View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Apollo

Journal of Automated Reasoning (2020) 64:857–877
https://doi.org/10.1007/s10817-019-09536-w

A Relaxation of Üresin and Dubois' Asynchronous Fixed-Point Theory in Agda

Matthew L. Daggitt[1] · **Ran Zmigrod**[1] · **Timothy G. Griffin**[1]

Received: 22 March 2019 / Accepted: 25 October 2019 / Published online: 10 December 2019
© The Author(s) 2019

Abstract

Üresin and Dubois' paper "Parallel Asynchronous Algorithms for Discrete Data" shows how a class of synchronous iterative algorithms may be transformed into asynchronous iterative algorithms. They then prove that the correctness of the resulting asynchronous algorithm can be guaranteed by reasoning about the synchronous algorithm alone. These results have been used to prove the correctness of various distributed algorithms, including in the fields of routing, numerical analysis and peer-to-peer protocols. In this paper we demonstrate several ways in which the assumptions that underlie this theory may be relaxed. Amongst others, we (i) expand the set of schedules for which the asynchronous iterative algorithm is known to converge and (ii) weaken the conditions that users must prove to hold to guarantee convergence. Furthermore, we demonstrate that two of the auxiliary results in the original paper are incorrect, and explicitly construct a counter-example. Finally, we also relax the alternative convergence conditions proposed by Gurney based on ultrametrics. Many of these relaxations and errors were uncovered after formalising the work in the proof assistant Agda. This paper describes the Agda code and the library that has resulted from this work. It is hoped that the library will be of use to others wishing to formally verify the correctness of asynchronous iterative algorithms.

Keywords Asynchronous · Iterative algorithms · Formalisation · Agda

✉ Matthew L. Daggitt
matthewdaggitt@gmail.com

Ran Zmigrod
rz279@cam.ac.uk

Timothy G. Griffin
tgg22@cam.ac.uk

1 Department of Computer Science and Technology, University of Cambridge, Cambridge, UK

1 Introduction

1.1 A Theory of Asynchronous Iterative Algorithms

Let S be a set. Iterative algorithms attempt to find a fixed point $x^* \in S$ for a function $\mathbf{F} : S \to S$ by repeatedly applying the function to some initial starting point $x \in S$. The state after k such iterations, $\sigma^k(x)$, is defined as follows:

$$\sigma^k(x) \triangleq \begin{cases} x & \text{if } k = 0 \\ \mathbf{F}(\sigma^{k-1}(x)) & \text{otherwise} \end{cases}$$

The algorithm terminates when it reaches an iteration k^* such that $\sigma^{k^*+1}(x) = \sigma^{k^*}(x)$ and so $x^* = \sigma^{k^*}(x)$ is the desired fixed point.

Many iterative algorithms can be performed in parallel. Assume that the state space S and the function $\mathbf{F}$ are decomposable into n parts:

$$S = S_1 \times S_2 \times \cdots \times S_n \quad \mathbf{F} = (\mathbf{F}_1, \mathbf{F}_2, \ldots, \mathbf{F}_n)$$

where $\mathbf{F}_i : S \to S_i$ takes in a state and calculates the ith component of the new state. It is now possible to assign the computation of each $\mathbf{F}_i$ to a separate processor. The processors may be part of a single computer with shared memory or distributed across many networked computers. We would prefer our model to be agnostic to this choice, and so this paper will simply refer to the processors as *nodes*. Each node i continues to apply $\mathbf{F}_i$ locally and propagate its updated state to the other nodes who incorporate it into their own computations. We will refer to an asynchronous implementation of this scheme as δ. A rigorous mathematical definition of δ will be presented in Sect. 2.2.

If the nodes' applications of $\mathbf{F}_i$ are synchronised then the parallel computation δ will be identical to σ. However in many cases enforcing synchronisation may not be practical or even possible. For example in distributed routing, the overhead of synchronisation on a continental scale would be prohibitive to the operation of the protocol. However, when updates are performed asynchronously, the behaviour of δ depends on the exact sequence of node activations and the timings of update messages between nodes. Furthermore, δ may enter states unreachable by σ and hence δ may not converge even when σ is guaranteed to do so. This motivates the question: what properties of $\mathbf{F}$ are required to guarantee that the asynchronous computation δ always converges to a unique fixed point?

Depending on the properties of the state space S and the function $\mathbf{F}$, there are multiple answers to this question—see the survey paper by Frommer and Szyld [12]. For example many of the approaches discussed in [12] rely on the rich structure of vector spaces over continuous domains. Üresin and Dubois [20] were the first to develop a theory that applied to both discrete and continuous domains. They prove that if $\mathbf{F}$ is an *asynchronously contracting operator* (ACO), then δ will always converge to a unique fixed point. Their model makes only very weak assumptions about inter-node communication and allows messages to be delayed, lost, duplicated and reordered. Henceforth we will refer to Üresin and Dubois [20] as **UD**.

Proving that $\mathbf{F}$ is an ACO is dramatically simpler than directly reasoning about the asynchronous behaviour of δ. However, in many cases it remains non-trivial and so **UD** also derive several alternative conditions that are easier to prove in special cases and that imply the ACO conditions. For example, they provide sufficient conditions when S is partially ordered and $\mathbf{F}$ is order preserving.

Many applications of these results can be found in the literature including in routing [6,9], programming language design [10], peer-to-peer protocols [16] and numerical simulation [7].

1.2 Contributions

This paper makes several contributions to **UD**'s existing theory. The original intention was to formalise the work of **UD** in the proof assistant Agda [4] as part of a larger project to develop formal proofs of correctness for distributed routing protocols [8]. The proofs in **UD** are mathematically rigorous in the traditional sense, but their definitions are somewhat informal and they occasionally claim the existence of objects without providing an explicit construction. Given this and the breadth of fields these results have been applied to, in our opinion a formal verification of the results is a useful exercise.

During the process of formalisation, we discovered various relaxations of the theory. This includes: (i) enlarging the set of schedules for which it is possible to prove δ converges over, (ii) relaxing the ACO conditions and (iii) generalising the model to include fully distributed algorithms rather than just shared-memory models. Furthermore, it was found that two of **UD**'s auxiliary sufficient conditions were incorrect, and we demonstrate a counter-example: an iteration which satisfies the conditions yet does not converge to a unique fixed point. Finally, we also formalise (and relax) a recently proposed alternative sufficient condition based on metric spaces by Gurney [13].

We have made the resulting library publicly available [1]. Its modular design should make it easy to apply the results to specific algorithms without understanding the technical details and we hope that it will be of use to others who are interested in developing formal proofs of correctness for asynchronous iterative algorithms. In this paper we have also included key definitions and proofs from the library alongside the standard mathematics. We do not provide an introduction to Agda, but have tried to mirror the mathematical notation as closely as possible to ensure that it is readable. Interested readers may find several excellent introductions to Agda online [3]. Any Agda types that are used but not explicitly referenced can be found in version 0.17 of the Agda standard library [2].

There have been efforts to formalise other asynchronous models and algorithms such as real-time systems [18] and distributed languages [14,15]. However, as far as we know our work is the first attempt to formalize the results of **UD**.

This paper is a revised version of our ITP 2018 conference paper [23]. In particular, the following contributions are new: (i) showing that it is possible to enlarge the set of schedules that the iteration converges over, (ii) relaxing the ACO conditions. As a result, the main proof has been simplified sufficiently to include its Agda formalisation within this paper.

2 Model

This section introduces **UD**'s model for asynchronous iterations. There are three main components: (i) the schedule describing the sequence of node activations and the timings of the messages between the nodes, (ii) the asynchronous state function and (iii) what it means for the asynchronous iteration to converge. We explicitly note where our definitions diverge from that of **UD** and justify why the changes are desirable.

2.1 Schedules

Schedules determine the non-deterministic behaviour of the asynchronous environment in which the iteration takes place; they describe when nodes update their values and the timings of the messages to other nodes. Let V be the finite set of nodes participating in the asynchronous process. Time T is assumed to be a discrete, linearly ordered set (i.e. $\mathbb{N}$).

Definition 1 A *schedule* is a pair of functions:

- The activation function $\alpha : T \to \mathcal{P}(V)$.
- The data flow function $\beta : T \to V \to V \to T$.

where β satisfies:

- (S1) $\forall t, i, j : \beta(t + 1, i, j) \leq t$

We formalise schedules in Agda as a dependent record. The number of nodes in the computation is passed as a parameter n and the nodes themselves are represented by the type Fin n, the type of finite sets with n elements.

```
record Schedule (n : ℕ) : Set where
  field
    α        : (t : 𝕋) → Subset n
    β        : (t : 𝕋)(i j : Fin n) → 𝕋
    causality: ∀ t i j → β (suc t) i j ≤ t
```

It would be possible to implicitly capture causality by changing the return type of β to Fin t instead of $\mathbb{T}$. However, this would require converting the result of β to type $\mathbb{T}$ almost every time it wanted to be used. The simplification of the definition of Schedule is therefore not worth complicating the resulting proofs.

Generalisation 1 In the original paper **UD** propose a model where all nodes communicate via shared memory, and so their definition of β takes only a single node i. However, in distributed processes (e.g. internet routing) nodes communicate in a pairwise fashion. We have therefore augmented our definition of β to take two nodes, a source and destination. **UD**'s original definition can be recovered by providing a data flow function β that is constant in its third argument.

Generalisation 2 **UD**'s definition of a schedule also has two additional liveness assumptions:

- (S2) $\forall t, i \quad : \exists t' : t < t' \wedge i \in \alpha(t')$
- (S3) $\forall t, i, j : \exists t' : \forall t'' : t' < t'' \Rightarrow \beta(t'', i, j) \neq t$

Assumption (S2) states that every node will always activate again at some point in the future and (S3) states that every message is only used for a finite amount of time. In practice they represent the assumption that every node and every link between pairs of nodes continue to function indefinitely.

Why have these assumptions been dropped from our definition of a schedule? We argue that unlike causality, (S2) and (S3) are not fundamental properties of a schedule but merely one possible set of constraints defining what it means for the schedule to be "well behaved". Any useful notion of the asynchronous iteration converging will require it to do so in a *finite* amount of time, yet (S2) and (S3) require the schedule to be well behaved for an *infinite* amount of time. This is hopefully an indication that (S2) and (S3) are unnecessarily strong

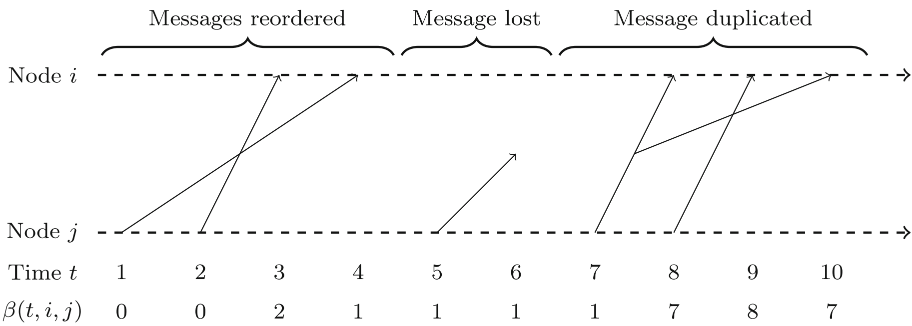

Fig. 1 Behaviour of the data flow function β. Messages from node j to node i may be reordered, lost or even duplicated. The only constraint is that every message must arrive after it was sent

assumptions. This is discussed further in Sect. 2.3 where we will incorporate relaxed versions of (S2) and (S3) into our definition of convergence, and in Sect. 3.1 we will show that there exist schedules which do not satisfy (S2) and (S3) and yet still allow the asynchronous iteration to converge.

Generalisation 3 Although not explicitly listed in their definition of a schedule, **UD** assume that all nodes activate at time 0, i.e. $\alpha(0) = V$. Such synchronisation is difficult to achieve in a distributed context and fortunately this assumption turns out to be unnecessary.

We should explicitly highlight that one of the advantages of **UD**'s theory is that there is no requirement for the data flow function to be monotonic, i.e. that messages arrive in the order they were sent:

$$\forall t, t' : t \leq t' \Rightarrow \beta(t, i, j) \leq \beta(t', i, j)$$

Although this assumption is natural in many settings, it does not hold for example if the nodes are communicating over a network and different messages take different routes through the network. Figure 1 demonstrates this and other artefacts of asynchronous communication that can be captured by β.

2.2 Asynchronous State Function

We now define the asynchronous state function δ. We formalise the state space $S = S_1 \times \cdots \times S_n$ in Agda using a (Fin n)-indexed IndexedSetoid from the Agda standard library, i.e. n sets each equipped with some suitable notion of equality.

Definition 2 Given a function $\mathbf{F}$ and a schedule (α, β) the *asynchronous state function* is defined as:

$$\delta_i^t(x) = \begin{cases} x_i & \text{if } t = 0 \\ \delta_i^{t-1}(x) & \text{else if } i \notin \alpha(t) \\ \mathbf{F}_i(\delta_1^{\beta(t,i,1)}(x), \delta_2^{\beta(t,i,2)}(x), \ldots, \delta_n^{\beta(t,i,n)}(x)) & \text{otherwise} \end{cases}$$

where $\delta_i^t(x)$ is the state of node i at time t when the iteration starts from state x.

Initially node i adopts x_i, the ith component of the initial state. At a subsequent point in time then if node i is inactive then it simply carries over its state from the previous time

step. However if node i is in the set of active nodes then it applies $\mathbf{F}_i$ to the state of the computation *from node i's local perspective*, e.g. the term $\delta_1^{\beta(t,i,1)}(x)$ is the contents of the most recent message node i received from node 1 at time t. This definition is formalised in Agda as follows:

```
δ′ : S → ∀ {t : 𝕋 } → Acc _<_ t → S
δ′ x {zero} (acc rec) i = x i
δ′ x {suc t} (acc rec) i with i ∈? α (suc t)
... | no _ = δ′ x (rec t (n<1+n t) ) i
... | yes _ = F (λ j → δ′ x (rec (β (suc t) i j) (s≤s (causality t i j))) j) i
```

Those unfamiliar with Agda may wonder why the additional Acc argument is necessary. Agda requires that every program terminates, and its termination checker ensures this by verifying that the function's arguments get structurally smaller with each recursive call. While we can see that δ will terminate, in the case where i activates at time t, the time index of the recursive call $\beta(t+1, i, j)$ is only smaller than $t+1$ because of causality, and so a naive implementation fails to pass the Agda termination checker. The Acc type helps the termination checker see that the function terminates by providing an argument that always becomes structurally smaller with each recursive call. The *rec* argument for the second case has the type: ∀ *s* → *s* < *t* + *1* → Acc _<_ *s*. Therefore in order to generate the next Acc one must prove the time really does strictly decrease. For the second recursive case this is proved by s≤s (causality *t i j*), where causality proves $\beta(t+1, i, j) \leq t$ and s≤s is a proof that if $x \leq y$ then $x+1 \leq y+1$ and hence that $\beta(t+1, i, j)+1 \leq t+1 \Leftrightarrow \beta(t+1, i, j) < t+1$. This additional complexity can be hidden from the users of the library by defining a second function of the expected type:

```
δ : S → 𝕋 → S
δ x t = δ′ x (<-wellFounded t)
```

by using the proof <-wellFounded which shows the natural numbers are well-founded with respect to < and which has type ∀ *t* → Acc _<_ *t*

Note that our revised definition of a schedule contains only what is necessary to define the asynchronous state function δ and nothing more. This provides circumstantial evidence that the decision to remove assumptions (S2) and (S3) from the definition was a reasonable one, as they are extraneous when defining the core iteration.

2.3 Correctness

Before exploring **UD**'s conditions for the asynchronous iteration δ to behave correctly, we must first establish what "behave correctly" means. An intuitive and informal definition might be as follows:

> The asynchronous iteration, δ, behaves correctly if for a given starting state x and all well-behaved schedules (α, β) there exists a time after which the iteration will have converged to the fixed point x^*.

What is a "well-behaved" schedule? As in many cases, it is initially easier to describe when a schedule is not well-behaved. For example, if a node i never activates then the iteration cannot be expected to converge to a fixed point. Equally, if node i never succeeds in sending a message to node j then a fixed point is unlikely to be reached.

UD incorporated their notion of well-behavedness into the definition of the schedule itself in the form of assumptions (S2) and (S3). These state that nodes continue to activate indefinitely and links will never fail entirely. As discussed previously in Sect. 2.1, this guarantees that the schedule is well-behaved forever. However, intuitively they are unnecessarily strong assumptions as both the definition of correctness above and the definition used by **UD** require that δ converges in a finite amount of time.

We now explore how to relax (S2) and (S3), so that the schedule is only required to be well-behaved for a finite amount of time. Unfortunately this is not as simple as requiring "every node must activate at least n times" and "every pair of nodes must exchange at least m messages", because the interleaving of node activations and message arrivals is important. For example node 1 activating n times followed by node 2 activating n times is unlikely to allow convergence as there is no opportunity for feedback from node 2 to node 1.

The right notion of a suitable interleaving of messages and activations turns out to be that of a *pseudoperiodic* schedule, as used by **UD** in their proof of convergence.

Definition 3 A schedule is *infinitely pseudoperiodic* if there exists functions $\varphi : \mathbb{N} \to T$ and $\tau : V \to \mathbb{N} \to T$ such that:

- (P1) $\varphi(0) = 0$
- (P2) $\forall i, k : \varphi(k) < \tau_i(k) \leq \varphi(k+1)$
- (P3) $\forall i, k : i \in \alpha(\tau_i(k))$
- (P4) $\forall t, i, j, k : \varphi(k+1) < t \Rightarrow \tau_i(k) \leq \beta(t, i, j)$

Note that **UD** refer to such schedules simply as *pseudoperiodic*. We have renamed it *infinitely pseudoperiodic* for reasons that will hopefully become apparent as we unpick the components of the definition.

First of all we define a period of time as a pair of times:

```
record TimePeriod : Set where
  field
    start : 𝕋
    end   : 𝕋
```

We do not include that start ≤ end as in turns out that it will always be inferrable from the context and hence including the proof in the record only leads to duplication.

Assumption (P1) for an infinitely pseudoperiodic schedule simply says that the initial time of interest is time 0. This turns out to be unnecessary and any starting point will do and hence we leave it unformalised. Assumptions (P2) and (P3) guarantee that every node activates at least once between times $\varphi(k)$ and $\varphi(k+1)$. We will call such a period an activation period.

Definition 4 A period of time $[t_1, t_2]$ is an *activation period* for node i if i activates at least once during that time period.

```
record _IsActiveIn_ (i : Fin n) (period : TimePeriod) : Set where
  constructor mkₐᵢ
  open TimePeriod period
  field
    tₐ       : 𝕋
    s<tₐ     : start < tₐ
    tₐ≤e     : tₐ ≤ end
    i∈α[tₐ]  : i ∈ α tₐ
```

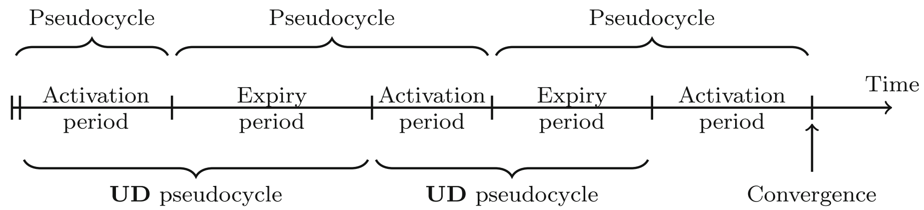

Fig. 2 A sequence of activation periods and expiry periods. Section 3.1 will show that convergence occurs at the end of an activation period. One consequence of **UD**'s definition of pseudocycle is that convergence occurs after "n and a half" pseudocycles. By redefining a pseudocycle it is possible to align the end of a pseudocycle with the end of the activation period. The realignment requires an additional expiry period at the start of the sequence, however this is fulfilled by the trivial expiry period at time 0 of 0 length. Realigning the pseudocycles in this way consequently simplifies both the definition and the proof of convergence

Assumption (P4) says that any message that arrives after $\varphi(k+1)$ must have been sent after $\tau_i(k)$, i.e. $\varphi(k+1)$ is long enough in the future that all messages sent before node i activated have either arrived or been lost. This motivates the following definition:

Definition 5 A period of time $[t_1, t_2]$ is an *expiry period* for node i if every message that arrives at i after t_2 was sent after time t_1.

```
record MessagesTo_ExpireIn_ (i : Fin n) (period : TimePeriod) : Set where
  constructor mkₑ
  open TimePeriod period
  field
    start≤end : start ≤ end
    expiryᵢ    : ∀ {t} j → end < t → start ≤ β t i j
```

UD call the period of time between $\varphi(k)$ and $\varphi(k+1)$ a pseudocycle. In such a period of time every node activates and subsequently all the messages sent before its activation time expire. The sequence $\varphi(k)$ therefore forms an infinite sequence of pseudocycles.

We argue that a pseudocycle is more naturally defined the other way round, i.e. all messages sent to node i before the start of the pseudoperiod should expire and *then* the node should activate. As shown in Fig. 2 and proved in Sect. 3.1, this alteration aligns the end of the pseudocycle with the moment the iteration converges. This has the consequence of simplifying the definition of what it means for the asynchronous iteration to converge in a finite time as well as the subsequent proofs that the iteration converges.

Definition 6 A period of time $[s, e]$ is a *pseudocycle* if there exists a time t such that $[s, t]$ is an expiry period and $[t, e]$ is an activation period.

```
record Pseudocycle (period : TimePeriod) : Set_ where
  open TimePeriod period
  field
    start≤end  : start ≤ end

    mid        : Fin n → 𝕋
    start≤midᵢ : ∀ i → start ≤ mid i
    midᵢ≤end   : ∀ i → mid i ≤ end
```

```
  β[s,m]      : ∀ i → MessagesTo i ExpireIn [ start , mid i]
  α[m,e]      : ∀ i → i IsActiveIn [ mid i , end ]
```

The notion of a pseudocycle is related to the iteration converging, as during a pseudocycle the asynchronous iteration will make at least as much progress as that of a single synchronous iteration. This will be shown rigorously in Sect. 3.1.

Definition 7 A period of time is a *multi-pseudocycle of order* k if it contains k disjoint pseudocycles.

```
data MultiPseudocycle : ℕ → TimePeriod → Set_ where
  none : ∀ {t} → MultiPseudocycle 0 [ t , t ]
  next :∀ {s} m {e k} →
         Pseudocycle [ s , m ] →
         MultiPseudocycle k [ m , e ] →
         MultiPseudocycle (suc k) [ s , e ]
```

We define a schedule to be k-pseudoperiodic if it contains k pseudocycles. **UD** show that a schedule satisfies (S2) and (S3) if and only if the schedule is ∞-pseudoperiodic. Therefore **UD**'s definition of convergence implicitly assumes that all schedules are ∞-pseudoperiodic. By removing (S2) and (S3) from the definition of a schedule we can relax our definition to say that the schedule only needs to be k^*-pseudoperiodic for some finite k^*. Our definition of what it means for δ to converge therefore runs as follows:

Definition 8 The iteration *converges over a set of states* X_0 if there exist a state x^* and a number k^* such that for all starting states $x \in X_0$ and schedules then if the schedule is k^* pseudoperiodic in some time period $[s, e]$ then for any time $t \geq e$ then $\delta^t(x) = x^*$.

```
record Converges {ℓ} (X₀ : IPred Sᵢ ℓ) : Set _ where
  field
    x*         : S
    k*         : ℕ
    x*-fixed   : F x* ≈ x*
    x*-reached :∀ {x} → x ∈ᵢ X₀ →
                (ψ : Schedule n) →
                ∀ {s e : 𝕋} → MultiPseudocycle ψ k* [ s , e ] →
                ∀ {t} → e ≤ t → δ ψ x t ≈ x*
```

Note that prior to this, the definitions of δ and pseudocycles etc. have been implicitly parameterised by some schedule ψ (omitted in the Agda via module parameters). As the definition of Converges quantifies over all schedules, this dependency must now be made explicit. Another point that Agda forces us to make explicit, and which is perhaps not immediately obvious in the mathematical definition above due to the overloading of $\in$, is that when we write $x \in X_0$ we really mean $\forall i : x_i \in (X_0)_i$. The latter is represented in the Agda code by the indexed membership relation $_\in_i_$.

What are the practical advantages of this new definition of convergence?

- It is strictly weaker than **UD** definition, as it allows a strictly larger set of schedules to be used. For example the following schedule:

$$\alpha(t) = \begin{cases} V & \text{if } t \leq k^* \\ \varnothing & \text{otherwise} \end{cases} \quad \beta(t, i, j) = \begin{cases} t - 1 & \text{if } t \leq k^* \\ k^* & \text{otherwise} \end{cases}$$

  which is synchronous until time k^* after which all nodes and links cease to activate, is k^*-pseudoperiodic but not ∞-pseudoperiodic.
- It allows one to reason about the rate of convergence. If you know δ converges according to **UD**'s definition, you still have no knowledge about how fast it converges. The new definition bounds the number of pseudocycles required. Prior work has been done on calculating the distribution of pseudocycles [5,21] when the activation function and data flow functions are modelled by various probability distributions. Together with the definition of convergence above, this would allow users to generate a probabilistic upper bound on the convergence time.

The next section discusses under what conditions δ can be proved to fulfil this definition of convergence.

3 Convergence

This section discusses sufficient conditions for the asynchronous iteration δ to converge. The most important feature of the conditions is that they require properties of the function $\mathbf{F}$ rather than the iteration δ. This means that the full asynchronous algorithm, δ, can be proved correct without having to directly reason about unreliable communication between nodes or the exponential number of possible interleavings of messages and activations.

The section is split up into 3 parts. In the first we discuss the original ACO conditions proposed by **UD**. We show that they can be relaxed and then prove that they imply our new stronger notion of convergence defined in Sect. 2.3. In the second part we show that two further sufficient conditions proposed by **UD** are in fact insufficient to guarantee that δ converges to a unique fixed point, and we provide a counter-example. In the final section we formalise and relax the alternative ultrametric conditions proposed by Gurney [13] and his proof that they reduce to the ACO conditions.

3.1 ACO Conditions

UD define a class of functions called Asynchronously Contracting Operators (ACO). They then prove that if the function $\mathbf{F}$ is an ACO, then δ will converge to a unique fixed point for all possible ∞-pseudoperiodic schedules.

Definition 9 An operator $\mathbf{F}$ is an *asynchronously contracting operator (ACO)* iff there exists a sequence of sets $D(k) = D_1(k) \times D_2(k) \times \cdots \times D_n(k)$ for $k \in \mathbb{N}$ such that

- (A1) $\forall x : x \in D(0) \Rightarrow \mathbf{F}(x) \in D(0)$
- (A2) $\forall k, x : x \in D(k) \Rightarrow \mathbf{F}(x) \in D(k+1)$
- (A3) $\exists k^*, x^* : \forall k : k^* \leq k \Rightarrow D(k) = \{x^*\}$

The ACO conditions state that the space S can be divided into a series of boxes $D(k)$. Every application of $\mathbf{F}$ moves the state into the next box, and eventually a box containing only

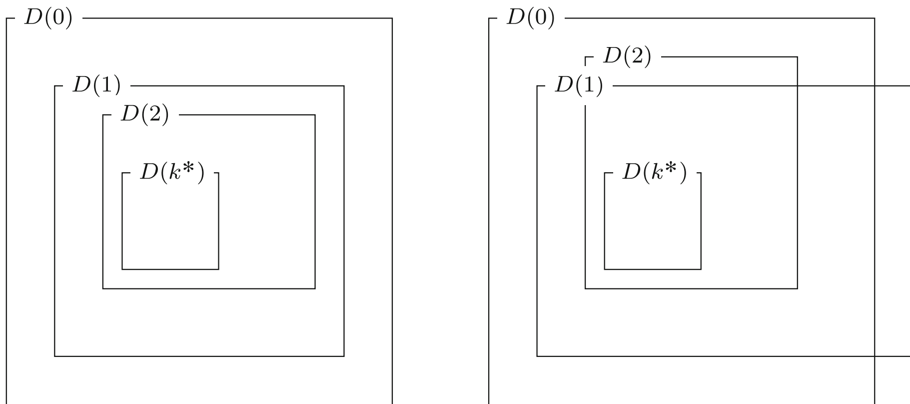

Fig. 3 Highlighting our change to the definition of the ACO conditions. Whereas **UD**'s definition required each set to be contained within the previous set (left), our definition do not make this assumption (right). Note that this figure is a simplification, as each set $D(k)$ is decomposable into $D_1(k) \times \cdots \times D_n(k)$ and so in reality the diagram should be n-dimensional

a single element is reached. Intuitively, the reason why it is possible to show that these conditions guarantee asynchronous convergence, instead of just synchronous convergence, is that each box is to be decomposable over each of the n nodes. Therefore, **F** remains contracting even if every node has not activated the same number of times. The definition of an ACO is formalised in Agda as follows:

```
record ACO ℓ : Set _ where
  field
    D          : ℕ → IPred Sᵢ ℓ
    F-resp-D₀ : ∀ {x } → x ∈ᵢ D 0 → F x ∈ᵢ D 0
    F-mono-D: ∀ {k x} → x ∈ᵢ D k → F x ∈ᵢ D (suc k)
    D-finish   : ∃₂ λ k* x* → ∀ {k} → k* ≤ k → IsSingleton (D k) x*
```

The variable ℓ is necessary to keep track of the universe level the family of sets D reside in. The sets themselves are implemented as a (Fin n)-indexed family of predicates. The code $\exists_2 \lambda k^* x^* \rightarrow$ can be read as "there exists two objects k^* and x^* such that".

Generalisation 4 The definition of an ACO in **UD** has the stronger assumption:

– (A1*) $\forall k : D(k+1) \subset D(k)$

whilst other related work in the literature [12] use:

– (A1**) $\forall k : D(k+1) \subseteq D(k)$

As shown in Fig. 3 assumption (A1*/A1**) implies that the sets $D(k)$ are nested. For any particular D, the assumption (A1) is strictly weaker than (A1*/A1**) as (A1*/A1**) + (A2) implies (A1) but (A1) + (A2) does not imply (A1*/A1**). However in general the two definitions of an ACO are equivalent because if the function **F** satisfies our definition of an ACO then the set of boxes defined by

$$C_i(0) = D_i(0)$$
$$C_i(k+1) = C_i(k) \cap D_i(k+1)$$

satisfy **UD** definition of an ACO. See our Agda library for a proof of this. However we argue that the relaxation is still useful as in practice (A1) is significantly easier to prove than (A1*/A1**) for users of the theorems.

Theorem 1 (Theorem 1 in **UD**) *If* **F** *is an ACO then* δ *converges over* $D(0)$.

Proof Assume that **F** is an ACO, and consider an arbitrary schedule (α, β) and starting state $x \in D(0)$. We initially describe some additional definitions in order to help structure the Agda proofs and increase their readability. The first definition is what it means for the current state of the iteration to be in $D(k)$:

```
StateOfNode_InBox_AtTime_ : Fin n → ℕ → 𝕋 → Set ℓ₃
StateOfNode i InBox k AtTime t = (tₐ : Acc _<_ t) → δ' x tₐ i ∈ D k i

StateInBox_AtTime_ : ℕ → 𝕋 → Set ℓ₃
StateInBox k AtTime t = ∀ i → StateOfNode i InBox k AtTime t
```

We define the messages sent to node i to be in box k at time t if every message that arrives at node i after t is in box k.

```
MessagesToNode_InBox_AtTime_ : Fin n → ℕ → 𝕋 → Set ℓ₃
MessagesToNode i InBox k AtTime t = ∀ {s} → t < s →
                                      ∀ {j} → (βₐ : Acc _<_ (β s i j)) →
                                      δ' x βₐ j ∈ D k j

MessagesInBox_AtTime_ : ℕ → 𝕋 → Set ℓ₃
MessagesInBox k AtTime t = ∀ i → MessagesToNode i InBox k AtTime t
```

Finally, the computation is in box k at time t if for every node i its messages are in box $k-1$ and its state is in box k.

```
ComputationInBox_AtTime_ : ℕ → 𝕋 → Set ℓ₃
ComputationInBox k AtTime t = ∀ i→ MessagesToNode i InBox (k ∸ 1) AtTime t
                                    × StateOfNode i InBox k AtTime t
```

The proof is then split into three main steps:
Step 1 The computation is always in $D(0)$. It is relatively easy to prove that the state is always in $D(0)$ by induction over the time t. The initial state x was assumed to be in $D(0)$, and assumption (A1) F-resp-D₀ ensures that the ith component remains in $D_i(0)$ whenever node i activates.

```
state∈D₀ : ∀ t → StateInBox 0 AtTime t
state∈D₀ zero   i (acc rec)= x∈D₀ i
state∈D₀ (suc t)i (acc rec)with i ∈? α (suc t)
... | no _ = state∈D₀ t i (rec t _)
... | yes_ = F-resp-D₀ (λ j → state∈D₀ (β (suc t) i j) j _) i
```

As the state is always in $D(0)$ then it is a trivial consequence that all messages must always be in $D(0)$.

```
messages∈D₀ : ∀ t → MessagesInBox 0 AtTime t
messages∈D₀ t i {s} t<s {j} = state∈D₀ (β s i j) j
```

Therefore the computation is always in $D(0)$.

```
computation∈D₀ : ∀ t → ComputationInBox 0 AtTime t
computation∈D₀ t i = messages∈D₀ t i , state∈D₀ t i
```

Step 2 Once the computation has entered $D(k)$ then it remains in $D(k)$. Suppose the state of node i is in $D(k)$ and the messages to i are in $D(k-1)$ at time s then we will show that the state remains in $D(k)$ for any later time e.

The proof proceeds by induction over e and k. If $e = 0$ then $s = e$ as $s \leq e$, and so the proof holds trivially. If $k = 0$ then we already know that the state is always in $D(0)$ by Step 1. For $k+1$ and $e+1$, if $s = e+1$ then again the proof holds trivially. Therefore $s < e+1$ otherwise we would contradict the assumption that time $e+1$ is after time s. If i is inactive at time $e+1$ then the result holds by the inductive hypothesis. Otherwise if i is active at time $e+1$ then assumption (A2) F-mono-D ensures that the result of applying $\mathbf{F}$ to node i's current view of the global state, is in $D(k+1)$ as we know from our initial assumption that all messages arriving at i are in $D(k)$.

```
state-stability :∀ {k s e i} → s ≤ e →
                 MessagesToNode i InBox (k ∸ 1) AtTime s ×
                 StateOfNode i InBox k AtTime s →
                 StateOfNode i InBox k AtTime e
state-stability {k}      {s} {zero} {i} z≤n    (_ , s∈Dk)= s∈Dk
state-stability {zero} {s} {suc e}{i} s≤1+e(_ , _)     = state∈D₀ (suc e) i
state-stability {suc k}{s} {suc e}{i} s≤1+e(m∈Dk , s∈D1+k) (acc _)
 with <-compare s (suc e)
... | tri≈ _ refl _        = s∈D1+k (acc _)
... | tri> _ _ s>1+e       = contradiction s≤1+e (<⇒≱ s>1+e)
... | tri< (s≤s s≤e) _ _with i ∈? α (suc e)
...  | no _ = state-stability s≤e (m∈Dk , s∈D1+k) _
...  | yes_ = F-mono-D (λ j → m∈Dk (s≤s s≤e) _) i
```

The corresponding lemma for messages is easy to prove as the definition requires that all future messages that arrive after time s at node i will be in box k and hence as $s \leq e$ so are all messages that arrive after time e.

```
message-stability :∀ {k s e i} → s ≤ e →
                   MessagesToNode i InBox k AtTime s →
                   MessagesToNode i InBox k AtTime e
message-stability s≤e m∈b e<t = m∈b (<-transʳ s≤e e<t)
```

It is then possible to prove the corresponding lemma for the entire computation.

```
computation-stability :∀ {k s e} → s ≤ e →
                       ComputationInBox k AtTime s →
                       ComputationInBox k AtTime e
```

```
computation-stability s≤e c∈Dₖ i = message-stability s≤e (proj₁ (c∈Dₖ i)) ,
                                   state-stability s≤e (c∈Dₖ i)
```

Step 3 After a pseudocycle the computation will advance from $D(k)$ to $D(k+1)$. Suppose all messages to node i are in $D(k)$ at time s and i activates at some point after s and before time e then the state of node i is in $D(k+1)$ at e. This can be shown by induction over e. We know that $e \neq 0$ as node i must activate strictly after s. If i activates at time $e+1$ then the state of node i after applying $\mathbf{F}$ must be in $D(k+1)$ by (A2) F-mono-D as all the messages it receives are in $D(k)$. If i does not activate at time $e+1$ then $[s, e]$ must still be an activation period for i and so it is possible to apply the inductive hypothesis to prove that node i is in $D(k+1)$ at time e and therefore also at $e+1$.

```
advance-state : ∀ {s e i k} → i IsActiveIn [ s , e ] →
                MessagesToNode i InBox k AtTime s →
                StateOfNode i InBox (suc k) AtTime e
advance-state {s} {zero} {i} (mkₐᵢ m () z≤n i∈αₘ)
advance-state {s} {suc e}{i} (mkₐᵢ m s<mm≤1+ei∈αₘ) m∈Dₖ (acc recₑ)
  with i ∈? α (suc e)
...| yes_ = F-mono-D (λ j → m∈Dₖ (≤-trans s<m m≤1+e) _) i
...| no i∉α₁₊ₑ with m ≟ suc e
... | yes refl      = contradiction i∈αₘ i∉α₁₊ₑ
... | no m≢1+e = advance-state (mkₐᵢ m s<m m≤e i∈αₘ) m∈Dₖ _
  where m≤e = ≤-pred (≤∧≢⇒< m≤1+e m≢1+e)
```

The analogous proof for messages runs as follows. If the computation is in $D(k)$ at time s and $[s, e]$ is an expiry period then we show that all messages i receives after time e must also be in $D(k)$. As $[s, e]$ is an expiry period then any message i receives after time e must have been sent after time s, and, as we know the computation is in $D(k)$ at time s, then the state of the computation must have been in $D(k)$ at every time after s by Step 2.

```
advance-messages : ∀ {s e k i} → MessagesTo i ExpireIn [ s , e ] →
                   ComputationInBox k AtTime s →
                   MessagesToNode i InBox k AtTime e
advance-messages (mkₑ _ exp) c∈Dₖ e<t {j} = state-stability (exp j e<t) (c∈Dₖ j)
```

Using these lemmas, it is now possible to show that during a pseudocycle the whole computation advances from $D(k)$ to $D(k+1)$. More concretely, after the expiry period messages to node i have moved from $D(k-1)$ to $D(k)$. After the subsequent activation period the state of node i has moved from $D(k)$ to $D(k+1)$ whilst the messages remain in $D(k)$.

```
advance-computation₁ : ∀ {s e k} →
                       Pseudocycle [ s , e ] →
                       ComputationInBox k AtTime s →
                       ComputationInBox (suc k) AtTime e
advance-computation₁ pp c∈Dₖ i = messagesᵉ∈Dₖ , stateᵉ∈Dₖ₊₁
  where
  open Pseudocycle pp
  messagesᵐ∈Dₖ = advance-messages (β[s,m] i) c∈Dₖ
```

```
messagesᵉ∈Dₖ = message-stability (midᵢ≤end i) messagesᵐ∈Dₖ
stateᵉ∈Dₖ₊₁   = advance-state (α[m,e] i) messagesᵐ∈Dₖ
```

It is therefore a trivial proof by induction to show that after n pseudocycles then the computation will have advanced from $D(k)$ to $D(k+n)$.

```
advance-computationₙ :∀ {s e k n} →
                     MultiPseudocycle n [ s , e ] →
                     ComputationInBox k AtTime s →
                     ComputationInBox (k + n) AtTime e
advance-computationₙ {_}{_}{k}  {zero}  none            =id
advance-computationₙ {s} {e} {k} {suc n} (next m pp mpp) = begin⟨_⟩
  ∴ ComputationInBox k           AtTime s  $⟨ advance-computation₁ pp ⟩
  ∴ ComputationInBox (suc k)     AtTime m  $⟨ advance-computationₙ mpp ⟩
  ∴ ComputationInBox (suc k + n) AtTime e  $⟨ subst _ (sym (+-suc k n)) ⟩
  ∴ ComputationInBox (k + suc n) AtTime e  □
```

The notation A $⟨ $A \Rightarrow B$ ⟩ ∴ B is an attempt to emulate standard mathematical logical reasoning that we have a proof of A, and a proof that A implies B and hence we have a proof of B.

Finally, the main result may be proved as follows. Initially the computation is in $D(0)$. Subsequently at time e after the end of the k^* pseudocycles the computation must be in $D(k^*)$. This implies that at any subsequent time t then the state of the computation must be in $D(k^*)$ and hence that $\delta^t(x) \in D(k^*)$. Therefore $\delta^t(x) = x^*$ by (A3) B-finish.

```
x*-reached :∀ {s e : 𝕋} → MultiPseudocycle k* [ s , e ] →
            ∀ {t : 𝕋} → e ≤ t →
            δ x t ≈ x*
x*-reached {s} {e} mpp {t} e≤t = begin⟨ computation∈D₀ s ⟩
  ∴ ComputationInBox 0 AtTime s  $⟨ advance-computationₙ mpp ⟩
  ∴ ComputationInBox k* AtTime e $⟨ state-stability e≤t ∘_ ⟩
  ∴ StateInBox k* AtTime t       $⟨ (λ prf i → prf i (<-wellFounded t)) ⟩
  ∴ δ x t ∈ᵢ D k*                $⟨ x∈D[k*]⇒x≈x* ⟩
  ∴ δ x t ≈ x*                   □
```

3.2 Synchronous and Finite Conditions

Even after relaxing (A1*) to (A1), the ACO sets $D(k)$ are not always intuitive or simple to construct. **UD** recognised this and provided several alternative sufficient conditions which are applicable in special cases. They then claim that these new conditions are sufficient for convergence by showing that they imply that **F** is an ACO.

The first set of sufficient conditions apply when there exists a partial order $\leq_i$ over each S_i. These are then lifted to form the order $\leq$ over S where $x \leq y$ means $\forall i : x_i \leq y_i$. **UD** then make the following claim, where σ is the synchronous state function:

Claim 1 (Proposition 3 in **UD**) The asynchronous iteration δ converges over some set $D = D_1 \times D_2 \cdots \times D_n$ if:

(i) $\forall x : x \in D \Rightarrow \mathbf{F}(x) \in D$

(ii) $\forall x, y : x, y \in D \wedge x \leq y \implies \mathbf{F}(x) \leq \mathbf{F}(y)$
(iii) $\forall x, t : x \in D \Rightarrow \sigma^{t+1}(x) \leq \sigma^t(x)$
(iv) $\forall x : x \in D \Rightarrow \sigma$ converges starting at x

UD attempt to prove this by first showing a reduction from these conditions to an ACO and then applying Theorem 1 to obtain the required result. However this claim is not true. While the asynchronous iteration does converge from every starting state in D under these assumptions, it does not necessarily converge to the *same* fixed point. The flaw in the original proof is that **UD** tacitly assume that the set $D(0)$ for the ACO they construct is the same as the original D specified in the conditions above. However the only elements that are provably in the ACO's $D(0)$ is the set $\{\sigma^k(x) \mid k \in \mathbb{N}\}$. We now present a counter-example to the claim.

Counterexample 1 Consider the degenerate asynchronous environment in which $|V| = 1$ and let $\mathbf{F}$ be the identity function (i.e. $\mathbf{F}(a) = a$). Let $D = \{x, y\}$ where the only relationships in the partial order are $x \leq x$ and $y \leq y$. Clearly (i), (ii), (iii) and (iv) all trivially hold as $\mathbf{F}$ is the identity function. However x and y are both fixed points, and which fixed point is reached depends on whether the iteration starts at x or y. Hence Claim 1 cannot be true.

It is possible to "fix" this claim by strengthening (iv) to "the synchronous iteration always converges to the *same* fixed point". Additionally it also turns out that the reduction to the ACO requires D to be non-empty so that there exists an initial state from which to begin iterating. The modified conditions are formalised in Agda as follows:

```
record SynchronousConditions p o : Set _ where

  field
    D                  : IPred Sᵢ p
    Dᵢ-cong            : ∀ {i} → (_∈ D i) Respects _≈ᵢ_
    _≤ᵢ_               : IRel Sᵢ o
    ≤ᵢ-isPartialOrder  : IsIndexedPartialOrder Sᵢ _≈ᵢ_ _≤ᵢ_

  _≤_ : Rel S o
  x ≤ y = ∀ i → x i ≤ᵢ y i

  field
    D-closed      : ∀ {x} → x ∈ᵢ D → F x ∈ᵢ D
    F-monotone    : ∀ {x y} → x ∈ᵢ D → y ∈ᵢ D → x ≤ y → F x ≤ F y
    F-decreasing  : ∀ {x} → x ∈ᵢ D → F x ≤ x

    x*            : S
    x*-fixed      : F x* ≈ x*
    k*            : ℕ
    σ-convergesTo-D and ∀ {x} → x ∈ᵢ D → σ k* x ≈ x*
```

Theorem 2 *If* $\mathbf{F}$ *obeys the synchronous conditions above,* $\mathbf{F}$ *is an ACO.*

Proof The sequence of sets $D(k)$ required by the definition of an ACO are defined as follows, where x_0 is some initial state in D:

$$D_i(k) = \{x \mid x \in D_i(0) \wedge x_i^* \leq_i x \leq_i \sigma_i^k(x_0)\}$$

For a full proof that these sets satisfy the ACO conditions please consult our Agda library [1]. □

One point of interest is that the sets defined above depend on the initial state x_0, and that $D(0)$ as defined only contains the synchronous iterates from x_0, and hence $D(0)$ is only a subset of D. It is still possible to show convergence for all states in D by constructing an ACO for each initial element and proving that $D(k^*)$ contains the same final element (which is possible due to the modification that σ now converges to a unique fixed point). However, even with our updated assumptions we have been unable to construct a single "unified" ACO for which $D(0) = D$. Whether or not it is possible to do so is an interesting open question.

3.3 Finite Conditions

UD also provide a set of sufficient conditions that are applicable for convergence over a finite set of values. Like Claim 1, they require that S is equipped with some indexed order.

Claim 2 (Proposition 4 in **UD**) The asynchronous iteration δ converges over D if:

(i) D is finite
(ii) $\forall x : x \in D \Rightarrow \mathbf{F}(x) \in D$
(iii) $\forall x : x \in D \Rightarrow \mathbf{F}(x) \leq x$
(iv) $\forall x, y : x, y \in D \wedge x \leq y \implies \mathbf{F}(x) \leq \mathbf{F}(y)$

UD's attempted proof for Claim 2 is a reduction to the conditions for Claim 1. Therefore like Claim 1, the conditions guarantee convergence but not to a unique solution. Similarly, the counterexample for Claim 1 is also a counterexample for Claim 2.

Unlike Claim 1, we do not have a proposed strengthening of Claim 2 which guarantees the uniqueness of the fixed point. This is because the finiteness of D does not help to ensure the uniqueness of the fixed point. Instead much stronger assumptions would be required to guarantee uniqueness (for example the existence of a metric space over the computation as discussed in the next section) and any such stronger conditions have the tendency to make the finiteness assumption redundant.

3.4 AMCO Conditions

Many classical convergence results for synchronous iterations rely on the notion of distance, and in suitable metric spaces the iteration can be proved to converge by showing that every application of the operator $\mathbf{F}$ moves the state closer (in non-negligible steps) to some fixed point x^*. There already exist several results of this type for asynchronous iterations. El Tarazi [11] shows that δ converges if each S_i is a normed linear space and there exists a fixed point x^* and a $\gamma \in (0, 1]$ such that for all $x \in S$:

$$||\mathbf{F}(x) - x^*|| \leq \gamma ||x - x^*||$$

However, this is a strong requirement as the use of the norm implies the existence of an additive operator over S and in many applications such an operator may not exist.

Recently Gurney [13] proposed a new, more general set of conditions based on ultrametrics [19]. Gurney does not name these conditions himself but for convenience we will call an $\mathbf{F}$ that satisfies them an *asynchronously metrically contracting operator* (AMCO). He then proves that $\mathbf{F}$ being an AMCO is equivalent to $\mathbf{F}$ being an ACO. We are primarily concerned with

applying the results to prove correctness and so we only formalise the forwards direction of the proof here. Before doing so, we define some terminology for different types of metrics.

Definition 10 A *quasi-semi-metric* is a distance function $d : S \to S \to \mathbb{N}$ such that:

- $d(x, y) = 0 \Leftrightarrow x = y$

Definition 11 An *ultrametric* is a distance function $d : S \to S \to \mathbb{N}$ such that:

- $d(x, y) = 0 \Leftrightarrow x = y$
- $d(x, y) = d(y, x)$
- $d(x, z) \leq \max(d(x, y), d(y, z))$

It should be noted that in these definitions the image of d is $\mathbb{N}$ rather than $\mathbb{R}^+$. This is important as it later allows us to prove that the distance between consecutive states in the iteration must reduce to 0 by the well-foundedness of the natural numbers over $<$.

Definition 12 An operator $\mathbf{F}$ is an *asynchronously metrically contracting operator* (AMCO) if for every node i there exists a distance function d_i and:

- (M1) S is non-empty
- (M2) d_i is a quasi-semi-metric
- (M3) $\exists\, d_i^{\max} : \forall x, y : d_i(x, y) \leq d_i^{\max}$
- (M4) $\forall x : x \neq \mathbf{F}(x) \implies d(x, \mathbf{F}(x)) > d(\mathbf{F}(x), \mathbf{F}(\mathbf{F}(x)))$
- (M5) $\forall x^*, x : (\mathbf{F}(x^*) = x^*) \wedge (x \neq x^*) \implies d(x^*, x) > d(x^*, \mathbf{F}(x))$

where $d(x, y) = \max_{i \in V} d_i(x_i, y_i)$

Assumption (M1) is not listed in Gurney [13] but was found to be required during formalisation as proofs in Agda are constructive. It should be noted that if (M1) does not hold (i.e. there are no states) then convergence is trivial to prove. Assumption (M2) says that two states are equal if and only if they occupy the same point in space. Assumption (M3) says that there exists a maximum distance between pairs of states. Assumption (M4) says that the distance between consecutive iterations must strictly decrease, and assumption (M5) says that for any fixed point x^* then applying $\mathbf{F}$ must move any state closer to that fixed point. These conditions are formalised in Agda as follows:

```
record AMCO : Set _ where
 field
  dᵢ : ∀ {i} → Sᵢ i → Sᵢ i → ℕ

 d : S → S → ℕ
 d x y = max 0 (λ i → dᵢ (x i) (y i))

 field
  element             : S
  dᵢ-quasiSemiMetric  : ∀ i → IsQuasiSemiMetric _≈ᵢ_ (dᵢ {i})
  dᵢ-bounded          : ∀ i → Bounded (dᵢ {i})
  F-strContrOnOrbits  : ∀ {x} → F x ≉ x → d (F x) (F (F x)) < d x (F x)
  F-strContrOnFP      : ∀ {x* x} → F x* ≈ x* → x ≉ x* → d x* (F x) < d x* x
```

Generalisation 5 Gurney's definition of an AMCO makes the stronger assumption:

– (M2*) d_i is an ultrametric

Users of the new modified conditions therefore no longer need to prove that their distance functions are symmetric or that they obey the max triangle inequality. This relaxation is a direct consequence of Generalisation 4 and reinforces our argument that Generalisation 4 truly is a relaxation.

Theorem 3 (Lemma 6 in [13]) *If* $\mathbf{F}$ *is an AMCO then* $\mathbf{F}$ *is an ACO.*

Proof Let x be an element in S by (M1). Then the fixed point k^* can be found by repeatedly applying (M4) to form the chain:

$$d(x, \mathbf{F}(x)) > d(\mathbf{F}(x), \mathbf{F}^2(x)) > d(\mathbf{F}^2(x), \mathbf{F}^3(x)) > \cdots$$

This is a decreasing in chain in $\mathbb{N}$ and there must exist a time k at which (M4) can no longer be applied. Hence $\mathbf{F}^k(x) = \mathbf{F}^{k+1}(x)$ and so $x^* = \mathbf{F}^k(x)$ is our desired fixed point. Let $k^* = \max_{i \in V} d_i^{max}$ by (M3). The required sets $D(k)$ are then defined as follows:

$$D_i(k) = \{x_i \in S_i \mid d_i(x_i, x_i^*) \leq \max(k^* - k, 0)\}$$

Due to space constraints we will not prove here that the sets $D(k)$ fulfil the required ACO properties. Interested readers may find the full proofs in our Agda library [1]. □

4 The Library

A library containing all of these proofs, as well as several others, is available publicly online [1]. It is arranged in such a way that hides the implementation details of the theorems from users. For example, among the most useful definitions contained in the main interface file for users are the following:

ACO⇒converges : ∀ {ℓ} → ACO F∥ ℓ → Converges F∥

AMCO⇒ACO : AMCO F∥ → ACO F∥ 0ℓ

AMCO⇒converges : AMCO F∥ → Converges F∥

where F∥ is simply a wrapped version of the function $\mathbf{F}$ that ensures that it is decomposable in the correct way. The same file also exports the definition of ACO, AMCO etc., which allows users to easily pick their conditions and theorem as desired.

5 Conclusion

5.1 Achievements

In this paper we have formalised the asynchronous fixed point theory of Üresin and Dubois' in Agda. Along the way we have proposed various relaxations by:

1. extending the model to incorporate iterations in a fully distributed environment as well as the original shared memory model.

2. showing how the ACO conditions can be tweaked to reduce the burden of proof on users of the theory.
3. reworking the theory to allow users to prove that the iteration still converges even when the schedule is only well behaved for a finite rather than an infinite length of time.

We have also described how an accordingly relaxed version of **UD**'s main theorem was successfully formalised. However our efforts to formalise Propositions 3 and 4 as stated in **UD**'s paper revealed that they are false. We hope that this finding alone justifies the formalisation process. We have proposed a fix for Proposition 3 but have been unable to come up with a similar practical alteration for Proposition 4. Finally, we have also relaxed and formalised the set of AMCO conditions based on the work by Gurney.

Our formalisation efforts have resulted in a library of proofs for general asynchronous iterations. The library is publicly available [1] and we hope that it will be a valuable resource for those wanting to formally verify the correctness of a wide range of asynchronous iterations. We ourselves have used the library to verify a proof about the largest possible set of distributed vector-based routing protocols that are always guaranteed to converge over any network [8].

5.2 Further Work

We are primarily interested in proving correctness and therefore we have only formalised that the **F** being an ACO is sufficient to guarantee that the asynchronous iteration converges. However Üresin and Dubois also show that if **F** converges then **F** is necessarily an ACO whenever the state space S is finite. The accompanying proof is significantly more complex and technical than the forwards direction and so would be an interesting extension to our formalisation.

Additionally it would be instructive to see if other related work such as [17,21,22], using different models, could be integrated into our formalisation.

Acknowledgements Matthew Daggitt is supported by an Engineering and Physical Sciences Research Council Doctoral Training grant.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Agda routing library. https://github.com/MatthewDaggitt/agda-routing/tree/jar2019. Accessed 09 Mar 2019
2. Agda standard library. https://github.com/agda/agda-stdlib, version 0.17. Accessed 20 Oct 2018
3. Agda tutorials (2019). https://agda.readthedocs.io/en/latest/getting-started/tutorial-list.html. Accessed 06 Feb 2019
4. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda–a functional language with dependent types. In: Wenzel, M., Nipkow, T. (eds.) Theorem Proving in Higher Order Logics, pp. 73–78. Springer, Berlin (2009)
5. Casanova, H., Thomason, M.G., Dongarra, J.J.: Stochastic performance prediction for iterative algorithms in distributed environments. J. Parallel Distrib. Comput. **58**(1), 68–91 (1999)
6. Chau, C.K.: Policy-based routing with non-strict preferences. SIGCOMM Comput. Commun. Rev. **36**(4), 387–398 (2006)
7. Chau, M.: Algorithmes parallèles asynchrones pour la simulation numérique. Ph.D. thesis, Institut National Polytechnique de Toulouse (2005)

8. Daggitt, M.L., Gurney, A.J.T., Griffin, T.G.: Asynchronous convergence of policy-rich distributed Bellman-Ford routing protocols. In: SIGCOMM Proceedings, ACM (2018)
9. Ducourthial, B., Tixeuil, S.: Self-stabilization with path algebra. Theor. Comput. Sci. **293**(1), 219–236 (2003)
10. Edwards, S.A., Lee, E.A.: The semantics and execution of a synchronous block-diagram language. Sci. Comput. Program. **48**(1), 21–42 (2003)
11. El Tarazi, M.N.: Some convergence results for asynchronous algorithms. Numer. Math. **39**(3), 325–340 (1982)
12. Frommer, A., Szyld, D.B.: On asynchronous iterations. J. Comput. Appl. Math. **123**(1), 201–216 (2000)
13. Gurney, A.J.T.: Asynchronous iterations in ultrametric spaces. Technical report (2017). arXiv:1701.07434
14. Henrio, L., Kammüller, F.: Functional active objects: typing and formalisation. Electron. Notes Theor. Comput. Sci. **255**, 83–101 (2009)
15. Henrio, L., Khan, M.U.: Asynchronous components with futures: semantics and proofs in Isabelle/HOL. Electron. Notes Theor. Comput. Sci. **264**(1), 35–53 (2010)
16. Ko, S.Y., Gupta, I., Jo, Y.: A new class of nature-inspired algorithms for self-adaptive peer-to-peer computing. ACM Trans. Auton. Adapt. Syst. **3**(3), 11:1–11:34 (2008)
17. Lee, H., Welch, J.L.: Applications of probabilistic quorums to iterative algorithms. In: Proceedings 21st International Conference on Distributed Computing Systems, pp. 21–28 (2001)
18. Meseguer, J., Ölveczky, P.C.: Formalization and correctness of the PALS architectural pattern for distributed real-time systems. In: International Conference on Formal Engineering Methods, pp. 303–320 (2010)
19. Schörner, E.: Ultrametric fixed point theorems and applications. Valuat. Theory Appl. **2**, 353–359 (2003)
20. Üresin, A., Dubois, M.: Parallel asynchronous algorithms for discrete data. J. ACM **37**(3), 588–606 (1990)
21. Üresin, A., Dubois, M.: Effects of asynchronism on the convergence rate of iterative algorithms. J. Parallel Distrib. Comput. **34**(1), 66–81 (1996)
22. Wei, J.: Parallel asynchronous iterations of least fixed points. Parallel Comput. **19**(8), 887–895 (1993)
23. Zmigrod, R., Daggitt, M.L., Griffin, T.G.: An Agda formalization of Üresin and Dubois' asynchronous fixed-point theory. In: International Conference on Interactive Theorem Proving, pp. 623–639. Springer (2018)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.