Longrightarrow} Q'_2$, $Q_1 \stackrel{\hat{\beta}}{\Longrightarrow} Q'_1$ and $Q'_1 \approx Q'_2$.

Note that this is an alternative characterization of confluence for fully convergent systems which is useful in that the original transitions to be matched are single transitions. The proof of the result is a simple modification of the one found in [2]. We proceed with a result on the preservation of confluence by CCS_v operators.

Lemma 4

1. $\mathbf{0}$ and $\overline{a}(\tilde{v})$ are confluent processes.
2. If P is confluent so are $P\backslash F$ and $a(\tilde{x}).P$.
3. If, for all $i \in I$, each P_i is confluent so is $\mathsf{cond}(e_1 \rhd P_1, \ldots, e_n \rhd P_n)$.
4. If P_1 and P_2 are confluent and $\mathsf{fic}(P_1) \cap \mathsf{fc}(P_2) = \emptyset$, $\mathsf{fic}(P_2) \cap \mathsf{fc}(P_1) = \emptyset$, then $P_1 \| P_2$ is also confluent.
5. if P is confluent and $\mathsf{fic}(P) \cap \mathsf{foc}(P) = \emptyset$, then $!P$ is confluent.

Of course, here, the guarded summation clause is missing.

A main motivation in [8] for studying confluence was to strengthen determinacy to an interesting property preserved by a wider range of process-calculus operators. Here, we are also interested in such compositional results in the setting of asynchronous processes. To achieve this, we observe that, despite the rational behind the treatment of input actions in the definition of confluence, it

is often the case that distributed systems are *input-enabled*. This notion, which has been fundamental in the development of the I/O-Automata of Lynch and Tuttle [7], captures that input actions of a system are not under the control of the system and are always enabled. This suggests that the execution of an input along a certain channel does not preclude the execution of another input along the same channel. As such, a confluence-type property can be expected to hold for input actions which we formulate as follows:

Definition 4. A CCS_v process P is F^i*-confluent*, where $F \subseteq \mathcal{L}$, if, for all derivatives Q of P and for all $a \in F$, if $Q \xrightarrow{a(\tilde{v})} Q_1$ and $Q \xrightarrow{a(\tilde{u})} Q_2$ then, there are Q_1' and Q_2' such that $Q_2 \stackrel{a(\tilde{v})}{\Longrightarrow} Q_2'$, $Q_1 \stackrel{a(\tilde{u})}{\Longrightarrow} Q_1'$ and $Q_1' \approx Q_2'$. □

We may see that F^i-confluence implies that, if at some point during execution of a process an input action becomes enabled, then it remains enabled. For the case that $F = \mathcal{L}$ we simply write i-confluence for $\mathcal{L}^i$-confluence. F^i-confluence is preserved by the following operators.

Lemma 5

1. $\mathbf{0}$ and $\overline{a}(\tilde{v})$ are i-confluent processes.
2. If P is F^i-confluent and $a \in F$, $P\backslash L$, $!P$, $a.\,P$ and $!a(\tilde{x}).\,P$ are also F^i-confluent.
3. If P_i, $i \in I$, are F^i-confluent so is $\mathsf{cond}(e_1 \rhd P_1, \ldots, e_n \rhd P_n)$.
4. If P_1 and P_2 are F^i-confluent so is $P_1 \| P_2$.

We conclude with our main result:

Theorem 1. Suppose $P = (P_1 \parallel \ldots \parallel P_n)\backslash L$, where (1) each P_j is confluent, (2) each P_j is F_j^i-confluent, where $F_j = \mathsf{fic}(P_j) \cap (\bigcup_{k \neq i} \mathsf{foc}(P_k))$, and $F_j \subseteq L$, and (3) $\mathsf{fic}(P_i) \cap \mathsf{fic}(P_j) = \emptyset$, for all $i \neq j$. Then P is confluent.

Proof: The proof, which is too long to include here in its full technical detail, employs Lemma 3. We show that any derivative Q of P is τ-inert by a case analysis on the possible internal actions of Q. Suppose that this arises by a communication of the form $Q_i \xrightarrow{a(\tilde{v})} Q_i'$ and $Q_j \xrightarrow{\overline{a}(\tilde{v})} Q_j'$. Then, the $\{a\}^i$-confluence of Q_i and the confluence of Q_j imply that any action enabled by Q_i and Q_j is still possible by Q_i' and Q_j'. Further, since $a \notin \mathsf{fic}(P_k)$, for all $k \neq i$, this transition cannot be precluded from arising. Then, Clause (1) of the lemma is easy to establish using the assumption that $\mathsf{fic}(P_i) \cap \mathsf{fic}(P_j) = \emptyset$, for all $i \neq j$, whereas Clause (2), employs similar arguments and the fact that $F_j \subseteq L$ for all j. □

4 Two Applications

We proceed to illustrate the utility of the CCS_v framework and its theory of confluence via the analysis of two distributed algorithms for leader-election in a distributed ring. We assume that n processes with distinct identifiers, chosen

from a totally-ordered set, are arranged around a ring. They are numbered 1 to n in a clockwise direction and they can communicate with their immediate neighbours in order to elect as the leader the node with the maximum identifier.

Our verification methodology consists of the following steps. First, we describe an algorithm and its specification as CCS_v processes, our aim being to establish that the two processes are bisimilar. We achieve this as follows: (1) We show that the process representing the algorithm contains at least one specification-respecting execution. (2) We show that this process is confluent. By confluence properties we then obtain the required result. We point out that the first of the algorithms we consider here was also proved correct in [6] using the I/O-Automata framework.

4.1 The LCR Algorithm

The LCR algorithm [6] is a simple, well-known algorithm for distributed leader-election with time complexity $O(n^2)$. Communication between the nodes of the ring can only take place in a clockwise direction. Figure 1 presents the algorithm architecture. Each node of the ring executes the following:

> It sends its identifier to its right neighbour. Concurrently, it awaits to receive messages from its left neighbour. For each incoming message, if it contains an identifier greater than its own, it forwards the message in a clockwise direction. If it is smaller it discards it and, if it is equal, it declares itself to be the leader.

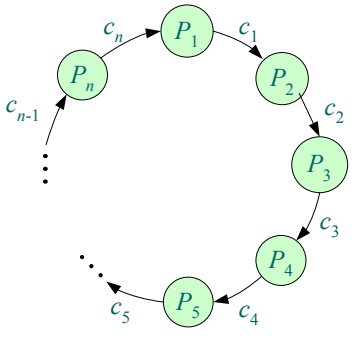

Fig. 1. The LCR-algorithm architecture

This informal description can be formalized in CCS_v: We assume a set of channels $c_1, \ldots, c_n$, where channel c_i connects processes P_i and P_{i+1}, for $1 \leq i < n$, and c_n connects processes P_n and P_1. For simplicity, hereafter, we write $i+1$ for 1 if $i = n$, and $i+1$ otherwise, and, similarly, $i-1$ for n if $i = 1$, and $i-1$, otherwise. The behavior of a node-process is described as follows:

$$
\begin{aligned}
P_i\langle u_i\rangle &\stackrel{\text{def}}{=} S_i\langle u_i\rangle \parallel R_i\langle u_i\rangle \\
S_i\langle u_i\rangle &\stackrel{\text{def}}{=} \overline{c_i}(u_i) \\
R_i\langle u_i\rangle &\stackrel{\text{def}}{=} !\, c_{i-1}(x).\, \mathsf{cond}(x < u_i \;\; \triangleright \;\; \mathbf{0},
\end{aligned}
$$

$$\begin{array}{lcl} x > u_i & \triangleright & \overline{c_i}(x), \\ \mathsf{true} & \triangleright & \overline{leader}(u_i)) \end{array}$$

In $P_i\langle u_i\rangle$, u_i is the unique identifier of the process. The process has c_{i-1} as its input channel and c_i as its output channel. It is composed of two parallel processes: a sender, $S_i\langle u_i\rangle$, and a receiver, $R_i\langle u_i\rangle$. The function of the sender process is to emit the node's identifier along channel c_i, whereas, the receiver is continuously listening along the input port of channel c_{i-1} for messages. On a receipt of a message, it discards it in case it is smaller than u_i, it forwards it in a clockwise direction in case it is larger (as expressed by the clause $\overline{c_i}(x)$), and it declares itself the leader along the common channel *leader*, if the received identifier is equal to its own.

The network is represented by as the parallel composition of the n nodes

$$LCR = (P_1\langle u_1\rangle \parallel \ldots \parallel P_n\langle u_n\rangle)\backslash L$$

where $L = \{c_1, \ldots, c_n\}$ contains all channels whose use is restricted within the system. The intended behavior of the algorithm is that the node with the maximum identifier is elected as the leader. In process-calculus terminology, we prove the following correctness result:

Theorem 2. $LCR \approx \overline{leader}(u_{max})$, where $u_{max} = \max(u_1, \ldots, u_n)$.

The proof is carried out in two steps. First, we show that LCR is capable of producing the required leader output and terminate. Then, we establish that it is confluent. The required result then follows easily from properties of confluence.

Lemma 6. $LCR \Longrightarrow \overset{\overline{leader}(u_{max})}{\longrightarrow} \Longrightarrow \approx \mathbf{0}$

Proof: Without loss of generality, we assume that node P_1 is the owner of u_{max}. Let us write $LCR_1 = (R_1\langle u_1\rangle \parallel P_2\langle u_2\rangle \parallel \ldots \parallel (P_n\langle u_n\rangle \parallel \overline{c_n}(u_1)))\backslash L$. From the definitions, it can be seen that $LCR \Longrightarrow LCR_1$ in $n-1$ transitions where, in the i^{th} transition, processes P_i and P_{i+1} communicate on channel c_i forwarding u_1 from the former to the latter. Clearly, these communications are enabled due to the fact that $u_1 > u_i$ for all $i \neq 1$. Consequently, we may derive:

$$\begin{array}{rcl} LCR_1 & \xrightarrow{\tau} & (Q_1\langle u_1\rangle \parallel P_2\langle u_2\rangle \parallel \ldots \parallel P_n\langle u_n\rangle)\backslash L \\ & \xrightarrow{\overline{leader}(u_1)} & (R_1\langle u_1\rangle \parallel P_2\langle u_2\rangle \parallel \ldots \parallel P_n\langle u_n\rangle)\backslash L \end{array}$$

where $Q_1\langle u_1\rangle = \overline{leader}(u_1) \parallel R_1\langle u_1\rangle$. Let us now consider the remaining processes. For P_2, we have that u_2 can be forwarded for at most $n-2$ steps. That is, there is a transition

$$(R_1\langle u_1\rangle \parallel P_2\langle u_2\rangle \parallel \ldots \parallel P_n\langle u_n\rangle)\backslash L \Longrightarrow (R_1\langle u_1\rangle \parallel R_2\langle u_2\rangle \parallel \ldots \parallel P_n\langle u_n\rangle)\backslash L$$

and, similarly, all u_i, $i > 2$, can proceed up to at most node P_1, and then become blocked:

$$(R_1\langle u_1\rangle \parallel R_2\langle u_2\rangle \parallel \ldots \parallel P_n\langle u_n\rangle)\backslash L \Longrightarrow LCR_2 = (R_1\langle u_1\rangle \parallel \ldots \parallel R_n\langle u_n\rangle)\backslash L$$

By observing the definitions of the processes, we conclude that, since all processes are only willing to receive, but none is ready to send, the resulting process is a deadlocked process, that is, a process bisimilar to **0**. This completes the proof. □

We now have our key observation:

Lemma 7. *LCR* is confluent.

PROOF: This follows from Theorem 1 of the previous section. We check its hypotheses: Each P_i is clearly a composition of two confluent components. Thus, by Lemma 4(4), it is confluent. Since, in addition, (1) by Lemma 5 all processes are i-confluent, (2) exactly one process takes input from each channel c_i and (3) each $c_i \in L$, by Theorem 1, we may conclude that *LCR* is a confluent process. □

From the two previous results, and since confluence implies τ-inertness, we have that $LCR \approx LCR_1$, $LCR_1 \approx \overline{leader}(u_{max}).\, LCR_2$ and $LCR_2 \approx \mathbf{0}$. Consequently, $LCR \approx \overline{leader}(u_{max})$, as required.

4.2 The HS Algorithm

The second algorithm that we study is the HS algorithm of Hirschberg and Sinclair [3]. It is distinguished from the LCR algorithm in that here communication between the ring nodes is bidirectional, and the time complexity is $O(n \lg n)$.

In the HS algorithm, execution at a node proceeds in phases. In phase k, a node forwards its identifier in both directions in the ring. The intention is that the identifier will travel a distance of 2^k, assuming that it does not encounter a node with a greater identifier and, then, it will follow the opposite direction returning to its origin node. If both tokens return to their origin, the node enters phase $k+1$ and continues its execution. If the identifier completes a cycle and reaches its origin in the outbound direction, then, the node declares itself as the leader. For the correct execution of the algorithm, messages transmitted in the ring are triples of the form $\langle id, dir, dist\rangle$, where id is a node identifier, $dir \in \{in, out\}$ is the direction of the message (*in* represents the inward direction and *out* the outward direction) and *dist* is the distance the node has still to travel.

Let us describe the precise behavior of an HS-node in CCS_v. The architecture considered is shown in Fig. 2. We assume a set of channels $\{c_{i,i-1}, c_{i,i+1} \mid 1 \leq i \leq n\}$, where channel $c_{i,j}$ connects process P_i to process P_j. (Note that we employ the same interpretation of the summation and substraction operators as described in Sect. 4.1.)

A node-process is then modeled as follows, where u_i is the identifier of the process and ϕ_i the phase of the process, initially set to 0.

$$P_i\langle u_i, \phi_i\rangle \stackrel{\text{def}}{=} (S_i\langle u_i, \phi_i\rangle \parallel R_i\langle u_i\rangle \parallel E_i\langle u_i\rangle)\backslash\{cph, elect\}$$

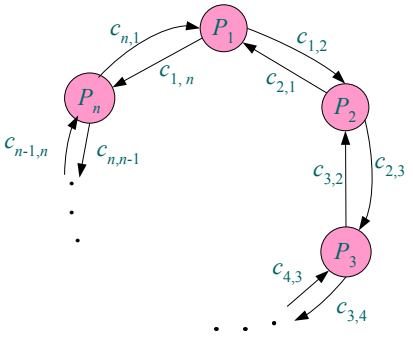

Fig. 2. The HS-algorithm architecture

Thus P_i is the parallel composition of three components responsible for sending and receiving messages and for electing a leader:

$$
\begin{aligned}
E_i\langle u_i\rangle &\stackrel{\text{def}}{=} \mathit{elect}.\,\overline{\mathit{leader}}(u_i)\\
S_i\langle u_i,\phi_i\rangle &\stackrel{\text{def}}{=} \overline{c_{i,i+1}}(u_i,\mathit{out},2^{\phi_i})\\
&\quad \parallel \overline{c_{i,i-1}}(u_i,\mathit{out},2^{\phi_i})\\
&\quad \parallel \mathit{cph}.\,\mathit{cph}.\,S_i\langle u_i,\phi_i+1\rangle\\
R_i\langle u_i\rangle &\stackrel{\text{def}}{=} !\,c_{i-1,i}(x,d,h).\\
&\qquad \mathsf{cond}((x<u_i) \quad \triangleright \quad \mathbf{0},\\
&\qquad\quad (x>u_i \wedge d=\mathit{out} \wedge h\neq 1) \quad \triangleright \quad \overline{c_{i,i+1}}(x,\mathit{out},h-1),\\
&\qquad\quad (x>u_i \wedge d=\mathit{out} \wedge h=1) \quad \triangleright \quad \overline{c_{i,i-1}}(x,\mathit{in},1),\\
&\qquad\quad (x\neq u_i \wedge d=\mathit{in}) \quad \triangleright \quad \overline{c_{i,i+1}}(x,\mathit{in},1),\\
&\qquad\quad (x=u_i \wedge d=\mathit{in}) \quad \triangleright \quad \overline{\mathit{cph}},\\
&\qquad\quad \mathsf{true} \quad \triangleright \quad \overline{\mathit{elect}}\,)\\
&\quad \parallel !\,c_{i+1,i}(x,d,h).\\
&\qquad \mathsf{cond}((x<u_i) \quad \triangleright \quad \mathbf{0},\\
&\qquad\quad (x>u_i \wedge d=\mathit{out} \wedge h\neq 1) \quad \triangleright \quad \overline{c_{i,i-1}}(x,\mathit{out},h-1),\\
&\qquad\quad (x>u_i \wedge d=\mathit{out} \wedge h=1) \quad \triangleright \quad \overline{c_{i,i+1}}(x,\mathit{in},1),\\
&\qquad\quad (x\neq u_i \wedge d=\mathit{in}) \quad \triangleright \quad \overline{c_{i,i-1}}(x,\mathit{in},1),\\
&\qquad\quad (x=u_i \wedge d=\mathit{in}) \quad \triangleright \quad \overline{\mathit{cph}},\\
&\qquad\quad \mathsf{true} \quad \triangleright \quad \overline{\mathit{elect}}\,)
\end{aligned}
$$

Thus, E_i awaits a notification (triggered by process R_i) that a leader-event should be produced. S_i emits the message $(u_i, \mathit{out}, 2^{\phi_i})$ in both directions along the ring. Concurrently, it waits the receipt of two confirmations via channel cph (emitted by process R_i) that the token has successfully travelled in both directions through the ring, in which case it increases the phase by 1. On the other hand, process R_i is listening on ports $c_{i,i-1}$ and $c_{i,i+1}$. The first summand of the process deals with the former channel, and the second with the latter. We consider the first summand, the second one is symmetric. If a message (x, d, h) is received on channel $c_{i-1,i}$, six cases exist:

1. If $x < u_i$, the message is ignored.
2. If the message is travelling in the outbound direction and $h > 1$, it is forwarded to node $i + 1$ with h decreased by 1.
3. If the message is travelling in the outbound direction and $h = 1$, it is sent back to node $i - 1$ to begin its inward journey.
4. If $x \neq u_i$ and the message is travelling its inward journey, it is forwarded towards its origin.
5. If the node's own identifier is received by the process while travelling its inward journey, process R_i emits a notification along channel *cph*.
6. Finally, if none of the above holds, implying that $x = u_i$ and $d = out$ (that is the identifier has survived performing a cycle around the ring), the node produces a notification (to be received by process E_i) that the node should be declared ring leader.

The network is represented by the parallel composition of the n nodes

$$HS = (P_1\langle u_i, 0\rangle \parallel \ldots \parallel P_n\langle u_n, 0\rangle)\backslash L$$

where all channels in $L = \{c_{i,i-1}, c_{i,i+1} \mid 1 \leq i \leq n\}$ are restricted within the system. The correctness criterion is expressed, again, as the following equivalence between the algorithm and its specification:

Theorem 3. $HS \approx \overline{leader}(u_{max})$, where $u_{max} = \max(u_1, \ldots, u_n)$.

As before, the proof is carried out in two steps: We show that HS is capable of producing the required leader output and terminate, and that it is confluent.

Lemma 8. $HS \Longrightarrow \overset{\overline{leader}(u_{max})}{\longrightarrow} \Longrightarrow \approx \mathbf{0}$

Proof: Without loss of generality, we assume that node P_1 is the owner of u_{max}. Let us write $HS_i = (P_1\langle u_1, i\rangle \parallel P_2\langle u_2, 0\rangle \parallel \ldots \parallel P_n\langle u_n, 0\rangle)\backslash L$. From the definitions, it can be seen that

$$HS \Longrightarrow HS_1 \Longrightarrow HS_2 \Longrightarrow \ldots \Longrightarrow HS_k$$

where $k = \lceil \lg n \rceil$. Specifically, the transition $HS_i \Longrightarrow HS_{i+1}$ consists of $2 \cdot 2 \cdot 2^i$ internal actions pertaining to the outward and inward journey of distance 2^i, of identifier u_1 in the clockwise and anticlockwise direction within the ring. Clearly, these communications are enabled due to the fact that $u_1 > u_i$ for all $i \neq 1$. Then, we may derive:

$$\begin{aligned} HS_k \ (\overset{\tau}{\longrightarrow})^{n+1} \ & HS_{k+1} = (P_{1,1}\langle u_1, k\rangle \parallel P_2\langle u_2, 0\rangle \parallel \ldots \parallel P_n\langle u_n, 0\rangle)\backslash L \\ \overset{\overline{leader}(u_1)}{\longrightarrow} \ & HS_{k+2} = (P_{1,2}\langle u_1, k\rangle \parallel P_2\langle u_2, 0\rangle \parallel \ldots \parallel P_n\langle u_n, 0\rangle)\backslash L \end{aligned}$$

where $P_{1,2}\langle u_1, k\rangle = (\overline{c_{i,i-1}}(u_i, out, 2^k) \parallel cph.\, cph.\, S\langle u_1, k\rangle \parallel R_1\langle u_1\rangle)\backslash\{cph, elect\}$ and $P_{1,1}\langle u_1, k\rangle = P_{1,2}\langle u_1, k\rangle \parallel \overline{leader}(u_1)$. These $n + 1$ internal steps correspond to a cycle of u_1 in a clockwise direction, and its return to its origin node while in

the outbound direction. This triggers R_1 to produce the message *elect*, received by process E_i which in turn proceeds to declare the node as the leader.

Let us now consider the remaining enabled communications. First note that the prefix $\overline{c_{i,i-1}}\ (u_1, out, 2^k)$ can trigger a cycle of u_1 in the anticlockwise direction, while all other processes will forward their identifiers in the clockwise and anticlockwise direction. It is easy to show that the journey of each of these identifiers will eventually become blocked, on reaching a node with a larger identifier, possibly P_1. This gives rise to the transition

$$HS_{k+2} \Longrightarrow HS_{k+3} = (P_{1,3}\langle u_1, k\rangle \parallel Q_2\langle u_2\rangle \parallel \ \ldots \ \parallel Q_n\langle u_n\rangle)\backslash L$$

where $P_{1,3}\langle u_1, k\rangle = (cph.\, cph.\, S\langle u_1, k\rangle \parallel R_1\langle u_1\rangle \| \overline{elect}\rangle)\backslash\{cph, elect\}$, and for $2 \leq i \leq n$, $Q_i\langle u_i\rangle = S'_i\langle u_i, \phi_i\rangle \parallel R_i\langle u_i\rangle \parallel E_i\langle u_i\rangle)\backslash\{cph, elect\}$ where $S'_i\langle u_i, \phi_i\rangle$ is one of the processes $cph.\, cph.\, S_i\langle u_i, \phi_i\rangle$ and $cph.\, S_i\langle u_i, \phi_i\rangle$. By observation, no communication is enabled in the resulting process, a fact that renders it bisimilar to **0**. This completes the proof. □

We now have our key observation:

Lemma 9. *HS* is confluent.

Proof: This follows from a multiple application of Theorem 1. To establish the confluence of a P_i we observe that, by Lemma 5, each of E_i, S_i and R_i is i-confluent. Lemma 4 and simple observation leads to the conclusion that the components are also confluent. Since the components share between them only the names *cph* and *elect*, which are hidden at the top level of the process, and no two components share an input name, by Theorem 1, each P_i is confluent.

Since, in addition, (1) all P_i are i-confluent, (2) exactly one process takes input from each channel $c_{i,j}$, and (3) each $c_{i,j} \in L$, by Theorem 1, we may conclude that *HS* is a confluent process. □

From the two previous results, and since confluence implies τ-inertness, we have that $HS \approx HS_{k+1}$, $HS_{k+1} \approx \overline{leader}(u_{max}).\, HS_{k+2}$ and $HS_{k+2} \approx \mathbf{0}$. Consequently, $HS \approx \overline{leader}(u_{max})$, as required.

5 Conclusions

We have considered an asynchronous process calculus and we have developed its associated theory of confluence. In doing this, our main objective has been the elaboration of concepts and techniques useful in proving the correctness of distributed algorithms. Specifically, we have given results for establishing the confluence of systems in a compositional manner and we have exploited the property of τ-inertness possessed by confluent systems for showing that systems are correct. Using these ideas, we have illustrated the correctness of two leader-election algorithms. As we have already mentioned, these two algorithms were also proved correct in [6] using the I/O-Automata framework. In our view, our proofs offer additional interesting insights in that the use of confluence aids towards the algorithms' understanding and simplifies their analysis.

Regarding the applicability of the proposed methodology, initially, it appears that it can be useful in a variety of contexts were confluent computations are running in parallel, e.g., parallel and distributed algorithms for function computation. In future work, we plan to further investigate the applicability of the methodology and, especially, the notion of F^i-confluence. (We are currently considering application of the results to verify distributed leader-election algorithms in networks of arbitrary topological structures). Further, we would like to extend the theory to a setting with mobility and location primitives. Naturally, the property of confluence is not satisfied in general by distributed algorithms, thus, a final research direction is the development of analogous results for algorithm verification which employ weaker partial-confluence properties.

References

1. G. Boudol. Asynchrony and the π-calculus. Technical Report RR-1702, INRIA-Sophia Antipolis, 1992.
2. J. F. Groote and M. P. A. Sellink. Confluence for process verification. In *Proceedings of CONCUR'95*, LNCS 962, pages 204–218, 1995.
3. D. S. Hirschberg and J. B. Sinclair. Decentralized extrema-finding in circular configurations. *Communications of the ACM*, 23(11):627–628, 1980.
4. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proceedings of ECOOP'91*, LNCS 512, pages 133–147, 1991.
5. X. Liu and D. Walker. Confluence of processes and systems of objects. In *Proceedings of TAPSOFT'95*, LNCS 915, pages 217–231, 1995.
6. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1997.
7. N. A. Lynch and M. R. Tuttle. An introduction to Input/Output Automata. *CWI-Quarterly*, 2(3):219–246, 1989.
8. R. Milner. *A Calculus of Communicating Systems*. Springer, 1980.
9. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
10. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts 1 and 2. *Information and Computation*, 100:1–77, 1992.
11. U. Nestmann. *On Determinacy and Non-determinacy in Concurrent Programming*. PhD thesis, University of Erlangen, 1996.
12. D. Park. Concurrency and automata on infinite sequences. In *Proceedings of* 5^{th} *GI Conference*, LNCS 104, pages 167–183, 1981.
13. A. Philippou and D. Walker. On transformations of concurrent object programs. In *Proceedings of CONCUR'96*, LNCS 1119, pages 131–146, 1996.
14. A. Philippou and D. Walker. On confluence in the π-calculus. In *Proceedings of ICALP'97*, LNCS 1256, pages 314–324, 1997.
15. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.
16. M. Sanderson. *Proof Techniques for CCS*. PhD thesis, University of Edinburgh, 1982.
17. D. Sangiorgi. A theory of bisimulation for the π-calculus. In *Proceedings of CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 127–142. Springer, 1993.
18. C. Tofts. *Proof Methods and Pragmatics for Parallel Programming*. PhD thesis, University of Edinburgh, 1990.