An Agda Formalisation of an Asynchronous Routing Protocol

Lex M. van der Stoep
Queens' College

UNIVERSITY OF CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the Computer Science Tripos, Part III.

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
UNITED KINGDOM

Email: lmv34@cam.ac.uk

June 1, 2020

Acknowledgements

Thanks to Tim Griffin, for supervising the project, introducing me to automated theorem proving and for exciting me about research in Computer Science. Thanks to Matthew Daggitt, for teaching me about Agda. Thanks to the community of Queens' College and the Computer Laboratory, for being a continuous source of learning, support and inspiration. Thanks to my parents, for everything.

Declaration of Originality

I, Lex M. van der Stoep of Queens' College, being a candidate for Computer Science Tripos, Part III, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 11,139[1]

Signed:

Date:

[1]This includes all content in the main body of the project, excluding the abstract, bibliography and appendices.

Abstract

Routing protocols play an essential role in computer networks such as the Internet. They orchestrate the communication between routers by computing the paths that traffic takes between them. We present new results in modelling the asynchronous behaviour of such protocols.

One of the more successful frameworks for reasoning about these protocols is the algebraic approach. The problems that these protocols solve (and the algorithms they use) can be generalised to a generic algebraic structure. Using this approach, the behaviour of entire classes of protocols can be reasoned about within a single framework.

In this dissertation we bridge the gap between this high-level algebraic approach, and actual implementations of distributed Bellman-Ford routing protocols. We define a chain of increasingly more detailed models of the low-level mechanism, and show that the high-level model can indeed simulate the implementations. This contributes to the credibility of the algebraic approach to routing. We also show that the well-known high-level approach fails to correctly simulate the behaviour of hard-state protocols in an asynchronous world, by constructing a counterexample.

This dissertation builds on top of work done for my Part II dissertation. In that work, we modelled various features of so-called "hard-state vectoring routing protocols" in a synchronous world. However, this is a greatly simplified view of the actual mechanics of implementations. The reality is more complex: routers operate asynchronously and therefore messages may be delayed, reordered or even lost. In this dissertation, the focus is on modelling the behaviour of routing protocols in an asynchronous world. To tame the vast complexity that arises when reasoning about their behaviour, we use Üresin and Dubois' framework for asynchronous processes.

Our models and theorems have been formalised in the Agda theorem prover. The implementation is incorporated into an open-source library for reasoning about network routing problems, available for others to use and extend the formalised results. This formalisation is in line with the increasing emphasis on formal verification of software for critical infrastructure.

“With Agda, you must be patient.
You must see the world as she does.
Then you will be at peace.” ∼ Tim Griffin

Contents

Chapter 1

Introduction

Routing protocols play an essential role in computer networks such as the Internet. They orchestrate the communication between routers by computing the paths that traffic takes between them, aiming to select the best routes. The problem they solve is an instance of the general class of problems called path problems – problems of findings paths with certain properties in graphs.

Research in the 1970s showed that these problems (and algorithms which solve them) can be generalised to a generic algebraic structure [4]. This algebraic approach is one of the more successful frameworks for reasoning about routing problems [10]. For example, recent research by Sobrinho shows that if the underlying algebra obeys certain conditions, then the protocol can be guaranteed to convergence to a solution [23]. Such results are useful, since they can aid routing protocol researchers when designing new protocols with desired properties.

We consider the family of policy-rich distributed Bellman-Ford routing protocols. It encompasses protocols which, in the algebraic framework, use different algebras but all solve their problem using some variant of a distributed version of the Bellman-Ford algorithm. This family of protocols include the widely-used Border Gateway Protocol [22], and could include potential future routing protocols.

The vast expressiveness of such policy-rich routing protocols – their ability to represent complex routing policies – also gives rise to unwanted anomalies, such as non-convergence, non-determinism and other problems [11, 20]. The algebraic framework can be used to model such protocols. However, it abstracts away many implementation details.

The goal of this dissertation is to bridge the gap between the high-level algebraic approach and the low-level mechanism (i.e. implementations) of distributed Bellman-Ford proto-

cols. It builds on top of work done for my Part II dissertation [25]. In that work, I showed the functional equivalence between the high-level model (referred to as Γ_0) and a model of the low-level mechanism (referred to as Γ_3), in a *synchronous* world. However, this is a greatly simplified view of the actual mechanics of implementations. The reality is more complex: routers operate asynchronously and therefore messages may be delayed, reordered or even lost.

The aim of this dissertation is to model the behaviour of routing protocols in an *asynchronous* world. It is well-known that reasoning about asynchronous algorithms is significantly more complex than their synchronous counterparts. Research in the 1990s by Üresin and Dubois provides us with a framework for reasoning about the behaviour of parallel asynchronous algorithms [24]. They decouple the underlying algorithm from the distributed, asynchronous context in which it occurs, thereby reducing the complexity of reasoning about asynchronous processes.

This project shows that the high-level algebraic model is indeed capable of simulating the actual implementations of distributed Bellman-Ford routing protocols, even in an asynchronous world. The results are new and will contribute to the credibility of the algebraic approach to routing.

We also present a counterexample which shows that the well-known high-level model fails to correctly simulate hard-state routing protocols, in an asynchronous world with unreliable communication. The algebraic approach should therefore be used with caution when reasoning about such protocols.

The contributions are two-fold. First, we show that the high-level model can simulate the low-level mechanisms, even in an asynchronous context. Second, we implement the models and proofs in the Agda theorem prover [2, 21]. The proofs have been incorporated into Daggitt's publicly available library for reasoning about iterative asynchronous processes and network routing problems [6]. This formalisation is in line with the increasing emphasis on formal verification of software for critical infrastructure.

In summary, this work makes the following key contributions:

- Defines several increasingly more detailed models, which approximate the low-level implementations of asynchronous routing protocols.
- Formalises the models in Agda.
- Proves that the high-level asynchronous model Ω_0 is indeed capable of simulating an asynchronous low-level implementation (as modelled by Ω_2).
- Constructs a counterexample which demonstrates that the high-level model Ω_0 is

not always capable of simulating the behaviour of hard-state routing protocols in an asynchronous world (as modelled by Ω_3).

- Formalises the simulation proofs and most of the necessary lemmas in Agda. In total, 58 out of 64 proofs have been implemented in Agda. The remaining 6 lemmas have been proved "by hand".

1.1 Road Map

In Chapter 2, we introduce the Agda programming language, a framework for reasoning about routing protocols, a framework for reasoning about asynchronous algorithms, and recent research which constitutes the starting point of this dissertation.

In Chapter 3, we define the goals of the dissertation, and explain the approach that we take to reach these goals.

In Chapters 4, 5, 6 and 7 we present the results of this project: the definitions of increasingly more detailed routing models; the theorems relating them; and the Agda formalisation of it all. Each chapter presents one of the models, and consists of two sections: a "synchronous" section summarising the synchronous model from the Part II dissertation; and an "asynchronous" section which describes the results from this project. The last chapter presents a counterexample which demonstrates that the well-known high-level model fails to correctly simulate implementations of hard-state protocols, in an asynchronous world.

In Chapter 8 we conclude the project, and discuss potential directions for future research.

Chapter 2

Background

This chapter introduces the relevant concepts for this project. In Section 2.1, we introduce the Agda programming language, and how it can be used as a proof assistant. Then in Section 2.2, the algebraic approach to routing is explained. In Section 2.3 we introduce Üresin and Dubois' framework for reasoning about the behaviour of asynchronous iterative processes. Section 2.4 then presents recent research which uses that framework to model asynchronous routing protocols. In Section 2.5 the Part II dissertation is briefly summarised. That work, together with the results described in Section 2.4, constitute the starting point for this dissertation.

2.1 Agda

Agda is a pure functional programming language [2, 21]. It can be used for general-purpose programming, but it is mainly used as a proof assistant in the development of formal proofs. An important feature of the language is that it is *dependently typed* – types can depend on arbitrary values. Agda's type system is powerful enough to represent almost any proposition as a type whose elements are proofs of the proposition.

2.1.1 The Basics

Let us begin with an introduction to the basics of the Agda programming language. We will highlight some of its features and give some examples to illustrate their use.

Datatypes and Pattern Matching

Agda has a Haskell-like syntax and likewise supports defining datatypes and pattern matching on them. For example, the natural numbers can be defined as follows:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

The above definition states that there is a constant value zero of type ℕ. Furthermore, there is a function suc which takes a value n of type ℕ and returns a value suc n of type ℕ. The number 2 is represented by suc (suc zero). This definition of natural numbers is based on Peano arithmetic.

When defining functions, we can pattern match on the construction of values of given datatypes. Addition of natural numbers can be defined as follows:

```
_+_ : ℕ → ℕ → ℕ
zero    + n = n
(suc m) + n = suc (m + n)
```

The function produces a value based on the nature of the input. If the first argument is zero, then it simply returns the second argument n (this corresponds to $0 + n = n$). If the first argument is suc m, then it constructs returns a new value by recursively calling the function (this corresponds to $(m + 1) + n = (m + n) + 1$).

Termination Checking

Agda requires functions to be *total* – they should always terminate. Without termination, Agda cannot guarantee logical consistency. This is important when using Agda as a proof assistant. Thus, at compile-time, Agda will check for termination. It accepts only those recursive schemas that it can mechanically prove terminating [1]. An example of guaranteed termination is when one argument in a function always gets structurally smaller on each recursive call. In the example above (the _+_ function), it is the first argument that gets structurally smaller, and therefore Agda accepts it as a terminating function.

If Agda cannot prove a function to be terminating, it will issue a warning to the user. The user can then try and convince Agda that it is indeed terminating, by supplying it with a structurally decreasing argument. As an alternative, one can include the TERMINATING pragma. This switches off the termination checker of Agda for a specific function. Of course, this increases the risk of a logical inconsistency in the code.

2.1.2 Dependent Types

Agda is a dependently typed language – types can depend on values. This is helpful when using types to represent propositions, as mathematics uses quantification over individual

elements (e.g. $\forall x, y, z, n \in \mathbb{N}^+.\ n > 2 \Rightarrow x^n + y^n \neq z^n$). This type system is based on Martin-Löf and Sambin's intuitionistic type theory [19], which was developed to support constructive mathematics. The type system is powerful enough to represent (almost) any proposition as a type whose elements are proofs of the proposition [21].

A *dependent type* is a family of types, indexed by some value (or type). The most basic dependent type is the dependent function type, where we quantify over types in the function signature. An example of this is the well-known identity function.

```
identity : {A : Set} → A → A
identity x = x
```

The identity function returns the argument that it is given, and can do so for any type of argument[1]. The dependent function type is the way Agda encodes polymorphic types in its type system.

Let us now look at a more involved dependent type, namely the dependent product type.

Definition 1 (Dependent product type)**.** *A dependent product type* $\Pi_{x:A} B(x)$ *is the type that assigns to each value* $x : A$*, some element of the type* $B(x)$*. The corresponding elimination rule is:*

$$\frac{f : \prod_{x:A} B(x) \qquad a : A}{f(a) : B(a)}$$

A classic example of a dependent type is Vec A n – the type of lists of length n, whose elements are of type A. Indeed, the type Vec $\mathbb{N}$ 3 is the type of triples of natural numbers. It is defined as follows:

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A 0
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)
```

We can construct a value of some type in Vec in two ways. The empty list constructor [] has type Vec A 0, for any type A. The list concatenation constructor _::_ is a function which takes an element x of type A, a vector xs of length n (and thus of the type Vec A n), and returns a new concatenated list $x :: xs$ of length $n + 1$.

Encoding the length of a list in its type can be useful when defining certain operations on them. Take the head function, for example:

[1] The curly brackets in the type signature indicate that A is an *implicit argument*. Agda will infer its value from another argument (in this case x).

Constructive logic	Dependent type system
Proposition P	Type P
Proof p of P	Program $p : P$
Implication $P \Rightarrow Q$	Function type $P \to Q$
Conjunction $P \wedge Q$	Product type $P \times Q$
Truth $\top$	Unit type $\top$
Falsehood $\bot$	Empty type $\bot$
Universal quantification $\forall x.\ P(x)$	Dependent product type $\Pi_{x:A}.\ P(x)$

Table 2.1: The Curry-Howard correspondence.

$$\mathsf{head} : \forall\ \{A\ n\} \to \mathsf{Vec}\ A\ (\mathsf{suc}\ n) \to A$$
$$\mathsf{head}\ (x :: xs) = x$$

The function returns the first element in a list. For the function to work on lists of arbitrary length, it should be defined on the empty list as well, but by the nature of the function this is undefined. Using dependent types, we can avoid the case of empty lists by design – the function type only accepts non-empty lists as input. The dependent types can help reduce bugs by allowing the programmer to specify types more precisely, and thereby limit the set of inputs.

2.1.3 Proving in Agda

The Agda programming language can be used for general-purpose programming, but it is mainly used as a proof assistant in the development of formal proofs. Agda's powerful type system allows us to encode properties of values as types, and therefore proofs of such properties as programs. The Curry-Howard correspondence is the relationship between logic and type systems [18]. Table 2.1 summarises this correspondence, for constructive logic and a dependent type system.

Example 1. *(Modus ponens). An example of a property that we can encode and prove in Agda is modus ponens – a well-known rule of inference. In short, it states that "If P implies Q, and P holds, then Q must hold as well."*

In Agda, this property is encoded in a type as follows:

$$\mathit{modus\text{-}ponens} : \forall\ \{P\ Q : \mathsf{Set}\} \to (P \to Q) \to P \to Q$$

It quantifies over all possible types P and Q. The modus-ponens function then takes a function of type $P \to Q$, a value of type P, and returns a value of type Q.

The implementation of the proof of the property in Agda is straightforward: simply apply the given function to the given value! This will leave you with a term (proof) of type Q.

```
modus-ponens f p = f p
```

An important concept in logic is equality. In Agda, it is defined as follows:

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

It states that for all terms x of any type A, there is a term refl of type $x \equiv x$. This type of equality is called *propositional equality* – terms are only equal to themselves. Note that this also includes *computational equality* of the form $2 + 2 \equiv 4$, since the term $2 + 2$ reduces to 4.

We can use this notion of propositional equality for equational reasoning.

Example 2. *(Commutativity of $*$). Consider multiplication for the natural numbers. It is defined in Agda in a similar manner to addition, by structural recursion on the first argument.*

```
_*_ : ℕ → ℕ → ℕ
zero    * n = zero
(suc m) * n = n + m * n
```

Now, we wish to encode the property that the multiplication of natural numbers is commutative: $\forall x, y. x * y = y * x$. In Agda, the type representing this property closely resembles it:

```
*-comm : ∀ m n → m * n ≡ n * m
```

We proceed by induction on the first argument. This is the natural step, as this is how the multiplication itself is defined as well.

The first case is the base case, where m is zero.

```
*-comm zero n = begin
  zero * n ≡⟨ refl ⟩
  zero     ≡⟨ sym (*-zero' n) ⟩
  n * zero ■
```

*The term "begin … ■" is the proof that zero * n is equal to n * zero. It is a chain of equal terms, where the first term is equal to the last term by the transitivity of equality. The terms inside the angle brackets ≡⟨_⟩ are proofs that relate a term to the next one. Furthermore, the function sym takes any equality term of type $x \equiv y$, and returns a symmetric proof of type $y \equiv x$. The term *-zero' n has type n * zero ≡ zero.*

The second case is the inductive step, where we deal with suc *m.*

```
*-comm (suc m) n = begin
  (suc m) * n ≡⟨ refl ⟩
  n + m * n   ≡⟨ cong (n +_) (*-comm m n) ⟩
  n + n * m   ≡⟨ sym (*-suc n m) ⟩
  n * (suc m) ■
```

The term has the type (suc *m*) * *n* ≡ *n* * (suc *m*)*. Again, we relate the left-hand side to the right-hand side using equational reasoning. The term contains a recursive call* *-comm *m n, which relates to using the induction hypothesis in logic. The term furthermore contains the function* cong*, which returns a term of type f x* ≡ *f y, for any function f and term of type x* ≡ *y.*

The above examples shed light on how Agda can be used as a proof assistant in the development and checking of formal proofs. Agda code will fail to compile when Agda cannot show that a term has the type that is expected (i.e. when it is not a valid proof).

2.2 Algebraic Approach to Routing

Routing protocols orchestrate the communication between routers in computer networks. They compute the paths that traffic takes between them, aiming to select the best routes. The problem that is solved is an instance of a path problem – a class of problems where we aim to find paths with desired properties within graphs. Path problems include the shortest path problem and critical path analysis.

Research in the 1970s by Carré showed that these problems (and the algorithms which solve them) can be generalised and represented in an algebraic structure [4]. This high-level algebraic approach allows for reasoning about the dynamics of problems at hand, proving that they have certain properties, without having to worry about implementation details. For example, recent research by Daggitt and Griffin [7] uses this framework to give bounds on the rate of convergence of routing algorithms. In Part III, I attended the lecture series "Algebraic Path Problems (L11)" [12] by Griffin which presents this approach to analysing path problems.

2.2.1 Routing Algebras

In the algebraic approach, we use an algebra that obeys certain properties to represent the problem at hand. This allows us to use generic methods to solve the problem. We consider the definition as presented by Daggitt et al. [8].

Property	Definition
$\oplus$ is associative	$a \oplus (b \oplus c) = (a \oplus b) \oplus c$
$\oplus$ is commutative	$a \oplus b = b \oplus a$
$\oplus$ is selective	$a \oplus b \in \{a, b\}$
$\bar{0}$ is an annihilator for $\oplus$	$a \oplus \bar{0} = \bar{0} = \bar{0} \oplus a$
$\bar{\infty}$ is an identity for $\oplus$	$a \oplus \bar{\infty} = a = \bar{\infty} \oplus a$
$\bar{\infty}$ is a fixed point for all $f \in F$	$f(\bar{\infty}) = \bar{\infty}$

Table 2.2: Definitions of routing algebra properties

Definition 2. *A routing algebra is a tuple $(S, \oplus, F, \bar{0}, \bar{\infty})$ where:*

- *S is the set of route values. Each route is assigned a value $s \in S$.*
- *$\oplus : S \times S \to S$ is the choice operator, which returns the preferred route out of two.*
- *F is the set of edge weights. Each weight is a function $f : S \to S$, which takes a route and returns that route extended by the edge. More precisely, let $f_{i \to j}$ be a function belonging to some edge $i \to j$, and let $s_{j \to k}$ be a route from node j to k. Then the route from i through j to k is equal to $f_{i \to j}(s_{j \to k})$. The reason for having edge functions, as opposed to edge weights, is that it allows for the representation of complex routing policies more easily.*
- *$\bar{0} \in S$ is the trivial route from any node to itself.*
- *$\bar{\infty} \in S$ is the invalid route.*

Furthermore we assume that the structure has the following properties (see Table 2.2 for more complete definitions):

- *$\oplus$ is associative and commutative – the order in which routes are chosen is irrelevant.*
- *$\oplus$ is selective – the choice operator always returns one of the two given routes.*
- *$\bar{0}$ is an annihilator for $\oplus$ – the trivial route is always preferred to any other route.*
- *$\bar{\infty}$ is an identity for $\oplus$ – any route is preferred over the invalid route.*
- *$\bar{\infty}$ is a fixed point for all $f \in F$ – extending an invalid route is also an invalid route.*

We can now define a total ordering over routes using the choice operator $\oplus$ as follows:

$$\begin{aligned} a \leq b &\equiv a \oplus b = a \\ a < b &\equiv a \leq b \wedge a \neq b. \end{aligned} \tag{2.1}$$

For all routes a we have that $\bar{0} \leq a \leq \bar{\infty}$.

Definition 3 (Strictly increasing algebra)**.** *A routing algebra is strictly increasing if:*

$$\forall f, x.\, x \neq \bar{\infty} \implies x < f(x)$$

Example 3 (Shortest paths algebra)**.** *In this framework, we can now define an algebra which represents the well-known shortest paths problem. Consider the structure ($\mathbb{N}_\infty$, min, F_+, 0, ∞). The route values are simply the natural numbers extended with infinity ($\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$). The choice operator takes the minimum of two numbers. The edge functions are the set of functions defined as follows: $F_+ = \{f_s(a) = s + a \mid s \in S\}$ – they simply take a route value and add a constant to it. The trivial route is 0 and the invalid route is ∞. The ordering that arises from this structure is the usual ordering on natural numbers (extended with infinity).*

2.3 Asynchronous Algorithms

The aim of this dissertation is to show that the high-level algebraic model can correctly simulate a class of asynchronous routing protocols. Research in the 1990s by Üresin and Dubois provides us with a framework for reasoning about the behaviour of parallel asynchronous algorithms [24]. They show that the correctness of the asynchronous algorithm can be guaranteed by reasoning about its synchronous counterpart.

Let us first consider several definitions as presented by Daggitt et al. [9].

2.3.1 The Synchronous World

An *iterative algorithm* aims to compute a fixed point $x^* \in S$ for a function $\mathbf{F} : S \to S$, where S is some carrier set of data. The algorithm can be represented by a function σ, which iteratively applies $\mathbf{F}$ to some starting state $x \in S$.

$$\sigma^k(x) = \begin{cases} x & \text{if } k = 0 \\ \mathbf{F}(\sigma^{k-1}(x)) & \text{otherwise} \end{cases} \tag{2.2}$$

The algorithm has found its solution when the function σ has stabilised at some time k^*, such that $\sigma^{k^*+1}(x) = \sigma^{k^*}(x)$

Definition 4 (Convergence)**.** *An iterative algorithm δ converges at time k^*, from a starting state x, if there exists a state x^* (fixed point for $\mathbf{F}$) such that for all t, we have $\delta^{k^*+t}(x) = \delta^{k^*}(x) = x^*$.*

Definition 5 (Absolute convergence)**.** *An iterative algorithm δ converges absolutely if it always converges to the same stable state from all possible starting states.*

Parallelisable iterative algorithms can be modelled by decomposing the carrier set S and the function $\mathbf{F}$ into n parts:

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

where each $\mathbf{F}_i : S \to S_i$ represents a computing entity which takes the entire state, and computes the value of a part of the next state. This decomposing of state is the model of parallel computing that we will use. It can be used to model both internal parallelisation (inside a single computer) or external parallelisation (between various computers).

2.3.2 The Asynchronous World

The model defined above explicitly models parallel computation. It assumes that nodes communicate synchronously, and messages between nodes are delivered immediately. In reality, the communication tends to be far less well-behaved. Messages can be delayed, reordered, or even lost.

Various computing entities (each of which is represented by a $\mathbf{F}_i$) asynchronously share data with each other, as the entities use the shared state to compute their part of the next state. Furthermore, nodes (computing entities) may not always be active. A node can possibly be inactive for a while, before it starts computing again.

Both of these notions – data sharing and node activation – are captured by a *schedule*. We use schedules to capture the particular asynchronous behaviour that nodes exhibit.

Definition 6 (Schedule)**.** *Let V denote the set of nodes that perform the asynchronous computation together, and let $\mathbb{T}$ be some discrete, linearly ordered set to denote time. A schedule is a pair of functions:*

- *The activation function $\alpha : \mathbb{T} \to \mathcal{P}(V)$, where $\alpha(t)$ is the set of nodes which update their state at time t.*
- *The data flow function $\beta : \mathbb{T} \to V \to V \to \mathbb{T}$, where $\beta(t, i, j)$ is the time at which the information used by node i at time t was sent by node j.*

where β satisfies causality – information only travels forward in time:

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

We use these schedules to define *asynchronous iterative algorithms*. Like their synchro-

nous counterparts, the asynchronous algorithms iterate until they find a stable state $x^* \in S$ which is a fixed point for the function $\mathbf{F}$. The asynchronous computation is represented by a funciton δ.

Definition 7 (Asynchronous iterative algorithm)**.** *Given a function* $\mathbf{F}$ *and a schedule* (α, β)*, the asynchronous state function is defined as:*

$$\begin{aligned} \delta_i^0(x) &= x_i \\ \delta_i^t(x) &= \begin{cases} \delta_i^{t-1}(x) & \text{if } i \notin \alpha(t) \\ \mathbf{F}_i\left(\delta_1^{\beta(t,i,1)}(x), \delta_2^{\beta(t,i,2)}(x), \ldots, \delta_n^{\beta(t,i,n)}(x)\right) & \text{otherwise} \end{cases} \end{aligned} \tag{2.3}$$

This states that all nodes i have the initial value of x_i at time $t = 0$. In subsequent computations, if a node i is inactive (i.e. $i \notin \alpha(t)$), it will simply maintain its internal state from the previous time $t - 1$. If a node i is activated, it will apply $\mathbf{F}_i$ to the state (from its local perspective) of other nodes. It does so by taking their state at time $\beta(t, i, j)$, namely the data that i received at time t from node j[2].

2.3.3 UD's Asynchronous Convergence Theorem

In their paper, Üresin and Dubois show sufficient (and necessary) conditions to guarantee convergence of asynchronous iterative algorithms [24]. Recently, Daggitt et al. contributed to these results in two ways. They formalised Üresin and Dubois' asynchronous fixed-point theory in Agda [26] and relaxed the conditions that the original paper required for convergence [9].

To guarantee convergence, the theory requires two properties. The function $\mathbf{F}$ should be an *asynchronously contracting operator (ACO)*, and the schedule should be ∞*-pseudo-periodic*. The most important feature of the conditions is that they require properties of $\mathbf{F}$ and schedules (α, β), instead of the algorithm δ. This is useful, as the correctness (asynchronous convergence) can be proved without having to reason about the specific asynchronous behaviour of the nodes running the algorithm.

We start with the definition of an asynchronously contracting operator. At a high level, an ACO is guaranteed to convergence to a specific state x^*, regardless of the starting conditions.

Definition 8 (ACO)**.** *An operator* $\mathbf{F}$ *is an asynchronously contracting operator if, and only if, 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*

[2]Note that we can recover the synchronous algorithm σ by setting the schedule to be $(t \mapsto V, (t, i, j) \mapsto t - 1)$

- *(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)* $\exists k^*, x^*.\ \forall k.\ k^* \leq k \Rightarrow D(k) = \{x^*\}$

For an operator $\mathbf{F}$ to be an ACO, its state space S should be decomposable into a chain of boxes $D(k)$, where each box $D(k)$ is the set of states that are reachable at time t. Property *(A1)* requires the starting state box $D(0)$ to contain all possible states reachable (by applying $\mathbf{F}$) from any start state $x \in D(0)$. Property *(A2)* states that every application of $\mathbf{F}$ moves the state into the next box. Eventually, at time k^*, some singleton box will be reached *(A3)*. Indeed, the point of convergence x^* of the chain $\{D(k)\}$ is also the unique fixed point of $\mathbf{F}$ in $D(0)$.

The idea of *pseudocycles* is that each node is activated at least once during a pseudocycle, and after the end of a pseudocycle, the information that nodes use will not originate from the past pseudocycle.

Definition 9 (∞-pseudoperiodic)**.** *A schedule is ∞-pseudoperiodic if there exists functions $\varphi : \mathbb{N} \to \mathbb{T}$ and $\tau : V \to \mathbb{N} \to \mathbb{T}$ 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)* $\forall t, i, j, k.\ \varphi(k+1) < t \Rightarrow \tau_i(k) \leq \beta(t, i, j)$

The function φ defines pseudocycles of a schedule – the period between $\varphi(k)$ and $\varphi(k+1)$ is one pseudocycle. The function τ defines when each node i activates in a certain pseudocycle k.

Property *(P1)* forces the first pseudocycle to start at time 0. Properties *(P2)* and *(P3)* ensure that each node activates at least once between $\varphi(k)$ and $\varphi(k+1)$ – the duration of one pseudocycle. This guarantees that all nodes continue to activate indefinitely, and rules out schedules where certain nodes cease to activate. Property *(P4)* states that any information arriving after $\varphi(k+1)$ (the end of a pseudocycle) must have been sent after $\tau_i(k)$ – a time when node i activated in the previous pseudocycle. This property rules out schedules where a link between nodes fails and no more information is shared.

Daggitt et al. relax this class of schedules in their paper [9]. The definition above requires schedules to behave "properly" indefinitely. This is relaxed to schedules which are so-called *k^*-pseudoperiodic* – the schedules should behave properly up to the point k^* when

a fixed point is reached. This class of schedules is slightly larger than the class of ∞-pseudoperiodic schedules. The following example is included only in the new relaxed class of schedules:

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

which is a synchronous schedule until the convergence time k^*, after which all nodes stop communicating.

Theorem 1 (UD's theorem, relaxed by Daggitt et al.)**.** *If* $\mathbf{F}$ *is an ACO then* δ *converges over* $\boldsymbol{D}(0)$ *for* k^**-pseudoperiodic schedules.*

Proof. See Theorem 1 in [9]. □

2.4 Asynchronous Routing

The work in this dissertation utilises results by Daggitt et al. on the asynchronous convergence of distributed Bellman-Ford protocols [8]. Their paper makes three main contributions. First, they decouple the routing computations from the asynchronous context in which they occur. Second, they provide properties for routing algebras that are sufficient to guarantee asynchronous absolute convergence. Third, they formalise their results in the Agda theorem prover.

Üresin and Dubois' theorem is used to allow reasoning about the asynchronous behaviour of routing protocols (referred to as δ), by proving properties of their synchronous counterparts (referred to as σ). This simplifies the reasoning, as one does not need to worry about details of asynchronous computations, such as link failure, delayed messages, and more.

In their paper, the focus is on policy-rich Bellman-Ford routing protocols. This class of protocols includes the widely-used Border Gateway Protocols (BGP) [22] and the Routing Information Protocol (RIP) [14]. They take the algebraic approach to model the routing problem that these protocols aim to solve. In this framework, they consider two types of Bellman-Ford protocols, namely distance-vector (e.g. RIP) and path-vector protocols (e.g. BGP). For both, they give sufficient conditions for the routing algebras to guarantee convergence of the asynchronous protocols. For distance-vector protocols, a strictly increasing and finite routing algebra guarantees that the asynchonous computation δ converges absolutely (Theorem 2 in [8]). For path-vector protocols, where paths are used in the algebra to exclude routes with loops, an increasing path algebra is sufficient to guarantee asynchronous convergence of δ (Theorem 3 in [8]).

Another contribution of the paper is the formalisation of the above in the Agda theorem prover. The notion of routing algebras and the proofs which guarantee the asynchronous absolute convergence of the routing algorihms, are formalised. This includes the theorems by Üresin and Dubois [24] (convergence theorems for asynchronous algorithms) and by Gurney [13] (the existence of an ultrametric implies ACO). The Agda library [6] which contains all of these formalisations will be the starting point for the Agda formalisation of the results in this project.

2.5 Part II Dissertation

This dissertation builds on top of the work done for the Part II dissertation [25], in which we modelled routing protocols in a synchronous world. In this section we will explain the work done in the dissertation, and how it relates to this project.

The model defined in the dissertation is an approximation of the low-level mechanism. It models three principal implementation details of the class of hard-state vectoring routing protocols:

- The high-level algebraic model using a single routing matrix to represent the global routing state. In implementations of distributed routing protocols, there is no such notion of a shared central routing matrix containing all the optimal routes. The separation of state (called vectoring) is modelled by using an explicit per-router representation of the routing state.
- The abstract edge functions in the adjacency matrix A, are actually a composition of export, import and protocol specific functions. For example, the export or import functions could filter out routes and the protocol function could increment the hop count of a route.
- Routers use a so-called *hard-state* approach, as opposed to the soft-state approach [17]. They only share changes in their view of the network. A constant routing state results in no communication between routers.

The features are introduced successively to the high-level model Γ_0. First, the vectoring aspect is introduced, giving rise to the Γ_1 model. Then, the function composition feature is modelled on top of this, resulting in Γ_2. Finally, the hard-state aspect is introduced, and all features are modelled in the Γ_3 model (which is the approximation of the low-level mechanism). The main result of the dissertation (Theorem 5) is showing that Γ_0 is functionally equivalent to Γ_3 (i.e. they compute the same results, given equal inputs). The

proof structure used in the dissertation is to show a chain of functional equivalence

$$\Gamma_0 \rightsquigarrow \Gamma_1 \rightsquigarrow \Gamma_2 \rightsquigarrow \Gamma_3$$

and hence obtain overall functional equivalence $\Gamma_0 \rightsquigarrow \Gamma_3$. This approach provides a structured way of proving functional equivalence. It also allows the model of the low-level implementation to be extended, as more details can be modelled by appending another model Γ_{N+1} to the proof chain.

We will describe these synchronous models in more detail in Sections 4.1, 5.1, 6.1 and 7.1.

2.6 Summary

Agda is a functional programming language which can be used as a proof assistant. The algebraic approach to routing allows reasoning about the behaviour of routing protocols at a high-level. Üresin and Dubois' framework helps to tame the complexity of reasoning about the behaviour of asynchronous algorithms. Recent research by Daggitt et al. [8] and my Part II dissertation [25] constitute the starting point of this dissertation.

Chapter 3

Approach

The goal of the project is to show that the high-level algebraic model of the routing problem is capable of simulating the low-level mechanism of the family of distributed Bellman-Ford protocols, in an asynchronous world. In this chapter we outline the approach that we take to reach this goal.

In the Part II dissertation [25], we defined the synchronous models $\Gamma_0 \ldots \Gamma_3$. These models greatly simplified the actual behaviour of routing protocols. The reality is more complex: routers communicate asynchronously and therefore messages may be delayed, reordered or even lost. As a result, reasoning about asynchronous routing protocols is far harder than their synchronous counterparts.

This dissertation studies the behaviour of the routing protocols in the intricate asynchronous world. We study two types of asynchrony:

- Asynchrony between nodes. Messages can be delayed, reordered, duplicated, or even lost. This gives rises to anomalies such a non-convergence and other problems.
- Asynchrony inside nodes. Routers import incoming routes, process these, and export their own routes. Message queues exist inside routers which potentially delay messages.

We define the asynchronous models $\Omega_0 \ldots \Omega_3$. The difference with the synchronous models is that the asynchronous models are defined using schedules, which dictate how and when data gets shared between the routers. The notion of schedules was introduced by Üresin and Dubois [24], whose framework we will utilise to tame the complexity of reasoning about asynchronous algorithms.

We will show that the high-level model Ω_0 is capable of simulating the mechanics of the

low-level model Ω_2 of asynchronous distributed Bellman-Ford protocols. To do so, we show that Ω_0 can simulate Ω_1, and that Ω_1 can simulate Ω_2. Additionally, we found that the Ω_2 model is unable to simulate the Ω_3 model of asynchronous hard-state protocols. In Chapter 7 we construct a counterexample which illustrates this.

We start by defining what constitutes a *simulation*: a higher-level model simulates a lower-level model when the computations of the lower-level model can be mimicked by the higher-level model.

The behaviour of an asynchronous process is fully determined by the initial state and the schedule. Therefore, to relate two models, we define a *transformation* for the routing state, and a *schedule reduction*.

A lower-level model is by definition more detailed. As a result, it has a more complex routing state. A transformation will give an interpretation of the lower-level state, in the world of the higher-level model.

Definition 10 (Transformation)**.** *A transformation $\mathcal{T}_j : \Omega_j \to \Omega_i$ takes a routing state for the Ω_j model, and transforms it to a routing state for the higher-level Ω_i model ($i < j$). The transformation function gives a higher-level interpretation of the routing state.*

To simulate the lower-level process with a certain schedule, we need to give an appropriate schedule in the higher-level model.

Definition 11 (Schedule reduction)**.** *A reduction $r_j : S_j \to S_i$ takes a schedule ψ for some lower-level model Ω_j and transforms it to a schedule ψ' for some higher-level model Ω_i.*

With the definitions of a transformation and a schedule reduction at hand, we can formally define what it means for a higher-level model to simulate a lower-level model.

Definition 12 (Simulation)**.** *A higher-level model Ω_i can simulate a lower-level model Ω_j when there exists a transformation $\mathcal{T}_j$ and a schedule reduction r_j, such that the reduction preserves equivalence under the transformation, for all schedules ψ and starting states x.*

$$\mathcal{T}_j\big(\Omega_j\,(\psi)\,(x)\big) = \Omega_i\,\big(r_j(\psi)\big)\,\big(\mathcal{T}_j(x)\big)$$

Whenever a model Ω_j is able to simulate a model Ω_i, we obtain a relationship between the two models in the form of a commutativity diagram. Figure 3.1 illustrates this relationship.

Computing the results of the Ω_j model, and then interpreting these results in the higher-level world, is equivalent to transforming the starting state and schedule into the higher-level world and then computing the results of the Ω_i model.

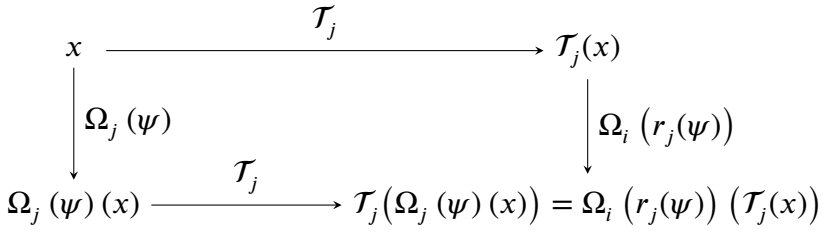

Figure 3.1: Commutativity of a lower-level model Ω_j and a higher-level model Ω_i, for some starting state x and schedule ψ. The Ω_i model simulates Ω_j using a transformation $\mathcal{T}_j$ and a schedule reduction r_j.

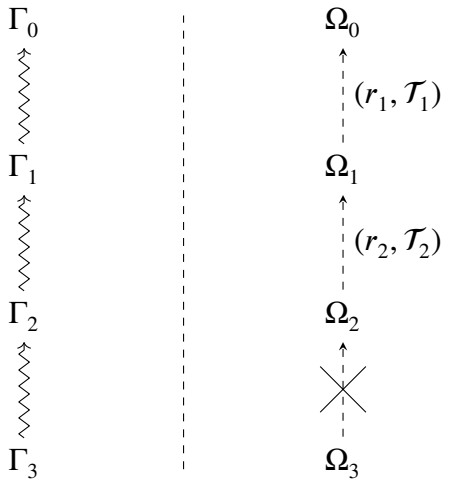

Figure 3.2: Relation between the synchronous and asynchronous worlds. An edge $\Gamma_i \rightsquigarrow \Gamma_j$ denotes that Γ_i is functionally equivalent to Γ_j. An edge $\Omega_i \dashleftarrow \Omega_j$ denotes that Ω_i can simulate Ω_j, using some transformation and schedule reduction.

In this project, we show that Ω_0 can simulate Ω_1, and that Ω_1 can simulate Ω_2. Furthermore, we show that Ω_2 cannot simulate Ω_3 by constructing a counterexample for which there cannot exist a valid schedule reduction. Figure 3.2 summarises the results in the synchronous world (from the Part II dissertation [25]) and the asynchronous world (this project).

The approach used in this project to prove our results (the chain $\Omega_0, \ldots, \Omega_2$ of functionally equivalent models) is extensible, and allows for more details to be modelled in future research.

Chapter 4

The Ω_0 Model: High-level

We take the algebraic approach to routing, building on previous work by Daggitt et al. [8]. Their definitions provided the starting point for the Part II dissertation [25], and consequently will be at the basis of this project too.

4.1 Synchronous

The high-level model of the routing problem is referred to as Γ_0 in the Part II dissertation. It abstracts away implementation details such as communication between routers, message queues inside routers, protocol-specific mechanisms and, indeed, asynchronicity.

A *routing algebra* $(S, \oplus, F, \bar{0}, \bar{\infty})$, as explained in Section 2.2, represents the problem at hand. It consists of a carrier set S, which is the set of routes between nodes (including the *trivial route* $\bar{0}$ from a node to itself, and the *invalid route* $\bar{\infty}$). The binary operator $\oplus$ is the *choice operator*, which takes two routes and returns the best one. The set F contains the set of edge functions $S \to S$, which take a route and extend it by one edge.

The routing state is represented by an $n \times n$ *routing matrix* Y over S, where n is the number of nodes. Each element $Y(i, j)$ is the value of the so-far optimal route from node i to node j.

The network topology is represented by an *adjacency matrix* A over F. Each edge $i \to q$ is represented by an edge function $A(i, q) : S \to S$ from F. Missing edges are represented by the constant function $f(s) = \bar{\infty}$. For example, let $s_{q \to j}$ be a route from node q to node j. This route can be extended along the edge $i \to q$, by applying the edge function to it. The result is a new route $A(i, q)\left(s_{q \to j}\right)$, from node i through q to j.

Finding a solution to the routing problems constitutes to finding a routing matrix X, such

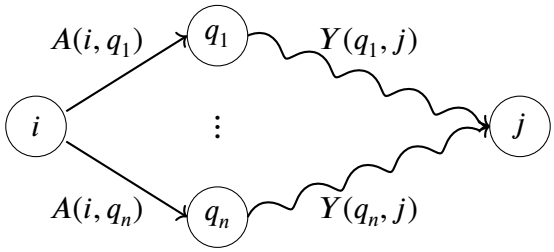

Figure 4.1: The application $A(Y)(i, j)$ of the adjacency matrix A to the routing state Y. The route $Y(q_k, j)$ is the best route found so far from node q_k to j.

that each route $X(i, j)$ is the optimal route from node i to j. Such a solution is computed by iteratively applying the adjacency matrix A to the current state Y. The matrix application is the result of each router choosing the best extension of routes offered by its neighbours (see Figure 4.1).

$$A(Y)(i, j) = \bigoplus_q A(i, q)\big(Y(q, j)\big) \tag{4.1}$$

The Γ_0 model represents one *synchronous* round of distributed Bellman-Ford computations.

$$\Gamma_0(Y) = A(Y) \oplus I \tag{4.2}$$

where I is the identity matrix, containing trivial routes along the diagonal (routes from nodes to themselves), and invalid routes everywhere else.

A solution to the routing problem is a routing state X in which each router has a *locally optimal route* – it cannot improve by unilaterally switching routes.

$$X = A(X) \oplus I \tag{4.3}$$

4.2 Asynchronous

The asynchronous version Ω_0 of the high-level model Γ_0 can be obtained by integrating the notion of a schedule. The Ω_0 model arises by changing the recursive call to the appropriate time (as dictated by the data flow function β), and by case splitting on whether a node i is being activated at time t (as dictated by the activation function α). We introduce some new operators to incorporate the activation function α and the data flow function β into the mechanics.

Definition 13 (Choice operator)**.** *The choice operator* $[\_,\_]_{\_}$ *takes two indexed sets* $X, Y : I \to Set$ *and a subset of the indices* $S \subseteq I$*, and returns a new indexed set. At each index of the new set, either one of the two sets is returned, depending on whether the index is an element of* S*.*

$$[X,Y]_S(i) = \begin{cases} X(i) & \text{if } i \in S \\ Y(i) & \text{otherwise} \end{cases}$$

This operator is used to select which data to take, depending on the activation function. It allows us to express asynchronous computations in a more algebraic fashion.

Lemma 1 (Active and inactive choice operator)**.**

$$[X,Y]_I = X \quad [X,Y]_\emptyset = Y$$

Proof. The results immediately follow from the definition of the choice operator. □

We generalise function matrix application to a matrix (as defined in the previous section, Eq. 4.1), to function matrix application to a triple-indexed function. Let us define the application as follows:

$$\big(A(f)\big)(i,j) = \bigoplus_q A(i,q)(f(i,q,j))$$

For $f(i,q,j) = X(q,j)$, we obtain our original matrix application $A(X)$.

Using the newly defined choice operator and the generalised matrix application, we can define the asynchronous model Ω_0 as follows:

$$\begin{aligned} \Omega_0^0(X) &\equiv X \\ \Omega_0^{t+1}(X) &\equiv \left[A(\Omega_0^{\beta(t+1)}(X)) \oplus I,\ \Omega_0^t(X)\right]_{\alpha(t+1)} \end{aligned} \tag{4.4}$$

How the schedules are incorporated into the model becomes clear when looking at individual elements of the resulting routing matrix.

$$\begin{aligned} \Omega_0^0(X)(i,j) &\equiv X(i,j) \\ \Omega_0^{t+1}(X)(i,j) &\equiv \begin{cases} \left(\bigoplus_q A(i,q)(\Omega_0^{\beta(t+1,i,q)}(X)(q,j))\right) \oplus I(i,j) & \text{if } i \in \alpha(t+1) \\ \Omega_0^t(X)(i,j) & \text{otherwise} \end{cases} \end{aligned} \tag{4.5}$$

We denote the type of schedules for Ω_0 with $\mathcal{S}_0$. It is a tuple (α, β) of an activation function α and a data flow function β.

Definition 14 (Synchronous schedule)**.** *The synchronous schedule ψ^{sync} is defined to be the schedule where all nodes update at all time, and receive data from other nodes instantly.*

$$\psi^{sync} = \begin{cases} \alpha(t) & = \{1 \dots n\} \\ \beta(t,i,j) & = t - 1. \end{cases}$$

Lemma 2 (Synchronous Ω_0)**.** *One can recover Γ_0 from Ω_0 by using the synchronous schedule ψ^{sync}.*

$$\forall t, X.\, \Omega_0^t\,(\psi^{sync})\,(X) = \Gamma_0^t\,(X)$$

Proof. Let X be some initial routing matrix.

We prove by induction on $t \in \mathbb{N}$.

Base case t = 0. Trivially holds by the definitions of Ω_0 and Γ_0.

Inductive step t + 1.

$$\begin{aligned} \Omega_0^{t+1}\,(\psi^{sync})\,(X) &= \left[A(\Omega_0^{\beta(t+1)}\,(X)) \oplus I,\ \Omega_0^t\,(X)\right]_{\alpha(t+1)} \quad \text{(by def. of } \Omega_0\text{)} \\ &= \left[\Gamma_0\big(\Omega_0^{\beta(t+1)}\,(X)\big),\ \Omega_0^t\,(X)\right]_{\alpha(t+1)} \quad \text{(by def. of } \Gamma_0\text{)} \\ &= \Gamma_0\big(\Omega_0^t\,(X)\big) \quad \text{(by def. of } \psi^{sync}\text{)} \\ &= \Gamma_0^{t+1}\,(X) \quad \text{(by } \mathbf{IH}\text{)} \end{aligned}$$

The Agda proof of this lemma is similar to the above "pen and paper" proof.

```
Ω₀'sync=Γ₀ : ∀ X {t} (acc_t : Acc _<_ t) → Ω₀' ψsync X acc_t ≈ₘ (Γ₀ ^ t) X
Ω₀'sync=Γ₀ X {zero} _          = ≈ₘ-refl
Ω₀'sync=Γ₀ X {suc t} (acc rec) = begin
  Ω₀' ψsync X (acc rec)          ≡⟨⟩
  [ Γ₀ X[t] , X[t] ] αsync (suc t) ≡⟨ [,]-T ⟩
  Γ₀ X[t]                        ≈⟨ Γ₀-cong (Ω₀'sync=Γ₀ X acc[t]) ⟩
  (Γ₀ ^ (suc t)) X               ■
  where open EqReasoning ℝ𝕄ₛ
        acc[t] : Acc _<_ t
        acc[t] = rec t ≤-refl
        X[t] : RoutingMatrix
        X[t] = Ω₀' ψsync X acc[t]
```

The most significant difference is that the Agda proof requires an *accessible argument*[1] Acc _<_ t. It is clear to the human reader that this function terminates, as each recursive call using the data flow function β has a smaller argument (property S1 from Definition 6: $\beta(t,i,j) < t$). However, Agda's termination checker fails to detect that this automatically. The Agda user needs to assist Agda in its search to prove termination. The Agda standard library [3] has the datatype Acc, which can be used to convince Agda's termination checker. The Acc datatype has an argument (*rec* in our proof) which becomes structurally smaller.

Note that acc[t] is a shorthand for this accessible argument, and that X[t] is a shorthand for the Ω_0 routing state after t iterations. The term [,]-T is the Agda implementation of Lemma 1.

The complexity of the accessible argument is hidden in $\Omega_0{}^{sync}{=}\Gamma_0$, by using the proof <-wellFounded that the natural numbers are *well-founded* (all numbers are accessible).

$$\Omega_0{}^{sync}{=}\Gamma_0 : \forall\, X\, t \to \Omega_0\ \psi^{sync}\ X\, t \approx_m (\Gamma_0\ \hat{}\ t)\ X$$
$$\Omega_0{}^{sync}{=}\Gamma_0\ X\, t\ =\ \Omega_0{}^{'sync}{=}\Gamma_0\ X\ (<\text{-wellFounded}\ t)$$

□

In this chapter, we defined the Ω_0 model. It is the high-level model of asynchronous routing protocols. We also introduced the choice operator [_, _] , which allows us to express asynchronous computations in a more algebraic fashion. Finally, we showed that the asynchronous Ω_0 model, running with the synchronous schedule ψ^{sync}, is equal to the synchronous model Γ_0.

The next chapters will define increasingly more detailed models of implementations of asynchronous distributed Bellman-Ford protocols, and show that Ω_0 is capable of simulating these lower-level models.

[1] The mathematical definition is that an element a in a set A is accessible for some binary relation $<$ over the set A, if there is no infinite descending chain starting at a.

Chapter 5

The Ω_1 Model: Vectoring

In this chapter, we model the vectoring feature of distributed Bellman-Ford protocols.

The routing state in the high-level model is represented using a $n \times n$ routing matrix. The vectoring aspect is modelled by transforming this global view to an explicit per-router representation of the routing state.

The routing state is a *routing vector* V of sets, instead of a matrix. Each $V(i)$ will be a *routing set* of (destination, value) pairs. This corresponds to a single node, and its route values to other nodes. We have that $V(i)$ is a set $\{(d_1, s_1), \ldots, (d_n, s_n)\}$, where the d_is (destinations) are unique and $s_i \neq \infty$. Each pair (d_j, s_j) represents the route s_j from node i to node j. The routing sets and routing vectors are implemented in Agda as follows:

```
RoutingSet : Set
RoutingSet = List (Fin n × Route)

RoutingVector : Set
RoutingVector = Vector RoutingSet n
```

A RoutingSet is a list of (Fin n × Route) pairs, where Fin n is the type of natural numbers less than n and Route is the type of route values. A RoutingVector is then simply a Vector of RoutingSets of length n.

5.1 Synchronous

We start by defining matrix application for routing vectors. It is similar to the matrix application for routing matrices (like Γ_0).

$$A \langle\!| V |\!\rangle (i) \equiv \widetilde{\bigoplus_q} A(i,q)[V(q)]$$

where function application to routing sets $f[X]$ is defined to apply the function f to all the routes, and filter out those that are invalid

$$f[X] \equiv \{(d, s') \mid (d, s) \in X, s' = f(s), s' \neq \infty\}$$

and where the operator $\widetilde{\bigoplus}$ is the choice operator for routing sets, returning a set of preferred (destination, route value) pairs.

Finally, we need an operator to transform a routing matrix to a routing vector.

$$\widetilde{X}(q) \equiv \{(d, s) \mid X(q, d) = s, s \neq \infty\} \tag{5.1}$$

Using these newly defined operators for the per-router representation of the routing state, we can define the Γ_1 model.

$$\Gamma_1(V) \equiv A(\!|V|\!)\,\widetilde{\bigoplus}\,\widetilde{I}$$

In the Part II dissertation, we showed the functional equivalence of this model and the high-level Γ_0 model.

Theorem 2 ($\Gamma_0 \rightsquigarrow \Gamma_1$)**.** *The models Γ_0 and Γ_1 are functionally equivalent.*

$$\Gamma_1(\widetilde{Y}) = \widetilde{\Gamma_0(Y)}$$

Proof. See Theorem 1 in [25]. □

5.2 Asynchronous

In the Ω_1 model, we capture asynchrony for the vectoring feature, as represented by Γ_1.

We obtain the asynchronous version Ω_1 in a similar way as Ω_0. A schedule S_1 for Ω_1 is equivalent to a schedule S_0 in Ω_0. As before, we generalise a matrix application operator. For the Ω_1 model, we take the matrix application for routing vectors, and generalise it.

$$A(\!|f|\!)_i = \widetilde{\bigoplus_q} A_{iq}(f(i, q))$$

Setting $f(i, q) = V(q)$ gives the original matrix application for routing vectors $A(\!|V|\!)$.

Now, we can define the Ω_1 model.

$$\begin{aligned} \Omega_1^0(V) &\equiv V \\ \Omega_1^{t+1}(V) &\equiv \left[A(\!|\Omega_1^{\beta(t+1)}(V)|\!)\ \widetilde{\bigoplus}\ \widetilde{I},\ \ \Omega_1^t(V)\right]_{\alpha(t+1)} \end{aligned} \tag{5.2}$$

Inspecting the individual elements, we get:

$$\begin{aligned}\Omega_1^0(V)_i &\equiv V_i \\ \Omega_1^{t+1}(V)_i &\equiv \begin{cases} \left(\widetilde{\oplus}_k A_{ik}[\Omega_1^{\beta(t+1,i,k)}(V)_k]\right) \widetilde{\oplus} \widetilde{I}_i & \text{if } i \in \alpha(t+1) \\ \Omega_1^t(V)_i & \text{otherwise} \end{cases}\end{aligned} \tag{5.3}$$

Lemma 3 (Synchronous Ω_1)**.** *The Ω_1 model using the synchronous schedule ψ^{sync} is equal to the synchronous model Γ_1.*

$$\forall t, V.\ \Omega_1^t\left(\psi^{sync}\right)(V) = \Gamma_1^t(V)$$

Proof. See Appendix A.1. □

5.2.1 Reduction $\Omega_1 \to \Omega_0$

The difference between the models Ω_0 and Ω_1 is the representation of the routing state. In Ω_0 the routes are stored in a routing matrix X – each $X(i,j)$ represents a route from node i to j. In Ω_1 the routes are stored in a vector V of routing sets – each $V(i)$ contains a set of (destination, route value) pairs (i.e. the pair (j,s) in $V(i)$ represents a route from node i to j with a value s).

The transformation function $\mathcal{T}_1 : \Omega_1 \to \Omega_0$ takes the routing vector state, and transforms it to the higher-level routing matrix state. We use "–" to denote this transformation. It is the inverse of the function $\sim$, which transforms a routing matrix to a routing vector. The transformation uses the routing set function lookup_d, which looks up a destination in a routing set, returning $\bar{\infty}$ if a destination is not present.

$$\left(\overline{\mathcal{V}}\right)_{ij} \equiv \text{lookup}_d(\mathcal{V}(i), j)$$

where

$$\begin{aligned}\text{lookup}_d(\emptyset, j) &= \bar{\infty} \\ \text{lookup}_d(\{(d,s)\} \cup S, j) &= \begin{cases} s & \text{if } d = j \\ \text{lookup}_d(S,j) & \text{otherwise} \end{cases}\end{aligned}$$

Indeed, we have that the transformation function – is an inverse of the function $\sim$.

Lemma 4 (Inverse – and $\sim$)**.** *For all routing matrices X and routing vectors $\mathcal{V}$, we have that*

$$\overline{(\widetilde{X})} = X \qquad \widetilde{(\overline{\mathcal{V}})} = \mathcal{V}$$

Proof. See Appendix A.1. □

It follows that the collection of routing matrices and the collection of routing vectors are *isomorphic* – the difference between Ω_0 and Ω_1 is truly merely a change in representation.

As the – and $\sim$ are inverse operators, we can deduce properties of – that are duals of properties of $\sim$. In the Part II dissertation [25], Lemma 3.3 and Lemma A.1 are distributivity properties of $\sim$ over $A(\_)$ and $\oplus$, respectively.

$$\begin{aligned}\widetilde{A(X)} &= A(\!|\widetilde{X}|\!) \quad \text{(Lemma 3.3 from [25])}\\ \widetilde{X \oplus Y} &= \widetilde{X} \mathbin{\widetilde{\oplus}} \widetilde{Y} \quad \text{(Lemma A.1 from [25])}\end{aligned}$$

Their dual properties hold for –.

Lemma 5 (Distributivity of – over $A(\!|\_|\!)$ and $\widetilde{\oplus}$)**.**

$$\overline{A(\!|V|\!)} = A(\overline{V}) \qquad \overline{X \mathbin{\widetilde{\oplus}} Y} = \overline{X} \oplus \overline{Y}$$

Proof.

$$\begin{aligned}\overline{X \mathbin{\widetilde{\oplus}} Y} &= \overline{\widetilde{\overline{X}} \mathbin{\widetilde{\oplus}} \widetilde{\overline{Y}}} \quad \text{(by inverse of – and } \sim\text{)}\\ &= \overline{\widetilde{\left(\overline{X} \oplus \overline{Y}\right)}} \quad \text{(by Lemma A.1 from [25])}\\ &= \overline{X} \oplus \overline{Y} \quad \text{(by inverse of – and } \sim\text{)}\end{aligned}$$

The proof of the distributivity of – over $A(\!|\_|\!)$ is similar. □

Finally, to show how the asynchronous models Ω_0 and Ω_1 are equivalent under a reduction, we will need the property that – distributes over $[\_,\_]_\_$.

Lemma 6 (Distributivity of – over choice operator)**.**

$$\overline{[U,V]_S} = \left[\overline{U},\overline{V}\right]_S$$

Proof. See Appendix A.1. □

The dynamics of the routing model does not change between Γ_0 and Γ_1. It is merely a change in representation. This is also true in the asynchronous world. Therefore, the reduction from Ω_1 to Ω_0 is simply the identity reduction.

Definition 15 (Schedule reduction r_1)**.**

$$r_1 \ : \ \mathcal{S}_1 \to \mathcal{S}_0$$
$$r_1(\alpha, \beta) = (\alpha, \beta)$$

Theorem 3 ($\Omega_0 \twoheadleftarrow \Omega_1$)**.** *The Ω_0 model can simulate the Ω_1 model, using the transformation $\mathcal{T}_1$ and schedule reduction r_1. For all times t, routing vectors $\mathcal{V}$ and all schedules ψ of type $\mathcal{S}_1$, we have that*

$$\mathcal{T}_1\big(\Omega_1^t\,(\psi)\,(\mathcal{V})\big) = \Omega_0^t\,(r_1(\psi))\,(\mathcal{T}_1(\mathcal{V}))$$

Proof. We prove using strong induction on $t \in \mathbb{N}$.

Base case $t = 0$:

$$\begin{aligned}
\mathcal{T}_1\big(\Omega_1^0\,(\psi)\,(\mathcal{V})\big) &= \mathcal{T}_1\,(\mathcal{V}) && \text{(by def. of } \Omega_1\text{)} \\
&= \Omega_0^0\,(r_1(\psi))\,\big(\mathcal{T}_1(\mathcal{V})\big) && \text{(by def. of } \Omega_0\text{)}
\end{aligned}$$

Inductive step:

$$\begin{aligned}
\mathcal{T}_1\big(\Omega_1^t(\psi)(\mathcal{V})\big) &= \overline{\Omega_1^t(\psi)(\mathcal{V})} && \text{(by def. of } \mathcal{T}_1\text{)} \\
&= \overline{\big[A\,(\!|\Omega_1^{\beta(t)}(\mathcal{V})|\!)\ \widetilde{\oplus}\ \widetilde{I},\ \Omega_1^{t-1}(\mathcal{V})\big]_{\alpha(t)}} && \text{(by def. of } \Omega_1\text{)} \\
&= \big[\overline{A\,(\!|\Omega_1^{\beta(t)}(\mathcal{V})|\!)\ \widetilde{\oplus}\ \widetilde{I}},\ \overline{\Omega_1^{t-1}(\mathcal{V})}\big]_{\alpha(t)} && \text{(by Lemma 6)} \\
&= \big[\overline{A\,(\!|\Omega_1^{\beta(t)}(\mathcal{V}))|\!)} \oplus \overline{(\widetilde{I})},\ \overline{\Omega_1^{t-1}(\mathcal{V})}\big]_{\alpha(t)} && \text{(by Lemma 5)} \\
&= \big[A\big(\overline{\Omega_1^{\beta(t)}(\mathcal{V})}\big) \oplus I,\ \overline{\Omega_1^{t-1}(\mathcal{V})}\big]_{\alpha(t)} && \text{(by Lemma 5)} \\
&= \big[A\big(\Omega_0^{\beta(t)}(\overline{\mathcal{V}})\big) \oplus I,\ \Omega_0^{t-1}(\overline{\mathcal{V}})\big]_{\alpha(t)} && \text{(by IH, and } \beta(t) < t\text{)} \\
&= \Omega_0^t(r_1(\psi))(\overline{\mathcal{V}}) && \text{(by def. of } \Omega_0\text{)} \\
&= \Omega_0^t(r_1(\psi))(\mathcal{T}_1(\mathcal{V})) && \text{(by def. of } \mathcal{T}_1\text{)}
\end{aligned}$$

□

In this chapter, we defined the Ω_1 model, which captures the vectoring aspect of distributed Bellman-Ford protocols. Furthermore, we showed that the high-level model Ω_0 is indeed capable of simulating the mechanics of Ω_1, by giving an appropriate transformation and schedule reduction.

Chapter 6

The Ω_2 Model: Composition

In this chapter, we model the composition feature of implementations of distributed Bellman-Ford protocols – routing computations consist of three steps: importing routes, selecting the best routes, and exporting routes.

6.1 Synchronous

In the high-level algebraic approach, every edge $i \to j$ is labelled with an edge function $A(i,j) : S \to S$. For Γ_2, we break through one layer of abstraction. Each edge function $A(i,j)$ models what actually is a composition of an export function $\mathrm{Exp}(j,i)$ (computations performed on outgoing routes), a protocol function $\mathrm{Prot}(i,j)$ (protocol specific computations) and an import function $\mathrm{Imp}(i,j)$ (computations performed on ingoing routes). We can express the adjacency matrix in terms of the export, import and protocol matrices:

$$A \equiv \mathrm{Imp} \circ \mathrm{Prot} \circ \mathrm{Exp}^T$$

Instead of applying the adjacency matrix to the routing vector, the application is now distributed over export, routing and import tables. The Γ_2 state is a triple $(\mathcal{V}, \mathcal{I}, \mathcal{O})$. The routing table $\mathcal{V}$ contains the best routes found so far. The import table $\mathcal{I}$ contains the imported routes. The export table $\mathcal{O}$ contains the exported routes. Figure 6.1 visualises the composition mechanism.

One cycle of the original routing computation now consists of three steps: exporting the current routes, importing the exported routes, and incorporating the imported routes into the new routing state. The Γ_2 model consists of three mutually dependent sub-models $\Gamma_{2,\mathcal{V}}$, $\Gamma_{2,\mathcal{I}}$ and $\Gamma_{2,\mathcal{O}}$:

$$\Gamma_2(\mathcal{V}, \mathcal{I}, \mathcal{O}) \equiv \left(\Gamma_{2,\mathcal{V}}(\mathcal{I}),\ \Gamma_{2,\mathcal{I}}(\mathcal{O}),\ \Gamma_{2,\mathcal{O}}(\mathcal{V})\right)$$

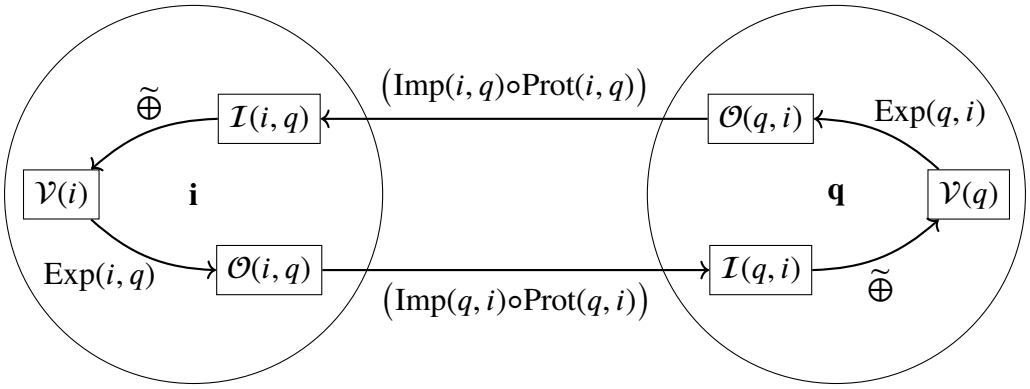

Figure 6.1: Function composition in Γ_2

The $\Gamma_{2,\mathcal{O}}$ function takes the interal routing state, and applies the export functions to it, placing the results in the output table $\mathcal{O}$. An example of such an export function $\mathrm{Exp}(i,j)$ is one which invalidates route going to competing Internet Service Providers.

$$\Gamma_{2,\mathcal{O}}(\mathcal{V}) \equiv \mathrm{Exp}\,(\!|\mathcal{V}|\!)$$

where the $F\,(\!|\mathcal{V}|\!)$ operator applies the function matrix to all the routing sets in the routing vector.

$$F\,(\!|\mathcal{V}|\!)(i,q) \equiv F(i,q)[\mathcal{V}(i)]$$

The $\Gamma_{2,\mathcal{I}}$ function takes the exported routes and applies the protocol and import functions to them, storing the results in the input table $\mathcal{I}$. An example of such a protocol function $\mathrm{Prot}(i,j)$ is one which increments the hop count of a route on each edge extension.

$$\Gamma_{2,\mathcal{I}}(\mathcal{O}) \equiv (\mathrm{Imp} \circ \mathrm{Prot})(\!|\mathcal{O}|\!)$$

where the $F(\!|\mathcal{O}|\!)$ operator applies the function matrix to all the routing sets in the transpose routing matrix $\mathcal{O}$.

$$F(\!|\mathcal{O}|\!)(i,q) \equiv F(i,q)[\mathcal{O}(q,i)]$$

Finally, the $\Gamma_{2,\mathcal{V}}$ function takes in the imported routes, chooses the best ones and incorporates them into the routing state $\mathcal{V}$.

$$\Gamma_{2,\mathcal{V}}(\mathcal{I}) \equiv \mathcal{I}^{\downarrow} \,\widetilde{\oplus}\, \widetilde{I}$$

where $\widetilde{I}$ contains the trivial zero-distance routes, and the operator $\mathcal{I}^{\downarrow}$ takes a matrix of routing sets and chooses the best routes available.

$$\mathcal{I}^{\downarrow}(i) \equiv \widetilde{\bigoplus_{q}} \mathcal{I}(i, q)$$

At each application of Γ_2, the information from one state table is propagated to another. Indeed, after three application of Γ_2, information has a completed one cycle. A single application of Γ_1 is functionally equivalent to a triple application of Γ_2.

Theorem 4 ($\Gamma_1 \rightsquigarrow \Gamma_2$)**.** *The models* Γ_1 *and* Γ_2 *are functionally equivalent.*

$$\Gamma_2{}^{\langle 3k \rangle}(\widetilde{I}, \emptyset, \emptyset) = \left(\Gamma_1{}^{\langle k \rangle}(\widetilde{I}), \mathcal{I}', \mathcal{O}'\right)$$

where $\mathcal{I}' = (\Gamma_{2,\mathcal{I}} \circ \Gamma_{2,\mathcal{O}})(\Gamma_1{}^{\langle k-1 \rangle}(\widetilde{I}))$*,* $\mathcal{O}' = \Gamma_{2,\mathcal{O}}(\Gamma_1{}^{\langle k-1 \rangle}(\widetilde{I}))$*, and where* $f^{\langle k \rangle}(x) = \left(\underbrace{f \circ \ldots \circ f}_{k}\right)(x)$.

Proof. See Theorem 3 in [25]. □

The essential lemma used in the proof of this theorem is that consecutively applying $\Gamma_{2,\mathcal{O}}$, $\Gamma_{2,\mathcal{I}}$ and $\Gamma_{2,\mathcal{V}}$ is equivalent to a single application of Γ_1.

Lemma 7.

$$\Gamma_1(\mathcal{V}) = \left(\Gamma_{2,\mathcal{V}} \circ \Gamma_{2,\mathcal{I}} \circ \Gamma_{2,\mathcal{O}}\right)(\mathcal{V})$$

Proof. See Lemma 3.6 in [25]. □

6.2 Asynchronous

The asynchrony in the Ω_2 model arises on the communication between components – both inside and between nodes. Asynchrony inside nodes can be due to queues that exist between the components inside a node. For example, there might be a queue between the import table and the routing table of a node. Asynchrony between nodes can be caused by latency on links, the reordering of packets sent over a network, and various other reasons that typically introduce asynchrony between nodes. Note that, since asynchronous computations are a generalisation of synchronous computations, we can obtain an asynchronous model without asynchrony inside nodes by simply setting the schedule for internal communication to be synchronous.

A schedule ψ (as defined in Section 2.3, Def. 6) models the asynchrony that occurs between nodes. It captures the notions of data sharing (using the data flow function β) between nodes and the activation of nodes (using the activation function α). However, this

definition of a schedule is not granular enough to model the asynchrony inside nodes.

To model asynchrony both inside and between nodes, we use a triple of schedules (ψ_V, ψ_I, ψ_O) for our Ω_2 model – one for each component $\mathcal{V}$, $\mathcal{I}$, $\mathcal{O}$. For example, if $i \notin \alpha_I(t)$, it means that at time t, node i will not update its input table $\mathcal{I}$. Using this more granular schedule, we can define the Ω_2 model. Note that $\mathcal{V}^k$ is shorthand for $\Omega_2^k(\mathcal{V}, \mathcal{I}, \mathcal{O}).\mathcal{V}$ – the routing table after k iterations of Ω_2.

$$\begin{aligned}
&\Omega_2^0 : (\mathcal{V}^0, \mathcal{I}^0, \mathcal{O}^0) \equiv (\mathcal{V}, \mathcal{I}, \mathcal{O}) \\
&\Omega_2^{t+1} : \begin{cases}
\mathcal{V}^{t+1} & \equiv \left[\Gamma_{2,\mathcal{V}}(\mathcal{I}^{\beta_V(t+1)}),\ \mathcal{V}^t\right]_{\alpha_V(t+1)} = \left[(\mathcal{I}^{\beta_V(t+1)})^{\downarrow} \widetilde{\oplus}\ \widetilde{I},\ \mathcal{V}^t\right]_{\alpha_V(t+1)} \\
\mathcal{I}^{t+1} & \equiv \left[\Gamma_{2,\mathcal{I}}(\mathcal{O}^{\beta_I(t+1)}),\ \mathcal{I}^t\right]_{\alpha_I(t+1)} = \left[(\mathrm{Imp} \circ \mathrm{Prot})(\!|\mathcal{O}^{\beta_I(t+1)}|\!),\ \mathcal{I}^t\right]_{\alpha_I(t+1)} \\
\mathcal{O}^{t+1} & \equiv \left[\Gamma_{2,\mathcal{O}}(\mathcal{V}^{\beta_O(t+1)}), \mathcal{O}^t\right]_{\alpha_O(t+1)} = \left[\mathrm{Exp}(\!|\mathcal{V}^{\beta_O(t+1)}|\!), \mathcal{O}^t\right]_{\alpha_O(t+1)}
\end{cases}
\end{aligned} \tag{6.1}$$

where the asynchronous component $\mathcal{I}(i,k)$, the routing set of routes that node i receives from node k, unfolds to the following:

$$\mathcal{I}_{ik}^{t+1} = \begin{cases} (\mathrm{Imp} \circ \mathrm{Prot})[\mathcal{O}_{ki}^{\beta_I(t+1,i,k)}] & \text{if } i \in \alpha_I(t+1) \\ \mathcal{I}_{ik}^t & \text{otherwise} \end{cases}$$

If the input table of node i is activated at time $t+1$, then it applies the protocol *Prot* and import *Imp* functions to the exported routing set $\mathcal{O}_{ki}^{\beta(t+1,i,k)}$ of node k (at time $\beta_I(t+1,i,k)$). Else, node i keeps its import routing table state $\mathcal{I}_{ik}^t$ from the previous iteration.

Lemma 8 (Synchronous Ω_2)**.** *The Ω_2 model using the synchronous schedule ψ^{sync} is equal to the synchronous model Γ_2.*

$$\forall t, \mathcal{V}, \mathcal{I}, \mathcal{O}.\ \Omega_2^t\,(\psi^{sync})\,(\mathcal{V}, \mathcal{I}, \mathcal{O}) = \Gamma_2^t\,(\mathcal{V}, \mathcal{I}, \mathcal{O})$$

Proof. See Appendix A.2. □

6.2.1 Reduction $\Omega_2 \to \Omega_1$

The Ω_2 model has three components: the routing table $\mathcal{V}$, the import table $\mathcal{I}$ and the export table $\mathcal{O}$, as opposed to the Ω_1 model which has a single routing table. The extra components in Ω_2 are $\mathcal{I}$ and $\mathcal{O}$. They contain left-over computations, that arise from the fact that Ω_2 models more implementation details (namely the explicit import and export of routes from and to other nodes).

We aim to show that Ω_2 can be simulated by Ω_1. To do so, we define a schedule reduction $r_2 : \Omega_2 \to \Omega_1$, which preserves equivalence under some appropriate transformation $\mathcal{T}_2$.

The transformation $\mathcal{T}_2 : \Omega_2 \to \Omega_1$ is a higher-level interpretation of the routing state of Ω_2 into the world of Ω_1. To interpret the state $(\mathcal{V}, \mathcal{I}, \mathcal{O})$ we simply discard the extra routing tables.

$$\mathcal{T}_2(\mathcal{V}, \mathcal{I}, \mathcal{O}) = \mathcal{V}$$

To reduce a schedule ψ for the Ω_2 model to a schedule for the Ω_1 model, we need to understand how the two models are related.

In Ω_1, the routing table is updated in a single step – routes from adjacent nodes are extended by applying the appropriate edge functions, and the best routes are selected. In Ω_2, this computation is decomposed into three steps, as illustrated in Figure 6.1.

Let us consider how a node i obtains some route s from adjacent node q to destination node j. The route is stored in the routing table $\mathcal{V}(q)$ of node q. First, the route is exported by applying some export function, and then stored in the export table $\mathcal{O}(q, i)$ of node q for routes being shared with node i (Eq. 6.1.$\mathcal{O}$). Second, the route is imported by node i by applying a protocol and an import function, and then stored in the import table $\mathcal{I}(i, q)$ of node i for routes received from node q (Eq. 6.1.$\mathcal{I}$). Third, node i selects the best imported routes and updates its routing table $\mathcal{V}(i)$ accordingly (Eq., 6.1.$\mathcal{V}$).

The above describes one *cycle of computation* in the Ω_2 model. Given a schedule ψ for Ω_2, we need to find a schedule ψ' for Ω_1 which collapses one cycle of computation in Ω_2 (using schedule ψ) into a singular computation step in Ω_1.

Figure 6.2 illustrates how data flows through time in the various components of Ω_2, for some example schedule. It displays how information stored in the routing table $\mathcal{V}(q)$ of node q, arrives at the routing table $\mathcal{V}(i)$ in one cycle of computation. For simplicity, indices of nodes are omitted from the figure.

The figure takes the perspective of data stored in the routing table $\mathcal{V}(i)$ of node i at time t. In the example schedule, the routing table $\mathcal{V}(i)$ of node i is activated at time t, causing it to update its routing table. It uses the data that was in its import table $\mathcal{I}(i, q)$ at time $\beta_V(t, i, i)$. Recall that the data flow function β_V describes the information flow from a node's import table to its routing table. At times $\beta_V(t, i, i)$ and $\beta_V(t, i, i) - 1$, the import table $\mathcal{I}$ is not activated. Therefore, the data in $\mathcal{I}(i, q)$ at time $\beta_V(t, i, i)$ is equal to the data at time $\beta_V(t, i, i) - 2$. At that point in time the table was activated, and used the data from $\mathcal{O}(q, i)$ at time $\beta_I(\beta_V(t, i, i) - 2)$ to update its contents. Following this cycle, we arrive at the table $\mathcal{V}(q)$ of node q time $\beta_O(\beta_I(\beta_V(t, i, i) - 2, i, q) - 1, q, q)$.

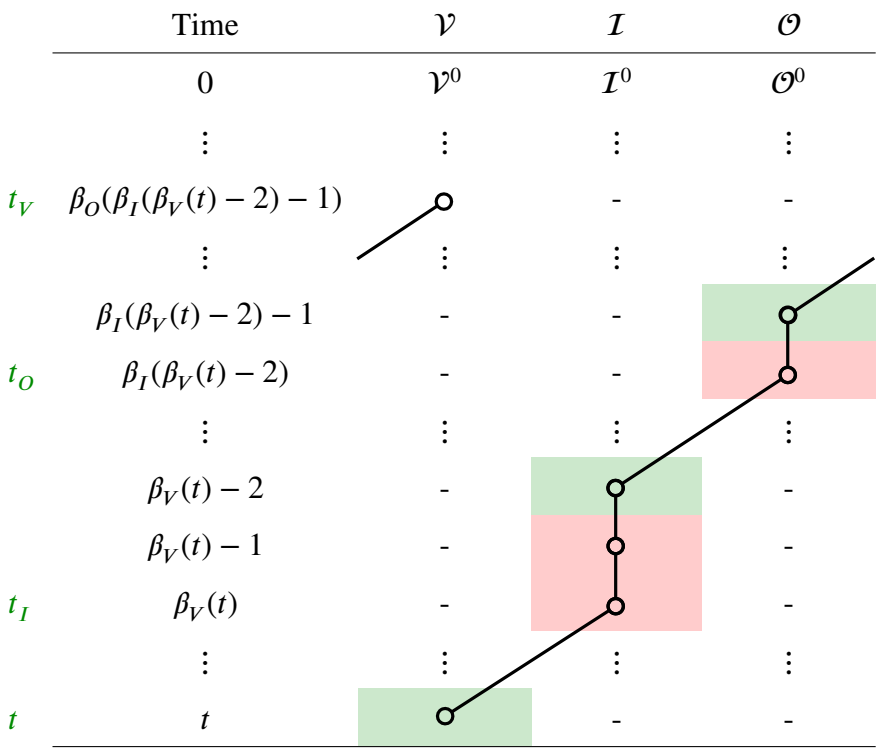

Figure 6.2: An example of one cycle of computation in Ω_2. A line indicates data sharing between nodes. Green cells indicate that a node is active. Red cells indicate that a node is inactive.

In the example computation, the data stored in the routing table $\mathcal{V}(q)$ of node q at time $\beta_O($ $\beta_I(\beta_V(t,i,i)-2,i,q)-1,q,q)$ is incorporated in the routing table $\mathcal{V}(i)$ of node i at time t. This constitutes one cycle of computation. Indeed, for this particular example, the data flow function β' of the reduced schedule in Ω_1 should have $\beta'(t,i,q)=\beta_O(\beta_I(\beta_V(t,i,i)-2,i,q)-1,q,q)$.

The main idea in relating Ω_2 and Ω_1 is to collapse one cycle of computation in Ω_2 into a single step of computation in Ω_1. As can be seen in the example, one cycle consists of three computational steps. To identify the times at which these three steps occur, we define a *data history function* ϕ.

Definition 16 (Data history function ϕ)**.**

```
φ : 𝕋 → Fin n → Fin n → 𝕋
φ zero  i q = zero
φ (suc t) i q with i ∈? α (suc t)
... | yes _ = β (suc t) i q
... | no  _ = φ t i q
```

where $\phi\ t\ i\ q$ is the time at which the information currently incorporated at node i was sent by node q.

If a node is activated ($i \in \alpha\ (t)$), then the data history function ϕ returns the data flow function β, as that is the time at which the most recent data was sent. If a node is not activated ($i \notin \alpha\ (t)$), then the data history function returns its value at t - *1*, as the node did not update its state and therefore we can recursively compute the result.

The data history function is a useful as it allows us to conveniently refer to the data used for the latest computation that was performed at a node[1]. Indeed, the state of the current node can be computed by applying the computation operator $\mathbf{F}$ to the data at the time given by the data history function ϕ.

Lemma 9. *For any asynchronous iterative computation δ (see Definition 7) specified by an operator $\boldsymbol{F}$, a time t and a starting state $\boldsymbol{x}$, we have that, for all nodes i*

$$\delta_i^t(\mathbf{x}) = \mathbf{F}_i\left(\delta^{\phi(t,i)}(\mathbf{x})\right) \qquad or \qquad \delta_i^t(\mathbf{x}) = \mathbf{x}_i$$

where the latter case occurs when a node i is never activated for all times $\leq t$.

[1]Note that the data history function is not specific to Ω_2. It can be used for any asynchronous iterative computation in the framework of Üresin and Dubois [24].

Proof. Let us consider an arbitrary node i. We prove by induction on $t \in \mathbb{N}$.

Base case $t = 0$:

$$\delta_i^0(\mathbf{x}) = \mathbf{x}_i \quad \text{(by def. of } \delta^t\text{)}$$

Inductive step $t + 1$: We proceed by cases on whether node i is activated at time $t + 1$.

Case $i \in \alpha(t + 1)$:

$$\begin{aligned}
& \delta_i^{t+1}(\mathbf{x}) \\
= \; & \mathbf{F}_i\left(\delta_1^{\beta(t+1,i,1)}(\mathbf{x}),\ \delta_2^{\beta(t+1,i,2)}(\mathbf{x}), \ldots, \delta_n^{\beta(t+1,i,n)}(\mathbf{x})\right) && \text{(by def. of } \delta \text{ and } i \in \alpha(t+1)\text{)} \\
= \; & \mathbf{F}_i\left(\delta_1^{\phi(t+1,i,1)}(\mathbf{x}),\ \delta_2^{\phi(t+1,i,2)}(\mathbf{x}), \ldots, \delta_n^{\phi(t+1,i,n)}(\mathbf{x})\right) && \text{(by def. of } \phi \text{ and } i \in \alpha(t+1)\text{)} \\
= \; & \mathbf{F}_i\left(\delta^{\phi(t+1,i)}(\mathbf{x})\right) && \text{(vector notation)}
\end{aligned}$$

Case $i \notin \alpha(t + 1)$: by the **IH**, we have that either $\delta_i^t(\mathbf{x}) = \mathbf{F}_i\left(\delta^{\phi(t,i)}(\mathbf{x})\right)$ or $\delta_i^t(\mathbf{x}) = \mathbf{x}_i$ (when node i is never activated for all times $\leq t$). We proceed by cases on the result of **IH**.

Case $\delta_i^t(\mathbf{x}) = \mathbf{F}_i\left(\delta^{\phi(t,i)}(\mathbf{x})\right)$:

$$\begin{aligned}
\delta_i^{t+1}(\mathbf{x}) &= \delta_i^t(\mathbf{x}) && \text{(by def. of } \delta \text{ and } i \notin \alpha(t+1)\text{)} \\
&= \mathbf{F}_i\left(\delta^{\phi(t,i)}(\mathbf{x})\right) && \text{(by \textbf{IH})} \\
&= \mathbf{F}_i\left(\delta^{\phi(t+1,i)}(\mathbf{x})\right) && \text{(by def. of } \phi \text{ and } i \notin \alpha(t+1)\text{)}
\end{aligned}$$

Case $\delta_i^t(\mathbf{x}) = \mathbf{x}_i$: node i is never activated for all times $\leq t$. We have that

$$\begin{aligned}
\delta_i^{t+1}(\mathbf{x}) &= \delta_i^t(\mathbf{x}) && \text{(by def. of } \delta^t \text{ and } i \notin \alpha(t+1)\text{)} \\
&= \mathbf{x}_i && \text{(by \textbf{IH})}
\end{aligned}$$

and, together with $i \notin \alpha(t + 1)$, that node i is never activated for all times $\leq t + 1$.

□

Now, using the ϕ function, we can collapse a cycle of three computations into a single step. If we take the perspective of data in the routing table $\mathcal{V}(i)$ at node i at time t, we apply the data history function ϕ three times to retrieve the time t_V at which the data was sent from the routing table $V(q)$ at node q.

This suggests the following reduction:

Definition 17 (Schedule reduction r_2)**.**

$$
\begin{aligned}
r_2 \qquad & : \mathcal{S}_2 \to \mathcal{S}_1 \\
r_2(\psi_V, \psi_I, \psi_O) \quad & = \begin{cases} \alpha'(t) & = \alpha_V(t) \\ \beta'(t,i,q) & = t_V \end{cases} \\
\textit{where} \qquad & t_I = \phi(\psi_V, t, i, i) \\
& t_O = \phi(\psi_I, t_I, i, q) \\
& t_V = \phi(\psi_O, t_O, q, q)
\end{aligned}
$$

The reduction takes a schedule ψ for Ω_2, and returns another ψ', such that a cycle of three computations in Ω_2 is collapsed into a single step of computation in Ω_1. The new activation function α' is simply equal to the activation function α_V of the routing table in Ω_2. After all, this is the table we are interested in, as it contains the actual routes (recall that the transformation $\mathcal{T}_2$ returns $\mathcal{V}$).

We aim to show that this reduction indeed lets Ω_1 simulate Ω_2, and therefore that it preserves equality under the transformation $\mathcal{T}_2$. To do so, we first utilise Lemma 9 to relate the state of the import table $\mathcal{I}$ to the export table $\mathcal{O}$, and the state of the export table $\mathcal{O}$ to the routing table $\mathcal{V}$.

Corollary 1. *For each node i and at any time t, we have that*

$$\mathcal{I}^t(i) = \Gamma_{2,\mathcal{I}}(\mathcal{O}^{\phi(t,i)})(i)$$

for a Ω_2 model starting in state $s_0 = (\emptyset, \emptyset, \emptyset)$.

Proof. By Lemma 9 we have that either $\mathcal{I}^t(i) = \Gamma_{2,\mathcal{I}}(\mathcal{O}^{\phi(t,i)})$ or $\mathcal{I}^t(i) = \mathcal{I}^0(i)$ holds.

Case $\mathcal{I}^t(i) = \Gamma_{2,\mathcal{I}}(\mathcal{O}^{\phi(t,i)})(i)$. The corollary holds.

Case $\mathcal{I}^t(i) = \mathcal{I}^0(i)$. Since the import table $I(i)$ of node i never activated (for times $\leq t$), we have that $\phi(t,i,q) = 0$ for all nodes q. Subsequently, the corollary holds:

$$
\begin{aligned}
\mathcal{I}^t(i) \quad & = \mathcal{I}^0(i) && \text{(by case)} \\
& = \emptyset && \text{(by def. of starting state } s_0\text{)} \\
& = \Gamma_{2,\mathcal{I}}(\emptyset) && \text{(importing an empty export table will be empty)} \\
& = \Gamma_{2,\mathcal{I}}(\mathcal{O}^{\phi(t,i)}) && \text{(as } \phi(t,i,q) = 0 \text{ for all } q\text{)}
\end{aligned}
$$

□

Corollary 2. *For each node i and at any time t, we have that* $\mathcal{O}^t(i) = \Gamma_{2,\mathcal{O}}(\mathcal{V}^{\phi(t,i)})(i)$ *for a Ω_2 model starting in state $s_0 = (\emptyset, \emptyset, \emptyset)$.*

These corollaries simplify the proof that Ω_2 can be simulated by Ω_1.

Theorem 5. *($\Omega_1 \leftsquigarrow \Omega_2$) The Ω_2 model can be simulated by the Ω_1 model; that is, the schedule reduction $r_2 : S_2 \to S_1$ preserves equivalence under the transformation $\mathcal{T}_2$. We have that for all Ω_2 schedules ψ, and the starting state $s_0 = (\emptyset, \emptyset, \emptyset)$:*

$$\mathcal{T}_2\left(\Omega_2^t(\psi)(s_0)\right) = \Omega_1^t(r_2(\psi))\,\mathcal{T}_2(s_0))$$

Proof. Let $\mathcal{V}_2^t$, $\mathcal{I}^t$, $\mathcal{O}^t$ denote the routing tables, import tables and export tables respectively, at time t in the Ω_2 model, for some schedule ψ and starting state s_0

Let $\mathcal{V}_1^t$ denote the routing table at time t in the Ω_1 model, for the reduced schedule $r_2(\psi)$ and reduced starting state $\mathcal{T}_2(s_0)$.

We prove by strong induction on $t \in \mathbb{N}$.

Base case $t = 0$:

$$\mathcal{V}_2^0 = \mathcal{V}_1^0 \quad \text{(by def. of } \Omega_2, \Omega_1 \text{ and } \mathcal{T}_2)$$

Inductive step $t > 0$: Let i be some arbitrary node. We proceed by case analysis on the activation of the node at time t (whether $i \in \alpha_V(t)$).

Case $i \notin \alpha_V(t)$:

$$\begin{aligned}
\mathcal{V}_2^t(i) &= \mathcal{V}_2^{t-1}(i) && \text{(by def. of } \Omega_2 \text{ and } i \notin \alpha(t)) \\
&= \mathcal{V}_1^{t-1}(i) && \text{(by } \mathbf{IH}) \\
&= \mathcal{V}_1^t(i) && \text{(by def. of } \Omega_1, \text{ reduction } r_2 \text{ and } i \notin \alpha(t))
\end{aligned}$$

Case $i \in \alpha_V(t)$: Let q be some arbitrary node. We consider the data that node i receives from node q.

Let t_I denote $\phi(t, i, i)$, t_O denote $\phi(t_I, i, q)$ and t_V denote $\phi(t_O, q, q)$.

We have that $\mathcal{V}_2^t(i) = \Gamma_{2,\mathcal{V}}(\mathcal{I}^{\beta(t,i)})(i) = \Gamma_{2,\mathcal{V}}(\mathcal{I}^{\phi(t,i)})(i) = \Gamma_{2,\mathcal{V}}(\mathcal{I}^{t_I})(i)$. The theorem follows by application of Corollaries 1 and 2, and Lemma 7 (described in the previous section, originally from the Part II dissertation [25]).

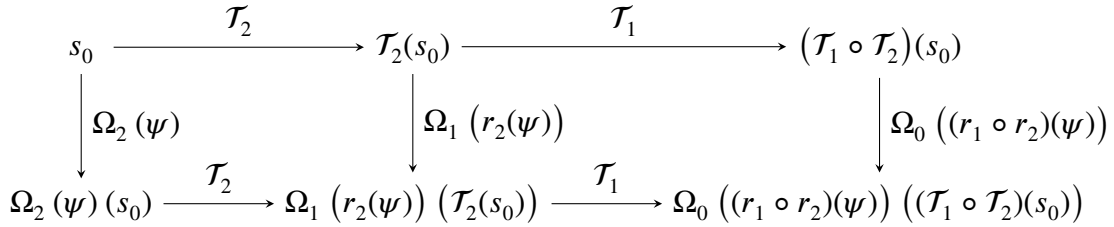

Figure 6.3: Commutativity of the low-level model Ω_2 and the high-level model Ω_0, for some starting state s_0 and schedule ψ. The Ω_0 model simulates Ω_2 using a schedule reduction ($r_1 \circ r_2$) and a transformation ($\mathcal{T}_1 \circ \mathcal{T}_2$).

$$
\begin{aligned}
\mathcal{V}_2^t(i) &= \Gamma_{2,\mathcal{V}}(\mathcal{I}^{t_I})(i) && \text{(by def. of } \Omega_2 \text{ and } i \in \alpha_V(t)) \\
&= (\Gamma_{2,\mathcal{V}} \circ \Gamma_{2,\mathcal{I}})(\mathcal{O}^{t_O})(i) && \text{(by Corollary 1)} \\
&= (\Gamma_{2,\mathcal{V}} \circ \Gamma_{2,\mathcal{I}} \circ \Gamma_{2,\mathcal{O}})(\mathcal{V}_2^{t_V})(i) && \text{(by Corollary 2)} \\
&= \Gamma_1(\mathcal{V}_2^{t_V})(i) && \text{(by Lemma 7)} \\
&= \Gamma_1(\mathcal{V}_1^{t_V})(i) && \text{(by } \mathbf{IH} \text{ and } t_V < t) \\
&= \Gamma_1(\mathcal{V}_1^{\beta'(t,i)})(i) && \text{(by def. of reduction } r_2) \\
&= \left[\Gamma_1(\mathcal{V}_1^{\beta'(t,i)}),\ \mathcal{V}_1^{t-1}\right]_{\alpha'(t)}(i) && \text{(by def. of reduction } r_2 \text{ and } i \in \alpha_V(t)) \\
&= \mathcal{V}_1^t(i) && \text{(by def. of } \Omega_1)
\end{aligned}
$$

Done. We have shown that for all nodes i, at all times t, the routing table $V_2(i)$ of the Ω_2 model is equal to the routing table $V_1(i)$ of the Ω_1 model.

□

The Ω_1 model is capable of simulating the mechanics of the Ω_2 model. It follows, by combining this with results of the previous section, that the high-level model Ω_0 is indeed capable of simulating Ω_2. Figure 6.3 illustrates this relationship in the form of a commutativity diagram.

In this chapter, we defined the Ω_2 model, which captures the composition aspect of distributed Bellman-Ford protocols. We showed that the higher-level Ω_1 model is capable of simulating the Ω_2 model, and therefore also that the high-level Ω_0 model can simulate the low-level Ω_2 model. This justifies the abstraction that is made by using the algebraic approach to routing.

Chapter 7

The Ω_3 Model: Hard-state

In this chapter, we introduce the hard-state aspect – each router only shares the changes they observe in their view of the network, instead of always sharing their full routing state [17]. This models the specific class of hard-state distributed Bellman-Ford protocols.

7.1 Synchronous

In the Part II dissertation, we defined a *difference operator*. It takes two sets X, Y, and returns a tuple (∇, Δ), where the set ∇ contains the elements to be deleted, and Δ contains the elements to be added.

$$\text{diff}(X, Y) \equiv (\underbrace{X - Y}_{\nabla}, \underbrace{Y - X}_{\Delta})$$

Given the original set X, the deletions ∇ and the additions Δ, we can compute the new set Y:

$$Y = (X - \nabla) \cup \Delta$$

The Γ_3 model adds two tables to the state, namely the ∇ and Δ tables, containing the deletions and additions, respectively. Instead of sharing the output table $\mathcal{O}$ with other nodes, the difference tables ∇ and Δ are shared. Figure 7.1 illustrates the mechanics of Γ_3. The old output table $\mathcal{O}$ and the new output table $\mathcal{O}'$ will satisfy the following:

$$\mathcal{O}'(i, j) = \big(\mathcal{O}(i, j) - \nabla(i, j)\big) \cup \Delta(i, j)$$

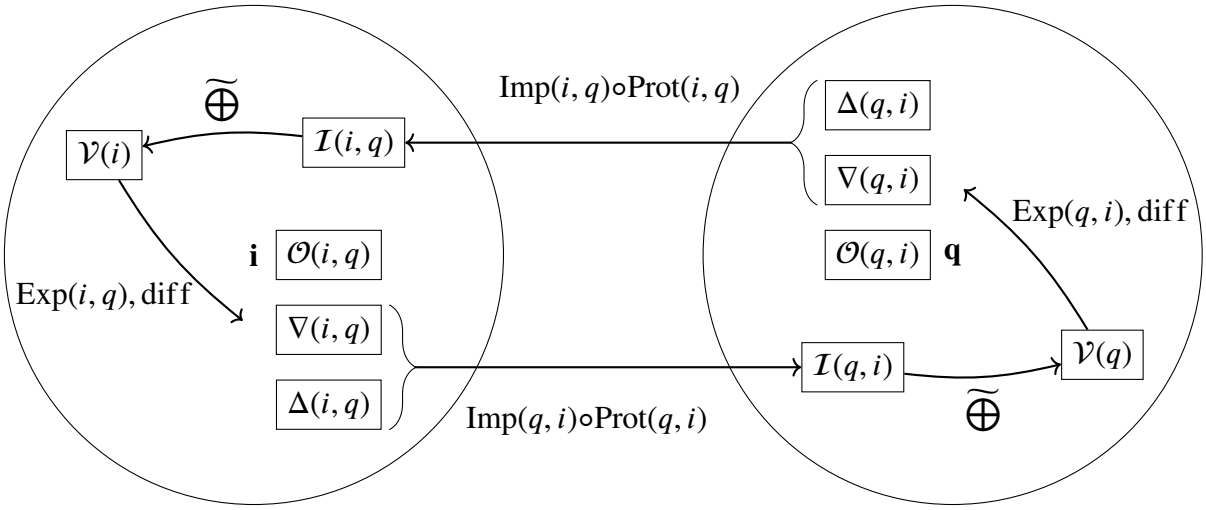

Figure 7.1: Hard-state in Γ_3

Like Γ_2, the Γ_3 model is split into various sub-models.

$$\Gamma_3(\mathcal{V},\mathcal{I},\mathcal{O},(\nabla,\Delta)) \equiv \big(\Gamma_{3,\mathcal{V}}(\mathcal{I}),\Gamma_{3,\mathcal{I}}(\mathcal{I},\nabla,\Delta),\Gamma_{3,\mathcal{O}}(\mathcal{V}),\Gamma_{3,D}(\mathcal{V},\mathcal{O})\big)$$

The $\Gamma_{3,\mathcal{V}}$ function, which updates the routing table, is equivalent to Γ_2's routing table update function.

$$\Gamma_{3,\mathcal{V}}(\mathcal{I}) \equiv \Gamma_{2,\mathcal{V}}(\mathcal{I}) \equiv \mathcal{I}^{\downarrow} \,\widetilde{\oplus}\, \widetilde{I}$$

The import table function $\Gamma_{3,\mathcal{I}}$ combines the current import table, together with the received deletions and additions:

$$\begin{aligned}\Gamma_{3,\mathcal{I}}(\mathcal{I},\nabla,\Delta) &\equiv \big(\mathcal{I} - (\mathrm{Imp}\circ\mathrm{Prot})(\!|\nabla|\!)\big) \cup (\mathrm{Imp}\circ\mathrm{Prot})(\!|\Delta|\!)\\ &= \big(\mathcal{I} - \Gamma_{2,\mathcal{I}}(\nabla)\big) \cup \Gamma_{2,\mathcal{I}}(\Delta)\end{aligned}$$

The output table function $\Gamma_{3,\mathcal{O}}$, which applies the export function to outgoing routes, is defined as in Γ_2.

$$\Gamma_{3,\mathcal{O}}(\mathcal{V}) \equiv \Gamma_{2,\mathcal{O}}(\mathcal{V}) \equiv \mathrm{Exp}(\!|\mathcal{V}|\!)$$

As mentioned, in Γ_3 only differences are shared between nodes. The $\Gamma_{3,D}$ function computes the difference tables ∇ and Δ, using the diff function.

$$\Gamma_{3,D}(\mathcal{V},\mathcal{O}) \equiv \mathrm{diff}\big(\mathcal{O},\Gamma_{3,\mathcal{O}}(\mathcal{V})\big)$$

The model Γ_3 differs from Γ_2 in that it models the hard-state aspect – nodes only share changes in their view of the routing state. Nevertheless, the computations performed by Γ_2 and Γ_3 are functionally equivalent.

Theorem 6 ($\Gamma_2 \rightsquigarrow \Gamma_3$)**.** *The models* Γ_2 *and* Γ_3 *are functionally equivalent.*

$$\Gamma_3{}^{\langle k \rangle}\left(\widetilde{I}, \emptyset, \emptyset, (\emptyset, \emptyset)\right) = \Gamma_2{}^{\langle k \rangle}\left(\widetilde{I}, \emptyset, \emptyset\right)$$

Proof. See Theorem 4 from [25]. □

The essential property that makes this difference structure work, is that function application preserves the structure.

Lemma 10. *Let* $\mathrm{diff}(A, B) = (\nabla, \Delta)$*, where A and B are matrices of routing sets. Let F be a matrix of edge functions. Then,*

$$F(\!|B|\!) = \left(F(\!|A|\!) - F(\!|\nabla|\!)\right) \cup F(\!|\Delta|\!)$$

Proof. See Lemma 3.7 from [25]. □

7.2 Asynchronous

As with Ω_2, we wish to model asynchrony both between and inside nodes. To do so, we use a quadruple of schedules $(\psi_V, \psi_I, \psi_O, \psi_D)$ for our Ω_3 model – one for each component $\mathcal{V}$, $\mathcal{I}$, $\mathcal{O}$, (∇, Δ). We add the notion of schedules to the synchronous Γ_3 model, obtaining the Ω_3 model:

$$\Omega_3^0 : (\mathcal{V}^0, \mathcal{I}^0, \mathcal{O}^0, (\nabla^0, \Delta^0)) \equiv (\mathcal{V}, \mathcal{I}, \mathcal{O}, (\nabla, \Delta))$$

$$\Omega_3^{t+1} : \begin{cases} \mathcal{V}^{t+1} & \equiv \left[\Gamma_{3,\mathcal{V}}(\mathcal{I}^{\beta_V(t+1)}),\ \mathcal{V}^t\right]_{\alpha_V(t+1)} = \left[(\mathcal{I}^{\beta_V(t+1)})^{\downarrow} \widetilde{\oplus}\ \widetilde{I},\ \mathcal{V}^t\right]_{\alpha_V(t+1)} \\ \mathcal{I}^{t+1} & \equiv \left[\Gamma_{3,\mathcal{I}}(\mathcal{I}^t, (\nabla^{\beta_I(t+1)}, \Delta^{\beta_I(t+1)})),\ \mathcal{I}^t\right]_{\alpha_I(t+1)} \\ & = \left[\left(\mathcal{I}^t - \Gamma_{2,\mathcal{I}}(\nabla^{\beta_I(t+1)})\right) \cup \Gamma_{2,\mathcal{I}}(\Delta^{\beta_I(t+1)}),\ \mathcal{I}^t\right]_{\alpha_I(t+1)} \\ \mathcal{O}^{t+1} & \equiv \left[\Gamma_{3,\mathcal{O}}(\mathcal{V}^{\beta_O(t+1)}),\ \mathcal{O}^t\right]_{\alpha_O(t+1)} = \left[\mathrm{Exp}(\!|\mathcal{V}^{\beta_O(t+1)}|\!),\ bigO^t\right]_{\alpha_O(t+1)} \\ (\nabla^{t+1}, \Delta^{t+1}) & \equiv \left[\Gamma_{3,D}(\mathcal{V}^{\beta_D(t+1)}, \mathcal{O}^{\beta_D(t+1)}),\ (\nabla^t, \Delta^t)\right]_{\alpha_D(t+1)} \\ & = \left[\mathrm{diff}(\mathcal{O}^{\beta_D(t+1)}, \Gamma_{2,\mathcal{O}}(\mathcal{V}^{\beta_D(t+1)})),\ (\nabla^t, \Delta^t)\right]_{\alpha_D(t+1)} \end{cases} \tag{7.1}$$

Lemma 11 (Synchronous Ω_3)**.** *The* Ω_3 *model using the synchronous schedule* ψ^{sync} *is equal to the synchronous model* Γ_3*.*

$$\forall t, \mathcal{V}, \mathcal{I}, \mathcal{O}, (\nabla, \Delta).\ \Omega_3^t\left(\psi^{sync}\right)\left(\mathcal{V}, \mathcal{I}, \mathcal{O}, (\nabla, \Delta)\right) = \Gamma_3^t\left(\mathcal{V}, \mathcal{I}, \mathcal{O}, (\nabla, \Delta)\right)$$

Proof. See Appendix A.3. □

7.2.1 Reduction $\Omega_3 \rightarrow \Omega_2$

The Ω_3 model has five components, as opposed to the Ω_2 model, which has three components. The Ω_3 model has additional difference tables ∇, Δ. These tables contain left-over computations, that arise from the fact that Ω_3 only transmits changes in the routing state. The transformation $\mathcal{T}_3 : \Omega_3 \rightarrow \Omega_2$ simply discards the difference tables.

$$\mathcal{T}_3(\mathcal{V}, \mathcal{I}, \mathcal{O}, (\nabla, \Delta)) = (\mathcal{V}, \mathcal{I}, \mathcal{O})$$

As mentioned, the Ω_3 models the hard-state approach – a router shares only the changes in its view of the network with other routers, instead of sharing its entire routing table at each iteration [17]. The advantage of this approach is that it significantly reduces the network traffic. The downside is that it does require a *reliable* schedule.

A reliable schedule is a schedule in which no messages are lost or reordered. Messages can be delayed, but they should be guaranteed to arrive, and to be received in order.

An implication of the use of the hard-state approach is that Ω_2 is not always able to simulate the mechanics of the Ω_3 model.

Theorem 7. *There does not exist a schedule reduction $r_3 : \mathcal{S}_3 \rightarrow \mathcal{S}_2$ such that it preserves equivalence under the transformation $\mathcal{T}_3$, for all Ω_3 schedules.*

$$\nexists r_3.\, \forall \psi.\, \mathcal{T}_3\left(\Omega_3^t\,(\psi)\,(s_0)\right) = \Omega_2^t\,(r_3(\psi))\,(\mathcal{T}_3(s_0))$$

Proof. We construct a counterexample consisting of an example network configuration, a routing algebra and a schedule ψ for Ω_3, for which there exists no schedule ψ' for Ω_2 such that the two models are equal under the transformation $\mathcal{T}_3$. The key idea is that a single message (route announcement) is lost, and that Ω_2 is unable to capture the granularity of this event.

The example algebra we use is the well-known shortest path algebra. Figure 7.2 shows the example network. It consists of three routers: **Alice**, **Bob** and **Charlie**. There is a unidirectional edge[1] from **Alice** to **Bob**, and from **Bob** to **Charlie**, both with a weight of 1. The solution to the routing problem of this example is the following routing state:

$$\begin{aligned} \underline{\textbf{A}}\textbf{lice} &: \{(A,0),(B,1),(C,2)\} \\ \underline{\textbf{B}}\textbf{ob} &: \{(B,0),(C,1)\} \\ \underline{\textbf{C}}\textbf{harlie} &: \{(C,0)\} \end{aligned}$$

[1] Note that the unidirectional edge means that network traffic can only flow from **Alice** to **Bob**. This, however, does not include routing information. The flow of routing information is dictated by the schedule.

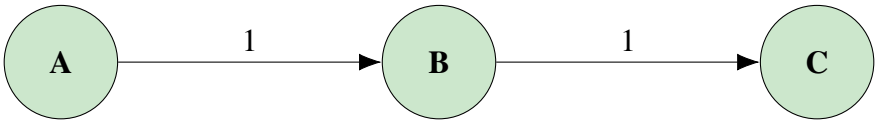

Figure 7.2: The network configuration of the counterexample. There are three routers: **Alice**, **Bob** and **Charlie**. They use the shortest-paths algebra. The edges are annotated with the cost associated with traffic flowing through that link.

Alice has a route to itself with the trivial weight 0, a route to **Bob** with a weight of 1 and a route to **Charlie** with a weight of 2. **Bob** has the trivial route to itself, and a route of weight 1 to **Charlie**. **Charlie** merely has the trivial route to itself.

The aim of the routing protocol is to compute this solution. We define an example schedule ψ for which it fails to compute this optimal solution. The failure of the protocol itself is *not* the problem here – the problem is that Ω_2 is unable to simulate this failure.

Let the example schedule ψ be the synchronous schedule, except that one message is never delivered: the message from **Bob** to **Alice** in which **Bob** announces that it has discovered the trivial route itself. The protocol fails to compute the optimal solution, as this message is only sent once (due to the hard-state approach), but never reaches **Alice**.

We show that there does *not* exist an Ω_2 schedule ψ', such that Ω_2 simulates this behaviour. Table 7.1 shows the routing state of the three routers as time progresses.

For the Ω_2 model to simulate the Ω_3 model (for this particular instance), we need to find an Ω_2 schedule ψ' such that at each time t, the $\mathcal{V}, \mathcal{I}, \mathcal{O}$ components of the two models are equal (i.e. they are equal under the transformation $\mathcal{T}_3$. However, in the process of the construction of such a reduced schedule ψ', a contradiction arises.

We proceed by explaining what is enforced on the schedule ψ' at each time t to try and make the reduction work. Recall that in the Ω_2 model, there are no difference tables ∇, Δ, and routers simply share their entire export table $\mathcal{O}$ with the network.

Time t = 0. The routing process is initialised using the initial empty routing state. The schedule is irrelevant.

Time t = 1. All routers in Ω_2 are required to activate their routing tables $\mathcal{V}$ ($\forall i. i \in \alpha'_V(1)$), as all routers in Ω_3 update their routing tables. There are no further restrictions.

Time t = 2. All routers in Ω_2 are required to activate their export tables $\mathcal{O}$ ($\forall i. i \in \alpha'_O(2)$), and to use the information generated at time $t = 1$. There are no further restrictions.

Time t = 3. The message $\{(B, 1)\}$ from **Bob** to **Alice** is lost. To simulate the lack of this route in **Alice**'s import table $\mathcal{I}$ in Ω_2, we can either set the table to be inactive ($\mathbf{A} \notin \alpha'_I(3)$),

Time	$\mathcal{V}$	$\mathcal{I}$	$\mathcal{O}$	(∇, Δ)
0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset, \emptyset$
1	$\{(A,0)\}$	$\emptyset$	$\emptyset$	$\emptyset, \emptyset$
2	$\{(A,0)\}$	$\emptyset$	$\{(A,0)\}$	$\emptyset, \{(A,0)\}$
3	$\{(A,0)\}$	$\emptyset$	$\{(A,0)\}$	$\emptyset, \emptyset$
4	$\{(A,0)\}$	$\emptyset$	$\{(A,0)\}$	$\emptyset, \emptyset$
5	$\{(A,0)\}$	$\emptyset$	$\{(A,0)\}$	$\emptyset, \emptyset$
6	$\{(A,0)\}$	$\{(C,2)\}$	$\{(A,0)\}$	$\emptyset, \emptyset$
7	$\{(A,0),(C,2)\}$	$\{(C,2)\}$	$\{(A,0)\}$	$\emptyset, \emptyset$
8	$\{(A,0),(C,2)\}$	$\{(C,2)\}$	$\{(A,0),(C,2)\}$	$\emptyset, \{(C,2)\}$

Time	$\mathcal{V}$	$\mathcal{I}$	$\mathcal{O}$	(∇, Δ)
0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset, \emptyset$
1	$\{(B,0)\}$	$\emptyset$	$\emptyset$	$\emptyset, \emptyset$
2	$\{(B,0)\}$	$\emptyset$	$\{(B,0)\}$	$\emptyset, \{(B,0)\}$
3	$\{(B,0)\}$	$\{(C,1)\}$	$\{(B,0)\}$	$\emptyset, \emptyset$
4	$\{(B,0),(C,1)$	$\{(C,1)\}$	$\{(B,0)\}$	$\emptyset, \emptyset$
5	$\{(B,0),(C,1)$	$\{(C,1)\}$	$\{(B,0),(C,1)\}$	$\emptyset, \{(C,1)\}$
6	$\{(B,0),(C,1)$	$\{(C,1)\}$	$\{(B,0),(C,1)\}$	$\emptyset, \emptyset$

Time	$\mathcal{V}$	$\mathcal{I}$	$\mathcal{O}$	(∇, Δ)
0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset, \emptyset$
1	$\{(C,0)\}$	$\emptyset$	$\emptyset$	$\emptyset, \emptyset$
2	$\{(C,0)\}$	$\emptyset$	$\{(C,0)\}$	$\emptyset, \{(C,0)\}$
3	$\{(C,0)\}$	$\emptyset$	$\{(C,0)\}$	$\emptyset, \emptyset$

Table 7.1: The routing state of the routers **Alice**, **Bob** and **Charlie**, respectively. The red cell indicates when the message $\{(B,0)\}$ sent to **Alice** by **Bob** was lost. The last time displayed for each router is the time at which the state of the router has converged.

or let it use **Bob**'s old export table ($\beta'_I(3, \mathbf{A}, \mathbf{B}) < 2$). **Bob** is required to update its import table $\mathcal{I}$ and its difference tables (∇, Δ). **Charlie** is required to update its difference tables. Note that at this time, **Charlie** has converged, and it not required to update any more.

Time t = 4. **Alice** is required to continue the same behaviour (either deactivate or use old data from **Bob**'s export table). **Bob** needs to update its routing table $\mathcal{V}$.

Time t = 5. Again, **Alice** is required to continue the same behaviour. **Bob** needs to update its output table $\mathcal{O}$ and difference tables (∇, Δ).

Time t = 6. **Alice** is required to update its import table $\mathcal{I}$. In Ω_3, this import table contains the $(C, 2)$ route, which **Alice** imported from **Bob**, who sent it at time $t = 5$. This forces the data flow function β'_I for **Alice** from **Bob** to refer to time $t = 5$ as well, as the route $(C, 1)$ did not exist in **Bob**'s export table before that time. We obtain the following constraint:

$$\beta'_I(6, \mathbf{A}, \mathbf{B}) = 5$$

However, this means that **Alice** will also receive the route $(B, 0)$ from **Bob**, as in the Ω_2 model, routers share their entire export table at each step. This leads to a *contradiction*, as **Alice** never received that route in the Ω_3 model.

This counterexample shows that there does not exist a reduction $r_3 : \Omega_3 \to \Omega_2$, such Ω_2 can simulate Ω_3, for all Ω_3 schedules.

□

We have shown that the Ω_2 model, and therefore also the high-level model, is not capable of simulating the mechanics of a hard-state distributed Bellman-Ford protocol in an asynchronous world. The result is that the high-level algebraic approach should be used with caution when reasoning about hard-state protocols, since it does not fully capture the behaviour of such protocols in the case of an unreliable connection.

It is important to note, however, that a hard-state protocol assumes a reliable connection, whereas the counterexample presented uses an unreliable schedule. In a context where it is certain that the communication will be reliable, the algebraic approach might still correctly simulate the behaviour of the protocol.

Chapter 8

Conclusion

Routing protocols are an essential part of computer networks such as the Internet. The algebraic approach to routing has proven to be a successful framework for modelling the behaviour of such protocols [7, 10, 23]. For example, it allows routing protocol researchers to develop safe-by-design algebras, such that the resulting protocols are guaranteed to converge [8].

This high-level algebraic model, however, abstracts away many details of actual implementations of such routing protocols. In this dissertation, we bridged the gap between the high-level model, and low-level implementations of the family of distributed Bellman-Ford protocols. More concretely, we have shown that, even in an asynchronous world, the high-level model is capable of simulating the mechanics of the low-level implementation.

Theorem 8 ($\Omega_0 \rightsquigarrow \Omega_2$)**.** *The high-level* Ω_0 *model can simulate the low-level* Ω_2 *model, using the transformation* $(\mathcal{T}_1 \circ \mathcal{T}_2)$ *and the schedule reduction* $(r_1 \circ r_2)$*.*

This result is new, and contributes to the credibility of the algebraic approach to routing.

Additionally, we found a counterexample which demonstrates that the well-known high-level model fails to correctly simulate asynchronous implementations of hard-state routing protocols (as modelled by Ω_3). The algebraic approach should therefore be used with caution when reasoning about hard-state protocols.

Our models and theorems have been implemented in the Agda theorem prover. The implementation is incorporated into an open-source library for reasoning about network routing problems, available for others to use and extend the formalised results [6].

8.1 Future Research

The project aimed to model the low-level implementation of a particular class of routing protocols. Future research could be focused on modelling more features. The approach taken in this project is modular, and thus allows for more detailed models to be added post hoc.

Hard-state. In Ω_3, we model the hard-state approach that routers employ – they only share the changes in their view of the network. We have shown that it is not possible for unreliable schedules in Ω_3, to be simulated by Ω_2. However, these protocols require reliable communication, and it would be therefore be interesting to investigate their mechanics for the restricted class of reliable schedules.

Dynamic networks. Recent research by Daggitt is on the notion of *dynamic networks* [5]. In such networks, a changing topology of computer networks is modelled – nodes can drop out and new nodes can join. The added complexity of such behaviour leads to new problems. It would therefore be interesting to incorporate this feature into the low-level model.

The results presented in this project have been formalised using the Agda theorem prover. In the process of formalisation, we found that implementing even the most trivial proofs can occasionally be a time-consuming process.

Automated proving in Agda. Isabelle is another proof assistant [15]. It comes with a "Sledgehammer" tool which converts propositions to be given to automatic theorem provers, and thus to be solved automatically [16]. Researching the possibility of such an interface between Agda and automatic theorem provers would be interesting, as it could greatly increase productivity when formalising in Agda.

8.2 Summary

In this dissertation, we modelled low-level implementations of distributed Bellman-Ford routing protocols, in an asynchronous world. We showed that the high-level algebraic model is indeed capable of simulating the mechanics of this low-level model. We also constructed a counterexample which demonstrates that the high-level model fails to correctly simulate asynchronous hard-state protocols.

Bibliography

[1] Agda community. Termination checking. 2020. URL `https://agda.readthedocs.io/en/v2.6.1/language/termination-checking.html`. [Last accessed 23 May 2020].

[2] Agda community. What is Agda? 2020. URL `https://agda.readthedocs.io/en/v2.6.1/getting-started/what-is-agda.html`. [Last accessed 23 May 2020].

[3] Agda community. The Agda standard library. 2020. URL `https://github.com/agda/agda-stdlib`. [Last accessed 23 May 2020].

[4] B. A. Carré. An algebra for network routing problems. *IMA Journal of Applied Mathematics*, 7(3):273–294, 06 1971.

[5] Matthew L. Daggitt. *An algebraic perspective on the convergence of vector-based routing protocols*. PhD thesis, University of Cambridge, 2019.

[6] Matthew L. Daggitt. Agda-routing. 2020. URL `https://github.com/MatthewDaggitt/agda-routing`. [Last accessed 23 May 2020].

[7] Matthew L. Daggitt and Timothy G. Griffin. Rate of convergence of increasing path-vector routing protocols. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 335–345. IEEE, 2018.

[8] Matthew L. Daggitt, Alexander J. T. Gurney, and Timothy G. Griffin. Asynchronous convergence of policy-rich distributed Bellman-Ford routing protocols. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 103–116, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5567-4. doi: 10.1145/3230543.3230561.

[9] Matthew L. Daggitt, Ran Zmigrod, and Timothy G. Griffin. A relaxation of Üresin and Dubois' asynchronous fixed-point theory in Agda. *Journal of Automated Reasoning*, pages 1–21, 2019. doi: https://doi.org/10.1007/s10817-019-09536-w.

[10] Michel Gondran and Michel Minoux. *Graphs, dioids and semirings: new models and algorithms*, volume 41. Springer Science & Business Media, 2008.

[11] Tim Griffin and Geoff Huston. BGP wedgies. *Network Working Group, RFC*, 4264, 2005.

[12] Timothy G. Griffin. Algebraic path problems (L11). 2019. URL https://www.cl.cam.ac.uk/teaching/1920/L11/. [Last accessed 23 May 2020].

[13] Alexander J.T. Gurney. Asynchronous iterations in ultrametric spaces. *Technical report*, 2017. URL https://arxiv.org/abs/1701.07434.

[14] Charles L Hedrick. Routing Information Protocol. Technical report, 1988.

[15] Iasbelle community. Isabelle. 2020. URL https://isabelle.in.tum.de. [Last accessed 23 May 2020].

[16] Isabelle community. Sledgehammer. 2020. URL https://isabelle.in.tum.de/website-Isabelle2009-1/sledgehammer.html. [Last accessed 23 May 2020].

[17] P. Ji, Z. Ge, J. Kurose, and D. Towsley. A comparison of hard-state and soft-state signaling protocols. *Networking, IEEE/ACM Transactions on*, 15:281 – 294, 05 2007. doi: 10.1109/TNET.2007.892849.

[18] Neel Krishnaswami. Types. 2019. URL https://www.cl.cam.ac.uk/teaching/1920/Types/. [Last accessed 23 May 2020].

[19] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[20] Danny McPherson, Vijay Gill, Daniel Walton, and Alvaro Retana. Border Gateway Protocol (BGP) persistent route oscillation condition. Technical report, RFC 3345, August, 2002.

[21] Ulf Norell. Dependently typed programming in Agda. In *International School on Advanced Functional Programming*, pages 230–266. Springer, 2008.

[22] Yakov Rekhter, Tony Li, and Susan Hares. A Border Gateway Protocol 4 (BGP-4). 2005.

[23] J. L. Sobrinho. An algebraic theory of dynamic network routing. *IEEE/ACM Transactions on Networking*, 13(5):1160–1173, October 2005. ISSN 1558-2566. doi: 10.1109/TNET.2005.857111.

[24] Aydin Üresin and Michel Dubois. Parallel asynchronous algorithms for discrete data. *Journal of the ACM (JACM)*, 37(3):588–606, 1990.

[25] Lex M. van der Stoep. An Agda formalisation of a hard-state vectoring routing protocol. 2019. Part II dissertation.

[26] Ran Zmigrod, Matthew L Daggitt, and Timothy G Griffin. An Agda formalization of Üresin and Dubois' asynchronous fixed-point theory. In *International Conference on Interactive Theorem Proving*, pages 623–639. Springer, 2018.

Appendix A

Mathematical Proofs

A.1 Ω_1: Vectoring

Lemma 3. (Synchronous Ω_1). *The Ω_1 model using the synchronous schedule ψ^{sync} is equal to the synchronous model Γ_1.*

$$\forall t, \mathcal{V}.\ \Omega_1^t\ (\psi^{sync})\ (\mathcal{V}) = \Gamma_1^t\ (\mathcal{V})$$

Proof. We prove by induction on $t \in \mathbb{N}$.

Base case t = 0. Trivially holds by the definitions of Ω_1 and Γ_1.

Inductive step t + 1. Let $\mathcal{V}$ be the initial routing vector. Note that $\mathcal{V}^k$ denotes the routing state of Ω_1 at time k, with an initial state $\mathcal{V}$.

$$\begin{aligned}
\Omega_1^{t+1}\ (\psi^{sync})\ (\mathcal{V}) &= \left[A\ (\!|\mathcal{V}^{\beta(t+1)}|\!)\ \widetilde{\oplus}\ \widetilde{I},\ \mathcal{V}^t\right]_{\alpha(t+1)} && \text{(by def. of } \Omega_1\text{)} \\
&= A\ (\!|\mathcal{V}^t|\!)\ \widetilde{\oplus}\ \widetilde{I} && \text{(by def. of synchronous schedule)} \\
&= \Gamma_1(\mathcal{V}^t) && \text{(by def. of } \Gamma_1\text{)} \\
&= \Gamma_1^{t+1}\ (\mathcal{V}) && \text{(by } \textbf{IH}\text{)}
\end{aligned}$$

□

Lemma 4. (Inverse – and $\sim$). *For all routing matrices X and routing vectors $\mathcal{V}$, we have that*

$$\overline{(\widetilde{X})} = X \qquad \widetilde{\overline{V}} = V$$

Proof. We start with the property that – is the left inverse of $\sim$; that is, $\forall X.\ \overline{(\widetilde{X})} = X$.

Let X be some routing matrix, and i, j some nodes. We perform a case analysis on whether the route $X(i,j)$ from node i to j is valid.

Case $\mathbf{X(i,j) = \bar{\infty}}$.

The routing set $\widetilde{X}(i)$ will not contain a route $(j, \bar{\infty})$. Therefore, $\overline{(\widetilde{X})}(i,j) = \bar{\infty} = X(i,j)$.

Case $\mathbf{X(i,j) \neq \bar{\infty}}$.

The routing set $\widetilde{X}(i)$ will contain a route (j,s), and no other routes for the destination node j. Therefore, $\overline{(\widetilde{X})}(i,j) = s = X(i,j)$.

Next, we show that $\sim$ is the left inverse of –; that is, $\forall \mathcal{V}.\ \widetilde{\overline{\mathcal{V}}} = \mathcal{V}$.

Let $\mathcal{V}$ be some routing vector, and i some node. We prove by mutual set inclusion that $(\widetilde{\overline{\mathcal{V}}})(i) = \mathcal{V}(i)$. Recall that a routing set contains at most one route (j,s) to a destination j.

$$\begin{aligned}
(j,s) \in (\widetilde{\overline{\mathcal{V}}})(i) &\Leftrightarrow \overline{\mathcal{V}}(i,j) = s, \text{ and } s \neq \infty && \text{(by def. of } \sim) \\
&\Leftrightarrow (j,s) \in \mathcal{V}(i) && \text{(by def. of –)}
\end{aligned}$$

We conclude that the operations – and $\sim$ are inverses of each other. □

Lemma 6. (Distributivity of – over choice operator).

$$\overline{[U,V]_S} = \left[\overline{U}, \overline{V}\right]_S$$

Proof. Let U, V be two routing vectors. Let i, j be some nodes.

$$\begin{aligned}
\left(\overline{[U,V]_S}\right)(i,j) &= \text{lookup}_d\left([U,V]_S(i),\ j\right) && \text{(by def. of –)} \\
&= \begin{cases} \text{lookup}_d\left(U(i),\ j\right) & \text{if } i \in S \\ \text{lookup}_d\left(V(i),\ j\right) & \text{otherwise} \end{cases} && \text{(by def. of choice operator)} \\
&= \begin{cases} \overline{U}(i,j) & \text{if } i \in S \\ \overline{V}(i,j) & \text{otherwise} \end{cases} && \text{(by def. of –)} \\
&= \left(\left[\overline{U},\ \overline{V}\right]_S\right)(i,j) && \text{(by def. of choice operator)}
\end{aligned}$$

□

A.2 Ω_2: Composition

Lemma 8. (Synchronous Ω_2). *The Ω_2 model using the synchronous schedule ψ^{sync} is equal to the synchronous model Γ_2.*

$$\forall t, \mathcal{V}, \mathcal{I}, \mathcal{O}.\ \Omega_2^t\left(\psi^{sync}\right)(\mathcal{V}, \mathcal{I}, \mathcal{O}) = \Gamma_2^t\left(\mathcal{V}, \mathcal{I}, \mathcal{O}\right)$$

Proof. We prove by induction on $t \in \mathbb{N}$.

Base case t = 0. Trivially holds by the definitions of Ω_2 of Γ_2.

Inductive step t + 1. Let $(\mathcal{V}, \mathcal{I}, \mathcal{O})$ be the initial routing state. Note that $\mathcal{I}^k$ denotes the import tables of Ω_2 at time k, with an initial state $(\mathcal{V}, \mathcal{I}, \mathcal{O})$.

$$\begin{aligned}
& \Omega_2^{t+1}\left(\psi^{sync}\right)(\mathcal{V}, \mathcal{I}, \mathcal{O}) \\
= & \begin{cases} \left[\Gamma_{2,\mathcal{V}}(\mathcal{I}^{\beta_V(t+1)}),\ \mathcal{V}^t\right]_{\alpha_V(t+1)} \\ \left[\Gamma_{2,\mathcal{I}}(\mathcal{O}^{\beta_I(t+1)}),\ \mathcal{I}^t\right]_{\alpha_I(t+1)} \\ \left[\Gamma_{2,\mathcal{O}}(\mathcal{V}^{\beta_O(t+1)}),\ \mathcal{O}^t\right]_{\alpha_O(t+1)} \end{cases} & \text{(by def. of } \Omega_2\text{)} \\
= & \left(\Gamma_{2,\mathcal{V}}(\mathcal{I}^t),\ \Gamma_{2,\mathcal{I}}(\mathcal{O}^t),\ \Gamma_{2,\mathcal{O}}(\mathcal{V}^t)\right) & \text{(by def. of synchronous schedule)} \\
= & \Gamma_2\left(\mathcal{V}^t, \mathcal{I}^t, \mathcal{O}^t\right) & \text{(by def. of } \Gamma_2\text{)} \\
= & \Gamma_2^{t+1}\left(\mathcal{V}, \mathcal{I}, \mathcal{O}\right) & \text{(by } \mathbf{IH}\text{)}
\end{aligned}$$

□

A.3 Ω_3: Hard-state

Lemma 11. (Synchronous Ω_3). *The Ω_3 model using the synchronous schedule ψ^{sync} is equal to the synchronous model Γ_3.*

$$\forall t, \mathcal{V}, \mathcal{I}, \mathcal{O}, (\nabla, \Delta).\ \Omega_3^t\left(\psi^{sync}\right)\left(\mathcal{V}, \mathcal{I}, \mathcal{O}, (\nabla, \Delta)\right) = \Gamma_3^t\left(\mathcal{V}, \mathcal{I}, \mathcal{O}, (\nabla, \Delta)\right)$$

Proof. We prove by induction on $t \in \mathbb{N}$.

Base case t = 0. Trivially holds by the definitions of Ω_3 and Γ_3.

Inductive step t + 1. Let $\left(\mathcal{V}, \mathcal{I}, \mathcal{O}, (\nabla, \Delta)\right)$ be the initial routing state. Note that $\mathcal{I}^k$ de-

notes the import tables of Ω_3 at time k, with an initial state $\left(\mathcal{V}, \mathcal{I}, \mathcal{O}, (\nabla, \Delta)\right)$.

$$
\begin{aligned}
& \Omega_3^{t+1}\left(\psi^{sync}\right)\left(\mathcal{V}, \mathcal{I}, \mathcal{O}, (\nabla, \Delta)\right) \\
= & \begin{cases}
\left[\Gamma_{3,\mathcal{V}}(\mathcal{I}^{\beta_V(t+1)}),\ \mathcal{V}^t\right]_{\alpha_V(t+1)} \\
\left[\Gamma_{3,\mathcal{I}}\left(\mathcal{I}^t, (\nabla^{\beta_I(t+1)}, \Delta^{\beta_I(t+1)})\right),\ \mathcal{I}^t\right]_{\alpha_I(t+1)} \\
\left[\Gamma_{3,\mathcal{O}}(\mathcal{V}^{\beta_O(t+1)}),\ \mathcal{O}^t\right]_{\alpha_O(t+1)} \\
\left[\Gamma_{3,D}(\mathcal{V}^{\beta_D(t+1)}, \mathcal{O}^{\beta_D(t+1)}),\ (\nabla^t, \Delta^t)\right]_{\alpha_D(t+1)}
\end{cases} && \text{(by def. of } \Omega_3\text{)} \\
= & \left(\Gamma_{3,\mathcal{V}}(\mathcal{I}^t),\ \Gamma_{3,\mathcal{I}}\left(\mathcal{I}^t, (\nabla^t, \Delta^t)\right),\ \Gamma_{3,\mathcal{O}}(\mathcal{V}^t),\ \Gamma_{3,D}(\mathcal{V}^t, \mathcal{O}^t)\right) && \text{(by def. of sync. schedule)} \\
= & \Gamma_3\left(\mathcal{V}^t,\ \mathcal{I}^t,\ \mathcal{O}^t,\ (\nabla^t, \Delta^t)\right) && \text{(by def. of } \Gamma_3\text{)} \\
= & \Gamma_3^{t+1}\left(\mathcal{V}, \mathcal{I}, \mathcal{O}, (\nabla, \Delta)\right) && \text{(by } \mathbf{IH}\text{)}
\end{aligned}
$$

□