Gossip in NetKAT

MSc Thesis *(Afstudeerscriptie)*

written by

Jana Wagemaker
(born March 14th, 1993 in Amsterdam, Netherlands)

under the supervision of **Prof Dr D. J. N. van Eijck**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:	**Members of the Thesis Committee:**
August 24th, 2017	Prof Dr R. M. de Wolf (chair)
	Prof Dr J. J. M. M. Rutten
	Dr C. Schaffner
	Prof Dr D. J. N. van Eijck

Institute for Logic, Language and Computation

Contents

Acknowledgements

First of all I would like to thank my supervisor, Jan van Eijck. He introduced me to this topic, and without our discussions, both technical and motivational, this thesis would not have been possible. I would also like to specially thank Malvin Gattinger. For just being a nice person to hang out with, but also for discussions about my thesis, reading it and providing incredibly thoughtful feedback, and mostly for his help in creating clarity and guidance in the Haskell maze. Without him the code would have been possibly non-existent but would have definitely been a lot less readable. He did not only provide tips and tricks, but also always managed to do this in such a way that I understood perfectly what was going on. And that comes from a person that did not know what the 'terminal' was a year ago.

I would like to thank Jan Rutten, for providing valuable help and feedback on the coalgebraic part of this thesis. He was always available for my probably often dumb beginner coalgebraic questions and motivational emails. I would also like to thank Tobias Kappé for reading parts of my thesis, providing valuable feedback and always answering questions about NetKAT. In this line there should also be a place for Nate Foster and Alexandra Silva, who both worked a lot on this topic and always answered my questions. Next I would like to thank Dora Achourioti, who is a great person in general, but also introduced me to the field of Logic during my bachelor and has always been a role model. A special thanks also to the committee members who have not been mentioned yet, Christian Schaffner and Ronald de Wolf, for being in this committee and taking the time to read this thesis. Christian, thank you as well for always providing a listening ear as my academic mentor throughout these two years.

And then lastly there is a place to thank my friends and family. Esteban, without your continuous support and faith in me this thesis would have not happened. Thank you also for reading this thesis and providing feedback. And for always giving me cuddles in exactly the right moments. Same goes for my mom and dad. Mom, thank you for always listening to me in desperate-thesis-moments and always believing that I could do it. Dad, thanks for feeding me nice and healthy food every week and still being kind no matter how stressed or grumpy I arrived. Thanks also to the other master of logic students for the nice times in Amsterdam, and then especially to the complexity people (could not have done it without you!!), Jakob, Lucy, Anthi, Pablo and Valery. Laura and Lucy, I would also like to thank both of you especially for setting up and

organising Ex Falso activities, they have been a lot of fun. Lastly I would like to thank my non logic friends, for being who they are and putting up with me during this masters and during the final months of crazy thesis writing.

Abstract

In this thesis we combine the study of NetKAT with the study of dynamic gossip. NetKAT is a sound and complete network programming language based on Kleene algebra with tests, originally used to describe packets traveling through networks. It has a coalgebraic decision procedure deciding NetKAT equivalences. Dynamic gossip is a field in which it is studied how information spreads through a gossip graph, where the links in the gossip graph are changeable (dynamic). So far no sound and complete logic is available that describes dynamic gossip in a satisfactory way. In this thesis we explore the application of NetKAT to dynamic gossip. It is shown that a change of perspective on NetKAT in which we keep track of the state of the switches in a network allows us to use NetKAT to model the Learn New Secrets Protocol, which is one of many gossip protocols. The addition to NetKAT to keep track of the state of switches is formalised and shown to remain sound and complete with respect to the packet-processing model. It is shown that the notions of weak successfulness, strong successfulness and sun graph are formalisable in NetKAT and that the theorem from dynamic gossip "Gossip graph G is a sun if and only if the LNS protocol is strongly successful on G" is also a theorem of NetKAT. In this thesis we thus extensively study NetKAT and explore a concrete application of the language: we show how the framework of NetKAT can be used to formalise and analyse dynamic gossip.

Chapter 1

Introduction

In this thesis we combine two fields of study that have not been combined before. The first of those fields is the study of dynamic gossip. Developed in [25], dynamic gossip studies how information diffuses in a dynamic network. Gossip protocols are procedures for spreading secrets among a group of agents, using a connection graph [6]. In the connection graph the agents are represented by the nodes, and the connections represent the ability of the agents to contact each other. It is assumed that every agent starts out having a unique secret. In dynamic gossip the links in the connection graphs can change and it is studied formally how the secrets of the agents spread throughout these changing graphs.

The second of those fields is the study of NetKAT. Recently, a new network programming language called NetKAT was developed. NetKAT emerged from a growing interest in high-level languages for programming networks. So far the development of such network programming languages had been largely done per network, specifically for the needs of that network, and not governed by any foundational principles. Networks have been built in the same way roughly since the 1970s. These old designs and this lack of foundational principles made it difficult for instance to incorporate new features in a language: how should the new features interact with the old ones, and how can one reason precisely about the code.

However, the recent rise of software-defined networking (SDN) has started to change the field. The revolutionary aspect of SDN is that there is a physical separation of the network control plane, which draws the network topology, and the forwarding plane, which decides what to do with incoming packets [27]. There is a controller machine managing network events such as new connections from hosts, topology changes, and shifts in traffic load by re-programming the switches accordingly. The controller has a global view of the network, which opens up possibilities for a variety of implementations using SDN.

To resolve aforementioned issues with network languages, domain-specific languages have been developed for SDN in recent years. One of such languages is NetKAT [2]. NetKAT is based on a solid mathematical foundation and comes equipped with a sound and complete equational theory, which makes it possible

to reason precisely about the code and when adding new constructs to determine how they should interact with the existing primitives. Formally, NetKAT is an instance of a mathematical structure called a Kleene algebra with tests (KAT), and it is proven to be sound and complete with respect to its denotational semantics. An additional advantage of NetKAT is that it unifies reasoning about switches and topology by incorporating the network topology in the programming language as well. We will see this in detail later. Incorporating network topology in the programming language is useful when one wants to answer almost any interesting question about the network such as "Can A send a packet to B?", or "Does traffic from A to B pass switch C?", as end-to-end behaviour of a network is determined both by switch-behaviours and network topology together.

In [2] it is shown that a decision procedure for NetKAT exists, based on the fact that Kleene algebras with tests have a decision procedure. The problem of deciding NetKAT equivalences is shown to be PSPACE-complete. In [8] the coalgebraic theory of NetKAT is developed, which leads to a new efficient algorithm for deciding equivalence using a specialised version of the Brzozowski derivative and bisimulations: the algorithm constructs a bisimulation between coalgebras built from NetKAT expressions via the Brzozowski derivative. An implementation of this algorithm is given in OCaml, a functional programming language.

It turned out that NetKAT can be used in practical applications such as for instance syntactic techniques for checking reachability [2]. In [8] experimental results are reported demonstrating that NetKAT and the coalgebraic decision procedure are competitive with state-of-the-art tools on several benchmarks including all-pairs connectivity, loop-freedom, and checking the correctness of compilers. For instance, it can verify reachability in a real-world campus network in less than a second on a laptop.

In this thesis we will start by exploring NetKAT in detail and its coalgebraic decision procedure. The second half of this thesis will be dedicated to investigating whether NetKAT can be applied to dynamic gossip Protocols. We have added a notion of state to NetKAT and show that this version of NetKAT is still sound and complete with respect to its denotational model. Then we show how to simulate a specific dynamic gossip protocol in NetKAT with state. We will try to argue that NetKAT with state can be an appropriate language to capture dynamic gossip.

Before we get further into the study of NetKAT, we will first explore the foundations that it is based on. First, we will discuss deterministic automata. We will explore a coalgebraic representation of automata, as that will be useful in understanding the coalgebraic decision procedure for NetKAT. Then we will look at the important concept of regular expressions and at the relation between automata and regular expressions via Kleene's theorem. Next we will describe Kleene algebras in detail and the extension of Kleene algebras to Kleene algebras with tests. Then we will continue with some generalities about coalgebraic theory with respect to automata, with the main result being presented that automata are coalgebras as well [20]. Furthermore, this chapter will demonstrate the relations between regular expressions and coalgebras and how this can be used for NetKAT. Chapter 5 concerns NetKAT and gives an overview of its syntax and

semantics and general results, including an overview of its coalgebraic decision procedure. In the next chapter we explore an extension of NetKAT and apply it to dynamic gossip protocols by simulating a specific gossip protocol in NetKAT with state and translating results and notions from dynamic gossip into NetKAT with state. In Chapter 7 we will give an implementation in Haskell of the simulation of the gossip protocol studied in Chapter 6, but it is not necessary to grasp this in order to follow the story of this thesis. At the end of Chapter 7, we present an example output which demonstrates that NetKAT correctly simulates the gossip protocol, which can be understood without studying the full implementation. The implementation chapter can also be used (also without understanding the actual code) to better understand the reasoning behind the protocol simulation presented in Chapter 6 as it gives a functional specification of the protocol.

In summary, the contributions of this thesis are as follows. We provide an overview of the study of NetKAT and its coalgebraic theory that is not yet presented as such anywhere else. In addition, we provide a different perspective on NetKAT by adding a notion of state. We show that we keep soundness and completeness, thereby demonstrating that NetKAT can be applied to dynamic gossip, which is a field of study NetKAT had not yet been applied to before. Lastly we give a new perspective on dynamic gossip, using a sound and complete language to model a certain type of dynamic gossip protocol and formalise notions from dynamic gossip.

Chapter 2

Automata Theory

As mentioned in the introduction, NetKAT is based on a Kleene algebra with tests. Kleene algebras can be roughly characterised as the algebras of regular expressions. This brings us to the beginning of this thesis: an introduction to automata theory and the relation between automata and regular expressions. This is the preparation that is needed to understand Kleene algebras and how they are used in NetKAT, which we will discuss in subsequent chapters.

2.1 Deterministic Automata

We will start by giving two essentially equivalent definitions of deterministic automata. One of the definitions is more natural in the setting of coalgebras, and the other one is the classical definition of deterministic automata. It will be explained later in the thesis why exactly we refer to the non-classical definition as the coalgebraic definition. We will also introduce some standard concepts for automata theory such as bisimulations and homomorphisms.

2.1.1 Coalgebraic Definition

Following [20], we define deterministic automata first as follows. Let Σ be a possibly infinite set of input symbols. A deterministic automaton with input alphabet Σ is then a triple $S = (S, o_S, t_S)$ consisting of a set of states S, an output function $o_S : S \to 2$ and a transition function $t_S : S \to S^\Sigma$. Note that 2 denotes the set $\{0, 1\}$ and $S^\Sigma = \{f \mid f : \Sigma \to S\}$ (all the functions from Σ to S). The output function o_S indicates whether a state is an accepting state ($o_S(s) = 1$) or a rejecting state ($o_S(s) = 0$). Intuitively, the transition function is a function that indicates which states can be moved to in response to an input: t_S assigns to each state s a function $t_S(s) : \Sigma \to S$ indicating which states can be reached from state s. In a deterministic automaton, this will always be only one state. More formally, $t_S(s)(a)$ represents which state is reached after an input symbol a has been consumed from state s. For convenience, we will sometimes

denote $o_S(s) = 1$ with $s \downarrow$, $o_S(s) = 0$ with $s \uparrow$ and $t_S(s)(a) = s'$ with $s \xrightarrow{a} s'$. If both the state space S of an automaton and the alphabet Σ are finite, the automaton is called a finite automaton. Here we place no restrictions on either Σ or S. Note that no state is designated as the start state, but when reasoning about automata we can appoint a specific state as the start state.

Let us introduce some standard terminology concerning languages. If Σ is an alphabet, we denote the set of all finite words over Σ with Σ^*. Concatenation of words w and w' is denoted with ww' and ϵ denotes the empty word. A language is now defined as any subset of Σ^*. The language accepted by a state s of an automaton $S = (S, o_S, t_S)$ is denoted by $\ell_S(s) = \{a_1...a_n | s \xrightarrow{a_1} s_1 \xrightarrow{a_2} ... \xrightarrow{a_n} s_n \downarrow\}$, where $s_1 = t_S(s)(a_1)$ and $s_{i+1} = t_S(s_i)(a_i)$ for $1 < i < n$. Note that if a state s is an accepting state, ϵ needs to be an element of $\ell_S(s)$.

Intuitively, following an example given by [17], a finite automaton with a designated start state s operates as follows. An input is a string x consisting of symbols from the alphabet Σ. Put a pebble down in the start state s. Then read the input string from left to right, one symbol at a time, and move the pebble according to the transition function t_S: if the next symbol of our string is a and we are in state p, then move the pebble to $t_S(p)(a)$. At the end of the string, the pebble will be in some state q. The string is said to be accepted by the finite automaton if $o_S(q) = 1$. In other words, if q is an accepting state.

2.1.2 Classical Definition

The coalgebraic definition of an automaton is not the classical one that can be found in for instance [17]. We will show that the classical definition and the one given in Section 2.1.1 are essentially the same. The reason we chose to present the definition from Section 2.1.1 for an automaton is because it allows for a natural translation into a coalgebraic representation, as we will see below.

In [17], a deterministic finite automaton is defined as a structure

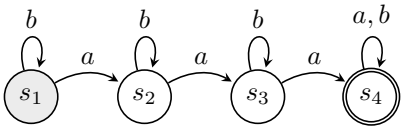
$$M = (Q, \Sigma, \delta, s, F)$$

where Q is a finite set of states, Σ is a finite input alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function such that if M is in state q and sees input a, it moves to state $\delta(q, a)$, $s \in Q$ is the start state and F is a subset of Q consisting of accepting states.

Example 1. To give an example, let us describe a simple four-state finite automaton. Let the set of states be $\{s_1, s_2, s_3, s_4\}$, the input alphabet $\{a, b\}$, the start state s_1, the set of accepting states $\{s_4\}$ and the transition function

$$\begin{aligned} \delta(s_1, a) &= s_2 \\ \delta(s_2, a) &= s_3 \\ \delta(s_3, a) = \delta(s_4, a) &= s_4 \\ \delta(q, b) &= q \text{ for } q \in \{s_1, s_2, s_3, s_4\} \end{aligned}$$

More insightful than the above description, is the diagram below:



The start state is shaded grey, and the accepting states are indicated by double circles. For this particular automaton, it can be seen that any string containing at least three a's will be an accepting string, and strings with fewer a's will not be.

As is stated in [20], it is not necessary to indicate one particular state as a start state as for the coalgebraic representation we are not particularly interested in it. It is also not required that the alphabet and set of states are finite. Then we get that both definitions of automata are equivalent, because of bijections between

$$\mathcal{P}(S) \cong (S \to 2) \text{ and } (S \times \Sigma \to S) \cong (S \to S^{\Sigma}).$$

Hence we have that there is a one-to-one correspondence between (S, F, δ) and (S, o_S, t_S).

2.1.3 Bisimulations and Homomorphisms

Two very important definitions for automata are the definitions of bisimulation and homomorphism. The definitions can be formulated in different ways, and we decided to follow [20]. Here the definitions for deterministic automata are given but they can be easily generalised to the definition for non-deterministic automata.

Definition 1. A bisimulation between two automata $S = (S, o_S, t_S)$ and $T = (T, o_T, t_T)$ is a relation $R \subseteq S \times T$ with for all $s \in S$, $s' \in T$ and $a \in \Sigma$:

$$\text{If } sRs' \text{ then } o_S(s) = o_T(s') \text{ and } t_S(s)(a)\, R\, t_T(s')(a).$$

Unions and compositions of bisimulations are bisimulations again. We write $s \sim s'$ whenever there exists a bisimulation R with sRs'.

Another important definition is the one of a homomorphism between two automata S and T.

Definition 2. A homomorphism between two automata $S = (S, o_S, t_S)$ and $T = (T, o_T, t_T)$ is any function $f : S \to T$ such that for all $s \in S$ and $a \in \Sigma$:

$$o_S(s) = o_T(f(s)) \text{ and } f(t_S(s)(a)) = t_T(f(s))(a),$$

The notions of automaton, homomorphism and bisimulation are closely related. A function $f : S \to T$ is a homomorphism if and only if the relation

$$R := \{(s, f(s)) | s \in S\}$$

is a bisimulation. For the left to right direction, we need to prove that $o_S(s) = o_T(f(s))$, which is immediate from the assumption that f is a homomorphism from S to T, and that $t_S(s)(a)\, R\, t_T(f(s))(a)$. For the latter fact, note that we have that

$$f(t_S(s)(a)) = t_T(f(s))(a),$$

so we need to prove that

$$t_S(s)(a) R f(t_S(s)(a))$$

This is immediate from the definition of R. For the right to left direction, we again have that the first condition for f to be a homomorphism from S to T is satisfied by definition. What is left to prove is that

$$f(t_S(s)(a)) = t_T(f(s))(a)$$

From the assumption that R is a bisimulation, we know that

$$t_S(s)(a) R t_T(f(s))(a)$$

From the definition of R we then immediately get that

$$f(t_S(s)(a)) = t_T(f(s))(a).$$

Also important to note is that bisimulations are automata themselves (we will use this in Chapter 4): Let R be a bisimulation between S and T, then we can take

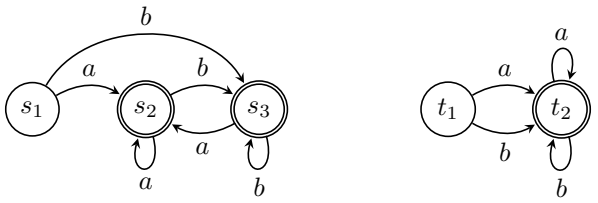
$$o_R : R \to 2 \text{ and } t_R : R \to R^{\Sigma}$$

such that for $(s, s') \in R$ and $a \in \Sigma$ we have

$$o_R((s, s')) = o_S(s) = o_T(s') \text{ and } t_R((s, s')) = (t_S(s)(a), t_T(s')(a))$$

This defines an automaton (R, o_R, t_R).

Example 2. To clarify, let us look at an example of two automata and a bisimulation between them. We take a simple alphabet, $A = \{a, b\}$, and consider automata $S = \{s_1, s_2, s_3\}$ and $T = \{t_1, t_2\}$. The circles represent the states, and a state with a double circle denotes an accepting state. The arrows represent the transitions between the states on the input letters labeling the arrows.



A bisimulation between them is $\{(s_1, t_1), (s_2, t_2), (s_3, t_2)\}$. To formally show this, note that

$$\begin{aligned} o_S(s_1) &= o_T(t_1) \\ o_S(s_2) &= o_T(t_2) \\ o_S(s_3) &= o_T(t_2) \end{aligned}$$

and

$$\begin{aligned} t_S(s_1)(a) &= s_2 R t_2 = t_T(t_1)(a) \\ t_S(s_1)(b) &= s_3 R t_2 = t_T(t_1)(b) \\ t_S(s_2)(a) &= s_2 R t_2 = t_T(t_2)(a) \\ t_S(s_2)(b) &= s_3 R t_2 = t_T(t_2)(b) \\ t_S(s_3)(a) &= s_2 R t_2 = t_T(t_2)(a) \\ t_S(s_3)(b) &= s_3 R t_2 = t_T(t_2)(b) \end{aligned}$$

So all requirements of a bisimulation are satisfied.

2.2 Regular Expressions

Regular expressions originated with Kleene, describing regular languages and what he called regular sets, which we will talk about later. Essentially, regular expressions are a sequence of characters defining a search pattern, making them widely used in search engines and for instance in search and replace functions in text editors. Of course 'patterns' is not a precisely defined term, but patterns using only the specific set of symbols given below, are called regular expressions. Later we will study the axiomatisation of regular expressions using Kleene algebras.

Let the set $\mathcal{R}$ of regular expressions over the finite input alphabet Σ be given by the following syntax:

$$r ::= \underline{0} \mid \underline{1} \mid a \in \Sigma \mid r_1 + r_2 \mid r_1 \cdot r_2 \mid r^*$$

We will often write $r_1 r_2$ instead of $r_1 \cdot r_2$.

We define a function $\lambda : \mathcal{R} \to \mathcal{L}$ which assigns to a a regular expression E the language $\lambda(E)$ it represents. Such a language is called a regular language. Before we can define λ, we need to define some so-called regular operators for languages $K \subseteq \Sigma^*$ and $L \subseteq \Sigma^*$:

$$\begin{aligned} K + L &= K \cup L \\ K \cdot L &= \{vw \mid v \in K, w \in L\} \\ K^* &= \bigcup_{n \geq 0} K^n \end{aligned}$$

with $K^0 = \{\epsilon\}$ and $K^{n+1} = K \cdot K^n = KK^n$. Now we define λ by induction on the structure of E:

$$\begin{aligned}
\lambda(\underline{0}) &= \varnothing \\
\lambda(\underline{1}) &= \{\epsilon\} \\
\lambda(a) &= \{a\} \\
\lambda(r_1 + r_2) &= \lambda(r_1) + \lambda(r_2) \\
\lambda(r_1 \cdot r_2) &= \lambda(r_1) \cdot \lambda(r_2) \\
\lambda(r^*) &= \lambda(r)^*
\end{aligned}$$

2.2.1 Regular Expressions as an Automaton

We can turn the set of regular expressions into a deterministic automaton $\mathcal{R}$. This idea comes from Brzozowski who proposed this in 1964 [4] and gives rise to the concept of a derivative of regular expressions, or the Brzozowski derivative. The automaton has all the regular expressions that can be formed from a finite input alphabet Σ as its state space, and a function $o_{\mathcal{R}}$ for determining acceptance and a function $t_{\mathcal{R}}$ giving the transitions to a next state. Let us denote $t_{\mathcal{R}}(s)(a)$ with r_a, with r a regular expression (the state). Then we have that $t_{\mathcal{R}}$ is defined as

$$\begin{aligned}
(\underline{0})_a &= \underline{0} \\
(\underline{1})_a &= \underline{0} \\
(a')_a &= \begin{cases} \underline{1} & \text{if } a = a' \\ \underline{0} & \text{if } a \neq a' \end{cases} \\
(r_1 + r_2)_a &= (r_1)_a + (r_2)_a \\
(r_1 r_2)_a &= \begin{cases} (r_1)_a r_2 & \text{if } o_{\mathcal{R}}(r_1) = 0 \\ (r_1)_a r_2 + (r_2)_a & \text{otherwise} \end{cases} \\
(r^*)_a &= r_a r^*
\end{aligned}$$

and $o_{\mathcal{R}}$ is defined as

$$\begin{aligned}
o_{\mathcal{R}}(\underline{0}) &= 0 \\
o_{\mathcal{R}}(\underline{1}) &= 1 \\
o_{\mathcal{R}}(a) &= 0 \\
o_{\mathcal{R}}(r_1 + r_2) &= o_{\mathcal{R}}(r_1) \vee o_{\mathcal{R}}(r_2) \\
o_{\mathcal{R}}(r_1 r_2) &= o_{\mathcal{R}}(r_1) \wedge o_{\mathcal{R}}(r_2) \\
o_{\mathcal{R}}(r^*) &= 1
\end{aligned}$$

In the definition of $o_{\mathcal{R}}$ we use that $\{0, 1\}$ can be given a lattice structure $(\{0, 1\}, \vee, \wedge, 0, 1)$ such that zero is neutral with respect to $\vee$ and 1 with respect to $\wedge$. We will come back to this in the chapter on Kleene algebras. Intuitively, a

regular expression r represents an accepting state whenever the empty string is a member of the regular language corresponding to r. The regular expression r_a (if r is a regular expression, then so is r_a [4]) denotes the language containing all words w such that aw is in the language denoted by r.

Later in this thesis, we will use the concept of regular expressions as an automaton in the coalgebraic decision procedure for NetKAT. Before we will see it at use there, we will revisit this automaton structure in the chapter about coalgebras.

2.3 Kleene's Theorem

The regular languages and finite automata introduced before, can be related in a very precise manner. This is done via an important theorem known as Kleene's theorem [12]:

Theorem 1. A set $A \subseteq \Sigma^*$ is the language accepted by a finite automaton M if and only if $A = \lambda(\alpha)$ for some regular expression α.

A proof of this theorem can be found in [17].

2.3.1 Regular Sets

To rephrase this theory and introduce some standard terminology, let us introduce the definition of a regular set according to [17]. A regular set is defined as a set $A \subseteq \Sigma^*$ such that A is the language accepted by some finite automaton. For Example 1, the set of strings accepted by the automaton were the strings containing at least three a's. Hence, the set

$$\{x \in \{a, b\}^* \mid x \text{ contains at least three } a\text{'s}\}$$

is a regular set. Note the difference here between a regular set and a regular language: A is a regular set if A is the language accepted by some finite automaton and A is a regular language if $A = \lambda(\alpha)$ for some regular expression α.

This means we can formulate Kleene's theorem as: A set $A \subseteq \Sigma^*$ is a regular set if and only if A is a regular language.

2.3.2 Examples

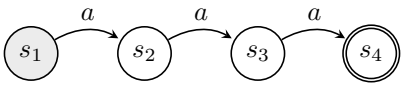
In this section we will demonstrate the principle of converting an automaton M into the corresponding regular expression α such that $\lambda(\alpha) = L(M)$ where $L(M)$ is the language accepted by M and vice versa for small automata. There are algorithms for converting automata into regular expressions and the other way around, but the demonstration here will be informal.

Example 3. We will start with the regular expression $(aaa)^* + (aaaaa)^*$ and we will construct a finite automaton M such that $L(M) = \lambda((aaa)^* + (aaaaa)^*)$.

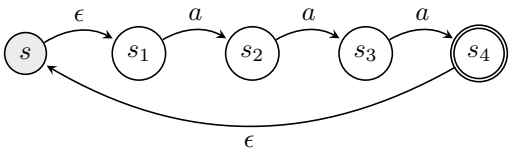
Note that

$$\lambda((aaa)^* + (aaaaa)^*) = \lambda((aaa)^*) \cup \lambda((aaaaa)^*) = \bigcup_{n\geq 0} (aaa)^n \cup \bigcup_{n\geq 0} (aaaaa)^n$$

We first take an automaton such that the set of strings accepted by the automaton is $L(aaa) = \{aaa\}$.

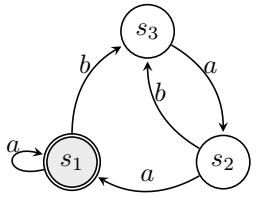


Then add a new start state s and ϵ-transition from s to the old start state and from the accepting state to s:



Do a similar thing for an automaton that accepts the set of strings $\lambda(aaaaa) = \{aaaaa\}$. Now we thus have two automata, one of which accepts the set of strings $\lambda((aaa)^*)$ and one of which accepts the set of strings $\lambda((aaaaa)^*)$. The automaton accepting $\lambda((aaa)^* + (aaaaa)^*)$ is simply the disjoint union of these two automata.

Example 4. Now let us look at an example the other way around, thus going from an automaton M to a regular expression α such that $\lambda(\alpha) = L(M)$. Consider the automaton



We will now define the regular expression representing this automaton. We see that s_1 is the only start and accept state. We will explain the reasoning behind our procedure as we go. To start with we pick state s_3 as the state that we will remove, but we could have chosen s_2 as well. We know that every path in the automaton that leads to acceptance must go from s_1 to s_1. Such a path either passes through s_3 or does not pass through s_3. In case the path includes s_3 it must go from s_1 to s_3, then pass s_3 finitely many times and lastly go from s_3 to s_1.

Via informal reasoning (looking at the automaton) we can see that the only path going from s_1 to s_1 without using state s_3 is the loop labeled with a, so we have that the regular expression

$$a^*$$

is a string that leads to acceptance. Now we will identify the regular expression that takes us from s_1 to s_3, then travels from s_3 to s_3 finitely many more times, and then takes us from s_3 to s_1. We see that the only path from s_1 to s_3 is a^*b. Similarly to travel from s_3 to s_3 we have a regular expression that looks like $\epsilon + ab + aaa^*b$. Finally, to travel from s_3 to s_1 we have the regular expression aaa^*.

Now we can conclude that $r = a^* + a^*b(\epsilon + ab + aaa^*b)aaa^*$ is the regular expression such that $\lambda(r)$ is the set of strings accepted by the automaton.

Chapter 3

Kleene Algebras

Now that we have seen regular expressions and their link to automata, we will formalise regular expressions a bit more. Roughly said, Kleene algebra is the algebra of regular expressions. More formally, Kleene algebras are algebraic structures with operators $+, \cdot, ^*, 0, 1$ satisfying certain axioms. Although Kleene did not define Kleene algebras, he laid the theoretical groundwork [12]. Originally, he studied regular expressions and their algebraical laws. Kleene posed the axiomatisation of these what he called 'regular events' as an open problem. Important contributions to this field were made by Conway [5], among others. See [16] for an overview. As we will see below, an important Kleene algebra is indeed the family of regular sets over a finite alphabet, but there are many more.

As mentioned before, many scholars have studied the axiomatisation of Kleene algebras since 1956. In fact, there are several inequivalent definitions of Kleene algebras that can be found in the literature. In this thesis we will follow the definition of Kleene algebras (KAs) as can be found in [16], which seems to be the most widely adopted definition. Kozen defines a Kleene algebra as any model of the equations and equational implications given in Section 3.1.

Kleene algebras can be found in a number of settings, and a very natural one is the field of automata and logic of programs, which is the field relevant to us. Namely, an important example of a Kleene algebra is the family of regular sets over a finite alphabet Σ, using the definition of regular sets given in Section 2.3.1. We then have the $+$, $\cdot$ and * operators as defined before, $\varnothing$ for 0, and $\{\epsilon\}$ for 1 [17]. The equational theory of this Kleene algebra has been called the algebra of regular events.

Another example of a Kleene algebra is the family of binary relations on a set with the operations of $\cup$ for $+$, the relational composition

$$R \cdot S = \{(x, z) \mid \exists y (x, y) \in R, (y, z) \in S\}$$

for $\cdot$, the empty relation for 0, the identity relation for 1 and reflexive transitive closure for *.

3.1 Axioms for Kleene Algebras

More formally [16], a Kleene algebra is an algebraic structure

$$\mathcal{K} = (K, +, \cdot, ^*, 0, 1)$$

satisfying the following equations and equational implications:

$$p + (q + r) \equiv (p + q) + r \tag{3.1}$$
$$p + q \equiv q + p \tag{3.2}$$
$$p + 0 \equiv p \tag{3.3}$$
$$p + p \equiv p \tag{3.4}$$
$$p(qr) \equiv (pq)r \tag{3.5}$$
$$1p \equiv p \tag{3.6}$$
$$p1 \equiv p \tag{3.7}$$
$$p(q + r) \equiv pq + pr \tag{3.8}$$
$$(p + q)r \equiv pr + qr \tag{3.9}$$
$$0p \equiv 0 \tag{3.10}$$
$$p0 \equiv 0 \tag{3.11}$$
$$1 + pp^* \equiv p^* \tag{3.12}$$
$$1 + p^*p \equiv p^* \tag{3.13}$$
$$q + pr \leq r \Rightarrow p^*q \leq r \tag{3.14}$$
$$q + rp \leq r \Rightarrow qp^* \leq r \tag{3.15}$$

where $\leq$ refers to the natural partial order on $\mathcal{K}$:

$$p \leq q \Leftrightarrow p + q \equiv q.$$

As we can see from axioms $(12 - 15)$, the *-operator behaves like the Kleene star operator or the reflexive transitive closure operator of relational algebra. Such structures can also be classified as idempotent semirings.

3.2 General Properties of Kleene Algebras

It follows quite easily that $\leq$ is a partial order:

- $\leq$ is reflexive: $a \leq a$ as $a + a \equiv a$ by (3.4)
- $\leq$ is transitive: $a \leq b$ and $b \leq c$ gives $a \leq c$ as we have that $a + b \equiv b$ and $b + c \equiv c$, which gives us that $a + c \equiv a + b + c \equiv b + c \equiv c$ and hence we have that $a \leq c$.
- $\leq$ is antisymmetric: $a \leq b$ and $b \leq a$ gives us that $a + b \equiv b$ and $b + a \equiv a$, and as $a + b \equiv b + a$, this gives us that $a \equiv b$.

Note also that all operators are monotone with respect to $\leq$.

Another important thing to notice is that $a + b$ is the least upper bound of a and b with respect to $\leq$. We have three cases to distinguish:

- $a \equiv b$, in this case $a + b \equiv a \equiv b$ and it is trivial that this is the least upper bound.
- $a \leq b$, in this case we have that $a + b \equiv b$, which means that $a + b$ is indeed an upper bound of both a and b. Now suppose that $a + b$ is not the least upper bound. So there exists an element d such that $d \leq a + b$ and $a \leq d$ and $b \leq d$. The latter two facts give us that $a + d \equiv d$ and $b + d \equiv d$ and $d \leq a + b$ gives us that $d + a + b \equiv a + b$. Combining that with the fact that $a + b \equiv b$ via substitution, we get that

  $$a + b \equiv d + a + b \equiv d + b \equiv b + d \equiv d$$

  Hence, we get that $a + b \equiv d$, which is a contradiction with the assumption that $a + b$ is not the least upper bound. Thus $a + b$ is the least upper bound of a and b.
- $b \leq a$, similar as for the previous case.

From this we can conclude that $\mathcal{K}$ is an upper semilattice with join given by $+$ and minimum element 0.

3.2.1 Soundness and Completeness

In [16], Kozen proves the soundness and completeness of the axioms stated in Section 3.1 for the algebra of regular events. In other words, we have that two regular expression α and β over Σ denote the same regular set if and only if $\alpha \equiv \beta$ is a theorem of Kleene algebra (i.e. a logical consequence of the axioms).

3.3 Kleene Algebra with Tests

One of the most well-studied extensions of Kleene algebra is Kleene algebra with tests [15], [18]. This is an algebraic structure consisting of a Kleene algebra with an embedded Boolean subalgebra. Its soundness and completeness with respect to relational models and language-theoretic models is proven in [18]. Formally, a Kleene algebra with tests (KAT) is a two-sorted algebraic structure

$$(K, B, +, \cdot, ^*, 0, 1, \neg)$$

where $\neg$ is a unary operator defined only on B such that

- $(K, +, \cdot, ^*, 0, 1)$ is a Kleene algebra,
- $(B, +, \cdot, 0, 1, \neg)$ is a Boolean algebra and
- $(B, +, \cdot, 0, 1, \neg)$ is a subalgebra of $(K, +, \cdot, ^*, 0, 1)$.

The axioms of KAT are the axioms of Kleene algebra listed in Section 3.1 plus additional axioms for Boolean algebras given below. Usually in work on KAT, the elements of B are referred to as tests and the elements of K as actions. In this paper, conform NetKAT standards, we will call them predicates and policies respectively. We use the letters $a, b, c, d, ..$ for predicates and $p, q, r, s, ...$ for policies. The additional axioms of a Boolean algebra needed are:

$$a + (b \cdot c) \equiv (a + b) \cdot (a + c) \tag{3.16}$$
$$a + 1 \equiv 1 \tag{3.17}$$
$$a + \neg a \equiv 1 \tag{3.18}$$
$$a \cdot b \equiv b \cdot a \tag{3.19}$$
$$a \cdot \neg a \equiv 0 \tag{3.20}$$
$$a \cdot a \equiv a \tag{3.21}$$

Chapter 4

Automata and Coalgebras

As we have seen in the second chapter, we can approach classical deterministic automata in terms of the notions of homomorphisms and bisimulation, which are also cornerstones of the theory of coalgebra. We do not aim to give an introduction to the field of coalgebra. However, for the treatment of automata theory in terms of coalgebraic notions, we will explain the necessary background information. The main sources used for this are [20], [21] and [22]. Later we will give a short introduction into the coalgebraic theory of NetKAT.

There is a lot one can say about coalgebras, but as that is not the focus of this thesis, we will not spend too much time on the general field of coalgebra. However, some attention will be spent to create a general feeling for the coalgebraic perspective.

In short, one can describe coalgebras as mathematical structures that capture the essence of dynamic or evolving systems. Many structures in theoretical computer science can naturally be represented as coalgebras, one of which is the one we are interested in: automata as coalgebras. But one can also imagine other examples, such as infinite data structures and modeling non-well-founded sets. It is often noted that algebraic operations are ways to construct complex objects out of simple ones, and coalgebraic operations on the other hand should be seen as ways to unfold or observe objects. [26]

For the coalgebras we will consider, we can make it a bit more concrete [22]. Destructors (or observers, or transition functions) tell us what we can observe about states in our state space. For instance, the operators HEAD and TAIL tell us what we can observe about the elements of an infinite list: HEAD gives the first element in the list, so this is the direct observation, and TAIL gives the remaining part of the list, which is effectively the 'next state'. One can imagine that we can use coinductive definitions on infinite lists.

Example 5. Let us give an example of a typically coalgebraic phenomenon, taken from [22]. Consider a black-box machine with one external button and one light. The machine performs a certain action only if the button is pressed and the light goes on only if the machine stops operating. In terms of states: if

the machine has reached an accepting state. In that case, pressing the button has no effect anymore. A person on the outside of this machine cannot observe directly in which internal state the machine is; the user can only observe the machines behaviour via the button and the light. In other words, the only thing that can be observed directly about a particular state of the machine is whether the light is on or not. Hence, whether the state is an accepting state or not. By repeatedly pressing the button, the user can observe how many times the button needs to be pressed for the light to go on.

In [22], this example is described mathematically as saying we have a set X denoting the unknown state space of the machine, and a function

$$\text{button: } X \to \{*\} \cup X$$

where $*$ is a new symbol not occurring in X. In a particular state $x \in X$, applying the function 'button', which corresponds to pressing the button, has outcome $*$, meaning that the machine has stopped operating and the light goes on, or it has an outcome in X, meaning that the machine has moved to another state in X. The set X together with the function 'button' constitute an example of a coalgebra.

There are many examples of coalgebras similar to this one, and an important similarity between them is that in each case there is a state space X about which no assumptions are made, there is a function defined on this state space allowing one to observe some aspect of the state directly or to move on to next states and there is no other access to this state space other than for observing or modifying via these specified operations. By making successive observations, one can say something about the behaviour of a state.

This 'behaviour' of a state is closely related to the notion of bisimulation. The only 'observable' fact about a state in an automaton is whether it is an accepting state or not. Further information about a state can be acquired by offering an input symbol which leads to a new state. Two states between which there exists a bisimulation relation are said to be observationally indistinguishable: one can observe the same facts and offering any input symbol will lead to the same new states (again two indistinguishable states).

We can formalise this a bit more. Let $F : Set \to Set$ be a functor on the category of sets and functions. One can view the functor as the type of the coalgebra and in this thesis we will only consider coalgebras of the deterministic automaton type, introduced below. An F-coalgebra is a pair (S, α_S) consisting of a set S and a function $\alpha_S : S \to F(S)$. The set S is called the carrier of the coalgebra, or the set of states, and the function α_S is called the F-transition structure. We will see later that the deterministic automata functor we will consider has some special properties that turn out to be quite convenient. [21]

In the next section we will discuss the notions of bisimulation and homomorphism in their coalgebraic representation and show that this representation is in fact a general definition and the classical definitions of homomorphism and bisimulation for automata we had in Section 2.1.3 can be recovered as a special case of this general definition. We will see how to view automata as

coalgebras and we will demonstrate the proof principle of coinduction. We will see that the set of all languages is a final coalgebra, which is a concept that will be explained in due time, and how this relates to the automaton of the set of regular expressions and to automata in general. Towards the end of the chapter we will hint at how all these connections are used in the coalgebraic theory for NetKAT.

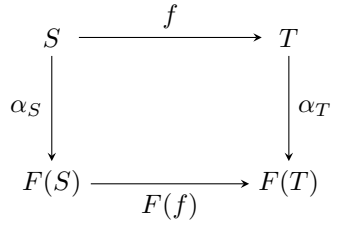
4.1 Bisimulations and Homomorphisms

We will start with the universal coalgebraic definitions of bisimulations and homomorphisms as can be found in [21]. Later on we will see that when viewing automata as coalgebras, these coalgebraic definitions coincide with the familiar definitions of bisimulation and homomorphism on automata.

Definition 3. Consider an arbitrary functor $F : Set \to Set$ and let (S, α_S) and (T, α_T) be two F-coalgebras. A function $f : S \to T$ is a homomorphism of F-coalgebras, or F-homomorphism, if

$$F(f) \circ \alpha_S = \alpha_T \circ f$$

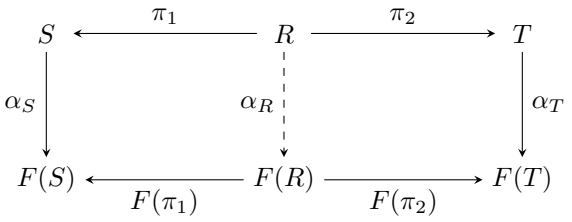
In diagrammatic representation:



If f is indeed a homomorphism, the diagram above commutes. The identity function on an F-coalgebra is always a homomorphism and the composition of two homomorphisms is again a homomorphism. Intuitively, a homomorphism is a function that is transition-preserving and reflecting.

A bisimulation between two coalgebras intuitively is a transition structure respecting a relation between sets of states.

Definition 4. Let $F : Set \to Set$ be an arbitrary set functor and let (S, α_S) and (T, α_T) be two F-coalgebras. A subset $R \subseteq S \times T$ is called an F-bisimulation between S and T if there exists an F-transition structure $\alpha_R : R \to F(R)$ such that the projections from R to S and T are F-homomorphisms. This is expressed by the following diagram:



Two states s and t are called bisimilar if there exists a bisimulation R such that $(s,t) \in R$.

4.2 Automata as Coalgebras

As can be found in [20], the reason to be interested in a coalgebraic representation of automata is that there exists a number of notions and results on coalgebras in general that can now be applied to automata. The reason why we introduced an alternative representation of automata is because it allows for a natural correspondence between automata and coalgebras [20]: automata can be seen as coalgebras.

Automata are coalgebras of the following functor $D : Set \to Set$. Let $D(S)$ for a set S be defined by $D(S) = 2 \times S^\Sigma$. How the functor D acts on functions we shall define below. For an automaton $S = (S, o_S, t_S)$, we had the functions $o_S : S \to 2$ and $t_S : S \to S^\Sigma$. We can combine them into a single function $(o_S, t_S) : S \to 2 \times S^\Sigma$, which is a function that maps a state $s \in S$ to the pair $(o_S(s), t_S(s))$. Our automaton S has thus been represented as a D-coalgebra: $(o_S, t_S) : S \to D(S)$. For a function $f : S \to T$ where both $(S, (o_S, t_S))$ and $(T, (o_T, t_T))$ are D-coalgebras, we have that the function $D(f) : (2 \times S^\Sigma) \to (2 \times T^\Sigma)$ is defined for any $x \in 2$ and $h \in S^\Sigma$ such that $D(f)((x,h)) = (x, f \circ h)$.

4.3 Instantiation of Definitions for Homomorphism and Bisimulation

4.3.1 Homomorphisms

We will now prove that the classical representation of bisimulation and homomorphism on automata are instances of the coalgebraic definition of the concepts of bisimulation and homomorphism.

Consider two D-coalgebras (hence, two automata) $(S, (o_S, t_S))$ and $(T, (o_T, t_T))$ with $(o_S, t_S) : S \to D(S)$ and $(o_T, t_T) : T \to D(T)$. Following the definition of a homomorphism for F-coalgebras given in Definition 3, a homomorphism of D-coalgebras is a function $f : S \to T$ if $D(f) \circ (o_S, t_S) = (o_T, t_T) \circ f$. Applying the definition of $D(f)$ given in the second paragraph of this section,

we get that

$$D(f) \circ (o_S, t_S)(s) = (o_S(s), f \circ t_S(s)) = (o_S(s), f(t_S(s)))$$

Using that

$$D(f) \circ (o_S, t_S) = (o_T, t_T) \circ f$$

we can conclude that

$$(o_S(s), f(t_S(s))) = (o_T, t_T) \circ f(s) = (o_T(f(s)), t_T(f(s))).$$

Looking at the last line, we can conclude that

$$o_S(s) = o_T(f(s)) \text{ and } f(t_S(s)) = t_T(f(s))$$

These two conclusions are exactly the requirements of a homomorphism between two automata we saw before.

4.3.2 Bisimulations

Now for bisimulations. Consider two D-coalgebras (hence, two automata) $(S, (o_S, t_S))$ and $(T, (o_T, t_T))$ with $(o_S, t_S) : S \to D(S)$ and $(o_T, t_T) : T \to D(T)$. Following the definition of a bisimulation for F-coalgebras given in Definition 4, a bisimulation R between D-coalgebras is a D-coalgebra structure $(o_R, t_R) : R \to D(R)$ such that the projections $\pi_1 : R \to S$ and $\pi_2 : R \to T$ are D-homomorphisms. Suppose that $(s, s') \in R$ with $s \in S$ and $s' \in T$. We need to show that

$$o_S(s) = o_T(s') \text{ and } t_S(s)(a) R t_T(s')(a)$$

for all $a \in \Sigma$. Given that π_1 and π_2 are D-homomorphisms, we get that

$$D(\pi_1) \circ (o_R, t_R) = (o_S, t_S) \circ \pi_1 \text{ and } D(\pi_2) \circ (o_R, t_R) = (o_T, t_T) \circ \pi_2$$

Writing the first half of these equations out using the definition of D on functions, we get that

$$\begin{aligned} D(\pi_1) \circ (o_R, t_R)(s, s') &= (o_R(s, s'), \pi_1 \circ t_R(s, s')) \\ &= (o_S, t_S) \circ \pi_1(s, s') \\ &= (o_S(s), t_S(s)) \end{aligned}$$

and

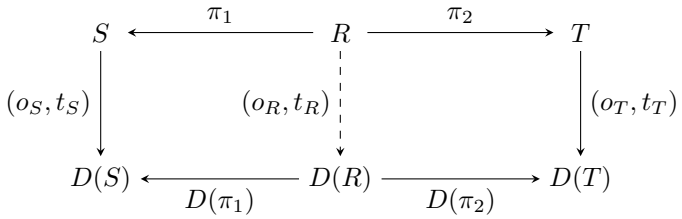
$$\begin{aligned} D(\pi_2) \circ (o_R, t_R)(s, s') &= (o_R(s, s'), \pi_2 \circ t_R(s, s')) \\ &= (o_T, t_T) \circ \pi_2(s, s') \\ &= (o_T(s'), t_T(s')) \end{aligned}$$

This immediately gives us that $o_R(s, s') = o_S(s)$ and $o_R(s, s') = o_T(s')$, and thus that $o_S(s) = o_T(s')$, as required.

To see that $(t_S(s)(a), t_T(s')) \in R$, we use the diagram first given above for F-coalgebras and repeated below for D-coalgebras. Suppose that $t_S(s)(a) = e$ for some $a \in \Sigma$. We know that $\pi_1(s, s') = s$, which implies that $t_S(\pi_1(s, s'))(a) = e$. From the fact that π_1 is a homomorphism, we then know that there must exist a pair $(u, w) \in R$ such that $t_R((s, s'))(a) = (u, w)$ and $\pi_1(u, w) = e$. This gives us that $u = e$ and thus that $(t_S(s)(a), w) \in R$. Now as π_2 is also a homomorphism and we know that $t_R((s, s'))(a) = (t_S(s)(a), w(a))$, we have that

$$\begin{aligned} D(\pi_2) \circ (o_R, t_R)(s, s')(a) &= (o_R(s, s'), \pi_2(t_S(s)(a), w(a))) \\ &= (o_R(s, s'), w(a)) \\ &= (o_T, t_T) \circ \pi_2(s, s')(a) \\ &= (o_T(s'), t_T(s')(a)) \end{aligned}$$

Hence we can conclude that $w = t_T(s')$ and $\big(t_S(s)(a), t_T(s')\big) \in R$, as required.



4.4 The set of all languages is an automaton

In this section we will demonstrate that the set of all languages is an automaton. The language recognised by a state s in an automaton is the behaviour (or the semantics) of s [24]. This means we can see the set of all subsets of Σ^*, the set of all languages over Σ, as the universe of all possible behaviours for automata. To turn this set into an automaton, we need some definitions.

Following [20] and [4], let

$$\mathcal{L} = \{L | L \subseteq \Sigma^*\}$$

be the set of all languages. For a word $w \in \Sigma^*$, the w-derivative of a language L is

$$L_w = \{v \in \Sigma^* | wv \in L\}$$

We also define an a-derivative, for $a \in \Sigma$:

$$L_a = \{v \in \Sigma^* | av \in L\}$$

We can use this derivative to turn the set $\mathcal{L}$ of all languages into an automaton $(\mathcal{L}, o_{\mathcal{L}}, t_{\mathcal{L}})$ with $o_{\mathcal{L}}(L) = 1$ if $\epsilon \in L$ and 0 otherwise and $t_{\mathcal{L}}(L)(a) = L_a$. This automaton is such that the language accepted by a state $L \in \mathcal{L}$ is exactly L itself. We will prove this later. We have the following rules for termination: $\varnothing \uparrow$, $\{\epsilon\} \downarrow$, $\{a\} \uparrow$, $(K + L) \downarrow$ if and only if $K \downarrow$ or $L \downarrow$, $(K \cdot L) \downarrow$ if and only if $K \downarrow$ and $L \downarrow$, and $K^* \downarrow$.

Hence, the set of all languages is a coalgebra with continuation and observation maps given by the Brzozowski derivative. In addition to the semantic definition, we have the following properties for calculating the a-derivative L_a of a language L:

$$
\begin{aligned}
\varnothing_a &= \varnothing \\
\{\epsilon\}_a &= \varnothing \\
\{b\}_a &= \begin{cases} \{\epsilon\}, & \text{if } b = a \\ \varnothing, & \text{if } b \neq a \end{cases} \\
(K + L)_a &= K_a + L_a \\
(K \cdot L)_a &= \begin{cases} K_a L, & \text{if } K \uparrow \\ K_a L + L_a, & \text{if } K \downarrow \end{cases} \\
(K^*)_a &= K_a K^*
\end{aligned}
$$

These rules are easily verified with the semantic definition.

4.5 ℓ_S is a homomorphism from S to $\mathcal{L}$

We have now seen that the set of all languages is an automaton. We will continue by discussing the function ℓ_S that maps a state of an automaton to the language it accepts. Using the set $\mathcal{L}$, ℓ_S is essentially a function from any automaton $S = (S, o_S, t_S)$ to $\mathcal{L}$: $\ell_S : S \to \mathcal{L}$. And it is not just a function: it is a homomorphism. According to the definition of homomorphism we saw in Section 2.1.3, we need to show that for all $s \in S$ and all $a \in \Sigma$ we have that:

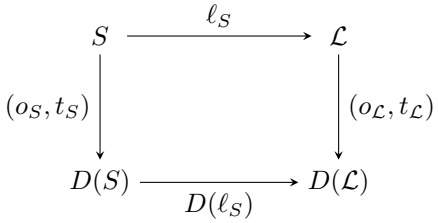
- $o_S(s) = o_{\mathcal{L}}(\ell_S(s))$
- $\ell_S(t_S(s)(a)) = t_{\mathcal{L}}(\ell_S(s))(a)$

Proof. For the first item, remember that s is an accepting state if and only if $\epsilon \in \ell_S(s)$. By definition of $o_{\mathcal{L}}$ we have that $\ell_S(s)$ is an accepting state if and only if $\epsilon \in \ell_S(s)$ as well. For the second item, take an arbitrary word $w \in \ell_S(t_S(s)(a))$. Hence, w is in the language accepted by state $t_S(s)(a)$. This means that aw is accepted by state s, and thus that $aw \in \ell_S(s)$. By definition,

$$t_{\mathcal{L}}(\ell_S(s))(a) = (\ell_S(s))_a = \{v \in \Sigma^* | av \in \ell_S(s)\}$$

As $aw \in \ell_S(s)$, we thus have that $w \in (\ell_S(s))_a$, and we have that $\ell_S(t_S(s)(a)) \subseteq t_{\mathcal{L}}(\ell_S(s))(a)$. The other direction is identical and hence we have that $\ell_S(t_S(s)(a)) = t_{\mathcal{L}}(\ell_S(s))(a)$, which completes the proof. □

In a coalgebraic representation seeing $\mathcal{L}$ as an automaton, the fact that ℓ_S is a homomorphism makes the diagram below commute:



4.6 Coinduction on $\mathcal{L}$

In this section we will present a proof relating states in $\mathcal{L}$ through bisimulation, which we will use later to prove a general coinduction principle. For the automaton $(\mathcal{L}, o_\mathcal{L}, t_\mathcal{L})$ we have that for all languages K and L and a bisimulation on $\mathcal{L}$:

$$\text{if } K \sim L \text{ then } K = L \tag{4.1}$$

Note that the converse trivially holds and that any bisimulation R on $\mathcal{L}$ has the property that if KRL then $K\downarrow \Leftrightarrow L\downarrow$ and for any $a \in \Sigma$ we have that $K_a R L_a$.

Proof. Assume that we have languages K and L such that $K \sim L$. We will first prove that $K \subseteq L$ by induction on the length of words. The base case is the empty string ϵ. Because R is a bisimulation and KRL, we have that $\epsilon \in K$ implies that $\epsilon \in L$. Now for the induction step. Consider an arbitrary word $w \in K$ of length $n+1$. Hence $w = aw'$ for some $a \in \Sigma$ and $w' \in \Sigma^*$ and w' has length n. We also have that $w' \in K_a$. As $K \sim L$ we know that for all $a \in \Sigma$, but in particular this a, we have that $K_a R L_a$. From our induction hypothesis we get $w' \in L_a$, which by definition gives us that $w \in L$. The proof of the converse $L \subseteq K$ is similar. Hence we get that $K = L$. □

Result 4.1 does not only hold for $\mathcal{L}$. In fact, it can be generalised to all final coalgebras: if two states in a final coalgebra are bisimilar, they are the same. We will show in the next section that $\mathcal{L}$ is indeed a final coalgebra. This result implies that to prove that two states s and s' in a final coalgebra are equal, it is enough to show there exists a bisimulation R on the final coalgebra such that $(s, s') \in R$. [24]

Example 6. To conclude this section, let us look at two (trivial) examples of how we can prove the equality of two languages using coinduction. For $K \subseteq \Sigma^*$, to show that $K + \varnothing = K$, we will prove that

$$R = \{(K + \varnothing, K) | K \in \mathcal{L}\}$$

is a bisimulation. Then, from the coinduction principle, we can conclude that $K + \varnothing = K$. First note that $(K + \varnothing)\downarrow$ if and only if $K\downarrow$, so the first condition of a bisimulation is already satisfied. Then we have that for any $a \in \Sigma$:

$$\begin{aligned}(K + \varnothing)_a &= K_a + \varnothing_a \\ &= K_a + \varnothing\end{aligned}$$

From the definition of R we know that $((K_a + \varnothing), K_a) \in R$, and thus that $((K + \varnothing)_a, K_a) \in R$. Hence we have proven that R is indeed a bisimulation. It follows that $K + \varnothing = K$.

Example 7. For $K \subseteq \Sigma^*$, to show that $K \cdot \{\epsilon\} = K$, we will prove that

$$R = \{(K \cdot \{\epsilon\}, K) | K \in \mathcal{L}\}$$

is a bisimulation. Then, from the coinduction principle, we can conclude that $K \cdot \{\epsilon\} = K$. First note that $(K \cdot \{\epsilon\})\downarrow$ if and only if $K\downarrow$ ($\{\epsilon\}\downarrow$ by definition), so the first condition of a bisimulation is already satisfied. We will only consider the situation where $K\downarrow$ as it is trivial when $K\uparrow$. Then we have that for any $a \in \Sigma$:

$$\begin{aligned}(K \cdot \{\epsilon\})_a &= K_a \cdot \{\epsilon\} + \{\epsilon\}_a \\ &= K_a \cdot \{\epsilon\}\end{aligned}$$

In the last line we use the result from Example 6 and the fact that $\{\epsilon\}_a = \varnothing$. From the definition of R we know that $((K_a \cdot \{\epsilon\}), K_a) \in R$, and thus that $((K \cdot \{\epsilon\})_a, K_a) \in R$. Hence we have proven that R is indeed a bisimulation, and thus that $K \cdot \{\epsilon\} = K$.

4.7 Coinduction

In this section, we will illustrate the principle of coinduction and we will relate it to the example of a bisimulation we saw in Section 2.1.3. In the concluding section of this chapter we will see how the coinduction principle is related to coalgebras of the deterministic automata type in general.

Theorem 2 (The Coinduction Principle)**.** Take two deterministic automata (S, o_S, t_s) and (T, o_T, t_T). Then for all $s \in S$ and $t \in T$

$$\text{if } s \sim t \text{ then } \ell_S(s) = \ell_T(t)$$

where $\ell_S(s)$ and $\ell_T(t)$ describe the languages accepted by state s and t. Note that any bisimulation R between S and T has the property that if sRt then $o_S(s) = o_T(t)$ and for any $a \in \Sigma$ $s_a R t_a$, where s_a denotes $t_S(s)(a)$. Now for the proof of the coinduction principle.

Proof. Assume that $s \sim t$. As we know that ℓ_S is a homomorphism, we know that $\ell_S(s) \sim \ell_T(t)$. As $\ell_S(s)$ and $\ell_T(t)$ are both languages, from equation 4.1 we get that $\ell_S(s) = \ell_T(t)$. □

This coinduction proof principle ensures that in order to determine whether two states s and t of two deterministic automata recognise the same language, it is sufficient to construct a bisimulation containing the pair (s, t). For example, the bisimulation example in Section 2.1.3, showed us that s_1 and t_1 recognise the same language, as well as s_2, s_3 and t_2.

4.8 The Set $\mathcal{L}$ is Final

From the coinduction proof principle we thus have that in order to prove the equality of languages K and L, one can show that there exists a bisimulation on $\mathcal{L}$ such that $K \sim L$ and from that conclude equality of K and L. We will use this principle to show that the automaton $\mathcal{L}$ is final among all automata, which means that for any automaton $S = (S, o_S, t_S)$ there exists a unique homomorphism from S to $\mathcal{L}$.

Proof. First of all, we know there is at least one homomorphism from S to $\mathcal{L}$: we saw in Section 4.5 that ℓ_S is a homomorphism from S to $\mathcal{L}$. Now we just need to prove that it is unique. Suppose we have two homomorphisms from S to $\mathcal{L}$: f and g. As $f(s)$ and $g(s)$ are both elements of $\mathcal{L}$, we have seen before that it suffices to prove that $f(s) \sim g(s)$. Then we can conclude that they are equal. Hence, we need to prove that

$$R = \{(f(s), g(s)) | s \in S\}$$

is a bisimulation. We thus need to prove that:

- $o_{\mathcal{L}}(f(s)) = o_{\mathcal{L}}(g(s))$
- $(t_{\mathcal{L}}(f(s))(a), t_{\mathcal{L}}(g(s))(a)) \in R$

For the first item, note that by definition of a homomorphism, we have that $o_S(s) = o_{\mathcal{L}}(f(s)) = o_{\mathcal{L}}(g(s))$, so this is immediate. For the second item, first note that again by definition of a homomorphism, we have that

$$t_{\mathcal{L}}(f(s))(a) = f(t_S(s)(a)) \text{ and } t_{\mathcal{L}}(g(s))(a) = g(t_S(s)(a))$$

As we know that

$$(f(t_S(s)(a)), g(t_S(s)(a))) \in R$$

by definition of R, we immediately get that

$$(t_{\mathcal{L}}(f(s))(a), t_{\mathcal{L}}(g(s))(a)) \in R.$$

□

Thus we have that the unique homomorphism from an automaton into the final coalgebra takes a state s to the set of strings accepted by s. Relating this to Kleene's theorem, which tells us that the language accepted by a finite

automaton is equal to a regular language, we get that the map taking a regular expression to the set of strings it represents (to its regular language) is the unique homomorphism into the final coalgebra. To elaborate on this a bit more, if we remember the structure of the regular expression automaton $\mathcal{R}$, consisting of regular expressions over a finite input alphabet, and view it as a coalgebra, the unique homomorphism $\ell_{\mathcal{R}}$ from $\mathcal{R}$ into the final coalgebra $\mathcal{L}$ maps a state, thus a regular expression, to the set of strings accepted by that state, which is indeed exactly the set of strings we get when we translate the regular expression into its belonging regular language. An exact proof of this can be found in [24], where it is proven that a word $w \in \Sigma^*$ is an element of the regular language belonging to regular expression r over Σ if and only if the state r_w is an accepting state.

Given the fact that $\mathcal{L}$ is final, we can prove that the language accepted by a state L in $\mathcal{L}$ is exactly L itself. From the finality of $\mathcal{L}$ we know that the identity function can be the only homomorphism from $\mathcal{L}$ to itself. This means that $\ell_{\mathcal{L}} : \mathcal{L} \to \mathcal{L}$ is the identity function. As this is the function that maps a state to the language it accepts, and it maps L to L, we get that the language accepted by L is L itself.

Remember the coinduction principle that we had, stating that for two deterministic automata (S, o_S, t_s) and (T, o_T, t_T) we have that for all $s \in S$ and $t \in T$

$$\text{if } s \sim t \text{ then } \ell_S(s) = \ell_T(t) \tag{4.1}$$

In fact, equation 4.1 is not just an implication, but an equivalence, meaning that the semantics induced by the map ℓ, the unique map into the final coalgebra, is equivalent with the bisimulation semantics. This is the case for many functors including the deterministic automata functor D we defined before. A general proof of this statement can be found in [21], where it is shown that for functors with certain properties two states in a coalgebra of that functor are bisimilar if and only if they are mapped to the same element in the final coalgebra.

To summarise and to relate this coalgebraic machinery to the coalgebraic decision procedure for NetKAT, we have seen how to relate automata and coalgebras, and how to use the nice properties of the deterministic automaton functor to relate bisimulations between states to the language accepted by states, through the use of the final coalgebra. We will see that for NetKAT we have the same property as before: two states are bisimilar if and only if they are mapped to the same element in the final coalgebra. We have seen how to turn the set of regular expressions into an automaton (and thus a coalgebra) such that the unique map into the final coalgebra, the automaton consisting of the set of all languages, maps a state to the language it accepts (its corresponding regular language). For NetKAT, a similar thing is done. Instead of regular expressions we will work with NetKAT expressions, and there will be a unique mapping from a NetKAT automaton to the final coalgebra, mapping a NetKAT expression e to the language represented by that expression and hence to the language accepted by the state represented by e.

Chapter 5

NetKAT

Now that all machinery we need is in place, we will delve into the syntax and semantics of NetKAT, and look at its coalgebraic theory. NetKAT was developed in [2] and [8], and all the information in the following sections comes from those two papers.

NetKAT was developed as a network programming language. Before we start with the syntactic and semantic details of NetKAT, let us look at it from an abstract perspective and discuss the semantic foundations. A network can be seen as an automaton that moves packets from node to node along the links in its topology. It is very natural to use regular expressions, the language of finite automata, to describe a network. In fact, regular expressions are a standard way to formulate the packet-processing behaviour of a network: a path through the network is encoded as a concatenation of processing steps (we will call those policies) $p \cdot q \cdot ...$, and a set of paths is encoded as a union of paths $p + q + ...$ Iterated processing is encoded using the Kleene star. A standard equational theory governing regular expressions, is, as mentioned before, a Kleene algebra.

Another important aspect of describing networks is switch-processing behaviour. The way a packet moves through a network is determined both by the network topology and switch-behaviour. Switch-processing behaviour concerns predicates to match packets (eg. check whether a packet has a certain property such as a particular destination) and consequently perform a certain action on the packet such as forwarding or transforming matching packets.

As we have seen before, Kleene algebra does not allow for reasoning about predicates, but a Boolean algebra does. A solution for combining those two is found in KAT, Kleene algebra with tests. We then have Kleene algebra for reasoning about network structure and Boolean algebra for reasoning about the predicates that define switch behaviour. In Section 3.3, the theory for KAT was developed. Kleene algebra with tests is very similar to Propositional Dynamic Logic (PDL), which is a mixture of Kleene algebra, Boolean algebra and modal logic [15], [7].

The axioms of NetKAT consist of the axioms of KAT and some additional NetKAT specific axioms. The axioms of KAT dictate the interactions between

primitive actions, predicates and other operators. We thus have a foundational structure and a consistent reasoning principle that can be used when modifying or adding primitives. In addition to the KAT axioms, we will present a finite set of equations that capture equivalences between NetKAT programs, needed for specification of and reasoning about transformations on packets. We then have an equational theory that enables reasoning about local switch-processing behaviours as well as global network structures. In [2] it is shown that this theory is sound and complete with respect to the denotational semantics. We will not repeat the soundness and completeness proofs, but the axioms will be stated and discussed below and a proof sketch will be given.

5.1 Preliminaries

Formally, a packet pk is a record with fields $(f_1, ..., f_n)$ mapping to fixed-width integers n. NetKAT assumes a finite amount of packet headers such as source address (src), destination address (dst), protocol type (typ) and two fields identifying the current location of the packet in the network: switch (sw) and port (pt). For simplicity, we assume that every packet contains the same fields. Following [2], we write $pk.f$ for the value in field f of pk and $pk[f := n]$ for the packet obtained from pk by updating field f to n.

In order to reason about the path taken by a packet through the network, NetKAT keeps track of a packet history recording the state of each packet as it travels from switch to switch. Formally, a packet history h is a non-empty sequence of packets. We write $pk :: \langle\rangle$ to denote a history with one element, $pk :: h$ to denote the history constructed by prepending pk on to h and $\langle pk_1, ..., pk_n \rangle$ for the history with elements pk_1 to pk_n. We denote the current packet as the first element of a history, other elements denote previously processed packets. Let H be the set of all histories.

Intuitively, NetKAT policies can be seen as programs. For instance, a policy regarding forwarding behaviour or a policy for access control. Atomic NetKAT policies filter and modify packets. A filter $(f = n)$ takes any input packet pk and outputs the singleton set $\{pk\}$ if field f of pk equals n, and $\{\}$ otherwise. A modification $(f \leftarrow n)$ takes any input packet pk and yields the singleton set $\{pk'\}$, where pk' is the packet obtained from pk by setting f to n. To build up more complicated policies, NetKAT has policy combinators union and sequential composition. The union combinator $(p + q)$ generates the union of the sets produced by applying both p and q to the input packet, while the sequential compositions combinator $(p \cdot q)$ first applies p to the input packet and then applies q to each packet in the resulting set and finally takes the union of all of the resulting sets. Note that NetKAT treats $+$ as conjunctive instead of nondeterministic choice in the usual Kleene algebra interpretation: after policy $p + q$ the packets resulting from applying policy p to the input packet will exist in the network as well as the packets resulting from applying policy q to the input packet. Policy iteration (p^*) denotes performing policy p finitely many times (possibly zero) on input packet pk and subsequently resulting packets. A

$$
\begin{array}{rll}
\text{Fields } f & ::= f_1 | ... | f_k & \\
\text{Packets } pk & ::= \{f_1 = v_1, ..., f_k = v_k\} & \\
\text{Histories } h & ::= pk :: \langle\rangle | pk :: h & \\
\text{Predicates } a, b & ::= 1 & \textit{Identity} \\
& \mid 0 & \textit{Drop} \\
& \mid f = n & \textit{Test} \\
& \mid a + b & \textit{Disjunction} \\
& \mid a \cdot b & \textit{Conjunction} \\
& \mid \neg a & \textit{Negation} \\
\text{Policies } p, q & ::= a & \textit{Filter} \\
& \mid f \leftarrow n & \textit{Modification} \\
& \mid p + q & \textit{Union} \\
& \mid p \cdot q & \textit{Sequential composition} \\
& \mid p^* & \textit{Kleene star} \\
& \mid dup & \textit{Duplication}
\end{array}
$$

Figure 5.1: Syntax of NetKAT

more formal definition is given in the following sections.

5.2 Syntax

Syntactically, NetKAT expressions can be predicates (a, b) and policies (p, q). Predicates include the constants true and false (1 and 0 respectively), tests $(f = n)$ and negation $(\neg a)$, disjunction $(a + b)$ and conjunction $(a \cdot b)$ operators. Policies include predicates, modifications $(f \leftarrow n)$, union $(p + q)$ and sequential composition $(p \cdot q)$, iteration (p^*), and a special policy that records the current packet in the history (dup). The complete syntax is presented in Figure 5.1.

5.3 Semantics

Semantically, every NetKAT predicate and policy is a function that takes a history h and produces a (possibly empty) set of histories $\{h_1, ...h_n\}$. Producing the empty set models dropping the packet, and producing a set with multiple entries models modifying the packet in several ways or sending it to multiple locations. When one wishes to implement this, one does not need to take histories into account as only the first packet in the history (the current packet) is modified by a policy. Histories are meant solely to facilitate reasoning about the code.

$$
\begin{aligned}
[\![p]\!] &\in H \to \mathcal{P}(H) \\
[\![1]\!]\ h &= \{h\} \\
[\![0]\!]\ h &= \{\} \\
[\![f = n]\!]\ (pk :: h) &= \begin{cases} \{pk :: h\}, & \text{if } pk.f = n \\ \{\}, & \text{otherwise} \end{cases} \\
[\![\neg a]\!]\ h &= \{h\} \setminus ([\![a]\!]\ h) \\
[\![f \leftarrow n]\!]\ (pk :: h) &= \{pk[f := n] :: h\} \\
[\![p + q]\!]\ h &= [\![p]\!]\ h \cup [\![q]\!]\ h \\
[\![p \cdot q]\!]\ h &= ([\![p]\!] \bullet [\![q]\!])\ h \\
[\![p^*]\!]\ h &= \bigcup_{i \in \mathbb{N}} F^i h \\
\text{where } F^0\ h &= \{h\} \text{ and } F^{i+1}\ h = ([\![p]\!] \bullet F^i)\ h \\
[\![dup]\!]\ (pk :: h) &= \{pk :: (pk :: h)\}
\end{aligned}
$$

Figure 5.2: Semantics of NetKAT

The denotational semantics of NetKAT are given in Figure 5.2. Note that there is no separate definition for predicates. This is because every predicate is a policy, and the reason there is a syntactic distinction between the two is to make sure that only predicates can be negated. Formally, a predicate behaves like a filter.

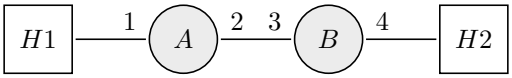
In light of the representation with histories, a filter $(f = n)$ returns either the empty set or $\{h\}$ when applied to a single history h, depending on whether field f of the first packet in the history h had value n or not. A modification $(f \leftarrow n)$ becomes a function that returns a singleton history in which the field f of the current packet has been updated to n. The union operator $(p + q)$ still denotes a function that returns the union of the sets generated by p and q, and sequential composition $(p \cdot q)$ is formally defined as Kleisli composition of the functions p and q. For functions of type $H \to \mathcal{P}(H)$, the Kleisli composition $(\bullet)$ of f and g is defined as

$$(f \bullet g)x = \bigcup \{gy \mid y \in fx\}$$

Policy iteration (p^*) formally denotes the union of semantic functions F_i of h, where each F_i is the Kleisli composition of functions denoted by p i times.

Example 8. Let us look at a simple example network and possible policies to familiarise ourselves with the NetKAT terminology. The following example can be found in [2]. Consider the network shown below. It consists of hosts 1 and 2

(squares) and switches A and B (circles) with ports $1, 2, 3$ and 4. In this case, the switches and hosts are connected in series.



Suppose that we want to implement two policies: one for transferring packets between the hosts and one to block SSH packets. For the forwarding policy, we want the policy to forward packets from host 1 to host 2 and vice versa. Formally, if a packet has destination H_1 it should travel to H_2 and the other way around. Putting this into NetKAT syntax, the forwarding policy looks as follows:

$$\begin{aligned} p_f := (\text{dst} = H_1 \cdot \text{sw} = A \cdot \text{pt} \leftarrow 1) + (\text{dst} = H_1 \cdot \text{sw} = B \cdot \text{pt} \leftarrow 3) + \\ (\text{dst} = H_2 \cdot \text{sw} = A \cdot \text{pt} \leftarrow 2) + (\text{dst} = H_2 \cdot \text{sw} = B \cdot \text{pt} \leftarrow 4) \end{aligned}$$

If we would label the ports for both switches the same, the policy could be made more concise, but throughout this thesis we have chosen to give each port a unique label. At the top level, this policy is the union of four sub-policies. For example, the first one updates the *pt* field of all packets destined for H_1 while at switch A to 1. The policy p_f now forwards packets between host 1 and host 2 via switches A and B.

Now we wish to extend the policy p_f to block SSH packets between the hosts. There are multiple ways to implement this, but we will simply create a filter and put that in sequence with our forwarding policy:

$$p_a := \neg(\text{typ} = \text{SSH}) \cdot p_f$$

This policy drops all input packets of type SSH, and otherwise forwards the packet according to policy p_f.

It would be more clever to not check all packets at all locations in the network to block certain types of packets, but to only filter at one of the switches, as packets between the hosts will cross both switches in any case. Hence, it would be sufficient to filter only at switch A or B:

$$p_{as} := (\text{sw} = A \cdot \neg(\text{typ} = \text{SSH}) \cdot p_f) + (\text{sw} = B \cdot p_f)$$

The policy p_{as} filters out packets of type SSH at switch A.

One can imagine that we would like to answer questions about this network such as "Can host 1 reach host 2?" or "Are p_{as} and p_a indeed equivalent?". By simply inspecting the policies we cannot answer these questions. We need to consider the network topology itself along with the specific switch policies in order to say anything interesting about the network. To incorporate network topology information into a NetKAT program, network topologies are modelled as the union of smaller policies that encode the behaviour of each link.

To model an internal link, a link between two switches, we use the sequential composition of a filter that matches packets located at one end of the link and a modification that updates the *sw* and *pt* fields to the location at the other end of

the link. Bi-directional links are encoded as the union of two uni-directional links along the same edge. To model a link on the outside of the network, between a host and a switch, just use a filter that keeps packets located at the ingress port. For our example network, the topology t is modelled as:

$$\begin{aligned} t := (\mathrm{sw} = A \cdot \mathrm{pt} = 2 \cdot \mathrm{sw} \leftarrow B \cdot \mathrm{pt} \leftarrow 3) + \\ (\mathrm{sw} = B \cdot \mathrm{pt} = 3 \cdot \mathrm{sw} \leftarrow A \cdot \mathrm{pt} \leftarrow 2) + \\ (\mathrm{sw} = A \cdot \mathrm{pt} = 1) + \\ (\mathrm{sw} = B \cdot \mathrm{pt} = 4) \end{aligned}$$

To model a packet moving through the network, it will encounter both switch policies and policies representing links in the network. As a packet might require an arbitrary number of steps (a topology could have a cycle), we model the overall behaviour of the network using the Kleene star operator:

$$(p_a \cdot t)^*$$

A little bit of thought on this expression will show that this policy processes packets that enter and exit the network at arbitrary locations. One might want to restrict this policy by only looking at packets entering and exiting the network at specified locations. For instance, we can have

$$p_{in,out} := (\mathrm{sw} = A \cdot \mathrm{pt} = 1) + (\mathrm{sw} = B \cdot \mathrm{pt} = 4)$$

We now encode the overall behaviour of the network as

$$p_{in,out} \cdot (p_a \cdot t)^* \cdot p_{in,out}$$

Using the equational theory below we could now answer the questions posed before such as "Can host 1 reach host 2?" and "Are p_{as} and p_a indeed equivalent?". As we will see later, the equational theory of NetKAT is complete with respect to its denotational semantics, so it can indeed answer all questions posed about equivalences of NetKAT expressions.

5.4 Equational Theory

Previously, we have seen the axioms of Kleene algebra and of Kleene algebra with tests. The axioms for NetKAT include all those axioms and some additional ones. The additional axioms are needed to make NetKAT complete for the underlying NetKAT packet model. Below are the additional NetKAT axioms listed. The other axioms can be found in Section 3.1 and 3.3.

$$(f \leftarrow n \cdot f' \leftarrow n') \equiv (f' \leftarrow n' \cdot f \leftarrow n), \text{ if } f \neq f' \qquad \text{MOD-MOD-COMM}$$

$$(f \leftarrow n \cdot f' = n') \equiv (f' = n' \cdot f \leftarrow n), \text{ if } f \neq f' \qquad \text{MOD-FILTER-COMM}$$

$$(\text{dup} \cdot f = n) \equiv (f = n \cdot \text{dup}) \qquad \text{DUP-FILTER-COMM}$$

$$(f \leftarrow n \cdot f = n) \equiv (f \leftarrow n) \qquad \text{MOD-FILTER}$$

$$(f = n \cdot f \leftarrow n) \equiv (f = n) \qquad \text{FILTER-MOD}$$

$$(f \leftarrow n \cdot f \leftarrow n') \equiv (f \leftarrow n') \qquad \text{MOD-MOD}$$

$$(f = n \cdot f = n') \equiv 0, \text{ if } n \neq n' \qquad \text{CONTRA}$$

$$\sum_i (f = i) \equiv 1 \qquad \text{MATCH-ALL}$$

where $\sum$ denotes taking the sum of all $f = i$ terms using the $+$ operator from Kleene algebra.

The packet axioms are very intuitive. The first three axioms ensure commutativity conditions. Updating two different fields will yield the same outcome regardless of the order, and performing a test and then an update on a different field will also be the same if one switches the order. The next three axioms specify modifications. They specify that performing a modification changing a field to n and then filtering packets where that field has value n is the same as performing just the modification, and performing a test to check if a field has value n and then modifying that value to n is the same as simply the filter. The PA-MOD-MOD axiom specifies that performing consecutive modifications on the same field is the same as performing simply the last modification. The last two axioms characterise filters. The first one specifies that performing two consecutive filters on the same field with different values will always result in dropping the packet as a field cannot be equal to two different values at the same time. And the last axiom states that if one does a union of filters with all values for a field, one will always keep the packet as one of them will come out as true. This axiom implies that packet values are taken from a finite domain, such as fixed-width integers.

5.5 Main results regarding NetKAT

In this section we will list the most important theoretical results for NetKAT. They are all taken from [2] and [8] where the full proofs of the theorems stated here can also be found.

Important results are the soundness and completeness of NetKAT axioms with respect to the denotational semantics defined before. Intuitively, this means that every equivalence provable from the NetKAT axioms also holds in the denotational model (soundness) and that every equivalence that holds in the denotational model is provable from the NetKAT axioms (completeness). We will not provide the full proofs, but a short proof sketch will be given for both theorems.

Theorem 3 (Soundness)**.** The NetKAT axioms are sound with respect to the semantics defined before. That is, if $\vdash p \equiv q$, then $[\![p]\!] = [\![q]\!]$.

Proof Sketch: One first proves that the packet-history model used in the denotational semantics is isomorphic to a binary relations model, as is done in [1]. Instead of modelling the policies and predicates as functions $[\![p]\!] \in H \to \mathcal{P}(H)$, policies and predicates are interpreted as a binary relation $[p] \subseteq H \times H$:

$$(h_1, h_2) \in [p] \leftrightarrow h_2 \in [\![p]\!](h_1)$$

You can see $[p]$ as the set of input-output pairs of the policy p. In the relational model, product is interpreted as relational composition and the remaining KAT operations translate under the isomorphism to the usual KAT operations on binary relations. As mentioned before, it is shown in for instance [18] that relational models with these operations satisfy the KAT axioms. We then know that NetKAT models with the packet-history semantics also satisfy the KAT axioms. It remains to show that the additional NetKAT packet axioms given in Section 5.4 are also satisfied. As shown in [1] this is done by arguments in relational algebra, found in for instance [23], and some axioms are special instances of equations (6)-(11) found in [3].

Theorem 4 (Completeness)**.** Every semantically equivalent pair of NetKAT expressions is provably equivalent using the NetKAT axioms. That is, if $[\![p]\!] = [\![q]\!]$, then $\vdash p = q$.

Proof Sketch: The proof of completeness is more complicated than for soundness, but we will not go into details. First, a language model is developed that plays the same role as regular sets of strings and guarded strings do for KA and KAT respectively. Then it is shown that packet-history and language model are isomorphic by proving that two NetKAT expressions have the same semantics (send the same packets to the same locations) if and only if the two expressions are mapped to the same set of strings in the language model. It is also proven that these sets of strings in the language model are in fact regular sets, meaning we can use the completeness of KA to prove the completeness of the NetKAT axioms.

Another important result is that deciding equivalences of NetKAT has the same complexity as deciding KAT or KA.

Theorem 5. The equational theory of NetKAT is PSPACE-complete.

The proof of Theorem 5 is given in [2]. In [8] a more sophisticated algorithm for deciding the equational theory of NetKAT is given, which is still exponential in the worst case, but a lot more efficient in a lot of practical examples. We will discuss this algorithm in a bit more detail in the next section.

5.6 NetKAT Coalgebra

As mentioned before, a coalgebraic decision procedure for NetKAT was developed in [8]. Central in this procedure is the use of bisimulation to characterise

equivalence. In order to apply these coalgebraic techniques to NetKAT, [8] first develops the coalgebraic theory of NetKAT. We will not give the full coalgebraic machinery for NetKAT, as that is not the focus of this thesis. What we will do instead, is give an overview of the coalgebraic theory for NetKAT and try to explain what it does and why it works, without going into too many details.

As we have seen before, we define coalgebras in terms of a set of states together with a function giving a direct observation of the current state and information about next states reachable from the current state. A NetKAT coalgebra consists of set of states S along with continuation and observation maps

$$\delta_{\alpha\beta} : S \to S \text{ and } \varepsilon_{\alpha\beta} : S \to 2$$

where α and β are atoms. We denote the set of atoms with At. Atoms are complete tests, meaning that they are of the form

$$f_1 = n_1 ... f_k = n_k$$

where $f_1, ..., f_k$ is the list of all fields of a packet. Besides complete tests, we also have complete assignments:

$$f_1 \leftarrow n_1 ... f_k \leftarrow n_k$$

The complete tests are often called atoms because they are the minimal nonzero elements of the Boolean algebra generated by the tests. Note that complete tests and complete assignments are in one-to-one correspondence according to the values of $n_1, ..., n_k$.

To show that every NetKAT expression can be written as a reduced string (a string where every test and assignment is complete) we need a simplified version of the NetKAT axioms. For reduced strings the NetKAT axioms, the ones specific for packet-processing, can be presented in a simpler form. Let α_p be the complete test corresponding to the complete assignment p and let p_β be the complete assignment corresponding to the complete test β. The NetKAT axioms for reduced strings are:

$$\alpha \text{ dup } = \text{ dup } \alpha \tag{5.1}$$

$$p\alpha_p = p \tag{5.2}$$

$$\alpha p_\alpha = \alpha \tag{5.3}$$

$$\alpha\alpha = \alpha \tag{5.4}$$

$$\alpha\beta = 0 \text{ if } \alpha \neq \beta \tag{5.5}$$

$$qp = p \tag{5.6}$$

$$\sum_{a \in At} \alpha = 1 \tag{5.7}$$

In [2] it is shown that every NetKAT expression can be rewritten to an equivalent reduced expression in which every test is a complete test and every assignment

is a complete assignment. To translate a NetKAT expression into a reduced expression, we only need to replace all atomic assignments $f \leftarrow n$ and tests $f = n$ with complete ones. This can be done as follows. Note that $\sum_{\alpha \in At}(\alpha \cdot \pi_\alpha) = 1$, using axioms 5.3 and 5.7.

$$\begin{aligned}(f \leftarrow n) &= 1 \cdot f \leftarrow n \\ &= (\sum_{\alpha \in At} \alpha \cdot p_\alpha) \cdot (f \leftarrow n) \\ &= \sum_{\alpha \in At} \alpha \cdot p'_\alpha\end{aligned}$$

where p' is the complete assignment resulting from p with the assignment to f replaced by $f \leftarrow n$. For tests we have

$$b = \sum_{\alpha \leq b} \alpha$$

The input to our NetKAT coalgebras (automata) will be reduced strings belonging to the set $U = At \cdot P \cdot (\text{dup} \cdot P)^*$, where P is the set of complete assignments and At the set of complete tests. U is a subset of all the reduced NetKAT strings. In [2] a mapping G is defined, mapping a reduced NetKAT expression e to a regular subset of U. This mapping can be seen as the mapping ℓ we had in the chapter on coalgebras. It is the mapping going from a regular expression to corresponding regular subsets. The continuation and observation maps for a NetKAT coalgebra can then also be given by a variant of the Brzozowski derivative for NetKAT. As we do not go into these details we do not need the exact definition, but it can be found in [8]. Intuitively, the Brzozowski derivative tells us how to move from one state in the NetKAT automaton to the next and whether a state is an accepting state or not.

To relate NetKAT expressions to NetKAT automata, a generalisation of Kleene's theorem is proven in [8]. This theorem states that any subset of U is equal to $G(e)$ for some NetKAT expression e if and only if it is the set of strings accepted by some finite NetKAT automaton.

In rough lines, what the coalgebraic decision procedure does is the following. It takes two NetKAT expressions e_1 and e_2 and translates those into reduced NetKAT expressions. Then the procedure normalises them (shown in [2]) such that when the mapping G is performed on the expressions the outcomes will be subsets of U. From now on e_1 and e_2 represent normalised reduced NetKAT expressions. This means that $G(e_1)$ and $G(e_2)$ are both subsets of U. Using Kleene's theorem for NetKAT, we then know for sure that there will be two automata corresponding to e_1 and e_2 such that these automata accept $G(e_1)$ and $G(e_2)$. The algorithm will construct these automata according to the Brzozowski derivative for NetKAT and will then check if they are bisimilar. In case they are bisimilar we know that $e_1 \equiv e_2$ and in case they are not bisimilar we have that $e_1 \not\equiv e_2$. To keep the complexity low of the part where it is checked whether the two automata are bisimilar the coalgebraic perspective on these automata is exploited as can be seen in [8] but is not discussed here.

Let us elaborate on why checking for bisimulation will answer the question whether $e_1 \equiv e_2$. The mapping G is the unique homomorphism from the NetKAT coalgebra (automaton) into the final coalgebra. Using the coinduction principle, we know that two states in two NetKAT automata are bisimilar if and only if they are mapped to the same set in the final coalgebra. In other words, two states s and t are bisimilar if and only if they accept the same language ($G(s) = G(t)$). This means that we can actually conclude that checking whether the two automata are bisimilar will give us the answer to the question if they accept the same language. In the case of the example where we had expressions e_1 and e_2, that would mean that we get an answer to the question whether $G(e_1) = G(e_2)$. As is proven in [2], we know that $[\![e_1]\!] = [\![e_2]\!]$ if and only if $G(e_1) = G(e_2)$ and by completeness of NetKAT we thus get an answer to the question whether $e_1 \equiv e_2$ as desired.

Chapter 6

Gossip through a State Vector Perspective on NetKAT

In this chapter we will explore dynamic gossip in NetKAT by adding a state vector perspective to NetKAT. As we have seen so far, NetKAT describes a single packet traveling through a network. One can for instance ask questions concerning the route taken by the packet or the topology of the network. Now we will extend the semantics in such a way that it allows for some form of indirect communication between switches via a state vector.

6.1 Indirect Switch Interaction

Ideally we wish to describe multiple packets in a network. This requires a sense of concurrency: which packet was where first? Did packet 1 pass before or after packet 2? A sense of concurrency would be useful for modelling dynamic gossip, but we can also think of consequences this can have such as that packet 1 can only pass if packet 2 has already passed at this switch.

In our case we are focused on modelling dynamic gossip. As we do not have the means to describe multiple packets, we instead follow [19] in adding a global 'state' to the network. In that paper a state extension of NetKAT is proposed. However, the formalisation given in that paper is based on extracting the static (standard) NetKAT version from NetKAT with state, whereas our formalisation will directly integrate NetKAT with state into standard NetKAT. Therefore, one can say that the extension proposed here is inspired by [19], but is not the same. A global 'state' in a network is interpreted to be comprised of states of the individual switches in the network.

Adding a state allows us to model indirect communication between switches via changing switch configurations. Intuitively, this means that each switch can

change configuration based on its state. For instance, a packet arriving at a switch makes the switch change configuration based on the values of certain fields on the packet. Information exchange could also go the other way around: a switch alters certain fields of an arriving packet based on which configuration the switch is in. We will give a more precise explanation below in combination with an explanation of how to model gossip.

One can think of various applications of NetKAT with state, but in this thesis we will focus solely on using it to model dynamic gossip protocols. Another application could be to use NetKAT with state in the study of how infections spread in a network while incorporating factors such as immunity of agents, as is done in for instance [11].

To state that NetKAT does not describe multiple packets is not entirely true. Even though NetKAT describes how a single packet travels through a network, it can implicitly describe the semantics of a set of packets. This happens when a policy results in multiple packets through the + operator and another policy is applied to this set sequentially. To formalise this explicitly, one can see the semantics as:

$$[\![p]\!]\ H = \bigcup_{h' \in H} [\![p]\!]\ h' \tag{6.1}$$

where p is an arbitrary policy, H a set of histories and $[\![p]\!]h'$ the usual definition of the NetKAT semantics stated in Figure 5.2. Intuitively, in Equation 6.1 one can see a policy on multiple packets as a policy on individual packets in parallel networks. However, as NetKAT interprets the + operator as a conjunctive, it is interpreted as that all those packets exist in the same network, but no interaction between them is possible.

6.2 NetKAT with state

Before turning to gossip protocols, we will describe NetKAT with state and its semantics. A network configuration is denoted with a vector $\vec{k}$, such that the entries denote the configuration of the different switches. A configuration is denoted with an integer and the integers encode NetKAT policies, thereby directly showing the forwarding behaviour of that switch. For simplicity we assume that the entries of $\vec{k}$ are labeled with switches in alphabetical order (entry 1 denotes the configuration of switch A). We call this vector a state vector. The configurations in the state vector are from a finite domain such as fixed-width integers. Therefore, the configurations an entry in the state vector can take are finite. The encoding between policies and integers is specified externally. In the implementation, we will handle the working with states a bit differently, but details on this will be given later.

6.2.1 Semantics

We first change the interpretation of the + operator to non-deterministic choice, as is usual with Kleene algebras. This change of interpretation does not change anything formally about standard NetKAT, it only changes the interpretation of a set of packets that comes out of a policy. Instead of interpreting a set of output packets as that all these packets exist in the network, they now all resemble possible outcomes that could have happened, but in each outcome only the outcome packet exists in the network. As we will see, changing the interpretation of the + operator to non-deterministic choice is more natural when modelling dynamic gossip.

Another change we make to the original NetKAT semantics is that the semantics will now describe how a packet and state vector and their history transform into a set of packets/state vectors and histories (or the empty set). Instead of the usual semantics of NetKAT that describe how a packet and its history transform into a set of packets and histories (or the empty set):

$$[\![p]\!] \ : H \to \mathcal{P}(H)$$

where H describes the set of all histories, we change it into:

$$[\![p]\!] \ : HV \to \mathcal{P}(HV)$$

where HV describes the set of all histories of all packets/state vector pairs we can have. In other words, HV is the set of all histories of the set $(PK \times \vec{K})$, where PK is the set of all packets and $\vec{K}$ the set of all state vectors.

Intuitively, when a packet/state vector pair is processed by a policy, the output is the set of packets/state vectors that can result from this policy. The state vectors denote the state of the network as a result of the route taken by the corresponding packet. For instance, if a packet is sent to two different locations A and B with the + operator, this will result in two different state vectors each reflecting either the move of the packet to A or the move to B. Hence, the outcome will be a set with two elements: a packet/state vector pair at A and a packet/state vector pair at B, where the state vector reflects the move to A or B respectively. Here it is demonstrated that it is more natural to interpret the + operator as non-deterministic choice: we cannot have a network in two states, so only one of the situations will have occurred.

This change of packet histories to histories of packet/state vector pairs has not changed anything essential in the semantics. Therefore, the definitions stay almost the same as in Figure 5.2, and we have just added a definition for a test and assignment on the state vector to obtain the definition in Figure 6.1. The state assignment operator can be understood similarly to the normal assignment operator: it assigns a certain value to a certain row in the vector thereby changing the configuration of that switch. The same goes for the state test operator: it tests whether a certain row in the state vector has a certain value, so whether a certain switch has that configuration. We denote a history of a packet/state vector pair with hv. The negation of the state test operator

$$
\begin{aligned}
[\![1]\!]\ (hv) &= \{hv\} \\
[\![0]\!]\ (hv) &= \{\} \\
[\![f = n]\!]\ ((pk, \vec{k}) :: hv) &= \begin{cases} \{(pk, \vec{k}) :: hv\}, & \text{if } pk.f = n \\ \{\}, & \text{otherwise} \end{cases} \\
[\![\neg a]\!]\ (hv) &= \{hv\} \setminus ([\![a]\!]\ (hv)) \\
[\![f \leftarrow n]\!]\ ((pk, \vec{k}) :: hv) &= \{(pk[f := n], \vec{k}) :: hv\} \\
[\![p + q]\!]\ (hv) &= [\![p]\!]\ (hv) \cup [\![q]\!]\ (hv) \\
[\![p \cdot q]\!]\ (hv) &= ([\![p]\!] \bullet [\![q]\!])\ (hv) \\
[\![p^*]\!]\ (hv) &= \bigcup_{i \in \mathbb{N}} F^i(hv) \\
\text{where } F^0\ (hv) &= \{hv\} \text{ and } F^{i+1}\ (hv) = ([\![p]\!] \bullet F^i)\ (hv) \\
[\![dup]\!]\ ((pk, \vec{k}) :: hv) &= \{(pk, \vec{k}) :: (pk, \vec{k}) :: hv\} \\
[\![state(x) = n]\!]\ ((pk, \vec{k}) :: hv) &= \begin{cases} \{(pk, \vec{k}) :: hv\}, & \text{if } \vec{k}(x) = n \\ \{\}, & \text{otherwise} \end{cases} \\
[\![state(x) \leftarrow n]\!]\ ((pk, \vec{k}) :: hv) &= \{(pk, \vec{k}(x) := n) :: hv\}
\end{aligned}
$$

Figure 6.1: Semantics for NetKAT with state

is the same as the negation of the normal test operator and is not mentioned separately in the figure.

6.2.2 Equational Theory, Soundness and Completeness

In this section we present a different view on NetKAT with state in order to demonstrate that NetKAT with state is not an extension of standard NetKAT but is merely a change of perspective. This will allow us to use the soundness and completeness of standard NetKAT to argue that NetKAT with state is sound and complete.

We could see the rows in the state vector as fields of the corresponding packet with which the state vector is paired. So the packet would have extra fields with the configuration of switch A, switch B etc. The state modifications and state tests on row S in the state vector would then simply be normal packet modifications and tests on the field corresponding to row S. For example, if we have the input $((pk, \vec{k}) :: hv)$ where pk is a packet and $\vec{k}$ is the state vector

$$\vec{k} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

with the configuration of switch A on row 1 and the configuration of switch B on row 2 encoded by the integers 1 and 2, we could also see it as that the packet pk gets extra fields C_A and C_B that get the values 1 and 2 respectively. Changing the configurations into extra fields of a packet can always be done as the packet and state vector are always paired together. Note that the values of the configuration still come from a finite set of possibilities, so we do not make the MATCH-ALL axiom invalid.

To clarify the distinction between packets and state vectors we have introduced a special operator for tests and assignments on the state vector. Formally speaking, these operators behave just like operators on packets and hence obey the same axioms. They are introduced simply to provide notational clarity and keep packets separate from configurations.

The equational theory of NetKAT given in Section 5.4 then stays in place for NetKAT with state and does not need to be altered. From this we can easily conclude that the proof system of NetKAT with state is still sound and complete with respect to the packet-processing model, based on the soundness and completeness of standard NetKAT.

The reason we choose not to represent gossip in the way of extra fields within a packet but with a state vector, is because using a separate state vector allows for a more elegant representation of the protocol. The separation between the packet and configuration seems very natural. In that way the packet is simply a communication facilitator whereas the configuration belongs to the switches as it represents the state of the network. One can also imagine that once multiple packets with NetKAT can be described, having the packet separate from the network configuration is desired. Note that this might require a slightly different formalisation, as the state of the network (the configurations of the switches) is then not so closely tied to a packet, but that is for future research.

6.3 Dynamic Gossip

Dynamic gossip protocols were developed and researched for the first time in [25]. A more in depth study can be found in [6]. In this thesis we do not wish to get into details about dynamic gossip, but we will give the information we need to understand one specific protocol. We will first demonstrate with an example and then later show more formally that NetKAT can simulate this specific protocol. We will also translate various results from dynamic gossip into NetKAT. So far no sound and complete axiomatisation has been found to describe dynamic gossip. In this thesis we claim that NetKAT can offer such a framework.

A gossip protocol is a procedure for spreading secrets among a group of agents, using a connection graph. It is assumed that everyone has a unique secret and each agent starts out with a secret only known to that agent. Calls between

agents involve only two agents. When a call takes place, all secrets known to the agents involved in the call are exchanged. Dynamic gossip protocols have the added feature that when a call takes place not only secrets are exchanged but also phone numbers, i.e. links in the gossip graph. Thereby each call in the gossip graph influences the distribution of secrets and the links in the graph. We assume that each agent starts out then with knowing only their own secret and an amount of phone numbers always including at least their own phone number.

The gossip procedure we study is not regulated by an outside authority. Instead, we use a distributed protocol from [25] called Learn New Secrets (LNS). The LNS protocol works as follows:

> While not every agent knows all secrets, let an agent x randomly select an agent y such that x knows y's phone number but does not know y's secret, and let x call y.

In essence such a protocol allows us to investigate how information spreads through a network.

We will not get into too many details about gossip graphs, but we need some terminology on how they are defined formally. A gossip graph G is represented as a triple (A, N, S), where A is the finite set of agents and N and S are binary relations on A. Nxy expresses that x has a link to y, or x knows y's phone number and Sxy expresses that x knows y's secret. We represent a call from x to y as a tuple xy. We denote G^{xy} to be the gossip graph after the call xy has taken place in G.

Definition 5 (*Gossip Graph*)**.** A *gossip graph* is a triple $G = (A, N, S)$, where A is the finite set of vertices or agents and $N \subseteq A^2$ and $S \subseteq A^2$ are relations on A. We denote G^{xy} to be the gossip graph after the call xy has taken place in G.

Definition 6 (*Call Sequence*)**.** A *call sequence* σ is a finite list of calls. $\sigma;\tau$ denotes the call sequence σ concatenated with the call sequence τ. ϵ denotes the empty call sequence. The empty call sequence performed on a graph G is G^ϵ and $G^\epsilon = G$. The graph G^σ is the result of call sequence σ on graph G.

This terminology was needed to show the following result: for a gossip graph where each agent in the beginning knows only their own secret and at least their own phone number, an agent will only know another agent's secret if he also knows his phone number. Formally, if we have a gossip graph $G = (A, N, S)$ with a calling sequence σ for G such that $S = \mathrm{id}$, we have that $S^\sigma xy$ only if $N^\sigma xy$. Hence:

Lemma 1. For any gossip graph $G = (A, N, S)$ with $S = \mathrm{id}$ and a calling sequence σ for G we have that $S^\sigma \subseteq N^\sigma$.

Proof. We will prove this by induction on the call sequence σ. The base case follows instantly as we always start with every agent knowing only their own secret and at least their own phone number. Hence we have that $S^\epsilon \subseteq N^\epsilon$.

For the induction step take a possible calling sequence σ for G. We assume that $S^\sigma \subseteq N^\sigma$ as our induction hypothesis. Let xy be a possible call in G^σ and let $(a, b) \in S^{\sigma;xy}$. We have two possibilities. Either $(a, b) \in S^\sigma$ or $(a, b) \in S^{\sigma;xy}$ but $(a, b) \notin S^\sigma$. In the first case, by the induction hypothesis we instantly know that $(a, b) \in N^\sigma$. And as knowledge does not get lost in these protocols (you never lose secrets or phone numbers but only gain new ones), we can conclude that $(a, b) \in N^{\sigma;xy}$. In the second case, we can assume without loss of generality that $a = x$. This means that $(y, b) \in S^\sigma$. From our induction hypothesis we get that $(y, b) \in N^\sigma$ and hence we have that $(x, b) = (a, b) \in N^{\sigma;xy}$ as all secrets and phone numbers that x and y know are exchanged in their call. □

6.4 Gossip in NetKAT with state

To express dynamic gossip in NetKAT, we need the following definitions. Let the switches in a NetKAT network describe the agents, and let the configuration of a switch denote the current knowledge of the agent concerning what phone numbers and what secrets he knows. We only consider networks where each switch is connected to each other switch to ensure that each agent could in theory communicate with each other agent. The initial links in the gossip graph are incorporated into the NetKAT model via the initial packet/state vector that enters the network. The packet plays the role of carrier pigeon facilitating the communication between agents and the state vector gets an initial configuration corresponding to the initial knowledge distribution in the gossip graph. The possession of phone numbers is incorporated locally by checking what configuration a switch is in. To simplify, we assume that only one call at a time takes place. That means even if we have multiple players, we cannot have player 1 and 2 calling and player 3 and 4 calling at the same time. This assumption is also common in existing work on dynamic gossip [6].

We wish to model the Learn New Secrets protocol. This means that we must allow for calls between agents only if certain conditions are satisfied, i.e. an agent does not know the secret of the agent he is going to call and he knows the phone number of the agent he will call. Roughly, our NetKAT policy describing this protocol will look as follows. First the input packet will be distributed to every agent in the network. Then, using the Kleene star, each iteration will represent one call that is made. The policy will include checking that the conditions that allow for a call are satisfied, and will update the configurations accordingly after a call has taken place. Using the nice property of the + operator that it behaves as non-deterministic choice, we will model all possible calls in formally separate versions of the network, so we do not need to use randomness.

The packets have four fields. The first two fields of a packet denote its location, so we have a field for at which agent a packet is, denoted with ag, and a field for at which port the packet is, denoted with pt. The other two fields are used to transfer knowledge of secrets and phone numbers. We will describe this as a packet knowing things, but in fact they are just transferring knowledge from one agent to another agent. There is a field S denoting the secrets a packet

knows and a field N denoting the phone numbers. As we are not interested in the actual value of the secrets and phone numbers but purely in whose secrets and phone numbers a packet possesses, we will denote for instance that a packet only knows agent A's secret as $S = \{a\}$.

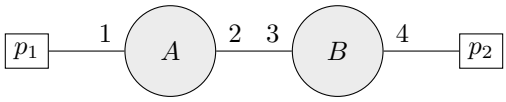
As one can see, we have extended the values a field can take from only integers to also sets of, in this case, agents. The justification for this is as follows. The values a field can take need to come from a finite set of possibilities. This is still the case, as the S and N fields draw their values from the power set of the set of agents, and we always work with a finite amount of agents. Moreover, when one is working with for instance three agents, instead of writing $S = \{a\}$ to represent that a packet only knows the secret of agent A, one can visualise this as giving the packet three fields S_a, S_b, S_c and then simply giving S_a the value 1 and S_b and S_c the value 0. As the set-theoretic notation is much more elegant, we will use that below.

The set-theoretic notation gives us an additional feature that we will make use of. It allows us to introduce a test of the form $e \in f$, which checks if e is an element of the value of the f field. We call this test the inclusion test and it will shorten notation of our gossip policy. Conceptually, $a \in S$ could be seen as the normal test $S_a = 1$. Hence, we know that NetKAT can express such an inclusion test, thereby justifying to use the more elegant notation. The negation of $e \in f$ will be denoted as $e \notin f$ as opposed to classical NetKAT negation of tests which would look like $\neg(e \in f)$.

The policy simulating the LNS protocol will consist out of three parts, that we will discuss in the next section via an example.

6.4.1 A worked out Example

Example 9. Consider two agents, A and B, and each agent has two ports: one port is their private 'home' port used to initialise the packet and a port connected to the other agent's port. The network then looks as follows:



The switches are denoted with circles and are called agents from now on and the packets that will be initialised in the network are denoted with p_1 and p_2, and are already visible in the network. Strictly speaking we start with one input packet, but as that packet is distributed to every home port in the network immediately, we chose to show the initial packets after this distribution. Note that this network could be used for any gossip graph with two agents, as we now have not specified yet what phone numbers the agents know initially (the links in the gossip graph). The distribution of phone numbers will be done via the configurations, and will show up in the initial state vector belonging to the input packet.

Configurations

However, before we can give the policy describing LNS for two agents, we need to specify which configurations we have. A configuration is a state an agent can be in, and thus represents that agent's knowledge of secrets and phone numbers. In essence a configuration is a policy belonging to an agent. We will encode the configurations with integers. Some integers give configurations for agent A and some for agent B (not all configurations can belong to each agent). The encoding is as follows:

$$1 : S = \{a\} \cdot N = \{a\}$$
$$2 : S = \{b\} \cdot N = \{b\}$$
$$3 : S = \{a\} \cdot N = \{a, b\}$$
$$4 : S = \{b\} \cdot N = \{a, b\}$$
$$5 : S = \{a, b\} \cdot N = \{a, b\}$$

A state vector in this case is a vector with two rows: the top row denotes the configuration of agent A and the bottom row the configuration of agent B. Note that some configurations do not occur in the encoding. This is because of Lemma 1, so that configurations where an agent knows the secret but not the phone number of an agent are not possible.

The Policy for modelling Dynamic Gossip

Before giving the full policy for making phone calls we will discuss how NetKAT can describe updating fields as opposed to overwriting them. The need for updating occurs when we wish to facilitate information exchange between a packet and a state vector: when a packet arrives at a switch, depending on the state of that switch, the secret and phone number fields of the packet will be updated with the knowledge of the switch and the configuration of the switch will be updated according to the information on the packet. This means that we wish to add new knowledge to the current value of a field in a packet. Note that we are always only adding knowledge and there will never be values disappearing from the S and N fields, as the knowledge in dynamic gossip protocols is monotone and does not get lost.

As NetKAT has no means of describing that a certain value will be added to the current value of a field but can only overwrite values, we are forced to test which value a field has now, and act according to that. Hence, we have to take the sum of policies each testing for a value of S, N and the state and then have the assignment behave accordingly by overwriting the fields such that a new value is added. Let us look at a tiny example to clarify this. For simplicity, let us assume we wish to just add a value to the secret field. We know that the secret field can only take a finite amount of values, and here we assume the secret field can take values that are all subsets of $\{a, b, c\}$. Then, the policy for

adding value $\{c\}$ to that field, would look like this:

$$\begin{aligned} pol_{add} := \; & S = \{a\} \cdot S \leftarrow \{a, c\} \\ & +S = \{b\} \cdot S \leftarrow \{b, c\} \\ & +S = \{c\} \cdot S \leftarrow \{c\} \\ & +S = \{a, b\} \cdot S \leftarrow \{a, b, c\} \\ & +S = \{a, c\} \cdot S \leftarrow \{a, c\} \\ & +S = \{b, c\} \cdot S \leftarrow \{b, c\} \\ & +S = \{a, b, c\} \cdot S \leftarrow \{a, b, c\} \\ & +S = \{\} \cdot S \leftarrow \{c\} \end{aligned}$$

We will use this principle below to describe the policy for dynamic gossip. When it comes to the implementation, we will add values to fields directly, as NetKAT does not lack expressiveness but just lacks the tools to express it neatly and it will save a lot of space in the implementation.

Staying close to the original use of NetKAT, the policy we will describe consists of three parts. As mentioned before, the packets are merely the communication facilitators. Therefore, each agent starts with a packet on their home port with empty S and N fields. This is their carrier pigeon. We do not start with only one agent having a pigeon-packet, as each agent in the protocol is allowed to make the first call. To elaborate on this, if agent x does not start with a pigeon packet, there will be no output that is a consequence of a call sequence starting with agent x making a call. This is undesirable, because we want to study every possible call sequence. Therefore, every agent starts with a packet. To ensure each agent starts with a packet, the original input packet is first distributed to every home port in the network. This is the first part of the policy describing the LNS protocol and is denoted with pol_{dstr}.

The second part of the LNS policy gives the forwarding behaviour of the agents. Thus, depending on what phone numbers and secrets an agent knows, packets are forwarded to the ports connected to the agents an agent will call. As the packet started empty but now needs to transfer information, its S and N fields will be updated according to the state (and thus the knowledge) of the agent.

The third part concerns making phone calls. Here a packet will move from an agent, for instance agent A, to another agent, let us call him agent B, have information exchange with that agent, and then move back to agent A and change the state of that agent too. Now we have seen a call between agent A and B and the state vector has been updated accordingly. Subsequently, any other two agents could have a call. To incorporate this, after the calling happened, the S and N fields of the packet will be emptied, hence making it a clean carrier pigeon again, and a copy of this packet with accompanying state vector will be distributed to every home port in the network. In this way the updated state vector will be used in any subsequent calls.

To elaborate on the update of the state vector, note that as long as we do not have an operator for concurrency, we cannot describe how to actually handle

multiple packets that might all influence the state vector differently. For that, we came up with the solution to just distribute an empty packet with corresponding state vector to all home ports in the network at the end of each round to ensure that the new state vector is used in the next round of calls.

Let us denote the second part with $pol_{forward}$ and the third part with pol_{call}. The full protocol will then be described by:

$$pol_{gossip} := pol_{dstr} \cdot (pol_{forward} \cdot pol_{call})^*$$

This statement is further discussed after the example. It might seem that an agent is making multiple calls at the same time if he has the phone numbers of more than one agent, but by the way the + operator is designed in NetKAT calls are not made at the same time, but will both be the only call made in that round in a separate version of the network. For instance, if agent A will make a call to agent B and C because he has both their phone numbers, there will be an outcome (a packet/state vector pair) where the call to B has taken place and an outcome where the call to C has taken place. In the next round, so a next iteration of the Kleene star, we might have an outcome representing agent A first having called B and then now calling C and agent A first having called C and then now calling B. But each outcome represents its own unique sequence of calls where no two calls are made at the same time (per iteration of the Kleene star only one call takes place per outcome).

The first part is given by:

$$pol_{dstr} := \ ag \leftarrow A \cdot pt \leftarrow 1 + ag \leftarrow B \cdot pt \leftarrow 4$$

As this policy contains no filters, every packet/state vector pair will be distributed to the home port of agent A and agent B as a consequence of this policy. Hence, starting with any input packet/state vector, the result of this policy is that this packet is distributed to the home ports in the network and every agent can thus make the first call. Note that, as we interpret the + operator as non-deterministic choice, formally the input packet/state vector is thus distributed to only one agent's home port in every sequence of calls that we will model. However, as we model every sequence of calls, this does not make a difference.

The second part is $pol_{forward}$ given below, and for convenience, the integer encoding of the states is repeated below the policy.

$$\begin{aligned} pol_{forward} := \ & ag = A \cdot pt = 1 \cdot State(A) = 3 \cdot S \leftarrow \{a\} \cdot N \leftarrow \{a,b\} \cdot (b \notin S) \cdot (b \in N) \cdot pt \leftarrow 2 \\ & + ag = B \cdot pt = 4 \cdot State(B) = 4 \cdot S \leftarrow \{b\} \cdot N \leftarrow \{a,b\} \cdot (a \notin S) \cdot (a \in N) \cdot pt \leftarrow 3 \end{aligned}$$

$$\begin{aligned} &1 : S = \{a\} \cdot N = \{a\} \\ &2 : S = \{b\} \cdot N = \{b\} \\ &3 : S = \{a\} \cdot N = \{a,b\} \\ &4 : S = \{b\} \cdot N = \{a,b\} \\ &5 : S = \{a,b\} \cdot N = \{a,b\} \end{aligned}$$

Intuitively, this policy says the following. We have 2 sub-policies combined via the + operator. The first sub-policy states that if a packet is at agent A's home port and agent A knows B's phone number but not his secret, the packet is updated according to agent A's knowledge and is then moved to port 2, which is the port linked to agent B. We have a similar sub-policy describing what a packet at switch B will do. The extra tests for whether the agent knows the number of the callee and not the secret of the callee are strictly speaking superfluous: this is tested all at once by checking the state of an agent. However, when generalising the policy to n agents instead of two, it is easiest to use the test for which state an agent is in only to make sure the S and N fields of the packet are updated according to the agent's knowledge, and specifying if the call takes place by using the inclusion tests. Hence, $(b \notin S) \cdot (b \in N)$ and $(a \notin S) \cdot (a \in N)$ are included at the end of $pol_{forward}$. It also makes it explicit that the call only takes place if these two conditions are satisfied.

Note that the situation where an agent has no phone number other than their own will now drop out (that trace will result in the empty set). This is desired, as in such a situation that agent will definitely not make a call in that round and hence does not need a packet in this round (he might gain one in the next round and if he has then learned a phone number by an agent calling him, he can also make a call himself).

Also note that in the case the agent already knows everything, which is state 5, this packet/state vector pair drops out. An agent who knows all secrets is no longer interested in making calls to other agents and therefore does not need his carrier pigeon anymore. With the packet dropping out we also lose the corresponding state vector, but as we are working with the Kleene star we can still see that an agent knew all secrets in the previous round. Also, in possible subsequent rounds if the agent gets another packet, the corresponding state vector will denote he knows all secrets, as the state vector gets passed and updated every round and knowledge does not get lost.

Before we give pol_{call}, let us discuss it in a bit more detail. We will present two identical versions of pol_{call}. The first one stays very close to NetKAT and demonstrates conceptually how NetKAT could express the desired policy. This might be only interesting for the reader familiar with NetKAT. The second one is a lot more readable but makes use of syntactic sugar to abbreviate standard NetKAT.

Here we will discuss the first version of pol_{call}, which can be skipped if one only wishes to see the readable version. In pol_{call} we need to describe all possible calls that can happen between A and B. This means we have a set of policies describing calls from A to B (the first half) and a set describing calls from B to A (the second half). These are identical but opposite. Let us consider the policies describing calls from A to B. The policy of A calling B describes all possible combinations we can have of the state of B and the values of the S and N fields and the updates of those fields and the configuration of A and B in response to the call. Note that the testing what the state of agent B is corresponds to testing what his knowledge is and testing the S and N fields of the packets correspond to testing what the knowledge of agent A is as the packet

comes from A and is carrying that agent's information.

In the situation of two agents, the update of the state vector and packet fields in response to the call is the same in all situations we can have before the call takes place. In other words, in the case of two players, regardless of the state of agent A and B and what knowledge they have, after one call everyone knows all secrets and phone numbers. Therefore, using distributivity, we can shorten the policy a bit, as can be seen on the next page.

The policy of A calling B consists of three parts. The first part describes the packet moving from A to B followed by all possible combinations we can have of the state of B and the values of the S and N fields. The second part describes the updates of the S and N fields and of the configuration of A and of B in accordance with their earlier values and how the packet moves back from B to A. The third part describes how after that the S and N fields will be emptied and a copy of the packet is distributed to A's home port (note that the packet has already moved back to A and does not need to change agent in that case) and to B's home port. Further explanation is given after the policy.

The third part of pol_{gossip} is then given by:

$$
\begin{aligned}
pol_{call} := & \Big(\sum_{i=1}^{5} (ag = A \cdot pt = 2 \cdot ag \leftarrow B \cdot pt \leftarrow 3 \cdot state(B) = i \cdot S = \{a\} \cdot N = \{a\}) \\
& + \sum_{i=1}^{5} (ag = A \cdot pt = 2 \cdot ag \leftarrow B \cdot pt \leftarrow 3 \cdot state(B) = i \cdot S = \{a\} \cdot N = \{a, b\}) \Big) \\
& \cdot \big(S \leftarrow \{a, b\} \cdot N \leftarrow \{a, b\} \cdot state(B) \leftarrow 5 \cdot ag \leftarrow A \cdot pt \leftarrow 2 \cdot state(A) \leftarrow 5 \big) \\
& \cdot \big((S \leftarrow \varnothing \cdot N \leftarrow \varnothing) \cdot (pt \leftarrow 1 + ag \leftarrow b \cdot pt \leftarrow 4) \big) \\
& + \Big(\sum_{i=1}^{5} (ag = B \cdot pt = 3 \cdot ag \leftarrow A \cdot pt \leftarrow 2 \cdot state(A) = i \cdot S = \{b\} \cdot N = \{b\}) \\
& + \sum_{i=1}^{5} (ag = B \cdot pt = 3 \cdot ag \leftarrow A \cdot pt \leftarrow 2 \cdot state(A) = i \cdot S = \{b\} \cdot N = \{a, b\}) \Big) \\
& \cdot \big(S \leftarrow \{a, b\} \cdot N \leftarrow \{a, b\} \cdot state(A) \leftarrow 5 \cdot ag \leftarrow B \cdot pt \leftarrow 3 \cdot state(B) \leftarrow 5 \big) \\
& \cdot \big((S \leftarrow \varnothing \cdot N \leftarrow \varnothing) \cdot (pt \leftarrow 4 + ag \leftarrow a \cdot pt \leftarrow 1) \big)
\end{aligned}
$$

Note that we test for values of the states of A and B that they will never have, but that is not a problem as in that case the packet/state vector pair will just drop out for that sequence as desired.

Intuitively, this scary expression states the following. The calls from A to B and from B to A can all be seen to consist out of three parts. The first state in

the sum that B will actually have (so where the trace will not end in the empty set) is state 2. Then the first sub-policy (note that we take the entire sub-policy, so

$$ag = A \cdot pt = 2 \cdot ag \leftarrow B \cdot pt \leftarrow 3 \cdot state(B) = 2 \cdot S = \{a\} \cdot N = \{a\}$$
$$\cdot S \leftarrow \{a, b\} \cdot N \leftarrow \{a, b\} \cdot state(B) \leftarrow 5 \cdot ag \leftarrow A \cdot pt \leftarrow 2 \cdot state(A) \leftarrow 5$$
$$\cdot S \leftarrow \varnothing \cdot N \leftarrow \varnothing) \cdot pt \leftarrow 1 + ag \leftarrow b \cdot pt \leftarrow 4$$

found using distributivity) tells us that if a packet is at port 2 of agent A, it will move to port 3 of agent B. If the state of B and the values of the S and N fields are all as specified in the sub-policy, the S and N field will both be updated to $\{a, b\}$. Then the state of B will be updated and the packet will move back to port 2 of agent A. The state of agent A is then also updated. Next the packet will be distributed to both A and B's home port. Here we choose to not follow the network route as the distribution to the home ports is really not part of the packet's path but merely a preparation for the next round. During the phone call we do try to follow the network's paths, as in that way the protocol resembles an actual network as much as possible, and it is convenient to have configuration changes only happen at the switch they concern if one is working with multiple packets in the future.

The rest of the sub-policies are all variations of this first one. They check for different combinations of S and N values and a state, and respond accordingly. In fact, for each state the agent at which the packet arrives can be in, we checked for two S and N values. We do not need to test all combinations, as these two combinations are the only possible ones. This is because a packet coming from agent A will always know A's phone number and secret, and an agent never knows a secret without knowing the corresponding phone number. Also, the situation where the agent knows all secrets or the secret of the agent he calls is not interesting, as the agent would not be calling in such a situation.

Closely inspecting the sub-policies one can see that some are still superfluous, as the response is the same everywhere (changing S and N both to $\{a, b\}$), but all options are included to demonstrate the principle if one would be working with more than two agents.

To make pol_{call} more understandable, we will now present pol_{call} with some syntactic sugar. However, as NetKAT cannot directly express the shortening of notation that we introduce, we have first presented the long version of pol_{call} to demonstrate the principle. We will introduce an $\mathcal{M}$ operator, which will merge the values of a specific field of the packet and of a specific switch in the state vector. In other words, instead of testing for all combinations of the state the callee can be in and the values of the S and N fields of the packet, we will directly write $\mathcal{M}\ S\ b$ to denote that the S field of the packet and of the callee, in this case agent B, should be merged and they are thus exchanging their secrets. In the implementation we will follow the same strategy to not have the complexity of our method be too big.

Formally, $\mathcal{M}$ is defined as:

$$[\![\mathcal{M}\ f\ n]\!]((pk, \vec{k}) :: hv) = \begin{cases} \{(pk[f := union], \vec{k}(n) := m) :: hv\}, & \text{if } f = S \text{ or } f = N \\ \{\}, & \text{otherwise} \end{cases}$$

where *union* represents the union of the current value of the f-field of the packet with the current value of f in entry n of the state vector and m is the state that agent n will get as a result of adding the current value of the f field of the packet to the current value of the f field of the state. Thus, the merge operator $\mathcal{M}$ can only be used for the fields S and N (secrets and phone numbers) as that is the only field that has a value in the entries of the state vector.

This syntactic sugar transforms pol_{call} into:

$$\begin{aligned} pol_{call} := {} & ag = A \cdot pt = 2 \cdot ag \leftarrow B \cdot pt \leftarrow 3 \cdot (\mathcal{M}\ S\ b) \cdot (\mathcal{M}\ N\ b) \cdot ag \leftarrow A \cdot pt \leftarrow 2 \\ & \cdot (\mathcal{M}\ S\ a) \cdot (\mathcal{M}\ N\ a) \cdot S \leftarrow \varnothing \cdot N \leftarrow \varnothing \cdot (pt \leftarrow 4 + ag \leftarrow a \cdot pt \leftarrow 1) \\ & \\ & + ag = B \cdot pt = 3 \cdot ag \leftarrow A \cdot pt \leftarrow 2 \cdot (\mathcal{M}\ S\ a) \cdot (\mathcal{M}\ N\ a) \cdot ag \leftarrow B \cdot pt \leftarrow 3 \\ & \cdot (\mathcal{M}\ S\ b) \cdot (\mathcal{M}\ N\ b) \cdot S \leftarrow \varnothing \cdot N \leftarrow \varnothing \cdot (pt \leftarrow 1 + ag \leftarrow b \cdot pt \leftarrow 4) \end{aligned}$$

The first sub-policy can be read as: if a packet is at port 2 of agent A, then move it to port 3 of agent B, have agent B exchange information with the packet (the call) and move the packet back to port 2 of agent A. There the packet will exchange information with agent A (updating the caller with the knowledge of the callee) and will be subsequently emptied and distributed to every home port in the network. The second sub-policy describes the call from B to A.

6.5 General Definition for n Agents

Example 9 demonstrates the principle of how we wish to translate the LNS protocol into NetKAT, but if we wish to make any general claims, we need a definition of pol_{gossip} for n agents.

We will start by describing pol_{dstr} for n agents. We denote the home port of agent i with $home_i$.

$$pol_{dstr} := \sum_{i=1}^{n} distribute_i$$

where

$$distribute_i = ag \leftarrow i \cdot pt \leftarrow home_i$$

To describe $pol_{forward}$ for n agents we need sub-policies that describe the forwarding of the carrier pigeon of each agent to the ports connected to the agents he will make calls to. What calls an agent will make depends on what phone numbers he knows and whose secrets he knows, which is information

encoded in the state. In other words, we need the sum of policies where each term describes the calls that will be made by one specific agent, which consists of that agents' carrier pigeon copying the knowledge of the agent and being forwarded to the right port. In case an agent can make multiple calls (in case we have more than two agents) this term will consist of a sum itself. We denote the home port of agent i with $home_i$. Depending on the network, this port will have a specific number. To keep notation short we make use of the inclusion test.

To keep it general, we have a policy pol_{copy} which copies the knowledge of the agent (updates the S and N fields according to the information in the state) and a policy pol_{port} which forwards the packet to the appropriate port based on to whom the call is taking place. By appropriate port we mean that the packet is forwarded to the port that connects to a port of the callee. We suppose we have k states (configurations) the agents can be in and to denote the set of all agents we use $\mathcal{A}$.

To generalise:

$$pol_{forward} := \sum_{i=1}^{n} forward_i$$

where

$$forward_i := \sum_{j \in \mathcal{A} \setminus \{i\}} forward_{ij}$$

where

$$forward_{ij} := \sum_{s=1}^{k} forward_{ijs}$$

where

$$forward_{ijs} := ag = i \cdot pt = home_i \cdot State(i) = s \cdot pol_{copy} \cdot (j \notin S) \cdot (j \in N) \cdot pol_{port}$$

To summarise, $forward_{ijs}$ gives the forwarding policy for agent i calling agent j when the state of agent i is s. The policy $forward_{ij}$ gives the forwarding policy for agent i calling j while taking into account every possible state agent i can have. The policy $forward_i$ gives the policy describing the distribution of the packet to all the ports in accordance with what calls agent i will make. Lastly, the policy $pol_{forward}$ gives the full forwarding policy for all n agents. The reasoning behind the general definition of pol_{gossip} is the same reasoning as used in the implementation.

Now we will generalise pol_{call} making use of the newly introduced merge operator $\mathcal{M}$ to increase readability. We will describe all the calls that can take place between every agent. This means we need a sum of policies describing for each agent what the calls that agent can make will look like. An agent can make a call from every port he has except for his home port, as every port other

than his home port is uniquely connected to another agent. Hence, the calls one specific agent can make will be described by taking the sum of calls that can happen from each port of that agent. We describe the set of ports of agent i without counting his home port as $\mathcal{C}_i$. To generalise, we also have a policy pol_{dest} that will change the agent and port of the packet according to its location and the link in the network. In other words, pol_{dest} moves the packet from the caller to the callee. We denote the callee (which is uniquely determined by from which agent/port pair the call arises) with l. Last we make use of a policy pol_{home} which is a sum of modifications forwarding the packet to every home port in the network.

$$pol_{call} := \sum_{i=1}^{n} call_i$$

where

$$call_i := \sum_{p \in \mathcal{C}_i} call_{ip}$$

where

$$\begin{aligned} call_{ip} := ag = i \cdot pt = p \cdot pol_{dest} \cdot (\mathcal{M}\ S\ \mathrm{l}) \cdot (\mathcal{M}\ N\ \mathrm{l}) \cdot ag \leftarrow i \cdot pt \leftarrow p \\ \cdot (\mathcal{M}\ S\ \mathrm{i}) \cdot (\mathcal{M}\ N\ \mathrm{i}) \cdot S \leftarrow \varnothing \cdot N \leftarrow \varnothing \cdot (pol_{home}) \end{aligned}$$

To summarise, $call_{ip}$ gives the policy of a call taking place from the agent i at port p. This call can only have one callee by how the network is constructed. The policy $call_i$ gives a sum of all the calls that can take place by one agent and the policy pol_{call} gives all calls that can take place between all agents.

We now have a gossip policy formulated for n agents:

$$pol_{gossip} := pol_{dstr} \cdot (pol_{forward} \cdot pol_{call})^*$$

From now on, if we talk about pol_{gossip}, we mean this generalised version.

6.6 Discussion

In this section we will consider what the policy pol_{gossip} actually describes and how it simulates the Learn New Secrets protocol, and what NetKAT could say about dynamic gossip. We give a proof that a theorem of dynamic gossip is also a theorem in NetKAT, thereby strengthening our claim that NetKAT can be used as a logic for dynamic gossip.

The policy

$$pol_{gossip} := pol_{dstr} \cdot (pol_{forward} \cdot pol_{call})^*$$

outputs a list of packets and state vector pairs that all represent a specific sequence of calls that have been made. In each iteration of the Kleene star, an empty packet with corresponding state vector is distributed to all home ports in the network to ensure that the new state vector is used in the next round of calls. All possible call sequences will be in the output of pol_{gossip}.

Correctness of the Gossip Policy

We claim that the policy pol_{gossip} simulates the LNS protocol from [25] perfectly. The NetKAT policy pol_{gossip} describes all sequences of calls that we can have until a fixpoint is reached (using the Kleene star) and no new packet/state vector pairs and hence new situations arise. Each iteration in the pol_{gossip} policy results in one call and similar to what we had with gossip graphs we will denote a trace of agents that called one another via the pol_{gossip} policy with a call sequence σ. The input packet/state vector is a packet with a random location (any location works), and a state vector that corresponds to the distribution of knowledge in the gossip graph. The NetKAT model $\mathcal{N}_G$ corresponding to gossip graph G is formed as follows:

Definition 7 (*NetKAT model corresponding to Gossip Graph*)**.** The NetKAT model $\mathcal{N}_G$ corresponding to gossip graph $G = (A, N, S)$ is a totally connected NetKAT model with a number of switches equal to the number of agents in A, where each switch also has a private port not connected to any other agent. The input state vector is such that it resembles the S and N relations of the gossip graph: if Nxy then entry x of the state vector has y in its phone number field. Same for Sxy.

We will prove by induction on the call sequence that the distribution of secrets and phone numbers is the same in the gossip graph and the corresponding NetKAT model plus packet/state vector if we apply the same call sequence. This result will be used to argue that indeed the same calls can and will take place in both models as a result of the LNS protocol and the pol_{gossip} policy. By distribution of secrets and phone numbers we refer to the knowledge of each agent. In case of the NetKAT model the distribution of knowledge after call sequence σ is visible in the state vector corresponding to that trace. Applying a call sequence to a NetKAT model with initial packet/state vector should be understood as pol_{call}: when a call between agent x and y takes place they exchange secrets and phone numbers. If we denote the ports connecting agent x and y with i and j, the policy applied to perform a call is:

$$\begin{aligned} call_{xy} := {} & ag = x \cdot pt = i \cdot ag \leftarrow y \cdot pt \leftarrow j \cdot \mathcal{M}\ S\ y \\ & \cdot \mathcal{M}\ N\ y \cdot ag \leftarrow x \cdot pt \leftarrow j \cdot \mathcal{M}\ S\ x \cdot \mathcal{M}\ N\ x \end{aligned} \tag{6.1}$$

This is exactly as the part of pol_{call} where the actual call takes place.

Lemma 2. Call sequence σ applied to gossip graph $G = (A, N, S)$ results in the same distribution of secrets and phone numbers as when σ is applied to the corresponding NetKAT model with initial packet/state vector, $\mathcal{N}_G$.

Proof. Let us assume that σ only has calls between agents that exist in G. In case there are calls between agents not in G, they will also not exist in $\mathcal{N}_G$ and therefore will have no effect on either of the two models.

The base case is where σ is the empty call sequence. By construction the distribution of knowledge is the same.

Now assume as an induction hypothesis that after call sequence σ the distribution of knowledge is the same. Another call takes place: xy. We will show that $(a, b) \in S^{\sigma;xy}$ if and only if agent a will know b's secret in $\mathcal{N}_G$ after $\sigma; xy$, and $(a, b) \in N^{\sigma;xy}$ if and only if agent a will know b's phone number in $\mathcal{N}_G$ after $\sigma; xy$.

We will start by proving that if $(a, b) \in S^{\sigma;xy}$ then agent a will know b's secret in $\mathcal{N}_G$ after $\sigma; xy$. We have two situations to distinguish: either $(a, b) \in S^{\sigma}$ or $(a, b) \notin S^{\sigma}$. In the former case the result we need follows instantly by the induction hypothesis. In the latter case we can assume without loss of generality that $x = a$. Hence, we get that $(y, b) \in S^{\sigma}$. By our induction hypothesis we know that in the NetKAT state vector corresponding to the trace represented by σ agent y knew b's secret after σ as well. Then, as all secrets are exchanged in a call from x to y (can be seen in the policy $call_{xy}$ from the occurrences of the merge operator $\mathcal{M}$), we get that agent x will know b's secret after $\sigma; xy$. Thus we have that agent a will know b's secret in $\mathcal{N}_G$ after $\sigma; xy$. The converse of the statement is proven similarly.

That the phone numbers will be distributed in the same way can be proven in a similar manner and is omitted. □

To argue that the same calls will and can take place in a gossip graph through the LNS protocol and in the corresponding NetKAT model plus initial packet/state vector through pol_{gossip} we will use Lemma 2. When applying the LNS protocol to a gossip graph, all call sequences that arise are subject to two conditions: a call is only made if the caller does not know the secret of the callee and if the caller has the callee's phone number. This procedure continues until no new situations arise, meaning that either we have that every agent is an *expert* (knows all secrets) or no new calls can be made. The pol_{gossip} policy works in the same way: first the packet is distributed to every home port in the network to ensure that the first calls made are the same as in the gossip graph, and then in each iteration of the Kleene star one call is made and the call takes place if and only if the secret of the callee is not known to the caller and the caller knows the callee's phone number. By Lemma 2 we know that the distribution of secrets and phone numbers stays the same in the NetKAT model plus packet/state vector and in the gossip graph in each specific call sequence, meaning that the same calls can be made as in the gossip graph in the continuation of that trace. The iteration of the Kleene star stops after no new situation arises, meaning no more calls can take place, which happens after every agent has become an expert or when no agent has the phone number of an agent whose secret he does not know. An implementation of this protocol is given in Chapter 7.

Properties of Dynamic Gossip in NetKAT

Now we will discuss some properties of dynamic gossip that NetKAT can capture. We follow the terminology in [6] by distinguishing between two types of success for the LNS protocol.

Definition 8 (*Weakly Successful*)**.** The LNS protocol is *weakly successful* on a given gossip graph if there is a possible call sequence following the LNS protocol that leads to success (every agent knows all secrets).

and

Definition 9 (*Strongly Successful*)**.** The LNS protocol is *strongly successful* on a given gossip graph if every possible call sequence following the LNS protocol leads to success.

We also need the following result about NetKAT in the discussion that will follow. In [2] the following result on reachability is proven:

Lemma 3. For predicates a and b, a policy p and topology t, b is reachable from a if and only if $a \cdot dup \cdot (p \cdot t \cdot dup)^* \cdot b \not\equiv 0$

The *dup*'s are there to account for the individual hops a packet takes through the network thereby making up the packet's path. Intuitively, a is a test filtering packets such that only packets satisfying test a (starting at location a) are included and b is a test filtering packets such that only packets satisfying test b (ending at location b) are left.

To argue that NetKAT can be used as a logic for dynamic gossip we need to show that theorems from dynamic gossip are also theorems of NetKAT. One important theorem of dynamic gossip that we will discuss is the following:

Theorem 6. The LNS protocol is strongly successful on an initial gossip graph G if and only if G is a sun graph, where a sun graph is defined as a graph $G = (A, N, S)$ such that N is strongly connected on A after removing all agents that only know their own number (terminal nodes of the gossip graph). N is defined to be strongly connected on A if and only if for every $x, y \in A$ there is an N-path from x to y.

We will now show that NetKAT can express when the LNS protocol is weakly successful on a given gossip graph, when the LNS protocol is strongly successful on a given gossip graph and when the graph is a sun.

If we wish to capture with a NetKAT statement that the LNS protocol will be weakly successful or not for a specific gossip graph, we extend pol_{gossip} by sequentially adding a policy testing for the state such that the input packet/state vector represents the gossip graph, pol_{graph}, and a policy checking for success, $pol_{success}$. Checking if the input packet/state vector represents the gossip graph is done by testing whether all agents are in the appropriate state in the beginning of the policy. Checking for success would mean filtering out all situations where the agents do not know all secrets, which can be done by testing if the agents are all in the right state at the end of the policy. Note that we now not check if we have the right number of agents in the NetKAT model. This cannot be tested inside a NetKAT policy and we therefore assume this condition is always satisfied.

Testing whether the LNS protocol will be weakly successful on a specific gossip graph can be seen as a reachability question. We ask whether the input

packet/state vector, representing the gossip graph, can get to the output packet/state vector (where all agents know all secrets) via a route in the corresponding NetKAT model using pol_{gossip}. Relating this to how reachability is formulated in NetKAT, we have that the network topology is encoded in pol_{gossip}, and pol_{graph} and $pol_{success}$ represent a and b respectively. To accommodate for the necessary dup, we change pol_{gossip} to $pol_{dstr} \cdot (pol_{forward} \cdot pol_{call} \cdot dup)^*$. Note that the dup records the individual 'hops' a packet makes, and in this case it records all the states that a packet/state vector pair goes through.

Thus we can take any gossip graph, translate it into the appropriate NetKAT model with accompanying starting packets and state vectors, and run the extended pol_{gossip} policy on this to simulate the LNS protocol. Using the result from [2] on reachability, we know that the question we need to answer to know whether the LNS protocol is weakly successful on a specific gossip graph is equivalent to

$$pol_{graph} \cdot dup \cdot pol_{gossip} \cdot pol_{success} \not\equiv 0$$

Theorem 7. LNS is weakly successful on a given gossip graph G if and only if $pol_G \cdot dup \cdot pol_{gossip} \cdot pol_{success} \not\equiv 0$, where pol_G is pol_{graph} for gossip graph G.

We can now answer the question whether a successful call sequence exists. If we wish to know what the specific call sequence was, we can use the implementation. One could also make use of the histories of the packet/state vector pairs, but in order to do so the policy pol_{gossip} should be extended with dup's as is done above. If the pol_{gossip} policy is not extended with dup's, the histories of all the output packets will be empty as the history is only altered in case a dup occurs (see Figure 6.1).

Now we wish to formulate when the LNS protocol is strongly successful on a given gossip graph. Strongly successful means that every call sequence that we can have for the gossip graph will lead to the outcome where every agent knows all secrets (Definition 9). Relating this to NetKAT, we can conclude that a policy that starts by checking if the input packet/state vector represents the initial gossip graph correctly, then performs the gossip, and then checks whether the outcomes are successful must give the same results as the policy that only checks if the input packet/state vector represents the initial gossip graph and then performs the gossip, as all outcomes are supposed to pass the test of success. To translate this into a NetKAT policy, we introduce a policy pol_{fin} to represent a test checking whether the protocol has finished. The protocol has finished if no more calls can take place. The policy pol_{fin} thus comprises a number of tests to see if all agents are in a state where they can no longer make calls (they do not have the phone number of an agent whose secret they do not know). Let pol_G again represent the policy checking if the input packet/state vector resembles gossip graph G and $pol_{success}$ the policy testing whether all agents know all secrets. Note that the dup's are again included in pol_{gossip} and after the initial state to keep track of the different states a packet/state vector pair goes through, which allows us to say something about the trace created by a packet/state vector pair.

Theorem 8. LNS is strongly successful on a given gossip graph G if and only if $pol_G \cdot dup \cdot pol_{gossip} \cdot pol_{fin} \leq pol_G \cdot dup \cdot pol_{gossip} \cdot pol_{success}$.

Proof. $\Rightarrow$) If LNS is strongly successful on a given gossip graph G we know that all outcomes will lead to success. We will show that then for arbitrary input packet/state vector pair pk_1 we have:

$$[\![pol_G \cdot dup \cdot pol_{gossip} \cdot pol_{fin}]\!](pk_1) \subseteq [\![pol_G \cdot dup \cdot pol_{gossip} \cdot pol_{success}]\!](pk_1).$$

Suppose we have an element g of $[\![pol_{graph} \cdot dup \cdot pol_{gossip} \cdot pol_{fin}]\!](pk_1)$. In other words, g is an outcome packet/state vector in response to input pk_1. This means that the initial state of pk_1 passed the test of pol_G and there is a trace $\langle g, .., pk_1 \rangle$ (note that the first packet in the history is the final packet) in $[\![dup \cdot pol_{gossip}]\!]$ such that g, the final packet/state vector, passed the test that the protocol has finished. To be clear, by traces in $[\![dup \cdot pol_{gossip}]\!]$ we refer to all traces of input to output packets we can get as a result of this policy on all possible inputs.

As we had assumed LNS to be strongly successful, we know that g also satisfies $pol_{success}$. Hence we have a trace $\langle g, .., pk_1 \rangle$ in $[\![dup \cdot pol_{gossip}]\!]$ such that g passes the test $pol_{success}$ and pk_1 passes the test pol_G, leading to the conclusion that $g \in [\![pol_{graph} \cdot dup \cdot pol_{gossip} \cdot pol_{success}]\!](pk_1)$.

Now using completeness we get that $pol_{graph} \cdot pol_{gossip} \cdot pol_{fin} \leq pol_{graph} \cdot pol_{gossip} \cdot pol_{success}$.

$\Leftarrow$) This direction is similar to what we did for the right to left direction, except that we now make use of soundness. □

The next thing we wish to formulate is a NetKAT statement characterising when the given gossip graph is a sun. In a sun graph every agent must either only know their own number, or there must be a path from that agent to every other agent using the links created by the phone numbers relation (N). We can capture this in NetKAT by creating a policy that for each agent tests whether they can reach all other agents via a phone number path or only have their own number. In case this holds for each agent, they should all pass the test. We denote the state where an agent only has their own phone number with $self$. In other words:

Theorem 9. A graph $G = (A, N, S)$ is a sun graph if and only if
$pol_G \cdot \sum_{i \in A} ag = i \leq pol_G \cdot \sum_{i \in A} \Big(\prod_{j \in A \setminus \{i\}} \big(ag = i \cdot pt = home_i \cdot pol_{net} \cdot ag = j \cdot ag \leftarrow i \cdot pt \leftarrow home_i\big) + State(i) = self \Big)$.

where pol_G is the usual policy checking if the input represents the graph G correctly. This is done by testing whether each agent is in the correct state. The policy pol_{net} describes moving the packet/state vector through the network according to the phone number links. Let us assume we have n agents and k states and we denote the set of ports belonging to agent i minus his home port with $\mathcal{C}_i$.

$$pol_{net} := (pol_{forwardnet} \cdot pol_{callnet})^*$$

where

$$pol_{forwardnet} := \sum_{i=1}^{n} forward_i$$

and

$$pol_{callnet} := \sum_{i=1}^{n} call_i$$

These definitions are similar to what we had before, so we have a policy pol_{copy} copying the knowledge of the agent onto his carrier pigeon (in this case only what phone numbers an agent knows), a policy pol_{port} transferring a packet to the port connecting to a port of the agent that will be called (j) and a policy pol_{dest} moving the packet to the callee. We have pol_{home_c} to denote the policy of moving the packet to the home port of the callee.

$$\begin{aligned} forward_i &:= \sum_{j \in A \setminus \{i\}} forward_{ij} \\ forward_{ij} &:= \sum_{s=1}^{k} forward_{ijs} \\ forward_{ijs} &:= ag = i \cdot pt = home_i \cdot state(i) = s \cdot pol_{copy} \cdot j \in N \cdot pol_{port} \end{aligned}$$

$$\begin{aligned} call_i &:= \sum_{p \in \mathcal{C}_i} call_{ip} \\ call_{ip} &:= ag = i \cdot pt = p \cdot pol_{dest} \cdot N \leftarrow \varnothing \cdot pol_{home_c} \end{aligned}$$

Note that these definitions are similar to the definitions for pol_{gossip} for n agents we saw in section 6.5. The policy pol_{net} distributes packets everywhere they can go according to the initial N relation between the agents. In the $forward$-part a packet is forwarded to all ports connecting to agents whose phone number an agent has, and in the *call*-part the calls take place by moving the packet to the home port of the callee. As we wish to see if there is a path to every other agent, we do not move the packet back to the caller as we did before, but we leave the packet at the callee so that he can make the next calls and we can study the N-path. As phone numbers are not exchanged there is also no point in moving the packet back to the caller (the caller will be able to make the same calls in every round).

We will abbreviate

$$\prod_{j \in A \setminus \{i\}} \left(ag = i \cdot pt = home_i \cdot pol_{net} \cdot ag = j \cdot ag \leftarrow i \cdot pt \leftarrow home_i \right)$$

from now on with simply the word $product_i$.

Before we can give a proof of Theorem 9, we also need to prove that:

Lemma 4. For a gossip graph $G = (A, N, S)$ and input a packet/state vector pair g such that the packet starts at the home port of an agent from A, let us say agent x, with a state vector representing the distribution of knowledge of phone numbers in G, we have that $[\![product_x]\!](g) = \{g\}$ if and only if the agent x has an N-path to every other agent.

Proof. Suppose that $[\![product_x]\!](g) = \{g\}$. Note that no *dup* is used here, making the histories of the output packets empty. The assignments that take place are all only on the location fields (which agent and which port), and at the end of the policy these are restored to the original ones. From these facts we can conclude that this policy only performs tests on a packet, and its only output can be the empty set or the input itself (or subsets of the input in case the input consists of multiple packets). We get that for each other agent j in the network but x the policy $[\![product_x]\!](g)$ returns g. This can only be the case if pol_{net} has distributed g to every other agent starting from agent x, ensuring that agent x has an N-path to every other agent. The other way around follows the same reasoning. □

Now for the proof of Theorem 9:

Proof. $\Rightarrow$) Suppose that a graph $G = (A, N, S)$ is a sun. For each agent we must have that either there is an N-path to every other agent or they only know their own number. Following a similar strategy as for Theorem 8, take an element $g \in [\![pol_G \cdot \sum_{i \in A} ag = i]\!](pk_1)$. As we do not use *dup* anywhere or any assignments, we know that the history of g is empty and all other fields of the packet and state vector are the same as for the input. Hence, $pk_1 = g$. Thus g is such that it correctly captures the gossip graph (passes pol_G) and represents a packet/state vector belonging to a particular agent in the gossip graph. It then trivially passes one of the test $ag = i$ for the agent g started at. Let us denote the agent that g started at with x. As we assume that G is a sun, we know that either there is an N-path from x to every other agent or that x only knows his own number. Using Lemma 4 we can conclude that $g \in [\![pol_G \cdot \sum_{i \in A} \big(product_i + State(i) = self\big)]\!](g)$. As $pk_1 = g$ we get the desired result.

Using completeness we can conclude that

$$pol_G \cdot \sum_{i \in A} ag = i \leq pol_G \cdot \sum_{i \in A} \big(product_i + State(i) = self\big)$$

$\Leftarrow$) Similarly using soundness.

□

We now have the following important result:

Theorem 10. "LNS is strongly successful on a given gossip graph G if and only if G is a sun" can be expressed and proven as a theorem of NetKAT.

Proof. $\Rightarrow$) If LNS is strongly successful we know from Theorem 8 that $pol_{graph} \cdot pol_{gossip} \cdot pol_{fin} \leq pol_{graph} \cdot pol_{gossip} \cdot pol_{success}$. Similarly, a graph G is a sun is characterised by Theorem 9. From [6] we know that LNS is strongly successful on a given gossip graph G if and only if G is a sun. Thus we get that $pol_{graph} \cdot pol_{gossip} \cdot pol_{fin} \leq pol_{graph} \cdot pol_{gossip} \cdot pol_{success}$ if and only if $pol_G \cdot \sum_{i \in A} ag = i \leq pol_G \cdot \sum_{i \in A} \big(product_i + State(i) = self\big)$ must semantically hold. Using completeness of NetKAT we know this is a theorem of NetKAT $\square$

We leave it to future research to investigate whether the proof of Theorem 6 as given in [6] can be mirrored in NetKAT with the same steps, lemmas, etc.

We also wish to prove a theorem concerning making the same call twice. If agent A calls agent B and then calls agent B again this should be the same as calling agent B only once. To formalise this, we first need to formalise what we define as agent A calling agent B. We define a call from agent A to B as the carrier pigeon of agent A travelling to B, exchanging information and then travelling back to A and exchanging information again, denoted with pol_{ab}. To describe pol_{ab}, let agent A before the call be in state k_1 where $S = x_1$ and $N = y_1$ and let agent B in state k_2 where $S = x_2$ and $N = y_2$. Let the ports connecting agent A and B be i and j belonging to A and B respectively. Let k_3 denote the state where the knowledge regarding phone numbers and secrets of agent A and B is combined. Then we have that:

$$\begin{aligned} pol_{ab} := {} & ag = A \cdot pt = i \cdot ag \leftarrow B \cdot pt \leftarrow j \cdot S \leftarrow x_1 \cup x_2 \\ & \cdot N \leftarrow y_1 \cup y_2 \cdot state(B) \leftarrow k_3 \cdot ag \leftarrow A \cdot pt \leftarrow i \cdot state(A) \leftarrow k_3 \end{aligned}$$

We chose to define a call here without using the merge operator $\mathcal{M}$ as we did in Equation 6.1 because it is easier to prove Theorem 11 that way. Note that both policies give the same outputs.

Theorem 11. $pol_{ab} \cdot pol_{ab} \equiv pol_{ab}$

Proof. As before, suppose that agent A is in state k_1 where $S = x_1$ and $N = y_1$ and agent B is in state k_2 where $S = x_2$ and $N = y_2$. Let the ports connecting agent A and B be i and j belonging to A and B respectively. Let k_3 denote the state where the knowledge regarding phone numbers and secrets of agent A and B is combined. Performing the call between A and B once means that agent A and B combine their knowledge and will both be in state k_3. Performing the call twice after each other means combining the knowledge of agent A and B for a second time, but as none of them gained any new knowledge, this will both leave them in state k_3. Hence, after the first call between A and B we get an outcome of a packet located at agent A with S field $x_1 \cup x_2$ and N field $y_1 \cup y_2$, and a state vector where agent A and B are both in state k_3. After performing the second call, the outcome is a packet again located at agent A, and the S field will now be $((x_1 \cup x_2) \cup (x_1 \cup x_2)) = x_1 \cup x_2$ and the N field $y_1 \cup y_2$ via similar reasoning. We also had that the agents are in the same state after the first call and second call. Note that the pol_{ab} does not contain any *dup*, meaning that the history of the output packets is always empty. Hence, we get the same

outcome when the call is performed once or twice. Using completeness, we find that $pol_{ab} \cdot pol_{ab} \equiv pol_{ab}$. □

We now have only described the LNS protocol using NetKAT. Clearly, other protocols from the dynamic gossip literature can be formalised in a similar way such as the Call Me Once protocol. In that protocol a call between agents takes place if and only if the caller has the phone number of the callee and if they have not called before. This could be easily done making use of NetKAT's histories. Or, alternatively, this could be achieved by giving a packet a certain mark if a call between particular agents has taken place.

We have proven that it is a theorem of NetKAT that a gossip graph is a sun if and only if the LNS protocol is strongly successful on that graph. Another important theorem characterises when the LNS protocol is weakly successful [6]. In this section we have already described how to capture weakly successful in NetKAT, and future research could investigate how to translate that characterisation into a NetKAT theorem as well.

6.7 LNS for static gossip

So far we have been investigating whether NetKAT can provide a sound and complete logic to describe dynamic gossip protocols. Instead of looking at the LNS protocol for dynamic gossip, one can also study the LNS protocol for static gossip.

In the LNS protocol for static gossip the phone numbers do not get updated. This means that we have an initial distribution of phone numbers and this stays the same throughout the evaluation of the protocol. In other words, the links in the gossip graph are static. During a phone call between two agents they only exchange all the secrets that they know.

The LNS protocol for static gossip can be modelled in NetKAT in a way similar to what we did for the dynamic version. However, it is a bit simpler now. We no longer need to update the phone numbers. Other than that, the protocol stays the same. The policy that describes the protocol will be denoted with pol_{static}. As we had before, we have a first part and a second part:

$$pol_{static} := pol_{dstr} \cdot (pol_{forwardstatic} \cdot pol_{callstatic})^*$$

We will not repeat the explanation of this policy as it can be found in Section 6.4 and 6.5. Similarly to what we had in those sections, pol_{dstr}, $pol_{forwardstatic}$ and $pol_{callstatic}$ are defined as follows. The only difference with dynamic gossip is that the phone numbers do not get updated, which will become can be seen in the definition for $pol_{callstatic}$.

$$pol_{dstr} := \sum_{i=1}^{n} distribute_i$$

where

$$distribute_i = ag \leftarrow i \cdot pt \leftarrow home_i$$

and

$$pol_{forwardstatic} := \sum_{i=1}^{n} forward_i$$

where

$$forward_i := \sum_{j \in \mathcal{A} \setminus \{i\}} forward_{ij}$$

where

$$forward_{ij} := \sum_{s=1}^{k} forward_{ijs}$$

where

$$forward_{ijs} := ag = i \cdot pt = home_i \cdot State(i) = s \cdot pol_{copy} \cdot (j \notin S) \cdot (j \in N) \cdot pol_{port}$$

and

$$pol_{callstatic} := \sum_{i=1}^{n} call_i$$

where

$$call_i := \sum_{p \in \mathcal{C}_i} call_{ip}$$

where

$$\begin{aligned} call_{ip} := ag = i \cdot pt = p \cdot pol_{dest} \cdot (\mathcal{M}\ S\ \mathrm{l}) \cdot ag \leftarrow i \cdot pt \leftarrow p \\ \cdot (\mathcal{M}\ S\ \mathrm{i}) \cdot S \leftarrow \varnothing \cdot N \leftarrow \varnothing \cdot (pol_{home}) \end{aligned}$$

Hence, we have that pol_{static} gives a policy for the static LNS protocol for n agents. Let pol_G denote a policy testing if the input packet correctly resembled the initial gossip graph G and let $pol_{success}$ be a policy testing whether a packet/state vector is in a state of success (every agent knows all secrets). Similar to what we had before, we get that the question of whether the initial input packets/state vectors resembling the gossip graph can get to output packets/state vectors where every agent knows all secrets boils down to a reachability query in NetKAT.

We get the following theorem:

Theorem 12. The static LNS protocol is weakly successful on a given gossip graph G if and only if $pol_G \cdot dup \cdot pol_{static} \cdot pol_{success} \not\equiv 0$.

Establishing the precise complexity class of the question whether the static LNS protocol is weakly successful on a given gossip graph is an open problem [14]. The statement $pol_G \cdot dup \cdot pol_{static} \cdot pol_{success} \not\equiv 0$ is decidable using NetKAT's coalgebraic decision procedure, meaning that the question of whether the static LNS protocol is weakly successful on a given gossip graph is decidable in PSPACE. As this problem is already known to be in NP, perhaps NetKAT's coalgebraic decision procedure can be tailored to this specific question in order to achieve a nicer complexity bound. We leave this for future research.

Chapter 7

Implementation

In this chapter we give an implementation of the pol_{gossip} policy in NetKAT. As mentioned before, it is not necessary to understand the implementation to follow the story of this thesis, but it is a nice tool to see that the pol_{gossip} policy indeed gives the outcomes of the LNS protocol. Through the implementation we also present the functional specifications of the pol_{gossip} policy. This functional overview of the policy can be used to understand the pol_{gossip} policy better. The implementation might also be used as a reasoning tool for future research. The last part of the implementation gives the possible call sequences and which ones were successful and which ones failed according to the LNS protocol. An example output can be found at the end, which is probably nice to look at even if one wishes to skip the implementation. All of the code can be found in a public git repository: `https://github.com/janawagemaker/GossipKATS`.

7.1 Preliminaries

This part of the thesis is written using literate programming style [13]. This allows us to keep the code and written text in the same file and in sync with one another. The implementation of gossip in NetKAT uses the modules Data.List, Data.Maybe and Control.Monad.

```
module Netcall where
import Data.List
import Data.Maybe
import Control.Monad
```

To model gossip in NetKAT using Haskell, we need some new definitions.

```
newtype Host = Host Int deriving (Eq,Ord,Show)
newtype Port = Port Int deriving (Eq,Ord,Show)
newtype Switch = Switch Int deriving (Eq,Ord)

a,b,c :: Switch
a = Switch 0 ; b = Switch 1 ; c = Switch 2
```

We have hosts, ports and switches. To make our lives a bit simpler, we have named three switches that we will use in our examples later on: the agents a, b and c. Please remember that we let the switches denote the agents.

Another data type that we need is that of models. We have chosen to denote models as follows:

```
data NetKATM = Mo
    [Host]
    [Switch]
    [Port]
    [(Switch,[Port])]
    [((Switch,Port),(Switch,Port))]
    [(Host,Port)] deriving (Eq,Show)
```

The first three things are lists of hosts, switches and ports. The fourth entry gives all the ports belonging to a specific switch. The next one gives internal links in the network between two different switches and the last entry gives for each host to which switch/port pair they are connected. In case of gossip, hosts do not play a role. However, they are needed in the implementation because they represent the home port belonging to an agent. So the port to which a host is connected is the agent's home port.

In addition to models, we also need a few other data types. As described before, a packet is a list of fields with corresponding values. The values we need for our purposes for the fields are of the type agent, list of agents or ports. We also needed a type for state vectors, and they are denoted with a list of pairs of switches and states. The configurations are represented as a list of pairs of fields and list of switches. In the case of gossip a configuration is then a list of two pairs: one with the secret field and the list of agents whose secrets are known and the other with the phone number field and the list of agents whose phone numbers are known. In this way we can easily access the knowledge of an agent while programming. Of course this diverges from NetKAT terminology, but for the implementation we are only interested in the result, and not in how fast it could be computed using purely NetKAT. Lastly there is a type to shorten the notation for an internal link in the network.

```
type Field = String
data Value = S Switch | LS [Switch] | P Port deriving (Eq,Ord,Show)
type Packet = [(Field,Value)]
type StateVector = [(Switch, State)]
type State = [(Field, [Switch])]
type Element = Switch
type Internallinks = [((Switch,Port),(Switch,Port))]
```

7.1.1 Generating NetKAT gossip models

The models we need for gossip will be models where every agent is connected to every other agent. This is to ensure that in principle every agent could contact every other agent. Whether they have the right phone number and are actually

able to contact the other agent is represented in the state which the agent is in, as we have seen in Example 9.

To not have to hard code these networks every time we use a different number of agents, we have implemented a NetKAT gossip network generator.

```
netkatmGen :: [Host] -> [Switch] -> NetKATM
netkatmGen [] _ = error "no hosts"
netkatmGen _ [] = error "no switches"
netkatmGen h s | length h /= length s = error "wrong number"
               | otherwise = netGosGen h s where
   netGosGen x k = Mo hosts switches ports combiswitchport internallinks
        outerlinks where
            hosts           = x
            switches        = k
            ports           = makeList (portCount x k)
            combiswitchport = combineSwitchesPorts s ports
            internallinks   = internalLinks combiswitchport
            outerlinks      = combineHosts h combiswitchport
```

One can input a list of hosts and a list of switches, and the output is a connected NetKAT model. Please note that for gossip models one needs to input an equal number of hosts and switches, as the switches represent the agents and each one of them has one home port (thus one host).

The other functions that are used in generating the models are the following:

```
portCount :: [Host] -> [Switch] -> Int
portCount h s = length h * length s

makeList :: Int -> [Port]
makeList 0 = []
makeList n = Port n : makeList (n-1)

combineSwitchesPorts :: [Switch] -> [Port] -> [(Switch,[Port])]
combineSwitchesPorts [] [] = []
combineSwitchesPorts [] _  = []
combineSwitchesPorts _ []  = []
combineSwitchesPorts (w:ws) ps = loop (w:ws) (w:ws) ps where
  loop:: [Switch] -> [Switch] -> [Port] -> [(Switch,[Port])]
  loop [] _ [] = []
  loop _  _ [] = []
  loop [] _  _ = []
  loop (y:ys) xs ks = (y, port) : loop ys xs (ks\\port) where
    port = take (length xs) ks

internalLinks :: [(Switch,[Port])] -> [((Switch,Port),(Switch,Port))]
internalLinks [] = []
internalLinks (z:zs) = internalLinksPerSwitch (z:zs) ++ internalLinks (
     eliminate zs) where
  eliminate:: [(Switch,[Port])] -> [(Switch,[Port])]
  eliminate = map (\(x,_:ys) -> (x, ys))

internalLinksPerSwitch :: [(Switch,[Port])] -> Internallinks
internalLinksPerSwitch [] = []
internalLinksPerSwitch ((_,[]):_) = []
internalLinksPerSwitch [(_,_)] = []
internalLinksPerSwitch ((t, _ : ps):(w, fs):zs) = ((t,head ps), (w, head (tail
     fs))): internalLinksPerSwitch ((t,ps):zs)

combineHosts:: [Host] -> [(Switch,[Port])] -> [(Host,Port)]
combineHosts _ []  = []
combineHosts [] _  = []
combineHosts _ ((_,[]):_) = []
combineHosts (h:hs) ((_, p:_):zs) = (h, p):combineHosts hs zs
```

The function `portCount` gives the amount of ports in a network based on the lists of hosts and switches. As every agent needs to connect to every other agent plus their home port, and each of these connections use a new port, every agent needs an amount of ports equal to the number of agents. The function is a bit more general and uses the number of hosts and switches. The function `makeList` creates a list of ports. The next function is `combineSwitchesPorts` which gives the ports belonging to each switch. The next two functions produce the internal links between different agents based on what ports each agent has and the function `combineHosts` gives each agent a home port that is not used to connect to other agents.

7.2 NetKAT language in Haskell

To describe the NetKAT language in Haskell we first need data types for predicates and policies. After giving those, we will describe their semantics by implementing how Haskell should evaluate them. We have a few more possibilities for predicates and policies as normally for NetKAT. The reason for this is that they make the implementation a lot more efficient, but they do not enable any behaviour that we cannot express with the normal primitives of NetKAT with a state.

```
data Predicate = One
               | Zero
               | Test Field Value
               | TestEl Field Element
               | TestAllSecrets Field [Switch] Switch
               | Cup [Predicate]
               | Seq [Predicate]
               | Neg Predicate
               | StateTest Switch State
               deriving (Eq,Ord)

data Policy = Filter Predicate
        | Mod Field Value
        | PCup [Policy]
        | PSeq [Policy]
        | Star Policy
        | StateMod Switch State
        | Merge Field Switch
        | Call Switch Switch
    deriving (Eq, Ord)
```

The first data type describes predicates. We have a predicate for `one` and `zero`, a test on packets and a test on state vectors, and the option to union predicates, negate them and sequentially compose them. We also have a predicate for the inclusion test: `TestEl`. This predicate allows us to test whether a certain agent is an element of a certain field. We can use this to test whether an agent knows a phone number. A predicate that is only used in the implementation is `TestAllSecrets`. This predicate does not belong to the NetKAT language itself, but does not increase expressivity, and simplifies the implementation. The `TestAllSecrets` predicate allows us to test whether a certain field has a list

of agents as its corresponding value. This allows us to test whether an agent already knows all secrets, which is normally done via the state.

The next data type describes policies. A policy can be a predicate, and following NetKAT this is called `Filter`. A policy can also be a modification of either a packet or a state vector, and policies can be composed by union, sequentially or by the Kleene star. As can be seen, we have two policy types that are not part of NetKAT. The MERGE policy allows us to merge the values of a certain field in the packet and the state vector. We will demonstrate this below with an example to see how it is evaluated. Another policy is the CALL policy, enabling a call between two agents. This policy actually does nothing (as can be seen in the evaluation), but is used merely in the representation of the outcome to allow for the call sequence to be visible. More details on this below.

The reason predicates and policies are separated is because predicates can be negated and policies cannot. We also have special implementations of how to display predicates and policies, such that it resembles the looks of NetKAT as we are used to it.

7.2.1 Evaluating policies and predicates

To evaluate predicates and policies, the input is a list of pairs of packets and state vectors, and the output is of the same type. The evaluation shows what happens to the input packets in response to the concerned predicate or policy.

```
evalPred :: [(Packet, StateVector)] -> Predicate -> [(Packet, StateVector)]
evalPred []     _               = []
evalPred (x:xs) One             = x : evalPred xs One
evalPred _      Zero            = []
evalPred ((x,f):xs) (Test g w) | x ! g == w = (x,f) : evalPred xs (Test g w)
                               | otherwise = evalPred xs (Test g w)
evalPred ((x,f):xs) (TestEl g w) | w `elem` whatValue (x ! g) = (x,f) :
    evalPred xs (TestEl g w)
                                 | otherwise = evalPred xs (TestEl g w)
evalPred ((x,f):xs) (TestAllSecrets g w s) | w == (f ! s) ! g = (x,f) :
    evalPred xs (TestAllSecrets g w s)
                                           | otherwise = evalPred xs (
                                               TestAllSecrets g w s)
evalPred _  (Cup [])       = []
evalPred xs (Cup (p:ps))   = evalPred xs p ++ evalPred xs (Cup ps)
evalPred xs (Seq [])       = evalPred xs One
evalPred xs (Seq (p:ps))   = evalPred (evalPred xs p) (Seq ps)
evalPred (x:xs) (Neg predi) | null (evalPred [x] predi) = x : evalPred xs (Neg
    predi)
                            | otherwise = evalPred xs (Neg predi)
evalPred ((x,f):xs) (StateTest g w) | f ! g == w = (x,f) : evalPred xs (
    StateTest g w)
                                    | otherwise = evalPred xs (StateTest g w)

instance Show Predicate where
  show One = "one"
  show Zero = "zero"
  show (Test field value) = field ++ " = " ++ show value
  show (TestEl field element) = show element ++ "∈" ++ field
  show (TestAllSecrets field value switch)= show switch ++ "." ++ field ++ " =
      " ++ show value
  show (Cup xs) = intercalate " + " (map show xs)
  show (Seq xs) = intercalate " • " (map show xs)
  show (Neg x) = "¬" ++ " (" ++ show x ++ ")"
  show (StateTest entry value) = "State(" ++ show entry ++ ") = " ++ show value
```

```
evalPol :: [(Packet, StateVector)] -> Policy -> [(Packet, StateVector)]
evalPol [] _                = []
evalPol xs (Filter predi) = evalPred xs predi
evalPol xs (Mod g w)       = map (update g w) xs
evalPol _ (PCup [])        = []
evalPol xs (PCup (p:ps))   = evalPol xs p ++ evalPol xs (PCup ps)
evalPol xs (PSeq [])       = xs
evalPol xs (PSeq (p:ps))   = evalPol (evalPol xs p) (PSeq ps)
evalPol xs (Star pol)      = lfp extend xs where
  extend :: [(Packet, StateVector)] -> [(Packet, StateVector)]
  extend ps = nub $ ps ++ evalPol ps pol
evalPol xs (StateMod g w) = map (updateVec g w) xs
evalPol xs (Merge f s)     = map (mergeField f s) xs
evalPol xs (Call _ _)      = evalPol xs (Filter One)

instance Show Policy where
  show (Filter predi) = show predi
  show (Mod field value) = field ++ " ← " ++ show value
  show (PCup xs) = intercalate " + " (map show xs)
  show (PSeq xs) = "(" ++ intercalate " • " (map show xs) ++ ")"
  show (Star x) = show x ++ "*"
  show (StateMod entry value) = "State(" ++ show entry ++ ") ← " ++ show value
  show (Merge field switch) = "Merge (" ++ field ++ " , " ++ show switch ++ ")"
  show (Call _ _) = ""
```

For the evaluation of $f \leftarrow n$, which is denoted `Mod f n`, we need a helper function called `update`. It updates the right field of the packet. We have a similar helper function for the evaluation of $state(f) \leftarrow n$, called `updateVec`. For the evaluation of the Kleene star we make use of the least fix point to make sure it stops iterating after no new packets appear. The function searching the least fix point is called `lfp`. We also have a helper function for MERGE. Because that function is quite unreadable, what it does is demonstrated with an example below these functions. The function `mergeField` uses a function `whatValue` which unwraps a value into a list of switches.

```
update :: Field -> Value -> (Packet, StateVector) -> (Packet, StateVector)
update f v (p,s) = ([ (g, if g==f then v else w) | (g,w) <- p ], s)

updateVec :: Switch -> State -> (Packet, StateVector) -> (Packet, StateVector)
updateVec f v (p,s) = (p, [ (g, if g==f then v else w) | (g,w) <- s ])

lfp :: Eq a => (a -> a) -> a -> a
lfp f x | x == f x  = x
        | otherwise = lfp f (f x)

mergeField:: Field -> Switch -> (Packet, StateVector) -> (Packet, StateVector)
mergeField f i (p, s) = (newPacket, newSV) where
  newPacket = [ (g,  if g==f  then LS (sort $ nub $ ((s ! i) ! f) ++ whatValue
      w) else w) | (g,w) <- p ]
  newSV     = [ (sw, if sw==i then newST else pr) | (sw, pr) <- s ]
  newST = [ (g,  if g==f  then sort $ nub $ whatValue (p ! f) ++ w else w) | (g
      ,w) <- s ! i ]

whatValue :: Value -> [Switch]
whatValue (LS w)  = w
whatValue (P _)   = error "not the right type"
whatValue (S _)   = error "not the right type"
```

An example for what `mergeField` does:

```
*Netcall> mergeField "S" a ([("S", LS [b])],[(a,[("S",[c])])])
([("S",LS [b,c])],[(a,[("S",[b,c])])])
```

What happens here is that we told the S field of the packet to merge with the S field corresponding to switch A in the configuration. As a result the packet learned the secret of agent c and agent a learned the secret of agent b.

7.3 Implementing Dynamic Gossip

To implement dynamic gossip, we need the ingredients mentioned in Example 9: we need to be able to generate the first, second and third part of the policy, and then we can use the Kleene star to get the outcomes. The outcomes consist of the resulting configurations after every possible call sequence. After that we have written a function that filters these outputs and tells us which call sequences led to success (every agent knows every secret) or which call sequences failed.

What we now need is the first, second and third part of the protocol. The first part takes care of the initial distribution of the input packet. The code for this is:

```
polDistribute :: NetKATM -> Policy
polDistribute (Mo _ _ _ ((_, []):_) _ _) = error "no ports"
polDistribute (Mo _ _ _ [] _ _) = Filter Zero
polDistribute (Mo h s k ((w,p:_):zs) e f) = PCup [PSeq [Mod "ag" (S w),
                                                        Mod "pt" (P p)],
                                                  polDistribute (Mo h s k zs e f)]

polDistributeSimple:: NetKATM -> Policy
polDistributeSimple m = simplify (polDistribute m)
```

The function `polDistribute` creates a policy distributing the input to every home port in the network, and the function `polDistributeSimple` simplifies that policy by taking out redundancies. The exact code for simplifying can be found in the code appendix.

The second part tells us what calls will take place in that round. The implementation of this bit is the code that follows below. `genModel` gives the policy for what calls can take place that round based on a NetKAT model where every agent is connected to every other agent. Thus, a sub-policy is generated for every possible agent an agent could call, and based on the input packets/state vectors, these calls will take place or not: there are tests to see if the agent actually has the right phone number and does not know the secret of the agent he would call. The function `genModelSimple` simplifies that policy by taking out redundancies.

The forwarding policy, `genForwardSwitch`, is first generated per specific pair of a switch (agent) and its corresponding ports. In other words, this function outputs the policy for one specific agent concerning what calls he will be making. In the policy created one can already see the tests that will be performed in the code. These tests ensure we are following the LNS protocol. There is a

test to check if the agent does not know the secret of the agent he will call and a test if he has the phone number of the agent he will call. If these tests fail, the call does not take place. If one wishes to use the code to implement different protocols, here one could put in different conditions that need to be satisfied before a call takes place. The function `decide` that is used there gives the right modification for the port corresponding to what agent will be called. The function `genForward` then gives the calling policy for all agents.

```
genModel:: NetKATM -> Policy
genModel (Mo _ p _ d e _) = genForward p d e

genModelSimple:: NetKATM -> Policy
genModelSimple m = simplify (genModel m)

genForward:: [Switch] -> [(Switch, [Port])] -> Internallinks -> Policy
genForward _ _ [] = error "no links"
genForward _ [] _ = Filter Zero
genForward [] _ _ = error "no switches"
genForward t ((s,p):xs) z = PCup [genForwardSwitch t (s,p) z, genForward t xs z
    ]

genForwardSwitch:: [Switch] -> (Switch, [Port]) -> Internallinks -> Policy
genForwardSwitch ws (s,p) z = loop ws where
  loop [] = Filter Zero
  loop (v:vs) | v == s    = loop vs
              | otherwise = PCup [ PSeq [ Filter (Test "ag" (S s) )
                                        , Filter (Test "pt" (P  (head p)))
                                        , Merge "S" s
                                        , Merge "N" s
                                        , Filter (Neg (TestEl "S" v))
                                        , Filter (TestEl "N" v)
                                        , decide s v z ]
                                 , loop vs ]

decide:: Switch -> Switch -> Internallinks -> Policy
decide _ _ [] = Filter Zero
decide s t (((h,w),(i,o)):zs) | (h == s) && (i == t) = Mod "pt" (P w)
                              | otherwise =
                                if (i == s) && (h == t)
                                  then Mod "pt" (P o)
                                  else decide s t zs
```

The third part of the protocol concerns the actual call. When the packets have been distributed to the right ports (the ports connected to the agents they should be making calls to), the packet needs to go back and forth and updates need to be performed accordingly. At the end a copy of the packet needs to be distributed to each agent's home port. Similar to what we had before, we have a function `genCallSimple` that takes out redundancies in the policy. We first have a function `callPolPort`, generating the call policy for a specific switch/port pair. The function `intLink` gives the beginning of the calling policy testing for a packet's location and then moving it to the right agent and port (the one he is calling to). Based on a packet's location there is only one agent he could be calling, as each port is only connected to one other port. Then the function `callPolSwitch` makes the calling policy for a switch and all its ports, and lastly the function `makePolCall` creates the full calling policy.

```
genCall:: NetKATM -> Policy
genCall (Mo _ p _ d e _) =  makePolCall p d e

genCallSimple:: NetKATM -> Policy
genCallSimple m = simplify (genCall m)

makePolCall :: [Switch] -> [(Switch, [Port])] -> Internallinks -> Policy
makePolCall [] _ _ = Filter Zero
makePolCall (x:xs) e z = PCup [ callPolSwitch (x, e ! x) x z e
                              , makePolCall xs e z ]

callPolSwitch:: (Switch, [Port]) -> Switch -> Internallinks -> [(Switch, [Port
    ])] -> Policy
callPolSwitch (_, []) _ _ _     = error "where is the home port"
callPolSwitch (_, [_]) _ _ _   = Filter Zero
callPolSwitch (s, _:ps) t z e = PCup [ callPolPort (head ps) s z e
                                     , callPolSwitch (s, ps) t z e ]

callPolPort:: Port -> Switch -> Internallinks -> [(Switch, [Port])] -> Policy
callPolPort p s z = loop where
  loop:: [(Switch, [Port])] -> Policy
  loop [] = Filter Zero
  loop (v:vs) = PCup [ PSeq   [ Call s (destination s p z)
                              , intLink p s z
                              , Merge "S" (destination s p z)
                              , Merge "N" (destination s p z)
                              , Mod "ag" (S s)
                              , Mod "pt" (P p)
                              , Merge "S" s
                              , Merge "N" s
                              , Mod "S" (LS [])
                              , Mod "N" (LS [])
                              , Mod "ag" (S (fst v))
                              , Mod "pt" (P (head (snd v))) ]
                     , loop vs]

intLink:: Port -> Switch -> Internallinks -> Policy
intLink p s z | isNothing (lookup (s, p) z) = check p s z
              | otherwise = PSeq [Filter (Test "ag" (S s))
                                     , Filter (Test "pt" (P p))
                                     , Mod "ag" (S (fst (z ! (s,p))))
                                     , Mod "pt" (P (snd (z ! (s,p)))) ]

destination:: Switch -> Port -> Internallinks -> Switch
destination s p z | isNothing (lookup (s, p) z) = checksecond s p z
                  | otherwise = fst (z ! (s,p))

checksecond:: Switch -> Port -> Internallinks -> Switch
checksecond _ _ [] = error "pair not found"
checksecond s p (((f,_),(w,t)):zs) | (s,p) == (w,t) = f
                                   | otherwise = checksecond s p zs

check:: Port -> Switch -> Internallinks -> Policy
check _ _ [] = error "pair not found"
check p s (((f,v),(w,t)):zs) | (s,p) == (w,t) = PSeq [Filter (Test "ag" (S s))
                                                      , Filter (Test "pt" (P p))
                                                      , Mod "ag" (S f)
                                                      , Mod "pt" (P v) ]
                             | otherwise = check p s zs
```

Now that we have both parts of the policy we need to represent the LNS protocol in place, we can create a full calling policy using the Kleene star. To evaluate the full calling policy we run the following code:

```
callSequence :: NetKATM -> [(Packet, StateVector)] -> [(Packet, StateVector)]
callSequence mo packstate = evalPol packstate (PSeq [polDistributeSimple mo,
                                                     Star (PSeq [genModelSimple
                                                          mo, genCallSimple mo
                                                     ])])
```

Based on a NetKAT model and an initial situation of a packet (in our case that will always be a packet starting anywhere) and a state vector (each agent will always know only their own secret and at least their own phone number), we get a set of outcome packets and configurations equal to performing the NetKAT policy

$$pol_{gossip} := pol_{dstr} \cdot (pol_{forward} \cdot pol_{call})^*$$

As it is desirable to see the call sequences behind those outcomes, we created a special evaluation function for policies that writes which calls has been made.

```
evalPolString :: [(String,(Packet, StateVector))] -> Policy -> [(String,(Packet
    , StateVector))]
evalPolString [] _                          = []
evalPolString ((s,x):xs) (Filter predi) | null $ evalPred [x] predi =
    evalPolString xs (Filter predi)
                                            | otherwise = (s, head (evalPred [x]
                                                predi)) : evalPolString xs (Filter
                                                predi)
evalPolString ((s,x):xs) (Mod g w)      = (s, head (evalPol [x] (Mod g w))) :
    evalPolString xs (Mod g w)
evalPolString _ (PCup [])               = []
evalPolString xs (PCup (p:ps))          = evalPolString xs p ++ evalPolString
    xs (PCup ps)
evalPolString xs (PSeq [])              = xs
evalPolString xs (PSeq (p:ps))          = evalPolString (evalPolString xs p) (
    PSeq ps)
evalPolString xs (Star pol)             = lfp extend xs where
  extend :: [(String,(Packet, StateVector))] -> [(String,(Packet, StateVector))
      ]
  extend ps = nub $ ps ++ evalPolString ps pol
evalPolString ((s,x):xs) (StateMod g w) = (s, updateVec g w x) : evalPolString
    xs (StateMod g w)
evalPolString ((s,x):xs) (Merge f w)    = (s, mergeField f w x) : evalPolString
     xs (Merge f w)
evalPolString ((s,x):xs) (Call z w)     = (s ++ show z ++ " calls " ++ show w
    ++ ", ", x) : evalPolString xs (Call z w)
```

To then get a full output of what call sequences have taken place and what the corresponding result with respect to the distribution of knowledge is we can use:

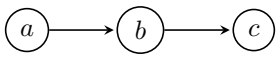
```
callSequenceString :: NetKATM -> [(String,(Packet, StateVector))] -> [(String,(
    Packet, StateVector))]
callSequenceString mo stringpackstate = evalPolString stringpackstate
                                        (PSeq [polDistributeSimple mo,
                                               Star (PSeq [genModelSimple mo,
                                                    genCallSimple mo])])
```

7.4 Success or failure of the LNS protocol

To directly see the call sequences that led to failure of the LNS protocol (not all agents know all secrets and no more calls can take place) and what call sequences led to success (all agents know all secrets), we created the following functions.

```
niceOutput:: NetKATM -> [(String,(Packet, StateVector))] -> [(String,(Packet,
    StateVector))]
niceOutput mo strpacst = filter (\z -> isUnique (fst z) (map fst $
    callSequenceString mo strpacst)) (callSequenceString mo strpacst)

isUnique:: String -> [String] -> Bool
isUnique _ []      = True
isUnique x (v:vs) | myPrefix x v = False
                  | otherwise    = isUnique x vs

myPrefix :: String -> String -> Bool
myPrefix [] [] = False
myPrefix [] _  = True
myPrefix _ []  = False
myPrefix (x:xs) (y:ys) = (x == y) && myPrefix xs ys

filteredOutput:: NetKATM -> [(String,(Packet, StateVector))] -> String
filteredOutput mo strpacst =
  "These are successful call sequences:\n  "
  ++ intercalate "\n  " successes
  ++ "\nThese are not successful call sequences:\n  "
  ++ intercalate "\n" failures where
    failures = nub $ failString mo outputs
    outputs = niceOutput mo strpacst
    successes = nub $ successString mo outputs

failString:: NetKATM -> [(String,(Packet, StateVector))] -> [String]
failString Mo{} [] = []
failString (Mo h s p z e f) (x:xs) | null $ evalPol [snd x] (successPol s s (Mo
     h s p z e f ))
                                       =  fst x : failString (Mo h s p z e f)
                                         xs
                                   | otherwise = failString (Mo h s p z e f) xs

successString:: NetKATM -> [(String,(Packet, StateVector))] -> [String]
successString Mo{} [] = []
successString (Mo h s p z e f) (x:xs) | null $ evalPol [snd x] (successPol s s
    (Mo h s p z e f ))
                                         = successString (Mo h s p z e f) xs
                                      | otherwise = fst x : successString (Mo h
                                          s p z e f) xs

successPol:: [Switch] -> [Switch] -> NetKATM -> Policy
successPol [] _ _ = Filter One
successPol (s:ws) x mo = PSeq [ Filter (TestAllSecrets "S" x s)
                              , successPol ws x mo ]

showOutput:: NetKATM -> [(String,(Packet, StateVector))] -> IO ()
showOutput mo strpacst = putStrLn $ filteredOutput mo strpacst
```

Example 10. We will now demonstrate the code with a few examples. Let us consider a gossip network with three agents, that initially starts like this:



To translate this initial situation to NetKAT, we need to start with three initial packets who all have the same initial state vector resembling the initial links between the agents in the network. To this end, there is the following code:

```
type Item = (Switch,([Switch],[Switch]))
type GossipGraph = [Item]

transfer:: GossipGraph -> (Packet, StateVector)
transfer gg = ([("ag", S a),
                ("pt", P (Port 1)),
                ("S", LS []),
                ("N", LS [])],
               statevector gg)

statevector:: GossipGraph -> StateVector
statevector [] = []
statevector ((x,(y,z)):xs) = (x, [("S", z),("N", y)])
                                      : statevector xs

gossipToNetkat:: GossipGraph -> (String, (Packet, StateVector))
gossipToNetkat gg = ("", transfer gg)
```

The function `gossipToNetkat` transfers a gossip graph to the packets and state vectors that encode it. As we wish to see the call sequences, the function also adds an empty string to each packet/state vector pair (no calls have been made).

The gossip example we have is encoded into the following gossip graph:

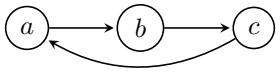
```
gossipExample:: GossipGraph
gossipExample = [ (a,([a,b],[a]))
                , (b,([b,c],[b]))
                , (c,([c],[c])) ]
```

The result of the LNS protocol on this gossip graph is then:

```
Netcall> showOutput (netkatmGen [ Host 1, Host 2 , Host 3] [a,b,c]) [(
    gossipToNetkat gossipExample)]
These are successful call sequences:
  a calls b, b calls c, a calls c,
  a calls b, a calls c, b calls c,
These are not successful call sequences:
  b calls c, a calls b,
```

This is exactly as we expected from the LNS protocol. [6]

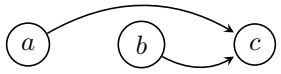
Two more examples will demonstrate a situation where every call sequence leads to success (LNS is strongly successful) and a situation where no call sequence will lead to success. The first initial gossip graph is:



The output from the implementation is then:

```
Netcall> showOutput (netkatmGen [ Host 1, Host 2 , Host 3] [a,b,c]) [(
     gossipToNetkat gossipExample2)]
These are successful call sequences:
  c calls a, b calls c, a calls b,
  c calls a, c calls b, a calls b,
  a calls b, b calls c, a calls c,
  b calls c, c calls a, b calls a,
  a calls b, a calls c, b calls c,
  a calls b, c calls a, b calls c,
  b calls c, a calls b, c calls a,
  b calls c, b calls a, c calls a,
  c calls a, a calls b, c calls b,
These are not successful call sequences:
```

The last example is the initial gossip graph:



The implementation then gives:

```
Netcall> showOutput ( netkatmGen [ Host 1, Host 2 , Host 3] [a,b,c]) [(
     gossipToNetkat gossipExample3)]
These are successful call sequences:

These are not successful call sequences:
  b calls c, a calls c,
a calls c, b calls c,
```

7.5 Code Appendix

```
(!) :: Eq a => [(a,b)] -> a -> b
(!) rel x = fromJust $ lookup x rel

instance Show Switch where
    show (Switch 0) = "a";
    show (Switch 1) = "b";
    show (Switch 2) = "c";
    show (Switch n) = "Switch " ++ show n

showStatesIntermediate :: [Switch] -> String
showStatesIntermediate z = loop 1 (states z) where
  loop:: Int -> [State] -> String
  loop _ []     = []
  loop n (x:xs) = "State " ++ show n ++ ": " ++ show x ++ "\n" ++ loop (n+1) xs

showStates :: [Switch] -> IO()
showStates z = putStrLn (showStatesIntermediate z)

import Data.GraphViz hiding (Star)
import Data.GraphViz.Types.Generalised
import Data.GraphViz.Types.Monadic

toGraph :: NetKATM -> Data.GraphViz.Types.Generalised.DotGraph String
toGraph (Mo hosts switches ports swlinks portlinks hostlinks) =
  graph' $ do
    let nodes = map show hosts ++ map show switches ++ map show ports
    mapM_ (\nid -> node nid [toLabel nid]) nodes
    mapM_ (\(sw,ps) -> mapM_ (\p -> edge (show sw) (show p) []) ps) swlinks
    mapM_ (\((_sw1,p1),(_sw2,p2)) -> edge (show p1) (show p2) []) portlinks
    mapM_ (\(h,p) -> edge (show p) (show h) []) hostlinks

writeToImage :: NetKATM -> IO ()
writeToImage x = do
  resultpath <- runGraphvizCommand Dot (toGraph x) Pdf "themodel.pdf"
  putStrLn $ "A picture of the model was saved as: " ++ resultpath

states:: [Switch] -> [State]
states [] = error "No agents"
states (x:xs) = nub (statesFin (x:xs) (x:xs)) where
  statesFin _ [] = error "No agents"
  statesFin [] _ = []
  statesFin (z:zs) y = genStates z y ++ statesFin zs y

genStates:: Switch -> [Switch] -> [State]
genStates _ [] = error "No list"
genStates x y = loop (subsetsEl x y) where
  loop = foldr (\z -> (++) (genState z y)) []

genState:: [Switch] -> [Switch] -> [State]
genState x y = loop (subsetOfSets x y) where
  loop:: [[Switch]] -> [State]
  loop = map (\z -> [("S", x), ("N", z)])

subsets:: (Eq a) => [a] -> [[a]]
subsets [] = [[]]
subsets (x:xs) = subsets xs ++ map (x:) (subsets xs)

subsetsEl:: (Eq a) => a -> [a] -> [[a]]
subsetsEl el set = filter (elem el) (subsets set)

subsetOf:: (Eq a) => [a] -> [a] -> Bool
subsetOf _ []      = False
subsetOf [] _      = True
subsetOf (x:xs)  y | x `elem` y = subsetOf xs y
                   | otherwise = False
```

```
subsetOfSets:: (Eq a) => [a] -> [a] -> [[a]]
subsetOfSets [] x = subsets x
subsetOfSets _ [] = []
subsetOfSets x y  = filter (subsetOf x) (subsets y)

-- | simplifying policies
simplify :: Policy -> Policy
simplify f = if simStep f == f then f else simplify (simStep f)

simStep :: Policy -> Policy
simStep (Filter predicate) = Filter predicate
simStep (Mod field value)  = Mod field value
simStep (PSeq [])          = Filter One
simStep (PSeq [f])         = simStep f
simStep (PSeq fs)          | Filter Zero `elem` fs = Filter Zero
                           | otherwise     = PSeq (nub $ map simStep (filter (
                              Filter One /=) fs))
simStep (PCup [])          = Filter Zero
simStep (PCup [f])         = simStep f
simStep (PCup fs)          | Filter One `elem` fs = Filter One
                           | otherwise     = PCup (nub $ map simStep (filter (
                              Filter Zero /=) fs))
simStep (Star f)           = Star (simStep f)
simStep (StateMod s p)     = StateMod s p
simStep (Merge f s)        = Merge f s
simStep (Call s w)         = Call s w

examplePacket4 :: Packet
examplePacket4 = [("ag", S a),("pt", P (Port 4)),("S" , LS []), ("N", LS [])]

exampleStateVec2 :: StateVector
exampleStateVec2 = [(a, [("S", [a]),("N", [a,b])]), (b, [("S", [b]),("N", [b])
    ])]

example7:: (String,(Packet, StateVector))
example7 = ("", example8)

example8 :: (Packet,StateVector)
example8 = (examplePacket4, exampleStateVec2)

gossipExample2:: GossipGraph
gossipExample2 = [ (a,([a,b],[a]))
                 , (b,([b,c],[b]))
                 , (c,([a,c],[c])) ]

gossipExample3:: GossipGraph
gossipExample3 = [ (a,([a,c],[a]))
                 , (b,([b,c],[b]))
                 , (c,([c],[c])) ]
```

Chapter 8

Conclusion

To conclude, we will summarise the main results of this thesis and we will discuss ideas for future work.

8.1 Summary of Results

The goal of this thesis was to explore the language of NetKAT and apply it to a new field of study: dynamic gossip. To that end, we have summarised and extended the literature in the following ways.

First of all we have presented an overview of automata theory, including regular expressions and automata and their relation (Kleene's theorem), Kleene algebras and a coalgebraic perspective on automata. Then we have combined all this knowledge to give an explanation of NetKAT and its coalgebraic decision procedure. This specific collection of information can hopefully contribute to the spreading of knowledge in the study of NetKAT.

Second, we have discussed a different perspective on NetKAT which demonstrated that NetKAT can be applied to the field of dynamic gossip. By adding a notion of state to the semantics, we got a handle on the configurations of the switches in the network, without losing the local perspective that NetKAT provides and its soundness and completeness with respect to the packet-processing model. In NetKAT with state each packet comes with a state vector representing the state of the network according to the history of that packet. Even though the idea of NetKAT with state existed already, it was never formalised in the way presented here. This thesis shows that NetKAT can be used to express information about the state of switches, and we have provided a first introduction into this concept. Besides dynamic gossip, this could perhaps be applied to other fields such as the spreading of infections while some agents are/become immune.

Lastly we showed that NetKAT can capture gossip protocols and theorems from dynamic gossip. We showed that NetKAT with state is capable of simulating the Learn New Secrets Protocol and we have translated a variety of notions of dynamic gossip into NetKAT. We have translated the notions of weakly successful

and strongly successful and have given a characterisation in NetKAT of when the gossip graph is a sun graph. We have shown that the theorem from dynamic gossip characterising when the LNS protocol is strongly successful is also a theorem of NetKAT. As dynamic gossip is a field without a sound and complete logic to describe it, applying NetKAT as a formal framework to dynamic gossip is a first exploration of an entirely new perspective that perhaps can capture dynamic gossip in a satisfactory way. We have presented an implementation, which is fun and can also be a reasoning tool for future research.

8.2 Future Work

This thesis points the way to a lot of potential future research. First of all, NetKAT is a field of study that has only recently gained attention. The applications of NetKAT are far from being exhausted. A very interesting future direction would be to describe multiple packets and their interactions in NetKAT. When describing real-life networks, it seems necessary to model multiple packets and how they would interact.

A concrete example of a network with multiple packets where one needs a handle on concurrency would be a firewall that allows packets to be sent from host 2 to host 1 only if host 1 has already sent a packet to host 2. This example actually resembles a method for externally opening ports on a firewall that happens in real life: port knocking. Firewall rules can be dynamically modified in response to specific connection attempts, thereby allowing a host to connect over previously closed ports. Port knocking is a method to prevent an attacker from getting useful results out of a port scan, because unless the attacker sends the correct sequence of connection attempts (knock sequence), the protected ports will appear closed.

Currently, the semantics of NetKAT does not allow for multiple packets and their interactions. A possibility might be to use Concurrent Kleene Algebra (CKA) [9], [10], but as the free model of CKA has not been developed yet this was not explored in this thesis. Perhaps the state perspective presented in this thesis could be of use in researching what such interactions between packets should look like.

Second, we have only created a beginning in applying NetKAT to the field of dynamic gossip. Other features of dynamic gossip that could be captured by NetKAT are different protocols and more validities that would be desired. The theorem from dynamic gossip characterising when the LNS protocol is strongly successful could be proven inside NetKAT and the theorem characterising when the LNS protocol is weakly successful could be both formulated and proven inside NetKAT. One could also go into the nature of what is exchanged by making use of NetKAT's histories. NetKAT's decision procedure could be used to answer questions like when did an agent get a secret and where did it come from.

A last interesting point would be to actually apply the coalgebraic decision procedure to decide for instance whether the LNS protocol is weakly successful on a given graph and compare the results to existing gossip tools for deciding whether

the LNS protocol is weakly successful. The coalgebraic decision procedure is exponential in the worst case as deciding NetKAT equivalences is a PSPACE-complete problem, but in most real-life scenarios it performs rather well. The coalgebraic decision procedure could also be used to verify existing results in dynamic gossip and the translations given in this thesis. To use the coalgebraic decision procedure it is needed that the policies used in gossip such as pol_{gossip} are translated into automata using the Brzozowski derivative for NetKAT and to use the existing NetKAT decision tools on gossip.

Bibliography

[1] C. J. Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. *NetKAT: Semantic Foundations for Networks*. Tech. rep. Cornell University, Computing and Information Science, Oct. 2013.

[2] C. J. Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. "NetKAT: Semantic Foundations for Networks". In: *SIGPLAN Not.* 49.1 (Jan. 2014), pp. 113–126. ISSN: 0362-1340. DOI: 10.1145/2578855.2535862. URL: http://doi.acm.org/10.1145/2578855.2535862.

[3] Allegra Angus and Dexter Kozen. *Kleene Algebra with Tests and Program Schematology*. Tech. rep. Ithaca, NY, USA, 2001.

[4] Janusz A. Brzozowski. "Derivatives of Regular Expressions". In: *J. ACM* 11.4 (Oct. 1964), pp. 481–494. ISSN: 0004-5411. DOI: 10.1145/321239.321249. URL: http://doi.acm.org/10.1145/321239.321249.

[5] J. H. Conway. *Regular Algebra and Finite Machine*. London: Chapman and Hall, 1971.

[6] Hans van Ditmarsch, Jan van Eijck, Pere Pardo, Rahim Ramezanian, and François Schwarzentruber. "Dynamic Gossip". In: *CoRR* abs/1511.00867 (2015). URL: http://arxiv.org/abs/1511.00867.

[7] Michael J. Fischer and Richard E. Ladner. "Propositional dynamic logic of regular programs". In: *Journal of Computer and System Sciences* 18.2 (1979), pp. 194–211. ISSN: 0022-0000. DOI: http://dx.doi.org/10.1016/0022-0000(79)90046-1. URL: http://www.sciencedirect.com/science/article/pii/0022000079900461.

[8] N. Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. "A Coalgebraic Decision Procedure for NetKAT". In: *SIGPLAN Not.* 50.1 (Jan. 2015), pp. 343–355. ISSN: 0362-1340. DOI: 10.1145/2775051.2677011. URL: http://doi.acm.org/10.1145/2775051.2677011.

[9] Tony Hoare, Stephan van Staden, Bernhard Möller, Georg Struth, and Huibiao Zhu. "Developments in concurrent Kleene algebra". In: *Journal of Logical and Algebraic Methods in Programming* 85.4 (2016), pp. 617–636. DOI: http://dx.doi.org/10.1016/j.jlamp.2015.09.012. URL: http://www.sciencedirect.com/science/article/pii/S2352220815000942.

[10] Tobias Kappé. Personal communication. June 2017.

[11] M. J. Keeling and K. T. Eames. "Networks and epidemic models". In: *Journal of the Royal Society Interface* 2.4 (2005), pp. 295–307. URL: http://doi.org/10.1098/rsif.2005.0051.

[12] S. C. Kleene. "Representation of events in nerve nets and finite automata". In: (1956). Ed. by C.E. Shannon and J. McCarthy, pp. 3–41.

[13] Donald Ervin Knuth. "Literate programming". In: *The Computer Journal* 27.2 (1984), pp. 97–111.

[14] Ioannis Kokkinis. Personal communication. July 2017.

[15] D. Kozen. "Kleene Algebra with Tests". In: *ACM Trans. Program. Lang. Syst.* 19.3 (May 1997), pp. 427–443. ISSN: 0164-0925. DOI: 10.1145/256167.256195. URL: http://doi.acm.org/10.1145/256167.256195.

[16] D. C. Kozen. "A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events". In: *Information and Computation* 110.2 (1994), pp. 366–390. ISSN: 0890-5401. DOI: http://dx.doi.org/10.1006/inco.1994.1037. URL: http://www.sciencedirect.com/science/article/pii/S0890540184710376.

[17] D. C. Kozen. *Automata and Computability*. Springer-Verlag Berlin Heidelberg, 1977.

[18] Dexter Kozen and Frederick Smith. *Kleene Algebra with Tests: Completeness and Decidability*. Tech. rep. Ithaca, NY, USA, 1996.

[19] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Černý. "Event-driven Network Programming". In: *SIGPLAN Not.* 51.6 (June 2016), pp. 369–385. ISSN: 0362-1340. DOI: 10.1145/2980983.2908097. URL: http://doi.acm.org/10.1145/2980983.2908097.

[20] J. J. M. M. Rutten. "Automata and coinduction (an exercise in coalgebra)". In: *CONCUR'98 Concurrency Theory: 9th International Conference Nice, France, September 8–11, 1998 Proceedings*. Ed. by Davide Sangiorgi and Robert de Simone. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 194–218. ISBN: 978-3-540-68455-8. DOI: 10.1007/BFb0055624. URL: http://dx.doi.org/10.1007/BFb0055624.

[21] J.J.M.M. Rutten. "Universal coalgebra: a theory of systems". In: *Theoretical Computer Science* 249.1 (2000), pp. 3–80. ISSN: 0304-3975. DOI: http://dx.doi.org/10.1016/S0304-3975(00)00056-6. URL: http://www.sciencedirect.com/science/article/pii/S0304397500000566.

[22] J.J.M.M. Rutten and B. Jacobs. "A Tutorial on (Co)Algebra and (Co)Induction". In: *EATCS Bulletin* 62 (1997), pp. 222–259.

[23] Gunther Schmidt. *Relational Mathematics*. New York, NY, USA: Cambridge University Press, 2010. ISBN: 0521762685, 9780521762687.

[24] A. Silva. *Kleene Coalgebra*. PhD thesis, University of Nijmegen, 2010.

[25] H. Van Ditmarsch, J. Van Eijck, P. Pardo, R. Ramezanian, and F. Schwarzentruber. "Gossip in Dynamic Networks". In: *Liber Amicorum Alberti, A Tribute to Albert Visser*. Ed. by J. Van Eijck, R. Iemhoff, and J. J. Joosten. College Publications, 2016, pp. 91–98.

[26] Yde Venema. "Algebras and Coalgebras". In: *Handbook of Modal Logic*. Ed. by Patrick Blackburn, Johan van Benthem, and Frank Wolter. Elsevier Science, 2006.

[27] S. Wilkins. *Software Defined Networking: Introduction To OpenFlow*. Dec. 2013. URL: `http://www.tomsitpro.com/articles/sdn-openflow-implementation,2-660.html`.