

Layers of Confluence for Actors

Ludovic Henrio

CNRS
Lyon, France
ludovic.henrio@cnrs.fr

Einar Broch Johnsen

University of Oslo
Oslo, Norway
einarj@ifi.uio.no

Åsmund Aqissiaq Arild

Kløvstad
University of Oslo
Oslo, Norway
aaklovst@ifi.uio.no

Violet Ka I Pun

Western Norway University of
Applied Sciences
Bergen, Norway
Violet.Ka.I.Pun@hvl.no

Yannick Zakowski

Inria
Paris, France
yannick.zakowski@inria.fr

Abstract

This paper introduces a novel proof technique to show that parallel or distributed programs exhibit confluent behaviour, even when the execution of these programs is inherently non-deterministic. The proposed method allows us to prove the confluence of programs for which standard properties such as strong confluence or commutativity of operations do not hold. Our technique builds on a method to prove the confluence of rewrite systems by de Bruijn, which we first adapt and formalise in Rocq. This method can be seen as a specialised induction principle for proving confluence. The paper further considers how this induction principle can be used in the context of programming languages. We show how the proof method can be instantiated to establish confluence conditions for programs in a small Actor-like programming language and demonstrate the application of the method to prove the confluence of a class of programs that cannot be proven to have deterministic behaviour by standard techniques.

CCS Concepts: • Theory of computation → Program analysis; Logic and verification; Operational semantics.

Keywords: formalisation, programming language semantics, deterministic behaviour, confluence

ACM Reference Format:

Ludovic Henrio, Einar Broch Johnsen, Åsmund Aqissiaq Arild Kløvstad, Violet Ka I Pun, and Yannick Zakowski. 2026. Layers of Confluence for Actors. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '26)*, January 12–13, 2026, Rennes, France. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3779031.3779104>



This work is licensed under a Creative Commons Attribution 4.0 International License.

CPP '26, Rennes, France

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2341-4/2026/01

<https://doi.org/10.1145/3779031.3779104>

1 Introduction

Parallel programs can be highly non-deterministic and produce a multitude of different executions. In order to reason about such programs, non-determinism needs to be controlled [36]. This paper introduces a novel semantic proof method to show that parallel programs exhibit *confluent* behaviour, i.e., that the non-determinism in the parallel and distributed computation is well-behaved in the sense that the produced results are deterministic. Inspired by a confluence proof for rewrite systems by de Bruijn [17, 19], the proposed method expands on classical reduction techniques based on commutative behaviour [18] by indexing the reductions and authorising reductions of lower indexes when proving commutativity. We show that the proposed method allows us to prove the confluence of programs for which strong confluence [33] does not apply. Specifically, we provide a mechanisation in Rocq [40] of the soundness of de Bruijn's methodology and of its application to proving the confluence of programs in a minimal Actor-like programming language.

1.1 Parallelism, Non-determinism and Races

While parallelism and distribution accelerate program execution, they also introduce interference, asynchrony, and non-determinism. Parallel programs exhibit *data races*, when different processes access shared memory in different orders, and *communication races*, when message delivery might occur in different orders [24]. These races make reasoning about parallel programs intrinsically difficult, because the races lead to an explosion in the possible program executions and thereby to an explosion in the program's reachable state space. As programmers, we can tackle this problem either by reducing the number of possible executions or by ensuring that the different executions are equivalent.

When reducing the number of possible executions, we tackle the state space explosion by ensuring that a program contains enough synchronisation to restrict its behaviour, by means of, e.g., lock patterns [4], transactional memory [28], futures [7] and promises [38], combined with sophisticated

verification techniques [16, 25, 41]. By adding synchronisation constructs to the parallel program, we effectively reduce the number of possible program executions to a size that we can handle. For example, binary locks can be used to protect a critical region, introducing atomic blocks and thereby eliminating interference between threads executing in parallel. This allows us to reason about the program by considering the effect of the atomic blocks independently, using sequential reasoning techniques.

Parallel programming languages can provide libraries that offer race-free (or “thread-safe”) constructs, such as hyperobjects in Cilk [22] and asynchronous operations over partitioned state spaces in X10 [12]. Actor languages [2, 5, 15, 27] promise race-free computations by design. Although they resolve data races, actor programs are still subject to communication races: the order in which messages are received by an actor can introduce non-determinism. Actor languages can be designed to resolve communication races and guarantee deterministic behaviour [11, 14, 26, 39], by restricting their expressivity or their underlying computational model.

By ensuring that different executions are equivalent, we need only consider some of the possible program executions to understand the program’s behaviour. Program reductions, first introduced by Lipton [37], allow us to reason about parallel code as if it had atomic blocks by ensuring that a program and its reduced version share behavioural properties of interest. The underlying technique is based on commutativity of program statements, and has many extensions; e.g., *purity* [21] allows procedures that leave the state invariant to be considered commutative. When searching for reachability properties, partial-order reduction (POR) techniques [1, 23] prune the execution space by considering execution paths modulo the reordering of commuting transitions.

1.2 Confluence: Beyond Commutativity

The correctness of the approaches discussed above relies on an underlying analysis of program executions and their confluence properties: Although the program execution may well be non-deterministic, executions are equivalent in the sense that they eventually reach the same state after further execution. If two statements alter separate parts of memory, they can obviously be executed in any order and we have *confluence by separation*. If the statements alter overlapping parts of memory, we say that the statements are commuting and have *confluence by commutativity* if the corresponding transitions exhibit the so-called *diamond property* [18] in the underlying reduction system (i.e., the operational semantics of the considered language): the two transitions, in either order, still lead to the same state. Thus, the diamond property is an instance of a *confluence property*.

In general, confluence properties are properties of rewrite systems that express whether, when two rewrite rules can be applied to a term, it is possible to obtain the same term again by further rewrites. In other words, they express whether two

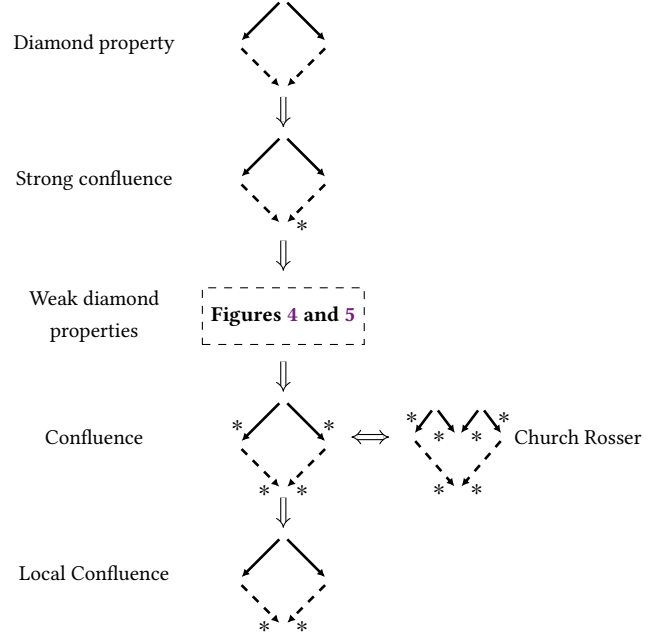


Figure 1. Relations between confluence properties.

different applications of rewrite rules are in conflict or not. Confluence properties differ in the number of steps needed to obtain the same term again after a diverging step. Figure 1 gives an overview of standard confluence properties, which were systematised by Huet [33] (see also textbooks [6, 18]), and situates the properties explored in this paper. In the figure, starred transitions indicate zero or more applications of a rewrite rule and dashed transitions indicate the steps needed to recover. Furthermore, the properties are ordered in terms of entailment. Diamond entails strong confluence, which in turn entails weak confluence and confluence (or equivalently Church Rosser); confluence entails local confluence. Local confluence does not entail confluence and is in general insufficient because, after a few diverging steps, the two terms might never converge. In programming language semantics, we often rely on a diamond property to ensure confluence, it is indeed sufficient to recursively apply the diamond property to prove confluence. Huet showed that we could relax the strict one-step commutation of the diamond by allowing one of the sides of the diagram to have several steps, but *only one side and this side cannot be chosen when doing the proof*. By symmetry, the critical pair must be closed both by making a single step from the left side and a single step from the right side. De Bruijn’s unpublished note [17, 19] on *weak diamond properties*, which paved the way for proof techniques based on decreasing diagrams [44], is weaker than strong confluence, yet sufficient to prove confluence. The weak diamond property is a generalisation of strong confluence that introduces an *induction* on a label associated to

each reduction. Confluence is generally the high-level property that is desired; in concurrent programming it means that the states resulting from all the possible concurrent executions are the same. Local confluence is rarely useful as it does not entail confluence and in programming language theory, controlling the execution to make a single step of execution does not make sense in general.

1.3 Contributions

Data races in programming languages are typically shown to be confluent by application of the diamond property [9]: for example, using so-called left- or right-movers to reorder a sequence of statements to normal form, thereby showing that a race does not affect the final state of a program. In contrast, communication races are generally solved by ensuring that the behaviour is independent of message ordering or that message ordering is deterministic [11, 39].

In this paper, we propose a proof method to show that a program behaves deterministically even in the presence of complex races. To this aim, we first mechanise the proof of a generic confluence theorem that relies on a layered view of the program semantics, namely de Bruijn's *weak diamond property*. We then identify conditions under which programs with communication races are confluent, and formalise these confluence properties for λ_{act} , a small actor-like calculus. In short, our contributions are as follows:

- a fully mechanised proof of de Bruijn's diagrammatic proof method for confluence of abstract reduction systems (Sec. 3);
- an analysis of sufficient properties to apply de Bruijn's diagrammatic proof method for confluence of actor-based programming languages (Sec. 4), based on a small actor language λ_{act} ; and
- a fully formalised demonstration of how the diagrammatic method can be applied to prove confluence of programs in λ_{act} (Sec. 5).

All our results are formalised in Rocq. Key definitions and lemmas in the paper are directly hyperlinked using the Rocq logo: .¹ We begin by discussing a motivating example and the high-level overview of our results in Sec. 2.

2 Motivation

The proof technique we introduce is not intrinsically tied to any specific programming paradigm. However, we illustrate it on actor languages, where deterministic behaviour has been extensively studied as a means to control the amount of concurrent behaviours without completely preventing them. There is a broad diversity in actor languages, but they are typically characterised by the fact that each actor is a computing entity that has exclusive access to its state, and only interacts with others via messages. We consider here a

```

1 ComputeF {
2   F(x) { // omitted
3     return e }
4 }
5 ComputeG {
6   G(x) { // omitted
7     return e }
8 }
9 Worker1, Worker2 {
10  Work(n) {
11    // accumulates the sum F(1)+G(1)+...
12    // computed by computeF and computeG
13    a := 0;
14    for i := 1 to n {
15      x := ComputeF.F(i);
16      a := a+x;
17      x := ComputeG.G(i);
18      a := a+x;
19    };
20    return a }
21 }
22 Client1 {
23   Init() { x := Worker1.Work(4); return x }
24 }
25 Client2 {
26   Init() { x := Worker2.Work(3); return x }
27 }
28 Client3 {
29   Init() { x := Worker1.Work(2); return x }
30 }

```

Figure 2. Motivating example.

programming paradigm where computing entities are internally single-threaded and interact by sending messages. In such a computational model, the only concurrent operation is the sending of a message to the same destination [11].

Generally, racy communications introduce nondeterminism by allowing different orders in the reception of messages. However, racy communications do not necessarily lead to non-deterministic behaviour; notably, if the racy communications only involve pure processes, or if there is separation of data between the racing processes. In this section, we first illustrate our proof technique on a program for which such deterministic behaviour can be expected, and stress the subtlety of the underlying formal argument. Finally, we formulate the class of properties we capture by our approach.

2.1 Motivating Example

Consider the program in Fig. 2, in which 7 computing entities are defined: ComputeF, ComputeG, Worker1, Client1, ..., Client3. In the example, each computing entity is declared in a style inspired by object-oriented languages. The statement $A.F(x)$ requires the computing entity A to execute the code defined in function F. We adopt a simplistic semantics: contrarily to most actor languages, the computation is not asynchronous and the invoker immediately waits for the result.

Let us have a closer look at the example. Three clients delegate work to a worker, taken among two available, in order to compute the summation of a sequence of computations: Client1 and Client3 call Worker1 (Lines 23 and 29), while Client2 calls Worker2 (Line 26). The summation values are

¹Artefact available at <https://doi.org/10.5281/zenodo.17712651>

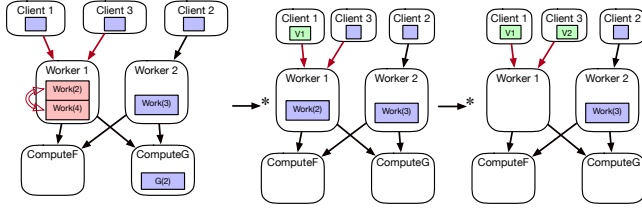


Figure 3. A partial execution of the example: race condition in red.

computed by invoking two methods *F* and *G* (Lines 15 and 17) which are respectively defined in *ComputeF* and *ComputeG*, whose computations are independent from each other.

Observe that racy communications occur when the two workers call *ComputeF* (respectively *ComputeG*) simultaneously, or when *Client1* and *Client3* call *Worker1* at the same time. Consider a possible partial execution as follows:

1. The three concurrent clients call method *Work* on the corresponding workers, *Clients 1* and *3* are in a race to call *Worker1*;
2. *Worker2* starts its execution first, performing calls to *F* and *G*, until the call to *G(2)* is being processed;
3. *Client1* wins the race: *Worker1* gets *Work(4)* scheduled before *Work(2)*.

The state of the execution at the end of (3) is depicted in Fig. 3 (left), where the race condition is illustrated in red (the bottom task is the first received). Indeed, a similar execution exists where *Client3* has won the race, and hence *Work(2)* gets served before *Work(4)*: let us refer to this alternate state as (3'). In the figure, a client waiting for a result contains a blue box, that becomes green with a value when the result is received. We therefore consider the following questions: (i) do both executions lead to the same result? (ii) how can we formally prove that they do? (iii) can we establish sufficiently general conditions to guarantee such confluence results?

Assuming an intuitive notion of purity for *F* and *G*, there is little doubt that (i) should hold: it boils down to arguing that “both executions of *Work(2)* and *Work(4)* commute.” Proving so is however non-trivial: let us sketch why both executions states (3) and (3') can reduce to the same state. The most straightforward way to reach the same configuration starting from (3) (resp. (3')) is illustrated in Fig. 3: First, serve fully *Work(4)* (resp. *Work(2)*)—doing so requires to finish the current pending execution of *G(2)* to free *computeG*. We end in the middle state, where the computed value *v1* (resp. *v1'*) is depicted in green. It remains to serve *Work(2)* (resp. *Work(4)*), yielding *v2* (resp. *v2'*): we end in the state in Fig. 3 (right), and must argue why these are identical in both executions.

For this argument to hold, these states must indeed be reachable: all intermediate computations must not diverge. Furthermore, both the computed values and the resulting

global states must be identical. The internal state of a computing entity, such as variable *a*, can differ by ensuring at the language level that it is restored at each method call. We formalise a sufficient condition of purity in Sec. 5.

Assuming an adequate notion of purity, we need to argue that different interleavings of two executions of *F* and three executions of *G* lead to the same state. While an ad hoc proof is possible, we seek an expressive and systematic proof technique. Such a technique should also account for some extensions—there may be an arbitrary number of clients and workers, and the workers (notably *ComputeF* and *ComputeG*) may delegate work to further workers.

Unfortunately, techniques such as Huet’s strong confluence theorem [33] fall short here. Strong confluence generalises the diamond property by leaving lenience for several reductions on one side of the diagram when closing on a critical pair (see Fig. 1). In the example however, we can only close the critical pair by performing several computations on each side of the diagram (for example 6 calls to *F()* and 7 calls to *G()* must be performed in the scenario above). The classical trick, used in λ -calculus [31], to rely on a more general reduction rule does not seem applicable. We need a weaker notion of confluence, and de Bruijn’s result on confluence of indexed reduction systems enters the stage [17, 19].

This paper demonstrates how to capture a large class of programs, encompassing our motivating example, for which confluence can be proven using de Bruijn’s result. Furthermore, all results are mechanised in the Rocq proof assistant.

2.2 Problem Statement: Confluence Theorems

The fact that a stateless entity can be queried by several entities in any order is common folklore. In this work, we investigate how to formally characterise and prove sound such folklore, i.e., sufficient conditions for ensuring confluence. The example above illustrates how confluence can be retrieved despite a very general setting: the host language as a whole is imperative, and arbitrary delegation to dynamically-computed threads can occur.

To characterise the configurations that make use of these features, the theorems we prove assume that we know the call graph between concurrent entities, i.e., which entity can call which other entities at runtime (or that we have a static approximation of this call graph). Namely, we prove two theorems for determinacy:

Problem 1: Single server determinacy. *Suppose that only a single entity can be accessed by others, and that its methods are pure in the sense that they terminate and leave the global state unchanged—the corresponding call graph is depicted in Fig. 11a. Then the program behaves deterministically.*

Considering the situation in Fig. 3, this theorem would be sufficient for clients directly delegating *ComputeF* and *ComputeG* to a single server, bypassing in particular the workers. It is nonetheless insufficient for the example as we have

a non-trivial chain of stateless servers. It is however crucially a chain, with no cycle that could introduce divergence depending on the schedule. We illustrate this general acyclicity constraint on the call graph in Fig. 11b. We prove determinacy in this more general scenario:

Problem 2: Chain of servers determinacy. *If the computing entities of the program are organised into a set of clients (that do not receive calls), and a set of pure servers that can all be given an index so that a server only calls servers at lower levels, and if methods are adequately pure, then the program behaves deterministically.*

We make precise the assumptions of this theorem, and notably the one of purity, in Sec. 5.1.

2.3 Methodology

In the rest of the article, we explain how we get to a mechanised proof in Rocq of the confluence for such an example. We first mechanise de Bruijn’s theorem for arbitrary rewriting systems (Sec. 3, 🍷). We then formalise λ_{act} (Sec. 4, 🍷), our minimalist actor calculus, and leverage de Bruijn’s result to prove the determinacy of appropriate configurations of λ_{act} exhibiting a layered call graph (Sec. 5.3, 🍷). To ease discharging the semantic assumptions over the program’s call graph, we prove correct a simple but sufficient analysis (🍷). Finally, we are ready to prove the example confluent by putting all ingredients together (🍷).

3 Layered Confluence of Rewriting Systems

Setting aside the specifics of actors, let us first turn our attention to the broader context of confluence for rewriting systems. The core observation we make is that while Huet’s strong confluence technique [33] falls short for the kind of systems we consider, a generalisation due to de Bruijn in an unpublished note [17], and revived in a modern presentation by Endrullis and Klop [19], fits the bill perfectly.

In this section, we restate de Bruijn’s result, outline its proof, and point out the adjustments that our Rocq formalisation led us to make. We refer to Endrullis and Klop [19] for a more detailed exposition of the proof, and its generalisation to abstract rewriting systems indexed by partial orders.

3.1 Preliminaries: Notations and Definitions

We consider an abstract rewriting system (ARS), i.e., a set of objects A equipped with a binary relation $\rightarrow \subseteq A \times A$. We write $\rightarrow^?$, $\rightarrow^*$, $\leftarrow$, and $\leftarrow^*$ for respectively the reflexive closure, the transitive closure, the inverse relation, and the transitive closure of the inverse relation of $\rightarrow$, and use similar notations for other relations. We write the composition and inclusion of relations in infix position, by respectively $\cdot$ and $\subseteq$. Given an object a , we write $a \nrightarrow$ if there exists no element b such that $a \rightarrow b$.

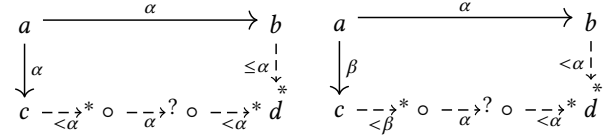


Figure 4. $D_1(\alpha)$: peaks $c \xleftarrow{\alpha} a \rightarrow_{\alpha} b$

Figure 5. $D_2(\alpha, \beta)$: peaks $c \xleftarrow{\beta} a \rightarrow_{\alpha} b$ ($\beta < \alpha$)

In a rewriting system, *confluence* can be formalised as follows: two partial evaluations of a system can always further reduce down to the same state.

Definition 3.1 (Confluence). A relation $\rightarrow \subseteq A \times A$ is *confluent* if $* \leftarrow \cdot \rightarrow^* \subseteq \rightarrow^* \cdot * \leftarrow$.

Jumping straight into a proof of confluence is a daring approach: intuitively, the quantification over two arbitrarily partial evaluations leaves too much leeway. One therefore looks for sufficient conditions whose challenges consider a pair of single steps of reduction, known as a *peak*. Figure 1 illustrates the most standard slice of this landscape. The *diamond property* is an ideal situation: the system may stray away from a step, but can immediately be reunified. Weaker but still sufficient, Huet’s *strong confluence* allows for one side to perform several steps to close the diagram. Interestingly, giving such lenience to both sides, i.e., considering *local confluence*, does not entail confluence [33].

As we have informally argued in Sec. 2.1, strong confluence does not hold for the systems we consider. We hence turn our attention to a lesser known result from the literature: *weak diamond* entails confluence. The proof of this result more specifically derives the *Church Rosser* property, an equivalent characterisation of confluence.

3.2 Confluence from Weak Diamond Property

De Bruijn’s intuition is to generalise Huet’s strong confluence proof, but over a *stratified* view of the ARS. At level 0, one proves strong confluence of the corresponding subsystem, while at subsequent levels, reductions at lower indices can be used more liberally, almost as in local confluence.

Definition 3.2. 🍷 An *indexed ARS* (iARS) $(A, (\rightarrow_{\alpha})_{\alpha \in I}, <)$ consists of a set of objects A , and a family $(\rightarrow_{\alpha})_{\alpha \in I}$ of relations $\rightarrow_{\alpha} \subseteq A \times A$ indexed by some totally ordered set $(I, <)$.

A standard ARS can be recovered as $\rightarrow \triangleq \bigcup_{\alpha \in I} \rightarrow_{\alpha}$. Furthermore, we write $\rightarrow_{<\beta}$ (resp. $\rightarrow_{\leq\beta}$) for the relation $\bigcup_{\alpha < \beta} \rightarrow_{\alpha}$ (resp. $\bigcup_{\alpha \leq \beta} \rightarrow_{\alpha}$). We are now ready to state the weak diamond property and its associated theorem.

Definition 3.3 (Weak diamond property). An indexed ARS $(A, (\rightarrow_{\alpha})_{\alpha \in I}, <)$ satisfies the weak diamond property if any peak $c \xleftarrow{\alpha} a \rightarrow_{\alpha} b$ can be closed by Fig. 4, and any peak $c \xleftarrow{\beta} a \rightarrow_{\alpha} b$, where $\beta < \alpha$, can be closed by Fig. 5.

Theorem 3.4 (Weak diamond theorem [17, 19], 🍷). *If an iARS satisfies the weak diamond property, then $\rightarrow$ is confluent.*

Remark that for $\alpha = 0$, there are no diagrams $D_2(\alpha, \cdot)$ to be considered. Furthermore, $D_1(\alpha)$ is the strong confluence of $\rightarrow_0$. At higher layers, $D_1(\alpha)$ is a strictly weaker proof obligation than the strong confluence of $\rightarrow_\alpha$: one can use additional reductions at lower layers. For D_2 , one can think of it as a weak form of commutativity of α steps with steps at lower layers.

Further remark that we made a minor adjustment to de Bruijn's original statement and proof: in both D_1 and D_2 , we allow for the bottom of the diagram to stutter by considering the reflexive closure $\rightarrow_\alpha^?$. This technical modification turns out to be convenient in practice in order to conclude when the considered peak is “fake” in the sense that $b = c$ (especially when $b \not\rightarrow$), or when it can be made equal by applying rules living at lower levels.

Proof structure. Endrullis and Klop describe de Bruijn's proof as “truly amazing”; indeed, the proof consists of a quite beautiful and miraculous interlocking of diagrams, depicted in Fig. 6 (our modified reflexive closures are marked in red).

The top of the figure shows the hypotheses of Theorem 3.4, D_1 and D_2 , the remaining intermediate diagrams are introduced during the proof. Arrows informally trace dependencies between diagrams: these are successively proven by reduction to a patchwork of previously established diagrams, starting from the assumptions over D_1 and D_2 . We define $CR(m)$ to mean the indexed ARS is confluent (i.e., satisfies Church Rosser) at level m , and $CR^*(m)$ to mean it is confluent at all levels strictly below m (i.e. $CR(k)$ for all $k < m$).

At the two extremities of this dependency graph is Church Rosser: more precisely, one proceeds by induction over m to establish Church Rosser up to level m ; i.e., assuming $CR^*(m)$, we prove $CR(m)$.

Adding optional steps. We refer to these variants with the suffix “opt” as in $D1opt$. $D1opt$ accounts for fake peaks. The other optional relation is in D_2 ; in fact, $D2opt$ has been used instead of D_2 several times in [19], but not in the main proof.

We show here that the proof can be adapted to these optional steps. The adaptation is non-trivial, as the optional steps in commutation diagrams add a lot of cases, but does not change the overall structure of the proof. Note that a few intermediate lemmas need more prerequisites, especially a stronger induction. Indeed we need at several places to use commuting diagrams for smaller indices to deal with the cases when the optional step does not occur. For example, we adapted the following lemma from [19]:

Lemma 3.9: Let $m \in I$ and assume $D7(m, k)$ for all $k < m$. Then we have $CR(m)$.

into a version that requires $CR^*(m)$ and $D5$ because of the optional steps:

Lemma 3.5 ($D7opt_Implies_CR$, ) .

$\forall k, k < m \wedge D7opt(m, k) \wedge D5opt(m) \wedge CR^*(m) \implies CR(m)$.

Appendix A illustrates one intricate step of the proof: the proof of the lemma that allows us to obtain $D7$ from other diagrams. It also illustrates how the optional steps require additional hypotheses in the lemmas, and is used.

Recovering the original theorem. As the original theorem is a particular case of the optional variant, we also prove that if $D1$ and $D2$ hold, then we also have confluence. In practice we derive de Bruijn's confluence theorem from the one with optional steps, but we also did the de Bruijn proof without the optional step directly () .

About the Rocq formalisation. The proof of the theorem has been fully formalised in Rocq, we highlight here a few points of interest.

Bugs. We identified a couple of minor mistakes in the original proof, most of them inconsequential. The main surprising mistake was this statement:

$$(\rightarrow_{\leq m})^* = (\rightarrow_{< m}^* \rightarrow_m \rightarrow_{< m}^*)$$

which does not hold if there is no reduction of level exactly m in the reduction on the left. Fortunately, the proofs could be adapted with a case analysis.

Clarified case distinction. Many proofs require additional inductions than the original paper proofs (typically on the length of a $\rightarrow^*$), the Rocq proof clarifies such details.

Direct induction. We have reworked the proof of Theorem 3.4 by reformulating the original proof by contradiction on the lowest level not satisfying confluence into a direct constructive proof by induction.

4 λ_{act} : A Minimal Actor Language

This section introduces λ_{act} , a minimal actor language designed to illustrate challenges that arise when proving the confluence of communicating pure entities with dependencies and FIFO service of requests, as discussed in Sec. 2.1. Due to its minimalism, λ_{act} misses many features of real actor languages: we discuss possible extensions in Sec. 6.1.

We first introduce the syntax of λ_{act} , then its semantics in two layers: a labelled transition system for thread-local computations, and a global transition system that synchronises threads on the emitted labels.

4.1 Syntax

Figure 7 introduces the syntax of λ_{act} . Commands ($c \in \text{Comm}$) are written in a standard, imperative core supporting iteration, sequence, and assignment. The particularity of the language stands on the right hand side of the assignment: additionally to evaluating pure computations, one may call t, m, v in order to delegate the execution of method $m(v)$ to thread t . The caller moves into a waiting state $\text{wait}(t)$ (in blue to emphasise this extended syntax is not part of the source) until it receives the result of the computation.

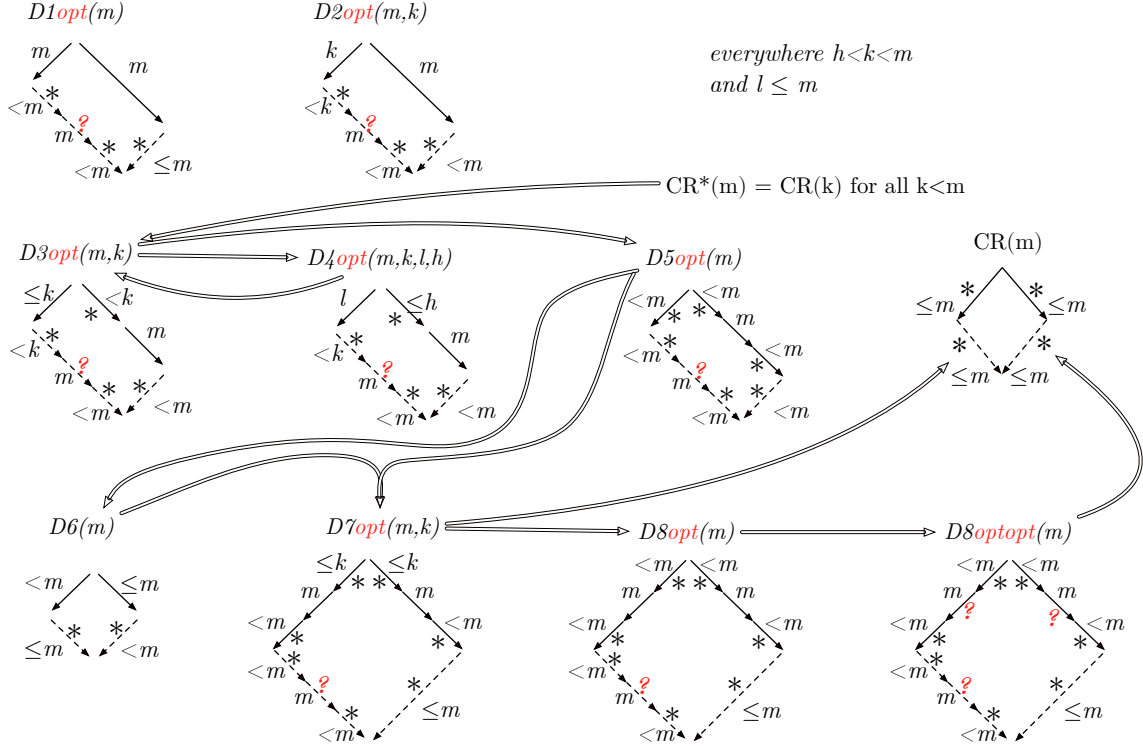


Figure 6. The diagrams of de Bruijn's proof; compared to [19] some steps are now optional (marked in red).

$$\begin{aligned}
 & x, x_l, x_g \in \text{Var} \quad t \in \text{Tid} \\
 & v \in \mathcal{V} ::= t \mid n \mid b \\
 & e \in \text{Expr} ::= x_l \mid x_g \mid v \mid op \ e \mid e \ op \ e \\
 & r \in \text{Rhs} ::= e \mid \text{call}(e, m, e) \mid \text{wait}(t) \\
 & c \in \text{Comm} ::= \text{skip} \mid x := r \mid c; c \mid \text{while } e \text{ do } c \\
 \\
 & C \in \mathcal{C} ::= \mathcal{M} \rightarrow (\text{Comm} \times \text{Var} \times \text{Expr}) \\
 & \rho \in \text{Store} ::= \text{Var} \rightarrow \mathcal{V} \\
 & ps \in \text{Stores} ::= \{L : \text{Store}; G : \text{Store}\} \\
 & \sigma \in \text{State} ::= \{e : \text{Comm}; ps : \text{Stores}\} \\
 & \text{tsk} \in \text{Task} ::= (\text{body} : \text{Comm}, \rho_l : \text{Store}, \text{ret} : \text{Expr}, \text{src} : \text{option Tid}) \\
 & a \in \text{Act} ::= \text{Idle} \mid \text{Running tsk} : \text{Task} \\
 & q \in \text{Queue} ::= \text{list Task} \\
 & \delta \in \text{Thread} ::= \{q : \text{Queue}; a : \text{Act}; \rho_g : \text{Store}\} \\
 & \Sigma \in \text{Conf} ::= \text{Tid} \rightarrow \text{Thread}
 \end{aligned}$$

Figure 7. Static and dynamic configurations for λ_{act} (🔴).

At runtime, a *task* ($\text{tsk} \in \text{Task}$) is made of a body under execution, a local store, the final return expression, and the id of the thread that has delegated this task—or None for initial threads, whose task has no source. A thread ($\delta \in \text{Thread}$) consists of three components. Firstly, it has typically a task a under focus—its activity is *Running*—unless it is waiting to serve a new one—its activity is *Idle*. Secondly, it may have been given other tasks: the ones not under focus are stored

in a FIFO queue q . Thirdly, it carries a global store ρ_g , which is not reset between task. Finally, a runtime configuration is a finite map of threads, indexed by their id.

An initial program is therefore embedded into a configuration made of threads, some of them initialised to an initial command, the others left *Idle*. All the queues are empty. We say that a thread is a server, and set $\text{isServer}(i)$ to true for its thread id, if it has no initial command. We define an initial global store for each thread.

4.2 Semantics

We equip λ_{act} with a small-step operational semantics split into thread-local rules and global synchronisation rules.

Thread-local reduction is defined (Fig. 8) as a labelled transition system between local configurations containing a command, and the local and global stores. We omit the label for silent steps, but note that the meta-variable l can range over such silent steps. Rules for sequence, iteration, and assignment of pure expressions are standard. An assignment of a delegated computation via a call reduces to one in a waiting state $\text{wait}(t)$ where t is the identity of the thread executing the computation; this reduction furthermore exhibits an emission label $!(v, m, t)$ registering that a request for executing $m(v)$ has been sent to t . Finally, a waiting state $\text{wait}(t)$ can be unlocked to a pure assignment to value v , exhibiting a label $?(t, v)$ that synchronises the received value

$$\begin{array}{c}
\frac{\llbracket e \rrbracket_{(\rho_l, \rho_g)} = v}{(x_l := e, (\rho_l, \rho_g)) \rightarrow (\text{skip}, (\rho_l[x_l \leftarrow v], \rho_g))} \quad \frac{\llbracket e \rrbracket_{(\rho_l, \rho_g)} = v}{(x_g := e, (\rho_l, \rho_g)) \rightarrow (\text{skip}, (\rho_l, \rho_g[x_g \leftarrow v]))} \quad \frac{(c_1, \rho_s) \xrightarrow{l} (c'_1, \rho_s')}{(c_1; c_2, \rho_s) \xrightarrow{l} (c'_1; c_2, \rho_s')} \\
\\
\frac{}{(\text{skip}; c_2, \rho_s) \rightarrow (c_2, \rho_s)} \quad \frac{\llbracket e \rrbracket_{(\rho_s)} = \text{true}}{(\text{while } e \text{ do } c, \rho_s) \rightarrow (c; \text{while } e \text{ do } c, \rho_s)} \quad \frac{\llbracket e \rrbracket_{(\rho_s)} = \text{false}}{(\text{while } e \text{ do } c, \rho_s) \rightarrow (\text{skip}, \rho_s)} \\
\\
\frac{\llbracket e_t \rrbracket_{(\rho_s)} = t \quad \llbracket e_v \rrbracket_{(\rho_s)} = v}{(x := \text{call}(e_t, m, e_v), \rho_s) \xrightarrow{!(v, m, t)} (x := \text{wait}(t), \rho_s)} \quad \frac{}{(x := \text{wait}(t), \rho_s) \xrightarrow{?(t, v)} (x := v, \rho_s)}
\end{array}$$

Figure 8. Dynamics: process-local rules (🚗).

$$\begin{aligned}
\Sigma(i)^s &= (c, (\rho_l, \rho_g)), \text{ if } \Sigma(i) = \{q, \text{Running}(c, \rho_l, e, \text{opt}), \rho_g\} \\
\Sigma(i)^q &= q, \text{ if } \Sigma(i) = \{q, a, \rho_g\} \\
\Sigma[i \stackrel{s}{\leftarrow} (c, (\rho_l, \rho_g))] &= \Sigma[i \leftarrow \{q, \text{Running}(c, \rho_l, e, \text{opt}), \rho_g\}], \\
&\quad \text{if } \Sigma(i) = \{q, \text{Running}(c', \rho_l', e, \text{opt}), \rho_g'\} \\
\Sigma[i \stackrel{q}{\leftarrow} q] &= \Sigma[i \leftarrow \{q, a, \rho_g\}], \text{ if } \Sigma(i) = \{q', a, \rho_g\} \\
\Sigma[i \stackrel{a}{\leftarrow} a] &= \Sigma[i \leftarrow \{q, a, \rho_g\}], \text{ if } \Sigma(i) = \{q, a', \rho_g\}
\end{aligned}$$

Figure 9. Getter and setter notation for threads.

with the thread that replies with this value. These labels are used to synchronise within the global semantics.

Before defining the global semantics, we introduce some notation for ease of presentation. Firstly, the getter notations $\Sigma(i)^s$, $\Sigma(i)^q$ retrieves the local state (command and local/global stores) and the queue of thread i , respectively. Secondly, the setter notations $\Sigma[i \stackrel{s}{\leftarrow} \sigma]$, $\Sigma[i \stackrel{q}{\leftarrow} q]$, $\Sigma[i \stackrel{a}{\leftarrow} a]$ update the local state, queue, and activity of thread i , respectively, with the values σ , q , and a . Formal definitions of both getter and setter notations can be found in Fig. 9.

The global semantics (Fig. 10) comprises four rules. Rule **Loc** allows any thread to perform a local computation in its focused task, where locality is characterised by the absence of a label on the thread-local reduction considered. Rule **CALL** performs a call if the local computation at thread i emits $!(v, m, j)$. The local state of i is updated, and a new task is added to the queue of callee thread j . Note that this new task comes with a local state mapping x to the value emitted at i . Conversely, rule **GET** synchronises the return of values. If thread i has completed a task destined for thread j and $\Sigma(j)^s$ can progress by $?(i, v)$, then i is set to **Idle** and j takes its receive step. Finally, a thread in an **Idle** state can pop its queue to put a new task under focus using rule **SERVE**.

Each of these transitions emits a label identifying the processes involved. We use these labels in Sec. 5 to create an indexed ARS. Note that these global labels have no semantics—we can safely erase them to obtain an unlabelled semantics.

$$\begin{array}{c}
\text{Loc} \\
\frac{\Sigma(i)^s \rightarrow \sigma'}{\Sigma \xrightarrow{\text{Loc}(i)} \Sigma[i \stackrel{s}{\leftarrow} \sigma']} \\
\\
\text{CALL} \\
\frac{\Sigma(i)^s \xrightarrow{!(v, m, j)} \sigma'_i \quad C(m) = (c, x, e)}{\Sigma \xrightarrow{\text{Call}(i, j)} \Sigma[i \stackrel{s}{\leftarrow} \sigma'_i][j \stackrel{q}{\leftarrow} \Sigma(j)^q :: (c, \{x \mapsto v\}, e, i)]} \\
\\
\text{GET} \\
\frac{\Sigma(i) = \{q, \text{Running}(\text{skip}, \rho_l^i, e_i, \text{Some } j), \rho_g^i\} \quad \llbracket e_i \rrbracket_{(\rho_l^i, \rho_g^i)} = v \quad \Sigma(j)^s \xrightarrow{?(i, v)} \sigma'_j}{\Sigma \xrightarrow{\text{Get}(j, i)} \Sigma[i \stackrel{a}{\leftarrow} \text{Idle}][j \stackrel{s}{\leftarrow} \sigma'_j]} \\
\\
\text{SERVE} \\
\frac{\Sigma(i) = \{tsk :: q, \text{Idle}, \rho_g\}}{\Sigma \xrightarrow{\text{Pop}(i)} \Sigma[i \leftarrow \{q, \text{Running } tsk, \rho_g\}]}
\end{array}$$

Figure 10. Dynamics: interleaving semantics (🚗).

4.3 Generic Properties of λ_{act}

We collect below generic properties of λ_{act} that are useful for the proof of confluence introduced in Sec. 5.

We first define two relations between threads and tasks. We say that a *thread t has a task for thread i* if there is a task with destination i in t 's queue, or being processed by t . We say that *thread i is waiting for thread t* if the next local step of i consists of performing a get operation from t .

Lemma 4.1 (Queue invariant, 🚗). *Thread i is waiting for thread j if and only if thread j has a task for thread i . Thus, if a thread is not waiting, then no thread has a task for it.*

Proof. We show that the invariant holds for all states reachable from an initial well-formed state, by induction on the reduction. The proof is not difficult, but relies on a large number of stability results. $\square$

Secondly, our language is inspired by actor-like languages and as such each thread has a single current task, and thus

features local determinacy. In practice, determinacy comprises a collection of lemmas that, taken together, show that except the value fetched by a get operation, the local behaviour is fully deterministic. As an illustrative example, consider the following state determinacy lemma.

Lemma 4.2 (Local determinism, 🚫). *Local reduction is a deterministic LTS.*

$$\sigma \xrightarrow{l} \sigma' \wedge \sigma \xrightarrow{l} \sigma'' \implies \sigma' = \sigma''$$

Finally, we characterise when certain reductions may commute with other operations, particularly with operations that manipulate the queue of tasks. This theorem will allow us to prove commutativity between certain reduction steps when proving confluence. We illustrate one concrete example, and will refer them as needed in Sec. 5.

Lemma 4.3 (Step and push, 🚫). *If a configuration can make a non-call step leading to a configuration in which a thread has a non-empty queue, then it could also take this reduction if we pop the top task from this queue, i.e.:*

$$\begin{aligned} \Sigma_q \xrightarrow{l} \Sigma' [i \stackrel{o}{\leftarrow} \Sigma' (i)^o :: \text{tsk}] \wedge l \text{ is not call} \\ \implies \exists \Sigma, \left(\Sigma \xrightarrow{l} \Sigma' \wedge \Sigma_q = \Sigma [i \stackrel{o}{\leftarrow} \Sigma (i)^o :: \text{tsk}] \right) \end{aligned}$$

We are now ready to turn to the proof of confluence.

5 On the Confluence of λ_{act}

In this section we prove confluence results for the two classes of λ_{act} programs that have been identified in Sec. 2.2. We first define some hypotheses that are required for both classes of programs, then for each class of problems we construct an appropriately indexed ARS, establish some key lemmas, and finally apply de Bruijn's theorem by constructing the necessary diagrams. Appendix B provides a few technical details on the Rocq formalisation of the confluence proofs.

5.1 Setting up the Proofs

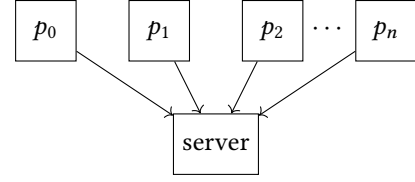
Call graph. We use call graphs to reason on the communication patterns between threads and identify the classes of programs we want to prove confluent. Call graphs can be safely approximated by static means, so this is a constraint that we can reasonably check.

Definition 5.1 (Call graph, 🚫). Given an initial state Σ_0 , the call graph G_{Σ_0} is the directed graph with vertices $\text{dom}(\Sigma_0)$ — the process ids — and if there exists an execution from Σ_0 in which i makes a call to j then there is an edge (i, j) .

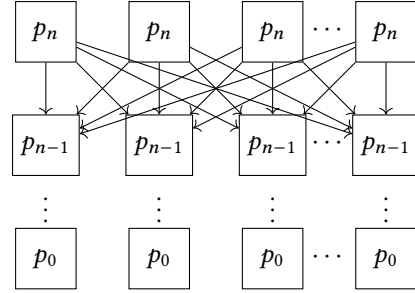
Firstly, we consider programs in which a single server processes tasks for a number of clients.

Hypothesis 1 (Single server, 🚫). *In this scenario, the clients may not communicate between each other, and the server may not make further calls to compute its results. This is formalised as follows: There is a single thread with id ts such that*

$$(i, j) \in G_{\Sigma_0} \implies j = ts \quad \wedge \quad \text{isServer}(ts).$$



(a) Single server call graph



(b) n -layered call graph

Figure 11. Classes of λ_{act} programs

Hypothesis 1 results in a call graph such as the one shown in Fig. 11a. These requirements can be statically enforced by designating one initially idle node as the server, making sure that only this node is called by initial computations, and ensuring that no task contain further method calls.

Secondly, we consider a generalisation in which each process is given a *level* and may only make calls to processes of strictly lower levels.

Hypothesis 2 (Layered call graph, 🚫). *In this scenario, a set of clients send tasks to any number of servers, which may make further calls to complete their tasks, but there can be no cycle in the call graph of servers. This is formalised as follows: there is a function *layer* and an integer *top* such that*

$$\begin{aligned} ((i, j) \in G_{\Sigma_0} \implies \text{layer}(j) < \text{layer}(i)) \\ \wedge (\text{layer}(i) < \text{top} \implies \text{isServer}(i)). \end{aligned}$$

Hypothesis 2 results in a call graph such as the one shown in Fig. 11b, possibly with fewer edges and vertices. This means we can associate a “level” to each thread so that each client is at the maximal level, and each call is made towards a server of a lower level.

In both cases, there is some non-determinism in the order in which the tasks are processed by the servers. Since the tasks may be stateful, the order could change the answers returned to the clients — consider for example a get and increment method on the same variable. We thus further constrain the programs with a purity requirement.

Purity. In principle, each method may arbitrarily change the server state, making any sort of confluence untenable. We could consider only completely pure methods that never assign a variable, but that would be very restrictive. Instead

we consider methods that may utilise the server state, but always clean up after themselves. For methods that make no calls, purity is directly expressed by taking the transitive closure of local rules. This is sufficient for the single server case (as the server never calls another actor).

Hypothesis 3 (Pure methods, ) *In this scenario, all methods in a class C reduce with a finite sequence of local rules, after these steps, the global state of the node is the same as at the beginning of the method. Formally, if $C(m) = (c, x, e)$, then for any local state $(c, (\rho_l, \rho_g))$ there exists a store ρ'_l such that $(c, (\rho_l, \rho_g)) \rightarrow^* (\text{skip}, (\rho'_l, \rho_g))$.*

This hypothesis also ensures that all methods terminate. Note that these methods can utilise local variables that are not constrained by purity to compute a result or modify global variables and restore them afterwards.

In the case of the n -layered call graph, this hypothesis needs to be generalised as the server makes non-local steps. We instead require that for all calls and replies that the server is involved in, it always reaches the same global state. This cannot be simply expressed by reachability of a final step as several possible executions (depending on the replies received) must be considered. The quantification on traces is thus a bit tricky; we express it as an inductive definition that takes as parameter the target global state that must be reached by the reduction when the task is finished.

Hypothesis 4 (Pure methods with calls, ) *We define a predicate bigstep inductively as follows*

- if the computation is finished, we reached the target global state: $\text{bigstep}(\text{skip}, (\rho_l, \rho_g)) = \rho_g$.
- if we do a local step, it reduces to a state that will reach the target global state:

$$\begin{aligned} \text{bigstep}(c', (\rho'_l, \rho'_g)) &= \rho''_g \wedge (c, (\rho_l, \rho_g)) \rightarrow (c', (\rho'_l, \rho'_g)) \\ &\implies \text{bigstep}(c, (\rho_l, \rho_g)) = \rho''_g \end{aligned}$$

- if we perform a call, it will be followed by a get (by definition of the semantics) and whatever value is obtained by the get we reduce to a state that will reach the target global state.

$$\begin{aligned} \text{bigstep}(c'', (\rho_l^2, \rho_g^2)) &= \rho''_g \wedge \\ (c, (\rho_l, \rho_g)) &\xrightarrow{\text{Call}(i,j)} (c', (\rho_l^1, \rho_g^1)) \wedge \\ (c', (\rho_l^1, \rho_g^1)) &\xrightarrow{\text{Get}(i,j)} (c'', (\rho_l^2, \rho_g^2)) \\ &\implies \text{bigstep}(c, (\rho_l, \rho_g)) = \rho''_g \end{aligned}$$

All methods in C verify the above predicate with the target global state equal to the initial one. If $C(m) = (c, x, e)$, then for any local state $(c, (\rho_l, \rho_g))$, we have $\text{bigstep}(c, (\rho_l, \rho_g)) = \rho_g$.

Note that this allows the function to return a different result depending on the value returned by a called function but not to store this value in its global state.

Assuming both call graph and purity hypotheses we can prove the two confluence theorems, solving the two problems presented in Sec. 2.2.

Confluence theorems for λ_{act} . In the rest of this section we prove confluence for the two classes of programs.

Theorem 5.2 (Confluence). *Provided all the methods that are called are pure, λ_{act} -programs with call graphs like the ones depicted in Fig. 11 are confluent.*

The proof employs de Bruijn's theorem. We detail the proof of the single server case before discussing the generalisations necessary for the n -layered case.

To apply de Bruijn's theorem, we need an indexed ARS. Given a labelled transition system we equip it with an order $(I, <)$ by defining a function $\mathcal{L}$ from labels to I , giving a step labelled by l the index $\mathcal{L}(l)$. In our case we assign an index to the labels of transitions in Fig. 10, defining an indexed ARS for which we prove de Bruijn's commuting diagrams.

5.2 Problem 1: Determinacy for Single Server

Single server programs have call graphs like the one in Fig. 11a. We define an ARS for them as follows. Note that, because of the single server hypothesis all calls are made to the server and all results are returned from the server.

Definition 5.3 ($\mathcal{S}_{\text{ss}}$, ) *Given a dedicated server node s , the two-level ARS $\mathcal{S}_{\text{ss}}$ is λ_{act} equipped with the order $0 < 1$ and assigning a level to each transition as follows:*

$$\mathcal{L}(\text{Get}(i, j)) = 0 \quad \mathcal{L}(\text{Call}(i, j)) = 1 \quad \mathcal{L}(\text{Pop}(i)) = 0$$

$$\mathcal{L}(\text{Loc}(i)) = \begin{cases} 0, & \text{if } i = s \\ 1, & \text{otherwise} \end{cases}$$

Before applying de Bruijn's proof, we make two important observations about this setup. First, Lemma 4.1 implies that, if a thread is not waiting, there is no task for it in the server. Second, it is always possible to reach a state where the server is idle and its queue is empty. In fact we need a variant of this property that leaves two requests in the queue, as follows:

Lemma 5.4 (Flushing, ) *Consider the configuration where the server's queue is $Q :: \text{tsk1} :: \text{tsk2}$, it is always possible to reach a state where the server is idle and its queue is reduced to $\text{tsk1} :: \text{tsk2}$. This flushing does not change any non-waiting threads. Furthermore, the similar configuration where the queue is $Q :: \text{tsk2} :: \text{tsk1}$ reduces to the same state except the remaining queue is $\text{tsk2} :: \text{tsk1}$.*

This lemma is necessary in the case where two tasks have been triggered concurrently. Hence:

Lemma 5.5 () *The D_1 diagrams for $\mathcal{S}_{\text{ss}}$ can all be closed.*

Proof. Since there are two levels, we need $D(0)$ and $D(1)$.

At level 0 things are (relatively) simple and we close all the diagrams by determinism of the local semantics (the proof indirectly relies on Lemma 4.2 and other similar lemmas).

At level 1, the critical pair is CALL/CALL. We use our two observations to close it by:

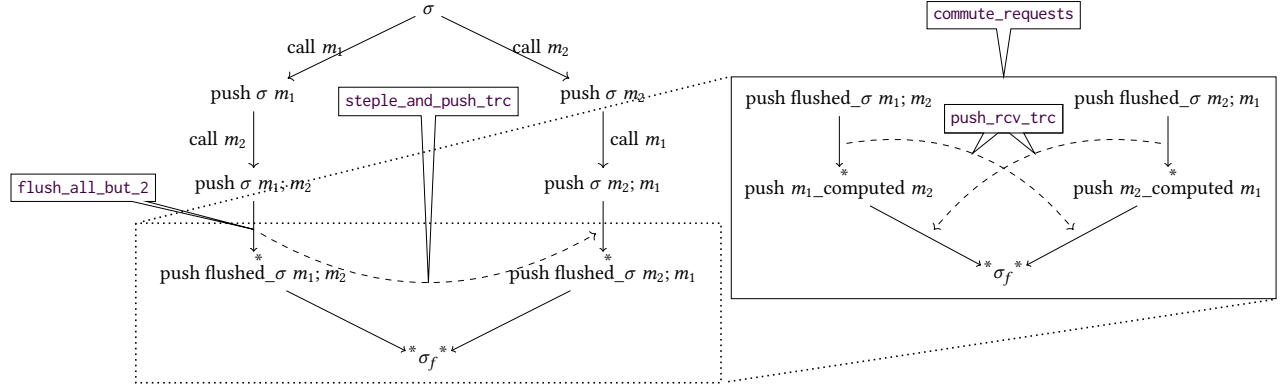


Figure 12. Overview of the proof

1. flushing the server,
2. processing the received task and return the result,
3. receiving the second task (it takes some work to show that we can still receive it after processing the first),
4. processing the second task and returning the result.

It remains to show that the final configurations are the same. This basically follows from Hyp. 3. $\square$

Lemma 5.6 (🚗). *The D_2 diagrams for $\mathcal{S}_{ss}$ can all be closed.*

Proof. Straightforward, since only one edge can be a call. $\square$

From D_1 and D_2 diagrams, we can apply de Bruijn’s theorem:

Theorem 5.7 (🚗). $\mathcal{S}_{ss}$ is confluent.

5.3 Problem 2: Determinacy for Layered Servers

We now study a more complex family of programs. We consider a top layer of clients sending tasks to an arbitrarily deep connected structure of servers as illustrated in Fig. 11b.

Definition 5.8 ($\mathcal{S}_{\text{layered}}$, 🚗). Given a *depth* n , the semantics of programs satisfying Hyp. 2 are defined by first assigning each node a level using $\text{lvl} : \text{Tid} \rightarrow \{0..n\}$; and then creating an iARS by assigning the following level to each transition, depending on its label:

$$\mathcal{L}(\text{Get}(i, j)) = \text{lvl}(j) \quad \mathcal{L}(\text{Call}(i, j)) = \text{lvl}(i)$$

$$\mathcal{L}(\text{Pop}(i)) = \text{lvl}(i) \quad \mathcal{L}(\text{Loc}(i)) = \text{lvl}(i)$$

Additionally we require that each node i whose level is strictly below n be initially empty and idle.

The principle of the proof is the same as before: we show that all diagrams of the forms $D_1(n)$ and $D_2(n, k)$ with $k < n$ can be closed by considering each case. Once again the most interesting case is CALL/CALL which we close by, (1) receiving the second task; (2) flushing the task queue of every thread below the callee in the diagram, plus flushing the callee apart from the two tasks; and (3) processing both tasks. We perform these steps on the two cases, depending on which call is performed first and must guarantee that the steps are

done similarly to ensure that we reach the same final state. Purity of methods (Hyp. 4) plays of course a crucial role in these proofs. The CALL/CALL commutation is illustrated in Fig. 12. Step (1) is trivial, we review the other steps below.

The theorems detailed below are only valid under the constraint that the involved configurations are reachable from the initial state, ensuring their well-formedness and the fact that methods are pure and the call graph is layered.

Compared to Problem 1, flushing, i.e., step (2), is more difficult since the callee may make arbitrarily many calls of its own. Flushing here means picking a level and constraining the execution until every process *below* it is empty and idle. This gives us a “clean” initial state for further computations.

Lemma 5.9 (n -layer flushing, 🚗). *Given a level $k < n$ and a configuration Σ , there exists a configuration Σ' such that $\Sigma \rightarrow^* \Sigma'$ and each thread i with $\text{lvl}(i) \leq k$ is idle and has an empty queue.*

Proof. By induction on k . For $k = 0$, it suffices to show that an arbitrary process of level 0 can become idle and empty. Since the level is smaller than n , the process must be a server and any ongoing work is the result of a method call. If the process is working on a task, the task is terminating by Hyp. 4. On level 0 there can be no calls, so the computation reaches skip with only local steps. We then return the answer to the waiting process, which is guaranteed to exist by Lemma 4.1. If the process is idle but its queue is non-empty, we pop the top task and proceed as before.

The inductive case is similar, where the computation may now make calls of its own: the induction hypothesis guarantees that the callee can be flushed and return an answer. $\square$

A variant of the lemma allows us to flush all the threads except two tasks of a given thread (🚗). Indeed, step (2) of the main proof requires to flush everything except the two tasks concerned by the CALL/CALL conflict. After flushing one side, we use the following lemma to ensure that the other side of the conflict can be flushed the same way.

Lemma 5.10 (Queue order invariance, 🍷). *Consider two well-formed configurations Σ, Σ' , a thread identifier i and tasks tsk_1, tsk_2 such that the configurations differ only in the order the tasks are pushed to i , i.e., $\Sigma(i)^Q = q :: tsk_1 :: tsk_2$ and $\Sigma'(i)^Q = q :: tsk_2 :: tsk_1$. If there exists an execution $\Sigma \xrightarrow{*}_{\leq \text{lvl}(i)} \Sigma_f$ with $\Sigma_f(i)^Q = q :: tsk_1 :: tsk_2$, then there is also an execution $\Sigma' \xrightarrow{*}_{\leq \text{lvl}(i)} \Sigma'_f$ such that Σ_f, Σ'_f differ only in the order of the tasks pushed to i , i.e., $\Sigma'_f(i)^Q = q :: tsk_2 :: tsk_1$.*

Proof. By induction on the execution. The proof mostly relies on Lemma 4.3, lifted to the iARS $\mathcal{S}_{\text{layered}}$ instead of the original reduction and to two tasks at the same time. $\square$

Having flushed every task before the two conflicting tasks and any process below them, we can show that the two methods left in the queue commute, corresponding to step (3). This important result is captured by the following lemma.

Lemma 5.11 (Requests commute, 🍷). *Given two well-formed configuration Σ, Σ' and tasks tsk_1, tsk_2 with $\text{src}(tsk_1) = j_1, \text{src}(tsk_2) = j_2$ such that process i is idle in both, $\text{queue}(\Sigma(i)) = [tsk_1, tsk_2]$, $\text{queue}(\Sigma'(i)) = [tsk_2, tsk_1]$, and both configurations are flushed below $\text{lvl}(i)$; there exists a configuration Σ_f such that $\Sigma \xrightarrow{*} \Sigma_f, \Sigma \xrightarrow{*} \Sigma_f, \Sigma_f(i)$ is idle and empty, and j_1, j_2 have received answers in Σ_f .*

Proof. By Lemma 4.1 we know that $j_1, j_2 \in \text{dom}(\Sigma)$ and both are waiting, and by Hyp. 4, $\text{body}(tsk_1)$ and $\text{body}(tsk_2)$ can be computed without changing the state of i . Thus we can pop the tasks in order, compute a return value for both tasks and return them to j_1, j_2 .

However, to guarantee that the two executions are indeed the same we need to prove commutativity of the whole request computation. Because of purity, we observe that the only difference in the configuration state between before and after the treatment of a task is one value answered and one less request in the queue, we thus rely on the following additional lemma to ensure that task treatment commute.

Lemma 5.12 (Push-receive invariance, 🍷). *Consider three reachable configurations Σ_t, Σ_r and Σ_l so that there is an execution from Σ_t to Σ_r at the level of process i (processing m_1). Suppose additionally that Σ_l is Σ_t where one task has been popped from the queue of process i and a reply has been sent from process i to a process k (of higher level), i.e., another method m_2 has been processed when going from Σ_t to Σ_l . Then, there is an execution from Σ_l to Σ_b at the level of process i (processing m_1) where Σ_b is Σ_r with one request less in the queue of process i and a reply sent from process i to process k .*

This lemma follows from the purity of methods (Hyp. 4). By applying it twice, we prove that a configuration exists that is reachable from both sides of the conflicting calls. $\square$

We now prove confluence for all systems of layered servers.

Theorem 5.13 (🍷). $\mathcal{S}_{\text{layered}}$ is confluent.

Checking the layering statically. We have encoded the motivating example from Fig. 2 in λ_{act} (🍷) and proved it confluent by applying Theorem 5.13. To discharge the theorem's hypothesis, we need in particular to index the threads involved and prove that the call graph is layered w.r.t. this index. However, the call graph is a semantic notion, quantifying over all reachable state. We hence provide a simple illustrative static analysis to over-approximate it (🍷).

The analysis assumes that all work is delegated to constant thread identifiers, otherwise it fails. To avoid having to compute a fixpoint, it takes as argument a user-provided oracle mapping each thread i to a set of methods over-approximating those that i may have to process during the execution. It then computes an over-approximation of the call graph, as captured by its soundness theorem (🍷). In particular, checking that the result of the analysis is layered is sufficient to prove that the call graph is layered (🍷). Finally, we prove confluence for the example in Fig. 2 (🍷).

6 Discussion and Related Work

In this section we briefly discuss language features that are missing for a fully-fledged language, and some related work.

6.1 Language Extensions

λ_{act} exhibits a small actor language core to facilitate the exposition of confluence proofs. We here consider three natural extensions, and the challenges they pose for our results.

Thread creation: It is straightforward to extend λ_{act} with thread creation, but the confluence theorems would require to reason on an abstraction of runtime threads which is not the scope of this paper; this is a problem of static analysis of processes, not of confluence.

Futures: To have more asynchrony and parallelism, we could use futures instead of actively waiting on calls. Such an extension however complicates the necessary notion of call-graph, and constitutes future work.

Confluence modulo equivalence: The notion of confluence we establish ensures that the diagrams are closed w.r.t. equal configurations, which may break for instance in presence of the generation of fresh identifiers. We have extended our results to work w.r.t. an arbitrary equivalence relation, but did not need it for λ_{act} .

6.2 Related Work

Determinism in programming languages has been studied previously. For example, Kahn process networks [34] already featured deterministic behaviour in a concurrent setting, with well identified computing entities. Several works try to control determinism to identify which parts of a program are deterministic or not. A recent survey [24] of different language-based techniques overviews *partially deterministic* languages that behave deterministically under given conditions, or for which sufficient properties have been formally

identified to ensure deterministic behaviour. Characterisations of such deterministic features are useful in many settings; for example, they identify when a recorded trace can be used to replay executions from minimally recorded data, e.g., for fault tolerance or debugging [8, 13, 43].

One particular application area for (partially) deterministic properties is actors and active objects [2, 7, 15]. Actors are single-threaded entities that communicate with others by asynchronous message sending. Actors access distinct memory locations, which prevents data races. The only reason an actor's internal behaviour may not be deterministic is due to message ordering (e.g., conflicting messages or non-FIFO mailbox usage) or when a time-dependent condition, such as the presence of a message in a queue or the end of a computation, is checked. Determinism in this setting was first formalised in the ASP (Asynchronous Sequential Processes) language [10, 11], and later revisited [29] to account for cooperative scheduling and provide a type system that ensures determinism. Lohstroh and Lee [39] proposed a deterministic actor language. The key ingredient for determinism is the *logical timing* of messages based on a protocol which combines physical and logical timing. The solution relies on messages with timestamps and providing a deterministic order for identical timestamps. We are not aware of a formal proof of correctness of the scheduling protocol; in fact, formalising conditions of confluence in this setting could allow for more asynchrony or stronger scheduling results. In the original Actor paradigms [2, 7] imperative aspects were provided as a *become* statement that changes the reaction to messages. While identifying purity based on the *become* statement can be specified, both informal and static identification of purity for such actors seems difficult. The other actor features are *creation*, discussed above, and *sending messages*, which is at the core of our model.

Partial order reduction (POR) [23] reasons about concurrent programs by a *reduction* based on *commuting* steps. The aim is to identify classes of equivalent interleavings up to commutativity, and then only consider one representative of each class. POR for actors has been extensively studied for program testing (e.g., [3, 42]). While both POR and our approach identify pairs of commuting steps to enable simpler analysis of concurrent programs, both the methods and goals differ. Research on POR focuses on providing algorithms to reduce the state space for different analysis tasks, while we seek to provide mechanised proofs of completely deterministic behaviour for constrained classes of programs. Recently, Farzan et al. [20] introduced *stratified* commutativity, in which commutativity relations are layered. Despite a resemblance to our methods, their approach indexes *commutativity relations* rather than steps of program behaviour.

The IsaFoR² library contains Isabelle/HOL formalisations of several confluence criteria in the setting of term rewriting

systems. Kohl and Middeldorp [35] formalise van Oostrom's development-closed criterion [45], while Hirokawa et al. [32] tackle a range of criteria focused on parallel critical pairs, including decreasing diagrams [44]. Like us, they identify and remedy small oversights in the paper proofs, highlighting the value of formalising such intricate results. De Bruijn's theorem is not among the ones formalised in Isabelle/HOL.

7 Conclusion

This paper proposes a novel proof technique to prove confluence of parallel programs, which allows us to prove confluence for parallel systems where other approaches fall short. Our method lifts a theorem developed by de Bruijn for abstract rewriting systems to programs in modern actor-based languages. We formalised de Bruijn's theorem in Rocq, resolving minor bugs in the original paper proof en passant. We further formalised the proposed proof technique for actor systems and demonstrated its use on two families of systems characterised by their call graphs and on a concrete example, which shows the practical applicability of this technique. It is highly interesting to extend the proposed proof technique to further language features and classes of programs.

8 Data-Availability Statement

The artefact for this submission is available on Zenodo [30].

A Details on the Weak Confluence Theorem: How to Obtain D7opt

We focus here on one particular lemma that states how to obtain D7. It is one of the most complex proofs of the weak confluence theorem. The former statement for this theorem, from [19], is the following:

Lemma 3.8: Let $m, k \in I$ with $k < m$. Assume $CR \star (m)$, $D5(m)$, $D6(m)$ and $D3(m, h)$ for all $h < m$. Then we have $D7(m, k)$.

First, we proved this lemma in Rocq (approx. 100 lines).

When introducing possible optional steps, the proof is similar except for the use of those optional steps:

Lemma D5opt_D6_and_D3opt_Implies_D7opt ($m:I$) ($k:I$):
 $k < m \rightarrow CRStar\ m \rightarrow D5opt\ m \rightarrow D6\ m \rightarrow$
 $(\forall\ h,\ h < m \rightarrow D3opt\ m\ h) \rightarrow D7opt\ m\ k.$

With the new version the proof reaches 300 lines (but factorised could possibly be improved with better abstractions and more evolved tooling on composition of relations).

The proof is done by induction on k and Fig. 13 summarises the proof sketch. At the center is the general case that performs the induction, when $k > 0$ and when none all the optional steps as conclusions of $D3opt$, $D7opt(k - 1, m)$ are indeed steps of level m . On the right is shown the base case of the induction, when $k = 0$ that relies on D1. Finally, on the left are shown two representative of the cases when the optional steps are empty, these cases are when one of

²Isabelle Formalisation of Rewriting, <http://cl-informatik.uibk.ac.at/isafor/>

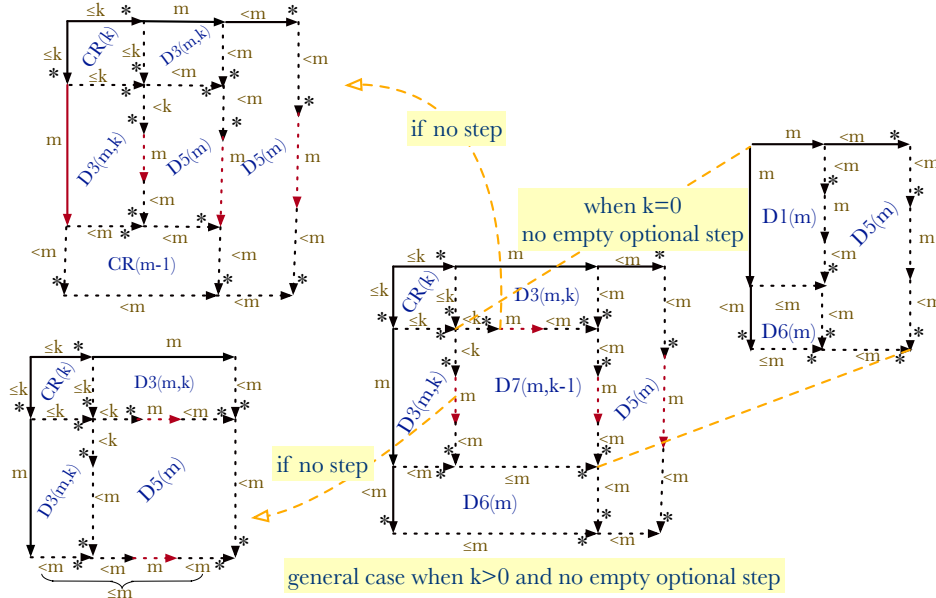


Figure 13. Sketch of the proof for D7

the two D3opt diagram has no m step in its conclusions. Optional steps induce several other cases to consider, simpler than the two shown as illustration.

B About the Rocq Mechanisation

B.1 A Note on the Mechanisation of Peaks

Utilising de Bruijn's proof requires closing the diagrams D_1 and D_2 for *all* possible peaks. However, many of these cases close immediately. We implement generic tactics for these cases. Given a peak $c \leftarrow a \rightarrow b$ there are three easy cases:

1. There is no peak: if we can prove that $b = c$, the diagram is closed by reflexivity. This is the case, e.g., if the two steps are local steps of the same process.
2. The diamond closes immediately: if the two steps commute because they act differently on the configuration. This is the case, for example, if the two steps are local steps of different processes.
3. The peak cannot exist in the first place. For example, it is not possible for the same process to take two different local steps from the same configuration.

These cases are solved by tactics (respectively) (1) `case_triangle`, (2) `case_square`, and (3) `case_impossible`. In each case, we implement the appropriate commutativity lemma for both D_1 and D_2 peaks and use a simple tactic to apply the correct version. They then generate the proof obligations and automatically discharge the trivial ones.

These tactics allow us to write concise proofs like the following snippet for the $\text{CALL}(t, t')/\text{CALL}(t_0, t'_0)$ case of D_2 .

```
is_eq t t0.
- case_triangle by triangle_call_call.
```

```
- is_eq t' t'0.
+ now apply delegate_delegate_D2.
+ now case_square by call_call_commute_diff.
```

Additionally, it is useful to compute the constraints for a given peak automatically. For example, if one step is a local computation at i and the other is call step from j we know that $i \neq j$. For this purpose we implement a `find_abs` tactic. It can be used to find a contradiction after applying `case_impossible`, but the constraints may also be useful to prove e.g., equality of the two corners in a triangular case. The tactic is implemented by defining a relation `relG` on pairs of labels and sets of constraints, then proving that the existence of a peak $c \xleftarrow{l} a \xrightarrow{l'} b$ implies `relG l l'`.

B.2 Rocq formalisation excerpt: Applying the amazing proof

De Bruijn's proof makes the confluence proofs for our classes of programs very short and sweet. Provided we have already proven all the diagrams can be filled (no small task), the proof itself is only a handful of lines!

Theorem `Confluence_layered`: `Confluent stepLayered_nolabel`.

Proof.

```
unfold Confluent in *.
rewrite Nolabel_is_stepany.
apply CR_m_Implies_CR.
apply Amazing_layered.
```

Qed.

Acknowledgments

This work is partially supported by the Research Council of Norway, within the project CROFlow (no. 326249).

References

- [1] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2017. Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction. *J. ACM* 64, 4 (2017), 25:1–25:49. doi:10.1145/3073408
- [2] Gul Agha. 1986. *Actors: a model of concurrent computation in distributed systems*. MIT Press.
- [3] Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. 2018. Systematic testing of actor systems. *Softw. Test. Verification Reliab.* 28, 3 (2018). doi:10.1002/STVR.1661
- [4] Gregory R. Andrews. 1991. *Concurrent programming - principles and practice*. Benjamin/Cummings.
- [5] Joe Armstrong. 2013. *Programming Erlang: software for a concurrent world*. The Pragmatic Bookshelf.
- [6] Franz Baader and Tobias Nipkow. 1998. *Term rewriting and all that*. Cambridge University Press, USA.
- [7] Henry G. Baker and Carl Hewitt. 1977. The incremental garbage collection of processes. In *Proc. Symposium on Artificial Intelligence and Programming Languages*, James Low (Ed.). ACM, 55–59. doi:10.1145/800228.806932
- [8] Françoise Baude, Denis Caromel, Christian Delbé, and Ludovic Henrio. 2007. Promised Messages: Recovering from Inconsistent Global States. In *Proceedings of the ACM SIGPLAN 2007 Symposium on Principles and Practice of Parallel Programming*. ACM Press, 154–155. Short paper.
- [9] Zoran Budimlic, Michael G. Burke, Vincent Cavé, Kathleen Knoke, Geoff Lowney, Ryan Newton, Jens Palsberg, David M. Peixotto, Vivek Sarkar, Frank Schlimbach, and Sagnak Tasirlar. 2010. Concurrent Collections. *Sci. Program.* 18, 3-4 (2010), 203–217. doi:10.3233/SPR-2011-0305
- [10] Denis Caromel and Ludovic Henrio. 2004. *A Theory of Distributed Objects*. Springer-Verlag.
- [11] Denis Caromel, Ludovic Henrio, and Bernard Serpette. 2004. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 123–134.
- [12] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005)*, Ralph E. Johnson and Richard P. Gabriel (Eds.). ACM, 519–538. doi:10.1145/1094811.1094852
- [13] Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. 2015. Deterministic Replay: A Survey. *ACM Surveys* 48, 2, Article 17 (sep 2015), 47 pages. doi:10.1145/2790077
- [14] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE! 2015)*, Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos A. Varela (Eds.). ACM, 1–12. doi:10.1145/2824815.2824816
- [15] Frank de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. 2017. A Survey of Active Object Languages. *ACM Comput. Surv.* 50, 5, Article 76 (Oct. 2017), 39 pages. doi:10.1145/3122848
- [16] Frank S. de Boer, Einar Broch Johnsen, Violet Ka I Pun, and Silvia Lizeth Tapia Tarifa. 2024. Proving Correctness of Parallel Implementations of Transition System Models. *ACM Trans. Program. Lang. Syst.* 46, 3 (2024), 9:1–9:50. doi:10.1145/3660630
- [17] N.G. de Bruijn. 1978. A note on weak diamond properties. Memorandum 78-08, url: <https://research.tue.nl/files/4292304/597942.pdf>.
- [18] Nachum Dershowitz. 2005. *Term Rewriting Systems by “Terese” (Marc Bezem, Jan Willem Klop, and Roel de Vrijer, eds.)*, Cambridge University Press, *Cambridge Tracts in Theoretical Computer Science* 55, 2003, hard cover: ISBN 0-521-39115-6, xxii+884 pages. Vol. 5. Cambridge University Press, USA. 395–399 pages. Issue 3. doi:10.1017/S1471068405222445
- [19] Jörg Endrullis and Jan Willem Klop. 2013. De Bruijn’s weak diamond property revisited. *Indagationes Mathematicae* 24, 4 (2013), 1050–1072. doi:10.1016/j.indag.2013.08.005 In memory of N.G. (Dick) de Bruijn (1918–2012).
- [20] Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. 2023. Stratified Commutativity in Verification Algorithms for Concurrent Programs. *Proc. ACM Program. Lang.* 7, POPL, Article 49 (Jan. 2023), 28 pages. doi:10.1145/3571242
- [21] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. 2005. Exploiting Purity for Atomicity. *IEEE Trans. Software Eng.* 31, 4 (2005), 275–291. doi:10.1109/TSE.2005.47
- [22] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. 2009. Reducers and other Cilk++ hyperobjects. In *Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2009)*, Friedhelm Meyer auf der Heide and Michael A. Bender (Eds.). ACM, 79–90. doi:10.1145/1583991.1584017
- [23] Patrice Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Lecture Notes in Computer Science, Vol. 1032. Springer. doi:10.1007/3-540-60761-7
- [24] Laure Gonnord, Ludovic Henrio, Lionel Morel, and Gabriel Radanne. 2022. A Survey on Parallelism and Determinism. *ACM Computing Surveys* 55, 10 (Sept. 2022), 210:1–210:28. doi:10.1145/3564529
- [25] Reiner Hähnle and Ludovic Henrio. 2024. Provably Fair Cooperative Scheduling. *Art. Sci. Eng. Program.* 8, 2 (2024). doi:10.22152/PROGRAMMING-JOURNAL.ORG/2024/8/6
- [26] Philipp Haller, Simon Geries, Michael Eichberg, and Guido Salvaneschi. 2016. Reactive Async: expressive deterministic concurrency. In *Proceedings of the 7th ACM SIGPLAN Symposium on Scala (SCALA@SPLASH 2016)*, Aggelos Biboudis, Manohar Jonnalagedda, Sandro Stucki, and Vlad Ureche (Eds.). ACM, 11–20. doi:10.1145/2998392.2998396
- [27] Philipp Haller and Martin Odersky. 2009. Scala Actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.* 410, 2-3 (2009), 202–220. doi:10.1016/j.tcs.2008.09.019
- [28] Tim Harris, Adrián Cristal, Osman S. Unsal, Eduard Ayguadé, Fabrizio Gagliardi, Burton Smith, and Mateo Valero. 2007. Transactional Memory: An Overview. *IEEE Micro* 27, 3 (2007), 8–29. doi:10.1109/MM.2007.63
- [29] Ludovic Henrio, Einar Broch Johnsen, and Violet Ka I Pun. 2020. Active Objects with Deterministic Behaviour. In *Proceedings of the 16th International Conference on Integrated Formal Methods (IFM 2020) (Lecture Notes in Computer Science, Vol. 12546)*, Brijesh Dongol and Elena Troubitsyna (Eds.). Springer, 181–198. doi:10.1007/978-3-030-63461-2_10
- [30] Ludovic Henrio, Åsmund Aqissiaq Arild Kløvstad, and Yannick Zakowski. 2025. *Layers of Confluence*. doi:10.5281/zenodo.17712651
- [31] J. Roger Hindley, B. Lercher, and Jonathan P. Seldin. 1972. *Introduction to combinatory logic*. Cambridge University Press.
- [32] Nao Hirokawa, Dohan Kim, Kiraku Shintani, and René Thiemann. 2024. Certification of Confluence- and Commutation-Proofs via Parallel Critical Pairs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024, London, UK, January 15-16, 2024*, Amin Timany, Dmitriy Traytel, Brigitte Pientka, and Sandrine Blazy (Eds.). ACM, 147–161. doi:10.1145/3636501.3636949
- [33] Gérard P. Huet. 1977. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems, In 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA,

- 31 October - 1 November 1977. *J. ACM* 27, 4, 797–821. doi:10.1145/322217.322230
- [34] Gilles Kahn. 1974. The semantics of a simple language for parallel programming. In *Information processing*. North-Holland, 471–475.
 - [35] Christina Kohl and Aart Middeldorp. 2023. A Formalization of the Development Closedness Criterion for Left-Linear Term Rewrite Systems. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16–17, 2023*, Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic (Eds.). ACM, 197–210. doi:10.1145/3573105.3575667
 - [36] Edward A. Lee. 2006. The Problem with Threads. *Computer* 39, 5 (2006), 33–42. doi:10.1109/MC.2006.180
 - [37] Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (1975), 717–721. doi:10.1145/361227.361234
 - [38] Barbara Liskov and Liuba Shrira. 1988. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI 2008)*, Richard L. Wexelblat (Ed.). ACM, 260–267. doi:10.1145/53990.54016
 - [39] Marten Lohstroh and Edward A. Lee. 2019. Deterministic Actors. In *Proceedings of the 2019 Forum for Specification and Design Languages (FDL 2019)*. IEEE, 1–8. doi:10.1109/FDL.2019.8876922
 - [40] Rocq development team. 2025. *The Rocq proof assistant*. <https://rocq-prover.org/> Version 9.0.
 - [41] Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307. doi:10.1016/J.TCS.2006.12.035
 - [42] Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. 2012. TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. In *Proceedings FMOODS/FORTE 2012 (Lecture Notes in Computer Science, Vol. 7273)*, Holger Giese and Grigore Rosu (Eds.). Springer, 219–234. doi:10.1007/978-3-642-30793-5_14
 - [43] Lars Tveito, Einar Broch Johnsen, and Rudolf Schlatte. 2020. Global Reproducibility through Local Control for Distributed Active Objects. In *Proc. 23rd International Conference on Fundamental Approaches to Software Engineering (FASE 2020) (Lecture Notes in Computer Science, Vol. 12076)*, Heike Wehrheim and Jordi Cabot (Eds.). Springer, 140–160.
 - [44] Vincent van Oostrom. 1994. Confluence by Decreasing Diagrams. *Theor. Comput. Sci.* 126, 2 (1994), 259–280. doi:10.1016/0304-3975(92)00023-K
 - [45] Vincent van Oostrom. 1997. Developing developments. *Theoretical Computer Science* 175, 1 (1997), 159–181. doi:10.1016/S0304-3975(96)00173-9

Received 2025-09-12; accepted 2025-11-13