# An Agda Formalization of Üresin & Dubois' Asynchronous Fixed-Point Theory

Ran Zmigrod, Matthew L. Daggitt, and Timothy G. Griffin

Computer Laboratory, University of Cambridge

**Abstract.** In this paper we describe an Agda-based formalization of results from Üresin & Dubois' "Parallel Asynchronous Algorithms for Discrete Data." That paper investigates a large class of iterative algorithms that can be transformed into asynchronous processes. In their model each node asynchronously performs partial computations and communicates results to other nodes using unreliable channels. Üresin & Dubois provide sufficient conditions on iterative algorithms that guarantee convergence to unique fixed points for the associated asynchronous iterations. Proving such sufficient conditions for an iterative algorithm is often dramatically simpler than reasoning directly about an asynchronous implementation. These results are used extensively in the literature of distributed computation, making formal verification worthwhile.
Our Agda library provides users with a collection of sufficient conditions, some of which mildly relax assumptions made in the original paper. Our primary application has been in reasoning about the correctness of network routing protocols. To do so we have derived a new sufficient condition based on the ultrametric theory of Alexander Gurney. This was needed to model the complex policy-rich routing protocol that maintains global connectivity in the internet.

## 1 Introduction

Many applications work with an iterative algorithm $\mathbf{F}$ and an initial state $\mathbf{x}(0)$ where successive states are computed as

$$\mathbf{x}(t+1) = \mathbf{F}(\mathbf{x}(t))$$

until a fixed point $\xi$ is reached at some time $t'$ when $\mathbf{x}(t') = \xi = \mathbf{F}(\xi)$. Here we assume that $\mathbf{x}(t)$ represents an $n$-dimensional vector in some state space. If we rewrite $\mathbf{F}$ as

$$\mathbf{F}(\mathbf{x}) = (\mathbf{F}_1(\mathbf{x}),\ \ldots,\ \mathbf{F}_n(\mathbf{x})),$$

then we can imagine that it may be possible to assign the computation of each $\mathbf{F}_i$ to a distinct processor. This might be performed in parallel with shared memory or in a completely distributed manner. However, enforcing correctness using global synchronization mechanisms may incur performance penalties that negate the gains from the parallelization. Furthermore, global synchronization is infeasible for applications such as network routing.

This leads to the question: When can we use the $\mathbf{F}_i$ to correctly implement an *asynchronous* version of **F**-iteration? There are many answers to this question that depend on properties of the state space and the function **F** – see the survey paper by Frommer & Syzld [9].

Many of the approaches discussed in [9] rely on the rich structure of vector spaces over continuous domains. However, our motivation arises from network routing protocols where the state space is comprised of discrete data. Happily, Üresin and Dubois [21] have developed a theory of asynchronous iterations over discrete state spaces. They prove that if **F** is an Asynchronously Contracting Operator (ACO, see Section 3), then the associated asynchronous iteration will always converge to the correct fixed point. Their proof uses very weak assumptions about inter-process communication (indeed, in the case that the state space is finite they show that ACO is a necessary condition as well). These weak assumptions are a good model for the case of distributed routing protocols where messages can be delayed, lost, duplicated or reordered. Henceforth we will refer to Üresin and Dubois [21] as **UD**.

Proving that a given **F** is an ACO can be dramatically simpler than reasoning directly about an asynchronous implementation. However, in many cases it still remains non-trivial and so **UD** also derive several sufficient conditions that imply the ACO condition. These conditions are typically easier to prove for many common iterative algorithms. For example, they provide sufficient conditions for special cases where the state space is a partial order and **F** is order preserving.

In this paper we describe an Agda [3] formalization of the sufficient conditions and associated proofs from **UD**. This represents one part of a larger project in which we are developing formalized proofs of the asynchronous convergence for policy-rich distributed Bellman-Ford routing protocols (see [5]). This work required formalizing a new sufficient condition not found in **UD**, based on the ultrametric theory of Gurney [11].

Many other applications of the results of **UD** can be found in the literature (for example, [4, 6, 7, 16]). 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. In our opinion a formal verification of the results is therefore a useful exercise.

There have been other efforts to formalize asynchronous computation such as Meseguer and Ölveczky [17] for real-time systems and Henrio, Khan, and Kammüller [13, 14] for distributed languages. However, as far as we know our work is the first attempt to formalize the results of **UD**.

Our Agda development can be found on Github [1]. We hope that this will be a valuable resource for others interested in asynchronous iterations.

## 2 Preliminaries

In this section we introduce the components of the model of asynchronous computation that underpin **UD**'s results together with their Agda formalizations. Naturally, when formalizing mathematical proofs, there are concerns over steps

that are considered trivial in the informal proof. We therefore highlight key features in the proof which are in practice significantly more complex than perhaps implied by the original reasoning.

**Definition 1.** *An* iterative algorithm *consists of an initial state* $\boldsymbol{x}(0)$ *and an operator* $\boldsymbol{F}$ *such that* $\forall t \in \mathbb{N},\ \boldsymbol{x}(t+1) = \boldsymbol{F}(\boldsymbol{x}(t))$.

We begin by formalizing the product state space $S = S_1 \times \cdots \times S_n$. This is encoded by a Fin $n$-indexed family of Setoids. The type S is a function that takes $i$ and returns the Carrier type of the $i$-th setoid. We can now formalize the iterative algorithm as follows:

```
sync-iter : S → ℕ → S
sync-iter x₀ zero    = x₀
sync-iter x₀ (suc K) = F (sync-iter x₀ K)
```

**Routing example.** We briefly outline how this work can be applied to reasoning about convergence of a very general class of internet routing protocols. Full details can be found in Daggitt, Gurney & Griffin [5].

Routing problems can be formalized as a tuple $(R,\ \oplus,\ E,\ \bar{0},\ \infty)$, where:

- $R$ is the set of routes,
- $\oplus : R \to R \to R$ is the choice operator, returning the preferred route,
- $E$ is a set of functions of the form $R \to R$ representing generalized edge weights,
- $\bar{0}$ is the trivial route from a node to itself,
- $\infty$ is the invalid route.

A network configuration is represented as an $n \times n$ adjacency matrix $\mathbf{A}$ over $E$. The state space is made up of $n \times n$ matrices $\mathbf{X}$ over $R$. Matrix addition, $\mathbf{X} \oplus \mathbf{X}'$, is just the pointwise application of $\oplus$. The application of $\mathbf{A}$ to state $\mathbf{X}$ is defined as

$$(\mathbf{A}(\mathbf{X}))_{ij} = \left( \bigoplus_k \mathbf{A}_{ik}(\mathbf{X}_{kj}) \right).$$

That is, each node $i$ choose the best extensions of the routes to $j$ advertised by its neighbors. Finally, the iterative algorithm $\mathbf{F}$ is defined as

$$\mathbf{F}(\mathbf{X}) = \mathbf{A}(\mathbf{X}) \oplus \mathbf{I}, \tag{1}$$

where $\mathbf{I}$ is the matrix defined as $\mathbf{I}_{ii} = \bar{0}$, and $\mathbf{I}_{ij} = \infty$ for $i \neq j$. As explained in [5], an asynchronous version of $\mathbf{F}$ provides a good model of Distributed Bellman-Ford (DBF) routing protocols. At each asynchronous iteration in the distributed setting, each node $i$ will compute only the $i$-th row of $\mathbf{F}(\mathbf{X})$ from the rows communicated by its adjacent neighbors.

Shortest paths routing is probably the simplest example where $\oplus = \min$ and $E$ is the set of all $f_w$ with $f_w(r) = w + r$.

### 2.1 Schedules

Schedules determine the asynchronous behaviour; they dictate when nodes release new information and the timing of that information propagating to other nodes. Let $I$ be the set of nodes participating in the asynchronous process.

**Definition 2.** *A* schedule $\zeta$ *is a pair of functions* $\alpha : \mathbb{N} \to \mathcal{P}(I)$ *and* $\beta : \mathbb{N} \to I \to I \to \mathbb{N}$ *which satisfy the following properties:*

**A1** $: \forall t \in \mathbb{N}, i, j \in I.\ \beta(t+1, i, j) \leq t$
**A2** $: \forall t \in \mathbb{N}, i \quad \in I.\ \exists t'.\ t < t' \wedge i \in \alpha(t')$
**A3** $: \forall t \in \mathbb{N}, i, j \in I.\ \exists t'.\ \forall t''.\ t' < t'' \Rightarrow \beta(t'', i, j) \neq t$

The activation function $\alpha$ takes a time $t$ and returns a subset of $I$ containing the nodes that updated their value at time $t$. The data flow function $\beta$ takes a time $t$ and two nodes $i$ and $j$ and returns the time at which the data used by $i$ at time $t$ was generated by $j$.

Assumption A1 captures the notion of causality by ensuring that data can only be used after it was generated. A2 says that each node continues to activate indefinitely. Lastly, A3 says that the data generated at time $t$ will only be used for a finite number of future updates.

**Generalization 1. UD** use a shared-memory model with all nodes communicating via shared memory, and so their definition of $\beta$ takes only a single node $i$. However this model does not capture processes in which nodes communicate in a pairwise fashion without shared memory (e.g. internet routing). We have therefore augmented our definition of $\beta$ to take two nodes, a source and destination. Their original definition can be recovered by providing a $\beta$ function that is constant in its third argument.

**Generalization 2. UD** assumed that all nodes are active initially (i.e. $\alpha(0) = I$), which is unlikely to be true in a distributed context. Fortunately this assumption turns out to be unnecessary.

We formalize schedules in Agda as a dependent record. The number of nodes in the computation is passed as a parameter and the nodes themselves are represented by the Fin n type. The three properties are named causality, nonstarvation, and finite respectively.

```
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
    nonstarvation : ∀ t i → ∃ λ k → i ∈ α (t + suc k)
    finite       : ∀ t i j → ∃ λ k → ∀ l → β (k + l) i j ≢ t
```

In the definition we use $\mathbb{T}$ as an alias for $\mathbb{N}$ to help semantically differentiate between times and other natural numbers. It would also be possible to implicitly

capture causality by changing the return type of β to Fin $t$ instead of $\mathbb{T}$. However, it turns out that in practice when using β we nearly always want a regular time, and therefore each call to β would require a conversion to $\mathbb{T}$. We thus decide to keep causality as an explicit field of Schedule.

Another choice made when designing the formalisation of nonstarvation and finite was to replace the conditions such as $\forall y.\ x \leq y \implies P(y)$ with $\forall y.\ P(x+y)$. This removes the need to pass around proof terms, and consequently often makes using these properties easier to use. This same technique is used throughout the rest of our library.

An asynchronously iteration can be constructed by combining an iterative algorithm with a schedule.

**Definition 3.** *An* asynchronous iteration *over a schedule* $\mathscr{S} = (\alpha,\ \beta)$*, an initial state* $\boldsymbol{x}(0)$*, and an operator* $\boldsymbol{F}$*, is denoted as* $(\boldsymbol{F},\ \boldsymbol{x}(0),\ \mathscr{S})$ *such that* $\forall t \in \mathbb{N}, i \in I$

$$\boldsymbol{x}_i(t+1) = \begin{cases} \boldsymbol{x}_i(t) & \text{if } i \notin \alpha(t+1) \\ \boldsymbol{F}_i(\boldsymbol{x}_0(\beta(t+1,i,0)), \ldots, \boldsymbol{x}_{n-1}(\beta(t+1,i,n-1))) & \text{otherwise} \end{cases}$$

We formalize this in Agda as follows:

```
async-Iter' : Schedule n → S → ∀ {t} → Acc _<_ t → S
async-Iter' 𝒮 x₀ {zero}  _          i = x₀ i
async-Iter' 𝒮 x₀ {suc t} (acc rec) i with i ∈? α 𝒮 (suc t)
... | yes _ = F (λ j → async-Iter' 𝒮 x₀
                        (rec (β 𝒮 (suc t) i j) (s≤s (causality 𝒮 t i j))) j) i
... | no  _ = async-Iter' 𝒮 x[0] (rec t ≤-refl) i
```

Those unfamiliar with Agda may wonder why the Acc argument is necessary. While we can see that this function will terminate as each recursive call goes from time $t$ to time $\beta(t,\ i,\ j)$ which is strictly smaller due to causality, the Agda termination checker cannot detect this without help. Acc is a data-type found in the Agda standard library that helps the termination checker by providing an argument to the function that always becomes structurally smaller with each recursive call. Using the proof that the natural numbers are well-founded, this complexity is hidden from the user in the main function:

```
async-iter : Schedule n → S → 𝕋 → S
async-iter 𝒮 x₀ t = async-iter' 𝒮 x₀ (<-wellFounded t)
```

## 3 Convergence theorem

**UD** define a class of **F**s called Asynchronously Contracting Operators (ACOs). They then prove that if an operator is an ACO, then it will converge to the correct fixed point for all possible schedules.

**Definition 4.** *An operator* $\boldsymbol{F}$ *is an* asynchronously contracting operator (ACO) *on a subset* $D(0)$ *of the state space* $S = S_0 \times S_1 \times \cdots \times S_{n-1}$ *iff there exists a sequence of sets* $D(K)$ *such that*

*(i)* $\forall K \in \mathbb{N}.\ D(K) = D_0(K) \times D_1(K) \times \cdots \times D_{n-1}(K)$

*(ii)* $\exists \xi \in S.\ \exists T \in \mathbb{N}.\ \forall K \in \mathbb{N}.$

$$K < T \Rightarrow D(K+1) \subseteq D(K)$$
$$K \geq T \Rightarrow D(K) = \{\xi\}$$

*(iii)* $\forall K \in \mathbb{N}.\ \boldsymbol{x} \in D(K) \Rightarrow \boldsymbol{F}(\boldsymbol{x}) \in D(K+1)$

The sequence $D(K)$ can be seen as a form of approximation for the process with each iteration providing a higher accuracy. Each set contains the possible states at a moment in time. $D(0)$ contains many possible states as the algorithm has just begun, and each set in the sequence removes some incorrect states. This occurs until $D(T) = \{\xi\}$ when the converged state has been found.

**Generalization 3.** The definition of ACO in **UD** used the clause $K < T \Rightarrow D(K+1) \subset D(K)$, where we have relaxed this to $K < T \Rightarrow D(K+1) \subseteq D(K)$. This relaxation is also found in the survey by Frommer & Szyld [9].

The definition of an ACO is captured in the following record type:

```
record ACO p : Set _ where
  field
    D             : ℕ → ∀ i → Sᵢ i → Set p
    D-decreasing  : ∀ K → D (suc K) ⊆ D K
    D-finish      : ∃₂ λ T ξ → ∀ K → IsSingleton ξ (D (T + K))
    F-monotonic   : ∀ K {t} → t ∈ D K → F t ∈ D (suc K)
```

The variable $p$ represents the universe level of the family of sets D, while the universe level of ACO is inferred automatically (Set _). The sets themselves are implemented as a double-indexed family of predicates over $\mathsf{S_i}\ i$.

The following theorem is the main sufficient condition proved in **UD**.

**Theorem 1.** *If* $\mathbf{F}$ *is an ACO on a set* $D(0)$*, then for all schedules* $\mathscr{S}$*, any asynchronous iteration* $\mathbf{x}(k) = (\mathbf{F}, \mathbf{x}(0), \mathscr{S})$ *with* $\mathbf{x}(0) \in D(0)$*, converges to the unique fixed point* $\xi$ *of* $\mathbf{F}$ *in* $D(0)$*.*

In order to prove this theorem, **UD** consider the concept of a *pseudo-periodic schedule*. It is then proved that every schedule (Definition 2) is in fact pseudo-periodic, which greatly simplifies reasoning about schedules. This is perhaps the least rigorous aspect of the work of **UD** , as they state this without proof.

**Definition 5.** *A schedule* $\mathscr{S} = (\alpha, \beta)$ *is* pseudo-periodic *if there exists an increasing function* $\varphi : \mathbb{N} \to \mathbb{N}$ *such that:*

*(i)* $\varphi(0) = 0$

*(ii)* $\forall K \in \mathbb{N}, i \in I.\ \exists t \in \mathbb{N}.\ i \in \alpha(t) \wedge \varphi(K) \leq t < \varphi(K+1)$

*(iii)* $\forall K, t \in \mathbb{N}, i, j \in I.\ t \geq \varphi(K+1) \implies \beta(t,i,j) \geq \tau_i(K) \geq \varphi(K)$

*where $\tau_i(K)$ is the earliest time after $\varphi(K)$ that element $i$ is updated.*

The intuition behind $\varphi$ is that by time $\varphi(K+1)$ every node is guaranteed to be using data generated at least as recently as $\varphi(K)$. Hence the interval $(\varphi(K), \varphi(K+1)]$ is known as the $k^{th}$ pseudo-period.

We formalize the pseudo-periodic property in Agda as follows:

```
record IsPseudoperiodic {n : ℕ} (𝒮 : Schedule n) : Set where
  open Schedule 𝒮
  field
    φ : ℕ → 𝕋
    τ : ℕ → Fin n → 𝕋

    φ-increasing : ∀ K → K ≤ φ K
    τ-active     : ∀ K i → i ∈ α (τ K i)
    τ-after-φ    : ∀ K i → φ K ≤ τ K i
    τ-expired    : ∀ K t i j → τ K j ≤ β (φ (suc K) + t) i j
```

Note that this represents a simplification of **UD**'s definition. We worked backwards from the proof of Theorem 1 and identified only those properties required. This simplification may have to change if we extend our library to include **UD**'s proof that the ACO condition is also necessary (in the case of finite state spaces).

**UD** assert that for any schedule there exist an infinite number of possible functions $\varphi$, but they do not provide any explicit constructions. This is one area where we had initial concerns when planning our proof strategy in Agda.

We start by defining nextActive, which takes a time $t$ and a node index $i$ and returns the first time after $t$ for which that $i$ is active.

```
nextActive' : (t k : 𝕋) {i : Fin n} → i ∈ α (t + suc k) → Acc _<_ k → 𝕋
nextActive' t zero {i} _ _ = suc t
nextActive' t (suc k) {i} i∈a[t+1+K] (acc rs) with i ∈? α t
... | yes i∈a = t
... | no i∉a rewrite +-suc t (suc k) = nextActive' (suc t) k i∈a[t+1+K] _

nextActive : 𝕋 → Fin n → 𝕋
nextActive t i with nonstarvation t i
... | (K , i∈a[t+1+K]) = nextActive' t K i∈a[t+1+K] (<-wellFounded K)
```

We then define allActive, which returns the first time after $t$ such that all nodes have activated since $t$.

```
allActive : 𝕋 → 𝕋
allActive t = max t (nextActive t)
```

We then need to define three auxiliary functions: $\text{pointExpiry}_{ij}$ returns a time after which $i$ does not use the data generated by $j$ at time $t$.

```
pointExpiryᵢⱼ : Fin n → Fin n → 𝕋 → 𝕋
pointExpiryᵢⱼ i j t = proj₁ (finite t i j)
```

$\mathsf{expiry_{ij}}$ returns a time after which $i$ only uses data generated by $j$ after time $t$.

```
expiryᵢⱼ : 𝕋 → Fin n → Fin n → 𝕋
expiryᵢⱼ t i j = List.max t (applyUpTo (pointExpiryᵢⱼ i j) (suc t))
```

$\mathsf{expiry_i}$ returns a time after which $i$ only uses data generated after time $t$.

```
expiryᵢ : 𝕋 → Fin n → 𝕋
expiryᵢ t i = max t (expiryᵢⱼ t i)
```

Using these we can define the function expiry that returns a time after which all nodes only use data generated after time $t$.

```
expiry : 𝕋 → 𝕋
expiry t = max t (expiryᵢ t)
```

Finally, we construct $\varphi$ as follows:

```
φ : ℕ → 𝕋
φ zero     = zero
φ (suc K) = suc (expiry (allActive (φ K)))
```

Therefore we find a time $t$ such that all nodes have been activated after $\varphi(K)$ and then $\varphi(K+1)$ is defined as the time after which all data used was generated after $t$. The function $\tau$ (as defined in property (iii) of pseudo-periodic schedules) is simply a special call to nextActive.

```
τ : ℕ → Fin n → 𝕋
τ K i = nextActive (φ K) i
```

We now prove that $\varphi$ and $\tau$ satisfy the properties required to be pseudo-periodic as given in Definition 5. The property $\varphi$-increasing is relatively simple, given that proofs that the various functions are increasing:

```
φ-increasing : ∀ K → K ≤ φ K
φ-increasing zero     = z≤n
φ-increasing (suc K) = s≤s (begin
  K                    ≤⟨ φ-increasing K ⟩
  φ K                  ≤⟨ allActive-increasing (φ K) ⟩
  allActive (φ K)      ≤⟨ expiry-increasing (allActive (φ K)) ⟩
  expiry (allActive (φ K)) ■)
```

The second property says that $\tau$ is always active and it can be satisfied by using properties of nextActive:

```
τ-active : ∀ K i → i ∈ α (τ K i)
τ-active K = nextActive-active (φ K)
```

The third property can be easily proved using the fact that nextActive is increasing:

```
τ-after-φ : ∀ K i → φ K ≤ τ K i
τ-after-φ zero    i = z≤n
τ-after-φ (suc K) i = nextActive-increasing (φ (suc K)) i
```

The final property states that at all points during a pseudo-period, no nodes use information generated in a previous pseudo-period. This is the most complex of the four properties to prove.

```
τ-expired : ∀ K t i j → τ K j ≤ β (φ (suc K) + t) i j
τ-expired K t i j = expiry-expired (begin
  expiry (nextActive _ j)  ≤⟨ expiry-monotone (nextActive≤allActive _ j) ⟩
  expiry (allActive (φ K)) ≤⟨ n≤1+n (expiry (allActive (φ K))) ⟩
  φ (suc K)                ≤⟨ m≤m+n (φ (suc K)) t ⟩
  φ (suc K) + t            ■) i j
```

As previously mentioned the construction of $\varphi$ is not discussed in **UD**. Nevertheless, filling this gap required significant effort in our Agda development.

The proof of Theorem 1 requires an additional fact about the functions $\tau_i$: for each $K$, once all $i$ have been updated after some time $t$, then $\mathbf{x}(t) \in D(K)$.

**Lemma 1.** $\forall t,\ K \in \mathbb{N},\ i \in I.\ \tau_i(K) \leq\ t \implies \boldsymbol{x}_i(t) \in D_i(K)$.

In **UD** Lemma 1 is proved by a fairly easy induction on $K$. However, in Agda the construction, called τ-stability, turned out to be more difficult. Several smaller lemmas were required, the biggest of which is that the asynchronous iteration remains within $D(0)$, the proof of which is called async[t]'∈D₀.

```
async[t]'∈D₀ : ∀ {t} (acc_t : Acc _<_ t) → async-Iter' 𝒮 x₀ acc_t ∈ D 0
async[t]'∈D₀ {zero}  _         i = x₀∈D₀ i
async[t]'∈D₀ {suc t} (acc rec) i with i ∈? α (suc t)
... | yes i∈a = D-decreasing 0 (F-monotonic 0 (λ j →
    async[t]'∈D₀ (rec (β (suc t) i j) (s≤s (causality t i j))) j)) i
... | no i∉a = async[t]'∈D₀ (rec t (s≤s ≤-refl)) i

τ-stability' : ∀ {t K i} (acc_t : Acc _<_ t) → τ K i ≤ t →
               async-Iter' 𝒮 x₀ acc_t i ∈ᵤ D K i
τ-stability' {_}    {zero}  {i} acc_t    _     = async[t]'∈D₀ acc_t i
τ-stability' {zero} {suc K} {i} _    τ≤0   =
    contradiction τ≤0 (<⇒≱ 0<τ[1+K])
τ-stability' {suc t} {suc K} {i} (acc rec) τ≤1+t with i ∈? α (suc t)
... | yes _ = F-monotonic K (λ j → τ-stability' _ (τ[1+K]-expired τ≤1+t)) i
... | no i∉a with τ (suc K) i ≟ suc t
...   | no τ≠1+t = τ-stability' _ (<⇒≤pred (≤+≠⇒< τ≤1+t τ≠1+t))
...   | yes τ≡1+t =
    contradiction (subst (i ∈ₛ_) (cong α τ≡1+t) (τ-active (suc K) i)) i∉a
```

```
τ-stability : ∀ {t K i} → τ K i ≤ t → asyncIter 𝒮 x₀ t i ∈ᵤ D K i
τ-stability {t} = τ-stability' (<-wellFounded t)
```

We now construct the final proof of convergence. To do this we must construct a time after which the result of the asynchronous iteration is always equal to the fixed point. **UD** prove that $\varphi(T+1)$, where $T$ is from the ACO, is the convergence time. This is because each pseudo-period, every node is updated at least once and a total of $T$ updates must occur before convergence. In the Agda, we first extract T and ξ from D-Finish. We then prove Theorem 1 as follows.

```
T : 𝕋
T = proj₁ D-finish

ξ : S
ξ = proj₁ (proj₂ D-finish)

tᶜ : 𝕋
tᶜ = ϕ (suc T)

async[tᶜ]∈D[T] : ∀ t → asyncIter 𝒮 x₀ (tᶜ + t) ∈ D T
async[tᶜ]∈D[T] t j = τ-stability (begin
  τ T j          ≤⟨ τ-expired T 0 j j ⟩
  β (tᶜ + 0) j j ≡⟨ cong (λ v → β v j j) (+-identityʳ tᶜ) ⟩
  β tᶜ j j       ≤⟨ β-decreasing j j 1≤tᶜ ⟩
  tᶜ             ≤⟨ m≤m+n tᶜ t ⟩
  tᶜ + t         ■)
  where open ≤-Reasoning

async-converge : ∀ K → asyncIter 𝒮 x₀ (tᶜ + K) ≈ ξ
async-converge K = D[T]≈{ξ} (async[tᶜ]∈D[T] K)
```

## 4 The library

**UD** show that being an ACO is a sufficient (and sometimes a necessary) condition for convergence. However in practice, constructing the sets $D(K)$ can still be a non-trivial exercise. Therefore, an extensive array of sufficient (but often not necessary) conditions have been constructed that in practice can be simpler and more intuitive to apply. These conditions are nearly always a reduction back to ACOs.

In this section we introduce three different sufficient conditions that are available in our library. The first two are from **UD** and the third is a modified version of a new sufficient condition found in a recent paper by Gurney [11] (which was essential for the results described in Daggitt, Gurney and Griffin [5]).

### 4.1 Synchronous iteration conditions

The first set of sufficient conditions makes use of the synchronous iteration of the algorithm, which **UD** refer to as $\mathbf{y}(t)$, as opposed to the asynchronous iteration $\mathbf{x}(t)$. The conditions involve the existence of partial orderings, $\preceq_i$, over each $S_i$, which are lifted to the order $\preceq$ over $S$ in the usual point-wise manner. To do this, each $\preceq_i$ must be a partial order, and is formalized in Agda as:

```
record S-poset p : Set (lsuc (a ⊔ ℓ ⊔ p)) where
  field
    _≤ᵢ_          : ∀ {i} → Rel (Sᵢ i) p
    isPartialOrderᵢ : ∀ i → IsPartialOrder (_≈ᵢ_ {i}) _≤ᵢ_
```

**Proposition 1.** *An operator* $\mathbf{F}$ *is an ACO over the set* $D(0)$ *with a start state* $\mathbf{y}(0) \in D(0)$ *if:*

*(i)* $\forall \mathbf{a} \in D(0).\ \mathbf{F}(\mathbf{a}) \in D(0)$
*(ii)* $\forall \mathbf{a}, \mathbf{b} \in D(0).\ \mathbf{a} \preceq \mathbf{b} \implies \mathbf{F}(\mathbf{a}) \preceq \mathbf{F}(\mathbf{b})$
*(iii)* $\forall K \in \mathbb{N}.\ \mathbf{y}(K+1) \preceq \mathbf{y}(K)$
*(iv) The sequence* $\{\mathbf{y}(K)\}$ *converges*

Proposition 1 indicates that if $\mathbf{y}(k)$ converges, the operator $\mathbf{F}$ is monotonic, and there exists some set $D(0)$ that is closed over $\mathbf{F}$, then $\mathbf{F}$ is an ACO.

The existing of a starting state $\mathbf{y}(0)$ and the condition (i) are shared with the second set of sufficient conditions described later in Section 4.2, and therefore we split them out into their own record type.

```
record StartingConditions p : Set (lsuc (a ⊔ ℓ ⊔ p)) where
  field
    D₀          : ∀ i → Sᵢ i → Set p
    D₀-closed   : ∀ x → x ∈ D₀ → F x ∈ D₀
    x₀          : S
    x₀∈D₀       : x₀ ∈ D₀
```

Therefore, the pre-conditions of Proposition 1 are formalized as:

```
record SynchronousConditions p : Set (lsuc (a ⊔ ℓ ⊔ p)) where
  field
    start           : StartingConditions p
    poset           : S-poset p
    F-monotone      : ∀ {x y} → x ∈ D₀ → y ∈ D₀ → x ≤ y → F x ≤ F y
    iter-decreasing : ∀ K → sync-iter x₀ (suc K) ≤ sync-iter x₀ K
    iter-converge   : ∃ λ T → ∀ t → sync-iter x₀ T ≈ sync-iter x₀ (T + t)
```

The reduction by **UD** of these conditions to an ACO runs as follows. The sequence of sets $D$ required by the definition of an ACO are defined as follows:

$$D(K) = \{\mathbf{x} \mid \xi \preceq \mathbf{x} \preceq \mathbf{y}(K) \wedge \mathbf{x} \in D_0\}$$

which is directly translated in Agda as:

```
D : ℕ → ∀ i → Mᵢ i → Set p
D K i = (λ x → (ξ i ≤ x) × (x ≤ sync-iter x₀ K i)) ∩ D₀ i
```

The proof that the sets $D(K)$ are decreasing is a direct application of iter-decreasing. The fixed point for the ACO is computed by calling sync-iter on the convergence time given by iter-converge.

**Routing example.** Classical routing theory [2] assumes that distributivity holds:

$$\forall e \in E : x, y \in S : e(x \oplus y) = e(x) \oplus e(y) \tag{2}$$

and under this assumption one can prove that every entry of every routing table improves monotonically with each iteration when the protocol starts from the initial state $\mathbf{I}$. Therefore for classical routing problems such as shortest-paths, it is fairly easy to construct an instance of SynchronousConditions.

### 4.2 Finite conditions

The next set of sufficient conditions are applicable when the initial set $D(0)$ is finite. Like Proposition 1, it requires that $\mathbf{F}$ is monotonic and $D(0)$ be closed over $\mathbf{F}$. Instead of reasoning about the synchronous iteration of the operator, it adds an additional requirement that $\mathbf{F}$ is non-expansive over $D(0)$.

**Proposition 2.** *An operator $\mathbf{F}$ is an ACO over the set finite $D(0)$ with a start state $\mathbf{x}(0) \in D(0)$ if:*

*(i)* $\forall \mathbf{a} \in D(0).\ \mathbf{F}(\mathbf{a}) \in D(0)$
*(ii)* $\forall \mathbf{a} \in D(0).\ \mathbf{F}(\mathbf{a}) \preceq \mathbf{a}$
*(iii)* $\forall \mathbf{a}, \mathbf{b} \in D(0).\ \mathbf{a} \preceq \mathbf{b} \implies \mathbf{F}(\mathbf{a}) \preceq \mathbf{F}(\mathbf{b})$

This can be formalized in a similar manner as Proposition 1.

```
record FiniteConditions p : Set (lsuc (a ⊔ ℓ ⊔ p)) where
  field
    start            : StartingConditions p
    poset            : M-poset p
    _≟_              : Decidable _≈_
    D₀-finite        : Finite-Pred D₀
    F-nonexpansive   : ∀ {x} → x ∈ D₀ → F x ≤ x
    F-monotone       : ∀ {x y} → x ∈ D₀ → y ∈ D₀ → x ≤ y → F x ≤ F y
    F-cong           : ∀ {x y} → x ≈ y → F x ≈ F y
```

The proof for Proposition 2 is a reduction to the conditions for Proposition 1. To do this we must show that the synchronous iteration decreases and converges. For convenience we define the Finite-Pred condition as the existence of a List that contains all elements of $D_0$.

The first goal is to prove that the synchronous iteration decreases, as $x_0$ is in $D_0$, this is a direct use of F-nonexpansive.

```
iter-decreasing : ∀ K i → iter x₀ (suc K) i ≤ iter x₀ K i
iter-decreasing K i = F-nonexpansive (closed-trans K) i
```

Proving that the synchronous iteration converges is more complex. This is due to the constructive nature of the proof, meaning that we must actually construct the fixed point $\xi$. To do this, we iterate until two consecutive steps are equal in which case we have converged (this exlaines the need for the two additional assumptions about equality: f-cong and $\_\stackrel{?}{=}\_$). However the Agda termination checker is not initially satisfied that this process will ever halt. We must therefore provide a value that strictly decreases each iteration and once again use the Acc type from the standard library.

As we know that all iterations are in $D_0$ from $D_0$-closed, the length of the list representing the elements of $D_0$ can be used to provide the decreasing values. Each iteration, the current value is removed from the list. Removing the current value is where the two additional assumptions of decidable equality and the preservation of equality by the operator are used.

**Routing example.** When starting the shortest-path routing iteration from arbitrary states, junk routes may be present that cause the well-known phenomenon of *count-to-convergence*. In order to guarantee the convergence from arbitrary states the routing protocol RIP [12] limits the longest path length is 15. This has the effect of making the domain finite, and hence one could imagine constructing an instance of FiniteConditions to prove the convergence of RIP from any state.

### 4.3 Ultrametrics

The notion of convergence has an intuitive interpretation in metric spaces. In such spaces, convergence is equivalent to every application of the operators $\mathbf{F}_i$ moving you closer (in discrete steps) to the fixed point $\xi$.

There do exist results of this type. For instance El Tarazi [8] shows that if there is a normed linear space over each the values at each node $i$, then convergence occurs if there exists a fixed point $\mathbf{x}^*$ and a $\gamma \in (0, 1]$ such that:

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

However in many ways this is a very strong sufficient condition as the existence of a norm over the operation space assumes the existence of an additive operator on the space. For many processes, including our example of network routing, this may not be true.

Instead there is a more general result by Gurney [11] based on ultrametrics. An ultrametric [19] is a metric where the standard triangle inequality has been replaced by the strong triangle inequality. As far as we are aware, this result seems to have appeared only in [11]. In fact [11] proves not only that the conditions imply an ACO but are actually equivalent to being an ACO and therefore

equivalent to saying the process converges. As with the theorems of **UD** we are primarily concerned with the usability of the theorems and therefore only prove the forwards direction.

**Definition 6.** *An* ultrametric space $(S, \Gamma, d)$ *is a set* $S$*, a totally ordered set* $\Gamma$ *with a least element* $0$*, and a function* $d : S \to S \to \Gamma$ *such that:*

**M1** *:* $d(x, y) = 0 \Leftrightarrow x = y$
**M2** *:* $d(x, y) = d(y, x)$
**M3** *:* $d(x, z) \leqslant \max(d(x, y), d(y, z))$

**Definition 7.** *A function* $f : S \to S$ *is* strictly contracting on orbits *in an ultrametric space* $(S, \Gamma, d)$ *if:*

$$x \neq f(x) \implies d(x, f(x)) > d(f(x), f(f(x)))$$

i.e. the distance between iterations strictly decreases.

**Definition 8.** *An operator* $f : S \to S$ *is* strictly contracting on a fixed point $x^*$ *in an ultrametric space* $(S, \Gamma, d)$ *if:*

$$x \neq x^* \implies d(x^*, x) > d(x^*, f(x))$$

**Theorem 2 (Gurney [11]).** *If there exists* $(S_i, \Gamma, d_i)$*, and we take* $S = \prod_i S_i$ *and* $d(\boldsymbol{x}, \boldsymbol{y}) = \max_i d_i(\boldsymbol{x}_i, \boldsymbol{y}_i)$ *then* $\boldsymbol{F}$ *is an ACO if:*

1. $\Gamma$ *is finite*
2. $\boldsymbol{F}$ *is strictly contracting on orbits over* $(S, \Gamma, d)$
3. $\boldsymbol{F}$ *is strictly contracting on a fixed point over* $(S, \Gamma, d)$
4. $S$ *is non-empty*

These conditions are constructed in Agda as:

```
record UltrametricConditions : Set (a ⊔ ℓ) 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
    dᵢ-isUltrametric      : ∀ {i} → IsUltrametric (𝕊 i) dᵢ
    F-strContrOnOrbits    : F StrContrOnOrbitsOver d
    F-strContrOnFP        : F StrContrOnFixedPointOver d
    d-bounded             : Bounded d

    element               : S
    _≟_                   : Decidable _≈_
    F-cong                : F Preserves _≈_ ⟶ _≈_
```

Note that in our formalisation we currently assume $\Gamma$ = Fin n for some $n$ in order to simplify the theory. We plan to generalize this at some point.

Our Agda proof is very similar to the original proof by Gurney [11]. One of the key differences is that Gurney assumes that $F$ is contracting where as we assume that $F$ is strictly contracting on a fixed point. This is because in our use-case it is not possible to construct a contracting metric. The relationship between the two properties is not entirely clear, but the resulting proofs are very similar.

**Routing example.** The Border Gateway Protocol [18] is used by all Internet Service Providers (ISPs) to maintain connectivity in the global internet. As explained in [5], distributivity (Eq. 2) cannot be guaranteed in this setting primarily because of the competing interests of service providers and the very expressive policy languages needed to implement these interests in routing.

Consequently, a great deal of research has been directed at finding sufficient conditions that guarantee convergence for policy-rich protocols such as BGP (see for example [10, 20]). One reasonable condition is that the algebra be *strictly increasing*:

$$\forall e \in E : x \in S : x = x \oplus e(x) \neq e(x) \tag{3}$$

This says that a route $x$ must be strictly more preferred than any extension $e(x)$.

However, now individual routing table entries are no longer guaranteed to improve monotonically, and so there is no natural ordering on the state space. Assuming Eq. 3, [5] show how to construct suitable ultrametrics $d_i$ over the routing tables in such a way that they fulfill the properties required by Theorem 2. It is based on the observation that the *worst* routing table entry in the state will always improve after each iteration.

## 5 Conclusion

In this paper we have taken the mathematically rigorous yet informal proof of Üresin and Dubois' theory regarding the convergence of asynchronous iterations [21] and formalized it constructively in Agda. This involved explicitly constructing the previously unspecified pseudo-periodic sequences and mildly weakening some assumptions. Furthermore, we have described our library of proofs and sufficient conditions for asynchronous convergence, including a recent, new ultrametric condition. We hope that our library of sufficient conditions will be a valuable resource for those wanting to formally verify the convergence of a wide range of asynchronous iterations. The library is available on Github [1].

We are primarily interested in proving convergence and therefore we have thus far only formalized the sufficient conditions from Üresin and Dubois and not their proof that the ACO condition is also necessary in the case of finite state spaces. This would be an interesting extension to our development. In addition it would be interesting to see if other related work such as [15, 22, 23], using different models, could be integrated into our formalization.

## References


1. Agda routing library, https://github.com/MatthewDaggitt/agda-routing/tree/itp2018
2. Baras, J.S., Theodorakopoulos, G.: Path problems in networks. Synthesis Lectures on Communication Networks 3(1), 1–77 (2010)
3. Bove, A., Dybjer, P., Norell, U.: A brief overview of agda – a functional language with dependent types. In: Theorem Proving in Higher Order Logics. pp. 73–78. Springer Berlin Heidelberg (2009)
4. Chau, C.k.: Policy-based routing with non-strict preferences. SIGCOMM Comput. Commun. Rev. 36(4), 387–398 (Aug 2006)
5. 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), to appear
6. Ducourthial, B., Tixeuil, S.: Self-stabilization with path algebra. Theoretical Computer Science 293(1), 219 – 236 (2003), Max-Plus Algebras
7. Edwards, S.A., Lee, E.A.: The semantics and execution of a synchronous block-diagram language. Science of Computer Programming 48(1), 21 – 42 (2003)
8. El Tarazi, M.N.: Some convergence results for asynchronous algorithms. Numerische Mathematik 39(3), 325–340 (1982)
9. Frommer, A., Szyld, D.B.: On asynchronous iterations. Journal of computational and applied mathematics 123(1), 201–216 (2000)
10. Griffin, T.G., Shepherd, F.B., Wilfong, G.: The stable paths problem and interdomain routing. IEEE/ACM Transactions on Networking 10(2), 232–243 (2002)
11. Gurney, A.J.T.: Asynchronous iterations in ultrametric spaces. Tech. rep. (2017), https://arxiv.org/abs/1701.07434
12. Hendrick, C.: Routing information protocol (RIP) (1988), RFC 1058
13. Henrio, L., Kammüller, F.: Functional active objects: Typing and formalisation. Electronic Notes in Theoretical Computer Science 255, 83 – 101 (2009), FOCLASA
14. Henrio, L., Khan, M.U.: Asynchronous components with futures: Semantics and proofs in isabelle/hol. Electronic Notes in Theoretical Computer Science 264(1), 35 – 53 (2010)
15. Lee, H., Welch, J.L.: Applications of probabilistic quorums to iterative algorithms. In: Proceedings 21st International Conference on Distributed Computing Systems. pp. 21–28 (Apr 2001)
16. Lee, H., Welch, J.L.: Randomized registers and iterative algorithms. Distributed Computing 17(3), 209–221 (2005)
17. Meseguer, J., Ölveczky, P.C.: Formalization and correctness of the PALS architectural pattern for distributed real-time systems. In: ICFEM. pp. 303–320 (2010)
18. Rekhter, Y., Li, T.: A Border Gateway Protocol (BGP) (1995)
19. Schörner, E.: Ultrametric fixed point theorems and applications. Valuation Theory and its Applications 2, 353–359 (2003)
20. Sobrinho, J.L.: An algebraic theory of dynamic network routing. IEEE/ACM Transactions on Networking 13(5), 1160–1173 (2005)
21. Üresin, A., Dubois, M.: Parallel asynchronous algorithms for discrete data. J. ACM 37(3), 588–606 (Jul 1990)
22. Üresin, A., Dubois, M.: Effects of asynchronism on the convergence rate of iterative algorithms. Journal of Parallel and Distributed Computing 34(1), 66 – 81 (1996)
23. Wei, J.: Parallel asynchronous iterations of least fixed points. Parallel Computing 19(8), 887 – 895 (1993)