

# Towards an Algebraic Specification of Quantum Networks

**Anita Buckley<sup>1</sup>, Pavel Chuprikov<sup>1</sup>, Rodrigo Otoni<sup>1</sup>,**  
**Robert Rand<sup>2</sup>, Robert Soulé<sup>3</sup> and Patrick Eugster<sup>1</sup>**

<sup>1</sup> Università della Svizzera italiana (USI), Switzerland

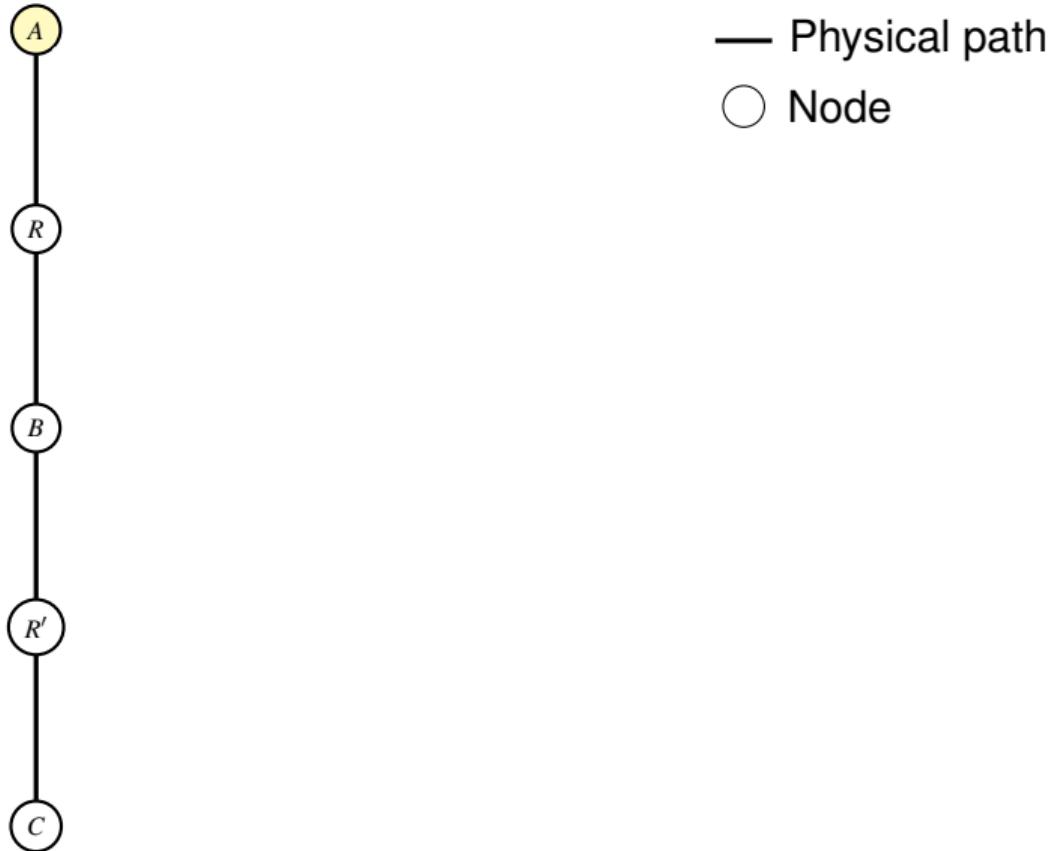
<sup>2</sup> University of Chicago, USA

<sup>3</sup> Yale University, USA

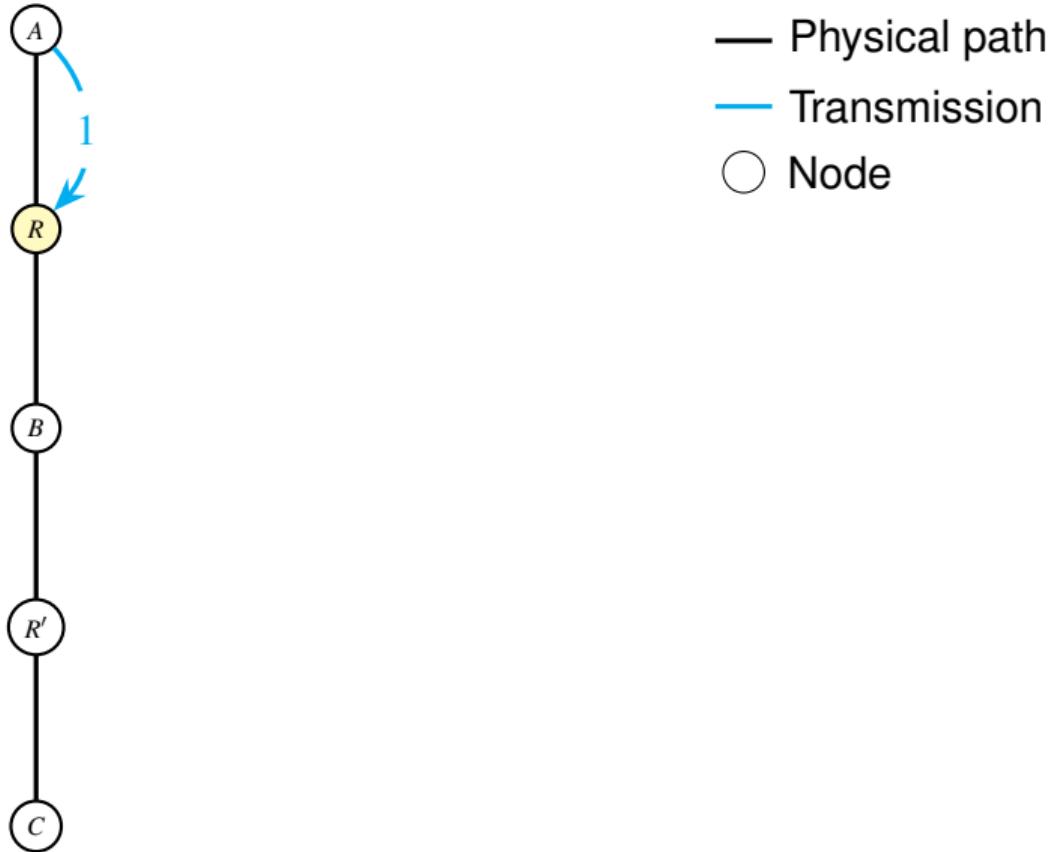
*New York City, 10th September 2023*

1st Workshop on Quantum Networks and Distributed Quantum Computing  
SIGCOMM 2023

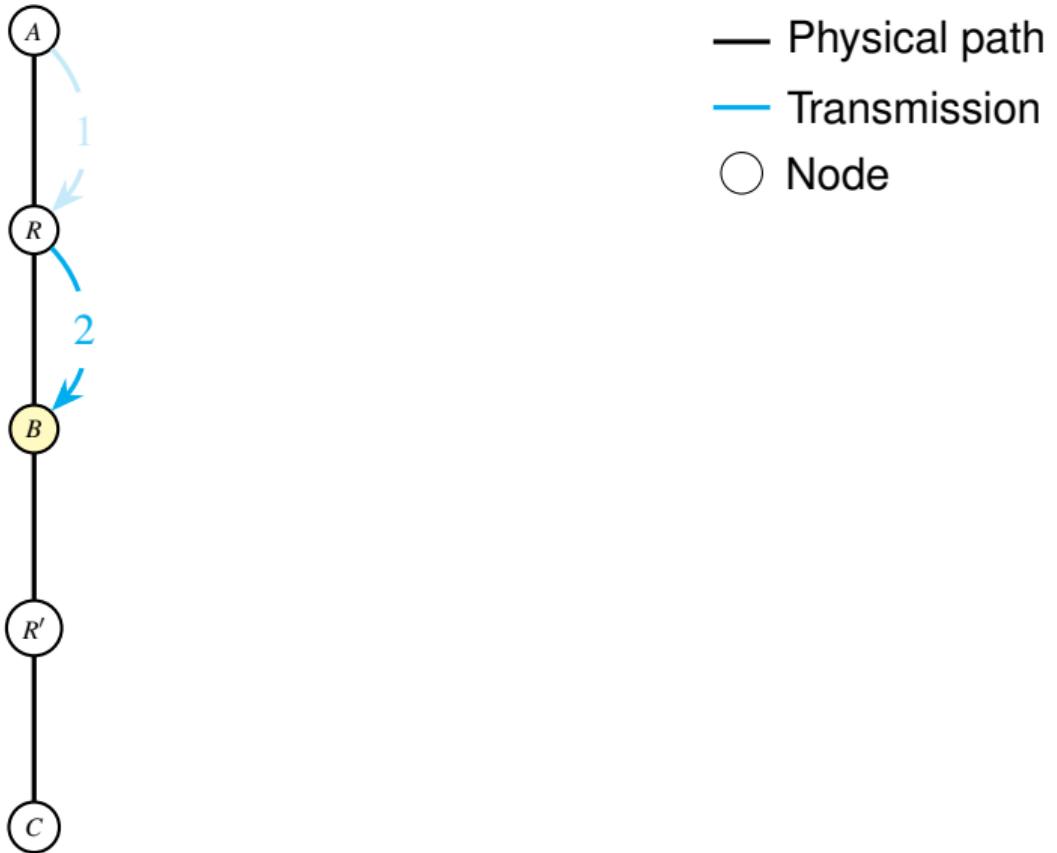
# Classical packet forwarding



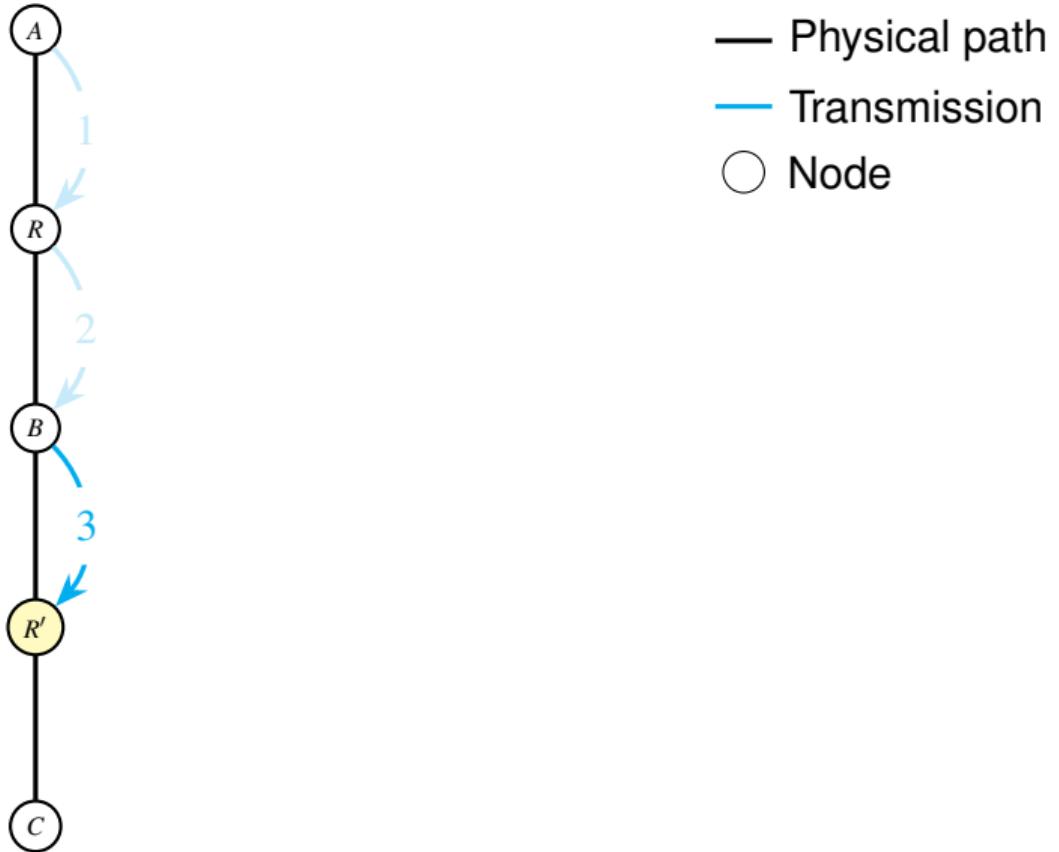
# Classical packet forwarding



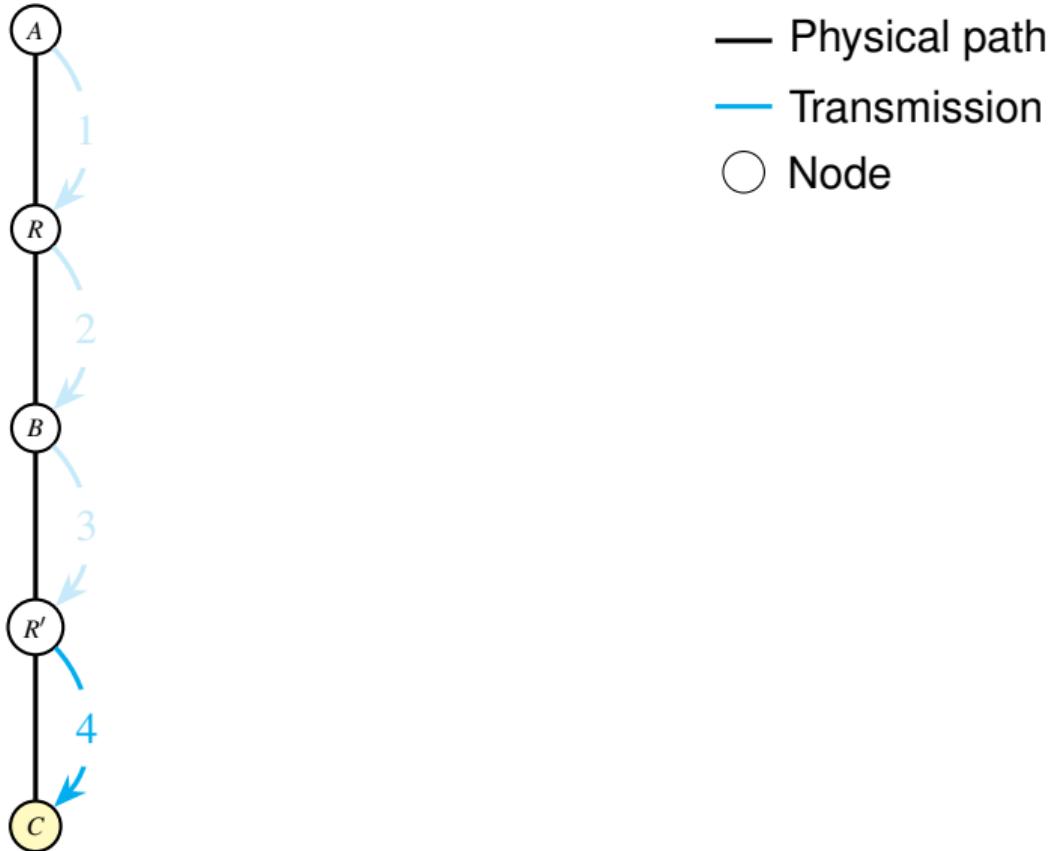
# Classical packet forwarding



# Classical packet forwarding



# Classical packet forwarding



## NetKAT: Semantic Foundations for Networks

Carolyn Jane Anderson  
Swarthmore College \*

Nate Foster  
Cornell University

Arjun Guha  
University of Massachusetts Amherst \*

Jean-Baptiste Jeannin  
Carnegie Mellon University \*

Dexter Kozen  
Cornell University

Cole Schlesinger  
Princeton University

David Walker  
Princeton University

### Abstract

Recent years have seen growing interest in high-level languages for programming networks. But the design of these languages has been largely ad hoc, driven more by the needs of applications and the capabilities of network hardware than by foundational principles. The lack of a semantic foundation left language designers with little guidance in determining how to incorporate new features, and provided no way to reason precisely about their code. This paper presents NetKAT, a new network programming language that is based on a solid mathematical foundation and comes equipped with a sound and complete equational theory. We describe the design of NetKAT, including primitives for filtering, modifying, and transmitting packets; union and sequential composition operators; and a Kleene star operator that handles preambles. We show that NetKAT is the union of a classical and well-known mathematical structure called a Kleene algebra with tests (KAT) and prove that its equational theory is sound and complete with respect to its denotational semantics. Finally, we present practical applications of the equational theory including syntactic techniques for checking readability, proving non-interference properties that ensure isolation between programs, and establishing the correctness of compilation algorithms.

**Categories and Subject Descriptors** D.3.2 [Programming Languages]: Language Classifications—Specialized application languages

**Keywords** Software-defined networking, Fretotic, Network programming languages, Domain-specific languages, Kleene algebra with tests, NetKAT

### 1. Introduction

Traditional network devices have been called ‘the last bastion of mainframe computing’ [9]. Unlike modern computers, which are

\* This work performed at Cornell University.

Permission to make digital or hard copies of all or part of this work for personal use or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright for works contained in this volume is held by others than ACM, if specified. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permission from [www.acm.org](http://www.acm.org).

POPL '14, January 22–24, 2014, San Diego, CA, USA.  
Copyright © 2014 ACM 978-1-4503-2544-8/14/01...\$15.00  
<http://doi.acm.org/10.1145/2535808.2535822>

implemented with commodity hardware and programmed using standard interfaces, networks have been built the same way since the 1970s: out of special-purpose devices such as routers, switches, firewalls, load balancers, and middle-boxes, each implemented with custom hardware and programmed using proprietary interfaces. This design makes it difficult to extend networks with new functionality and effectively impossible to reason precisely about their behavior.

However, a revolution has taken place with the recent rise of software-defined networking (SDN). In SDN, a general-purpose controller machine manages a collection of programmable switches. The controller responds to network events such as new connections, friend/host topology changes, and shifts in traffic load by reprogramming the switches. Because the controller itself has a global view of the network, it is easy to use SDN to implement a wide variety of standard applications such as shortest-path routing, traffic monitoring, and access control, as well as more sophisticated applications such as load balancing, intrusion detection, and fault-tolerance.

A key benefit of SDN is that it defines open standards that any vendor can implement. For example, the OpenFlow API [21] clearly specifies the capabilities and behavior of switch hardware and defines a low-level language for manipulating their configurations. However, programs written directly for SDN platforms such as OpenFlow are akin to assembly: easy for hardware to implement, but difficult for humans to write.

**Network programming languages.** In recent years, several different research groups have proposed domain-specific languages for SDN [5–7, 23–25, 31, 32]. The goal of these *network programming languages* is to raise the level of abstraction of network programs above hardware-oriented APIs such as OpenFlow, thus enabling them to be easily ported to different SDN applications. For example, the languages developed in the Fretotic project [30] support a two-phase programming model: (i) a general-purpose program responds to network events by generating a static forwarding policy; and (ii) the static policy is compiled and passed to a run-time system that configures the switches using OpenFlow messages. This model balances expressiveness—dynamic policies can be easily added by the general-purpose program—but generates a sequence of static policies for simplicity—forwarding rules are written in a simple domain-specific language with a small semantics, so programs can be analyzed and even verified using automated tools [7, 26].

Still, it has never been clear what features a static policy language should support. The initial version of Fretotic [6] used simple lists of predicate-action rules as policies, where the actions in-

# NetKAT: Semantic Foundations for Networks

Carolyn Jane Anderson  
Swarthmore College<sup>\*</sup>

Nate Foster  
Cornell University

Arijit Guha

Jean-Baptiste Jeannin  
Carnegie Mellon University<sup>\*</sup>

Dexter Kozen  
Cornell University

Cole Schlesinger  
Princeton University

## Abstract

Recent years have seen growing interest in high-level languages for programming networks. But the design of these languages has been largely ad hoc, driven more by the needs of applications and the capabilities of network hardware than by foundational principles. The lack of a semantic foundation left language designers with little guidance in determining how to incorporate new features, and provided no way to reason about the correctness of their code. This paper presents NetKAT, a new network programming language that is based on a solid mathematical foundation and comes equipped with a sound and complete equational theory. We describe the design of NetKAT, including primitives for filtering, modifying, and transmitting packets; union and sequential composition operators; and a Kleene star operator that binds previous programs. We show that NetKAT is the union of a classical and well-known mathematical structure called a Kleene algebra with tests (KAT) and prove that its equational theory is sound and complete with respect to its denotational semantics. Finally, we present practical applications of the equational theory including syntactic predicates for checking readability, proving non-interference properties that ensure isolation between programs, and establishing the correctness of compilation algorithms.

**Categories and Subject Descriptors** D.3.2 [Programming Languages]: Language Classifications—Specialized languages; D.3.2 [Programming Languages]: Language Classifications—Specialized languages.

**Keywords** Software-defined networking, Fretwork, Network programming languages, Domain-specific languages, Kleene algebra with tests, NetKAT.

## 1 Introduction

Traditional network devices have been called “the last bastion of mainframe computing” [9]. Unlike modern computers, which are

<sup>\*</sup>This work performed at Cornell University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright for works contained in this volume is held by their ACM members. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permission from [www.acm.org](http://www.acm.org).

POPL '14, January 22–24, 2014, San Diego, CA, USA.  
Copyright © 2014 ACM 978-1-4503-2544-8/14/01...\$15.00  
<http://doi.acm.org/10.1145/2539838.2539842>

## Probabilistic NetKAT

Nate Foster<sup>1</sup>, Dexter Kozen<sup>1</sup>, Konstantinos Mamouras<sup>2\*</sup>,  
Mark Reitblatt<sup>3\*</sup>, and Alexandra Silva<sup>4</sup>

<sup>1</sup> Cornell University

<sup>2</sup> University of Pennsylvania

<sup>3</sup> Facebook

<sup>4</sup> University College London

**Abstract.** This paper presents a new language for network programming based on a probabilistic semantics. We extend the NetKAT language with new primitives for expressing probabilistic behaviors and enrich the semantics from one based on deterministic functions to one based on measurable functions on sets of packet histories. To establish fundamental properties of the language, we prove that it is a complete semantics of the deterministic language, show that it satisfies a number of natural equations, and develop a notion of representation. We present case studies that show how the language can be used to model a diverse collection of scenarios drawn from real-world networks.

## 1 Introduction

Formal specification and verification of networks has become a reality in recent years with the emergence of network-specific programming languages and property-checking tools. Programming languages like Fretwork [21], Pyretic [36], Maple [32], FlowLog [38], and others are enabling programmers to specify the intended behavior of a network in terms of high-level constructs such as Boolean predicates and functions on packets. Verification tools like Header Space Analysis [21], VeriFlow [22], and NetKAT [12] are making it possible to check properties such as safety, liveness, loop freedom, and traffic isolation automatically.

However, despite many notable advances, these frameworks all have a fundamental limitation: they model network behavior in terms of deterministic packet-processing functions. This approach works well enough in settings where the network function is simple, or where the properties of interest only concern the forwarding paths used to carry traffic. But it does not provide satisfactory accounts of more complicated situations that often arise in practice:

- Congestion: the network operator wishes to calculate the expected degree of congestion on each link given a model of the demands for traffic.
- Failures: the network operator wishes to calculate the probability that packets will be delivered to their destination, given that devices and links fail with a certain probability.

<sup>\*</sup>This work performed at Cornell University.



### Abstract

Over the past 5–10 years, the rise of software-defined networking (SDN) has inspired a wide range of new research directions, hypotheses and paradigms for programming, managing, and debugging systems under behavior. Otherwise, these systems are *stateless*—languages for programming and managing networks, or verification, are *stateless* for reasons of consistency and debugging. In this paper, we present a solid framework, called Temporal NetKAT, capable of formalizing all of these areas at once. As its name suggests, Temporal NetKAT provides a formal semantics of temporal logic (time) over finite-linear temporal domains (linear, discrete). This allows us to write down concise programs in a pocket’s worth of lines, to reason about their temporal properties, and to program them directly to network hardware and network operator devices. In addition to making temporal programming easier, it provides a foundation for using new, general, program-based correctness properties. Using new, general, program-based correctness properties.

### Temporal NetKAT

Ryan Beckett  
Princeton University, USA  
ryanbecket@princeton.edu

Michael Greenberg  
Princeton College, USA  
mgreenber@princeton.edu



David Walker  
Princeton University, USA  
dwalker@princeton.edu

### Introduction

In software-defined networking, a general-purpose controller machine, or cluster of machines, manages a collection of hosts, programmable switches throughout a datacenter, and open API servers. OpenFlow [29]. In order to build reliable SDN networks, one requires a specification of necessary key abstractions, a platform for programming policies, and a way for monitoring packets of interest and sending them to the right places. One must also recall, that all of these components have been added in a myriad of platforms. For instance, Flowspec [36], BIRD [38], Linux’s Modem [42], NetKAT [11], and others have focus on efficient processing of incoming packets, depending on their semantics. For example, Flowspec [36] performs DREAM-like operations on flows [41], and others have focus on efficient processing functions like Header Space Analysis [21], VeriFlow [22], and NetKAT [12] among making it possible to check properties such as fairness, keep freedom, and traffic isolation automatically.

However, despite many notable advances, these frameworks all have a fundamental limitation: they model network behavior in terms of deterministic packet-processing functions. This approach works well enough in settings where the network infrastructure is simple, or where the properties of interest only concern the forwarding paths used to carry traffic. But it does not provide satisfactory accounts of more complicated situations that often arise in practice:

- Congestion: the network operator wishes to calculate the expected degree of congestion on each link given a model of the demands for traffic.
- Failures: the network operator wishes to calculate the probability that packets will be delivered to their destination, given that devices and links fail with a certain probability.

<sup>1</sup> Work performed at Cornell University.

Jean-Baptiste Jeannin<sup>\*</sup>  
Carnegie Mellon University

Dexter Kozen<sup>\*</sup>  
Cornell University

Cole Schlesinger<sup>\*</sup>  
Princeton University

## NetKAT: Semantic Foundations for Networks

Carolyn Jane Anderson  
Swarthmore College

Nate Foster  
Cornell University

Arijit Guha

### Abstract

Recent years have seen growing interest in high-level languages for programming networks. But the design of these languages has been largely ad hoc, driven more by the needs of applications and the capabilities of network hardware than by foundational principles. The lack of a semantic foundation for these languages makes little guidance in determining how to incorporate new features, and programming without a means to verify its property about their code.

This paper presents NetKAT, a new network programming language that is based on a solid mathematical foundation and comes equipped with a sound type system. It is built upon a solid mathematical foundation and comes equipped with a sound type system. It is built upon a solid mathematical foundation and comes equipped with a sound type system.

implemented w  
standard interfa  
the 1970s: out c  
firewalls, load  
with custom h  
faces. This de  
fundamentali  
their behavi  
However, i  
software-defi  
ruler machi  
The controli  
tions from h  
as program  
has a global  
ment a wide  
routing, traf  
plasticized i  
and fault-t  
A typ  
any vendor  
clearly spe  
and defini  
tions. How  
as OpenFl  
but differe

D.3.2 [Programming Lan  
Specialized application lan  
ing. Frenetic. Network pro  
languages. Kleene algebra

### Introduction

Formal specification and verification

of networks with the emergence of network-specific programming languages and property-checking tools. Programming languages like Frenetic [24], Pyretic [36], Maple [32], FlowLog [38], and others are enabling programmers to specify the intended behavior of a network in terms of high-level constructs such as Boolean predicates and functions on packets. Verification tools like Header Space Analysis [21], VeriFlow [22], and NetKAT [12] are making it possible to check properties such as fairness, keep freedom, and traffic isolation automatically.

Such advances have led to

mental limitations: they model network behavior in terms of deterministic packet-processing functions. This approach works well enough in settings where the network infrastructure is simple, or where the properties of interest only concern the forwarding paths used to carry traffic. But it does not provide satisfactory accounts of more complicated situations that often arise in practice:

- Congestion: the network operator wishes to calculate the expected degree of congestion on each link given a model of the demands for traffic.
- Failures: the network operator wishes to calculate the probability that packets will be delivered to their destination, given that devices and links fail with a certain probability.

<sup>\*</sup>Work performed at Cornell University.

## Probabilistic NetKAT

Nate Foster<sup>1</sup>, Dexter Kozen<sup>1</sup>, Konstantinos Mamouras<sup>2</sup>,  
Mark Reitblatt<sup>3</sup>, and Alexandra Siliva<sup>4</sup>

<sup>1</sup> Cornell University

<sup>2</sup> University of Pennsylvania

<sup>3</sup> Facebook

<sup>4</sup> University College London

**Abstract.** This paper presents a new language for network programming based on a probabilistic semantics. We extend the NetKAT language with primitives for expressing probabilistic behaviors and enrich the semantics from one based on deterministic functions to one based on measurable functions on sets of packet histories. We establish fundamental properties of probabilistic NetKAT and prove that it is a conservative extension of the deterministic language; we show that it satisfies a number of important properties such as closure and idempotence. We present case studies that show how the language can be used to model a diverse collection of semantics drawn from real-world networks.

### 1 Introduction

Formal specification and verification of networks have become a reality in recent years with the emergence of network-specific programming languages and property-checking tools. Programming languages like Frenetic [24], Pyretic [36], Maple [32], FlowLog [38], and others are enabling programmers to specify the intended behavior of a network in terms of high-level constructs such as Boolean predicates and functions on packets. Verification tools like Header Space Analysis [21], VeriFlow [22], and NetKAT [12] are making it possible to check properties such as fairness, keep freedom, and traffic isolation automatically.

However, despite many notable advances, these frameworks all have a fundamental limitation: they model network behavior in terms of deterministic packet-processing functions. This approach works well enough in settings where the network infrastructure is simple, or where the properties of interest only concern the forwarding paths used to carry traffic. But it does not provide satisfactory accounts of more complicated situations that often arise in practice:



## NetKAT: Semantic Foundations for Networks

Carolyn Jane Anderson  
Swarthmore College\*

Nate Foster  
Cornell University

Arijit Guha

Jean-Baptiste Jeannin  
Carnegie Mellon University\*

Dexter Kozen  
Cornell University

Cole Schlesinger  
Princeton University

### Abstract

Recent years have seen growing interest in high-level languages for programming networks. But the design of these languages has been largely ad hoc, driven more by the needs of applications and the capabilities of network hardware than by foundational principles. The lack of a semantic foundation for these languages makes it difficult to determine how to incorporate new features, and progress without a means to reason precisely about their code.

This paper presents NetKAT, a new way of thinking about network programming based on a solid mathematical foundation and comes equipped with a suite of tools—“the last bastion of correctness”

implemented w/ standard interfaces from the 1970s: out-of-firewalls, load with custom headers, with custom headers.

This de-abstracts their behavior.

However, *i* software-defined networking machine. The controller from h has a global port number, a wide range of traffic routing, traffic classification and fault-tolerance.

A vendor clearly specifies the definitions. How is OpenFlow but difficult

Network

forwarding for SDN grammars work

and application

purpose

forwards to a rule

message

to the port

with a certain probability.

\*Work performed at Cornell University.

### 1 Introduction

In software-defined networking, a general-purpose controller machine, or cluster of machines, manages a collection of hosts, programmers switches throughout a network, and devices, and performs all operations for programming and managing the network. In this paper, we present a solid framework, called Temporal NetKAT, capable of facilitating all of these tasks at once. As the name suggests, Temporal NetKAT provides a formal semantics of temporal logic over these many linear temporal and universal clauses. Together with Temporal NetKAT’s original predecessor, a programming language for writing linear temporal logic programs, we write down concise programs as a pocket’s worth of temporal logic and to make them easier to reason about. In addition, we extend the language with annotations that let us automatically generate programs from that base. This allows us to focus our attention on what that base does, as opposed to how to implement it. And NetKAT now generalizes proofs of path-based correctness properties. Using this new semantic framework, we show that the semantics of standard programs with traditional semantics, and, for actions of programs with traditional semantics, we have

verified ‘the last bastion of correctness’

on computers, which are

of this work for personal or commercial use, must not be distributed outside your organization, except by other NCM members, or republished, in whole or in part. It may be reproduced by a third party if: (a) it is not in copyrighted material; (b) it is not changed in any way; (c) it is not sold in any manner; and (d) credit is made to the source.

Copyright © IEEE. All rights reserved.

## Probabilistic NetKAT

Nate Foster<sup>1</sup>, Dexter Kozen<sup>1</sup>, Konstantinos Mamouras<sup>2\*</sup>,  
Mark Reitblatt<sup>3</sup><sup>4</sup>, and Alexandra Silva<sup>4</sup>

<sup>1</sup> Cornell University

<sup>2</sup> University of Pennsylvania  
<sup>3</sup> Facebook

<sup>4</sup> University College London

**Abstract.** This paper presents a new language for network programming based on a probabilistic semantics. We extend the NetKAT language with new primitives for expressing probabilistic behaviors and enrich the semantics from one based on deterministic functions to one based on measurable functions on sets of packet histories. To establish fundamental properties of the semantics, prove that it is a complete semantics of the deterministic one, we show that it satisfies a number of invariant equations, and develop a noisy semantics representation. We present case studies that show how the language can be used to model a diverse collection of scenarios drawn from real-world networks.

### 1 Introduction

Formal specification and verification of networks has become a reality in recent years with the emergence of network-specific programming languages and property-checking tools. Programming languages like Frenetic [31], Pyretic [36], and Maple [32], FlowLog [38], and others are enabling programmers to specify the intended behavior of a network in terms of high-level constructs such as Boolean predicates and functions on packets. Verification tools like Header Space Analysis [21], VeriFlow [22], and NetKAT [12] are making it possible to check properties such as safety, liveness, loop freedom, and traffic isolation automatically.

However, despite many notable advances, these frameworks all have a fundamental limitation: they model network behavior in terms of deterministic processing functions. This approach works well enough in settings where the forwarding paths tend to carry traffic. But it does not provide suitable accounts of more complicated situations that often arise in practice.

- Congestion: the network operator wishes to calculate the expected congestion on each link given a model of the demands for traffic. - Traffic: the network operator wishes to calculate the probability that traffic will be delivered to their destination, given that devices send traffic with a certain probability.

\*Work performed at Cornell University.

**PaLoNK: Functional Probabilistic NetKAT**  
ALEXANDER VANDENBROUCKE, EU-Louvain, Belgium  
TOM SCHROEVERS, EU-Louvain, Belgium  
This work is presented in part at LISA’19, a functional probabilistic network programming language that extends Probabilistic Network Logic (PaLoNL), a functional probabilistic network programming language for probabilistic correctness of networks. PaLoNL is also often shortened to PaLoNK.

This work is presented in part at LISA’19, a functional probabilistic network programming language that extends Probabilistic Network Logic (PaLoNL), a functional probabilistic network programming language for probabilistic correctness of networks. PaLoNL is also often shortened to PaLoNK.  
The following sections provide a brief overview of PaLoNL’s language and probabilistic semantics, followed by a formal presentation of its operational semantics. Finally, we conclude with some remarks on the standardization of PaLoNL and a short advertisement for the standardization process. We believe that the work we set out to do is very much a first step towards the development of a formal semantics for probabilistic network programming languages.

**Abstract**  
Over the past 5–10 years, the rise of software-defined networking (SDN) has inspired a wide range of new research, theories, hypotheses and languages for programming and managing networks, centered around the idea of separating the control plane from the data plane. In this paper, we present a solid framework, called Temporal NetKAT, capable of facilitating all of these tasks at once. As the name suggests, Temporal NetKAT provides a formal semantics of temporal logic over these many linear temporal and universal clauses. Together with Temporal NetKAT’s original predecessor, a programming language for writing linear temporal logic programs, we write down concise programs as a pocket’s worth of temporal logic and to make them easier to reason about. In addition, we extend the language with annotations that let us automatically generate programs from that base. This allows us to focus our attention on what that base does, as opposed to how to implement it. And NetKAT now generalizes proofs of path-based correctness properties. Using this new semantic framework, we show that the semantics of standard programs with traditional semantics, and, for actions of programs with traditional semantics, we have

Buckleynet et al.

Algebraic Specification of Quantum Networks

3/19

## NetKAT: Semantic Foundations for Networks

Carolyn Jane Anderson  
Swarthmore College\*

Nate Foster  
Cornell University

Arijit Guha

Jean-Baptiste Jeannin  
Carnegie Mellon University\*

Dexter Kozen  
Cornell University

Cole Schlesinger  
Princeton University

### Abstract

Recent years have seen growing interest in high-level languages for programming networks. But the design of these languages has been largely ad hoc, driven more by the needs of applications and the capabilities of network hardware than by foundational principles. The lack of a semantic foundation for these languages makes it difficult to determine how to incorporate new features, and progress without a means to reason precisely about their code.

This paper presents NetKAT, a new network programming language that is based on a solid mathematical foundation and comes equipped with a sound<sup>1</sup> semantics. It provides primitives for filtering, modifying, and sequential composition operators that iterates programs. We show axiomatic and well-studied mathematical results (KAT) and prove

implemented w standard interfaces the 1970s: out-of-firewalls, load with custom headers. This determinism is key to their behavior.

However, i software-defined networking machine. The control logic from hardware has a global implement a wide

### Probabilistic NetKAT

Nate Foster<sup>1</sup>, Dexter Kozen<sup>1</sup>, Konstantinos Mamouras<sup>2</sup>,  
Mark Reitblatt<sup>3</sup>, and Alexandra Silva<sup>4</sup>

<sup>1</sup> Cornell University  
<sup>2</sup> University of Pennsylvania  
<sup>3</sup> Facebook  
<sup>4</sup> University College London

**Abstract.** This paper presents a new language for network programming based on a probabilistic semantics. We extend the NetKAT language with new primitives for expressing probabilistic behaviors and enrich the semantics from one based on deterministic functions to one based on measurable functions on sets of packet histories. To establish fundamental properties of the semantics, we prove that it is a complete extension of the deterministic semantics, show that it satisfies a number of natural equations, and develop a sound type representation. We present a series of case studies that show how the language can be used to model a diverse collection of scenarios drawn from real-world networks.

### Temporal NetKAT

Ryan Beckett  
Princeton University, USA  
rbeckett@princeton.edu

Michael Greenberg  
Princeton College, USA  
mgreenberg@princeton.edu



David Walker  
Princeton University, US  
dwalk@princeton.edu

### DyNetKAT: An Algebra of Dynamic Networks \*

Georgiana Caltais<sup>2</sup>✉, Hossein Hojjati<sup>2</sup>✉, Mohammad Reza Mousavil<sup>3</sup>✉, and  
Hünkar Can Tunc<sup>4</sup>✉

<sup>1</sup> University of Konstanz, Germany & University of Twente, The Netherlands  
g.c.caltais@utwente.nl

<sup>2</sup> TelAS, Kharazmi University & University of Tehran, Iran  
hojjati@ut.ac.ir

<sup>3</sup> King Abdullah University of Science and Technology, Saudi Arabia  
mousavi@kaust.edu.sa

<sup>4</sup> University of Konstanz, Germany & Aarhus University, Denmark  
tunc@cs.au.dk

### Abstract

Over the past 5-10 years, the rise of software-defined networking (SDN) has inspired a wide range of new research, tooling, and deployment strategies for programming networks. However, these systems are ad-hoc—new languages for programming SDNs are starting to appear, and there is little support for reuse, consistency, and debugging. In this paper, we present DyNetKAT, a framework for building dynamic network protocols. DyNetKAT is based on NetKAT, the algebra of the formal language Temporal NetKAT, which extends the standard NetKAT semantics to handle time-varying temporal logic and introduces closure operators with TempKAT. TempKAT generalizes Temporal NetKAT to write down concise programs as a packet's path through a network and to make them easier to reason about. In addition, we extend DyNetKAT with a new general purpose language for writing sequential programs, ProgramKAT, which builds upon the semantics of TempKAT and NetKAT to encode proofs of path-compositionality, property preservation, and modular correctness. Using these semantics as a foundation, we show that the semantics of DyNetKAT are sound and complete, and that they support standard programmatic constructs, such as conditionals, loops, and parallel execution. We base our work on NetKAT, the algebra of the standard NetKAT semantics, and, for actions involving temporal logic, we build on TempKAT.

**Abstract.** We introduce a formal language for specifying dynamic updates for Software Defined Networks. Our language builds upon Network Kleene Algebras with Tests (NetKAT) and adds constructs for synchronization and multi-packet behaviour to capture the interaction between the network and its environment. We also provide a sound and complete, and ground-complete, axiomatization of our language. We exploit the equational theory and provide an efficient method for reasoning about safety properties. We implement our equational theory in DyNetKAT—a tool prototype, based on the Maude Rewriting Logic and the NetKAT tool, and apply it to a case study. We show that we can analyse the case study for networks with hundreds of switches using our tool prototype.

### Introduction

Software and verification of networks has become a reality in recent years with the emergence of network-specific programming languages and verification tools. Programming languages like Frenetic [21], Pyretic [36], FlowLog [38], and others are enabling programmers to specify the behavior of a network in terms of high-level constructs such as Boolean and functions on packets. Verification tools like Header Space Analysis [22], and NetKAT [12] are making it possible to check properties like security, loop freedom, and traffic isolation automatically.

Despite many notable advances, these frameworks all have a limitation: they model network behavior in terms of deterministic properties. This approach works well enough in settings where traffic is simple, or where the properties of interest only concern paths used to carry traffic. But it does not provide sufficient support for more complex situations that often arise in practice.

on the network operator wishes to calculate the expected traffic on each link given a model of the demands for traffic he or she. An operator wishes to calculate the probability that traffic delivered to their destination, given that devices are idle.

at Cornell University.

**PolyNK: Functional Probabilistic NetKAT**  
ALEXANDER VANDENBROUCKE, EUGENE BELIBAN  
TOM SCHRÖVERS, EU-LEMUR, BELGIUM  
This work is presented PolyNK, a functional probabilistic network programming language that extends Probabilistic NetKAT with LilaPINK. It enables probabilistic modeling of network behaviors for mobile devices, for static and vehicle routes, for reliable packet delivery, and for end-to-end flows. The language is designed for portable and efficient execution. The core idea was to apply state-of-the-art functional programming techniques to the standard NetKAT semantics. Recently, many improvements have been made to the language, and now it is a complete and expressive language, does not suffer from the standard NetKAT semantics' limitations, and provides a more advanced feature set.

We believe that our work is not yet finished. There is still much work to be done. We have developed a new version of the language, and we are currently working on improving its performance and ease of use. We are also working on improving the language's expressiveness and making it more suitable for real-world applications. We hope that our work will contribute to the development of a new generation of network programming languages.

## Concurrent NetKAT

Modeling and analyzing stateful, concurrent networks

Jana Wagemaker<sup>1</sup>  36, Nate Foster<sup>2</sup> , Tobias Kappé<sup>3</sup> , Dexter Kozen<sup>4</sup>  Jorijn Ruys<sup>2</sup>, and Alexandra Silva<sup>2</sup> 

<sup>1</sup> Radboud University, Nijmegen, The Netherlands

<sup>2</sup> Carnegie Mellon University

<sup>3</sup> Cornell University, Ithaca, New York, USA

<sup>4</sup> ILLC, University of Amsterdam, The Netherlands

**Abstract.** We introduce Concurrent NetKAT (CntrNetKAT), an extension of NetKAT with operators for specifying and reasoning about concurrent network semantics where multiple packets interact through state. We provide a model of the language based on partially-ordered multiset rewriting (pomesa), which uses a well-established mathematical structure to define the denotational semantics of concurrent languages. We provide a sound and complete axiomatization of this model, and we illustrate the use of CntrNetKAT through examples. More generally, CntrNetKAT can be understood as an algebraic framework for reasoning about programs with both local state (in packets) and global state (in a global store).

**Keywords:** Concurrent Kleene algebras, NetKAT, completeness, concurrency

## 1 Introduction

Kleene algebras (KA) is a well-studied formalism [20, 23, 34, 8] for analyzing and verifying imperative programs. Over the past few decades, various extensions of KA have been proposed for modeling increasingly sophisticated semantics. For example, Kleene algebras with tests (KAT) [11] models conditional control flow while NetKAT [3,10] models both flows in packet-switched networks.

A key limitation of NetKAT, however, is that the language is stateless and sequential. It cannot model programs composed in parallel, and it offers no way to reason algebraically about the effects induced by multiple concurrent programs. Meanwhile, the software-defined networking (SDN) paradigm has evolved to include richer functionality based on statistical programming including data aggregation and policy routing. In languages like P4 [4], issues of concurrency arise because the semantics depends on the order that packets are processed.

Given this context, it is natural to wonder whether we can add concurrency to NetKAT while retaining the elegance of the underlying framework. In this paper, we answer this question in the affirmative, by developing CntrNetKAT. In order to do this, we must overcome several challenges. A first hurdle is that networks exhibit many different forms of concurrent behavior. The most obvious source

## NetKAT: Semantic Foundations for Networks

Carolyn Jane Anderson  
Swarthmore College \*

Nate Foster  
Cornell University

Arijit Guha

Jean-Baptiste Jeannin  
Carnegie Mellon University \*

Dexter Kozen  
Cornell University

Cole Schlesinger  
Princeton University

### Abstract

Recent years have seen growing interest in high-level languages for programming networks. But the design of these languages has been largely ad hoc, driven more by the needs of applications and the capabilities of network hardware than by foundational principles. The lack of a semantic foundation left language designers with little guidance in determining how to incorporate new features, and programmers without a means to reason precisely about their code.

This paper presents NetKAT, a new network programming language that is based on a solid mathematical foundation and comes equipped with a sound and complete equational theory. We describe primitives for filtering, modifying, and sequential composition operators that iterates programs. We show anamorphisms and well-studied mathematical techniques with tests (KAT) and prove

implemented w standard interfaces the 1970s: out-of firewalls, load with custom rules. This defines their behavior as a function of their environment.

However, in *softstate-decoder* machine

The controller

comes from h

as program

has a globa

ment a wide



David Walker  
Stanford University, US  
<http://www.cs.stanford.edu/~walker/>



Jana Wagemaker  
Radboud University, NL  
<http://www.cs.ru.nl/~wajo/>

## DiNetKAT: An Algebra of Dynamic Networks \*

Georgiana Caltais<sup>1</sup> , Hossein Hajiost<sup>1</sup> , Mohammad Reza Mousavil<sup>2</sup> , and Hanuk Cun Tunc<sup>3</sup> 

<sup>1</sup> University of Konstanz, Germany & University of Twente, The Netherlands

[g.caltais@utwente.nl](http://cs.kit.edu/~caltais/)

<sup>2</sup> TELAS, Khatai University & University of Tehran, Iran

[hajiostab.com](http://hajiostab.com)

<sup>3</sup> King Abdullah University of Science and Technology, Saudi Arabia

[tunc@cse.cs.kit.edu](http://cse.cs.kit.edu/~tunc/)

working, a general p

of switches, mont

of switches through

flowfilter [29]. Is o

re required, we

planning the program

long packets of var

ation. Once the p

'been included in a t

and modified [36].

., and others have

been used to pro

grammatical gram

for writing programs

specifications for

switches [41]. Pat-Or

are pure, sequential,

programming language

and have been ap-

plied in order to

analyze and syn-

thesis. In order to

work with the

language, we built a

more specific lan-

guage for writing

switch programs

and switch config

**Abstract.** We introduce a formal language for specifying dynamic update software for Software Defined Networks. Our language builds upon Network Kleene Algebras with Tests (NetKAT) and adds constructs for synchronization and multi-threaded behaviour to capture the interaction between the different components of a network. We also add support for parallel programs and ground-complete axiomatization of our language. We exploit the equational theory and provide an efficient method for reasoning about sequential properties. We implement our equational theory in DiNetKAT – a tool prototype, based on the Maude Rewriting Logic and the NetKAT tool, and apply it to a case study. We show that we can analyse the case study for networks with hundreds of switches using our tool prototype.

Kleene Algebras, Software Defined Networks, Dynamic Update, Programming

## Probabilistic NetKAT

Nate Foster<sup>1</sup>, Dexter Kozen<sup>1</sup>, Konstantinos Mamouras<sup>2</sup>,  
Mark Reitblatt<sup>3</sup>, and Alexandra Silva<sup>4</sup>

<sup>1</sup> Cornell University

<sup>2</sup> University of Pennsylvania

<sup>3</sup> Facebook

<sup>4</sup> University College London

**Abstract.** This paper presents a new language for network programming based on a probabilistic semantics. We extend the NetKAT language with new primitives for expressing probabilistic behaviors and enrich the semantics from one based on deterministic functions to one based on measurable functions on sets of packet histories. We establish fundamental properties of the semantics, prove that it is a consequence of the determinism of NetKAT, and show that it satisfies a number of natural equations, and develop a notion of approximation. We present a grammar that shows how the language can be used to model a diverse collection of scenarios drawn from real-world networks.

## Introduction

Specification and verification of networks have become a reality in recent years, with the emergence of network-specific programming languages and verification tools. Programming languages like FLOWLOG [21], Pyretic [36], and FlowLog [36], among others are enabling programmers to specify the behavior of a network in terms of high-level constructs such as Boolean and functions on packets. Verification tools like Header Space Analysis [22], and NetKAT [12] are making it possible to check properties of network protocols, verify end-to-end reachability, and validate security, keep freshness, and traffic isolation automatically. Despite many notable advances, these frameworks all have a fundamental limitation: they model network behavior in terms of deterministic processes. Thus, approaches will enough in settings where only simple paths exist to carry traffic. But it does not provide suitable ways to handle complex situations that often arise in practice.

on the network operator wishes to calculate the expected traffic on each link given a model of the demands for traffic. If the network operator wishes to calculate the probability of a device delivered to their destination, given that devices are not independent. The Header Space Analysis (HS) [22] approach makes it possible to verify network performance by providing probabilistic guarantees for network protocols. The NetKAT approach is similar, but it is based on probabilistic logic. We believe that this work will be an important step in the direction of probabilistic network verification.

at Cornell University.

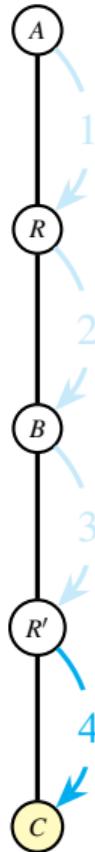
## PaLoNK: Functional Probabilistic NetKAT

ALEXANDER VANDENBROUCKE, EUGÈNE BELIOTT,  
TOM SCHRÖVERS, ERLING BENGTSSON

This work presents PaLoNK, a functional probabilistic network programming language that extends Probabilistic FlowLog [12] and Lala-Pickle [1] to enable probabilistic reasoning about network behaviors. PaLoNK extends FlowLog by adding probabilistic constructs, including probabilistic functions, probabilistic assignments, and probabilistic loops. It also allows the programmer to make probabilistic decisions based on network observations. The main contributions of PaLoNK are the design and implementation of a probabilistic functional language that supports probabilistic annotations, probabilistic assignments, and probabilistic loops. The language is designed to be expressive and flexible, allowing the programmer to express complex probabilistic behaviors in a clean and concise way.

In this paper, we present the design and implementation of PaLoNK. We show how PaLoNK can be used to model network behaviors in a functional probabilistic language. We demonstrate the expressiveness and flexibility of PaLoNK through several examples, showing how it can be used to model complex probabilistic behaviors in a functional probabilistic language. We believe that PaLoNK will be an important step in the direction of probabilistic network verification.

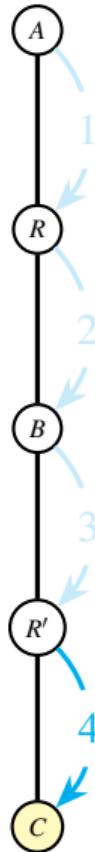
## NetKAT expression $p^*$ is encoding network's behavior



$$p \equiv (\text{SW} = A; \text{SW} \leftarrow R) + (\text{SW} = R; \text{SW} \leftarrow B) + (\text{SW} = B; \text{SW} \leftarrow R') + (\text{SW} = R'; \text{SW} \leftarrow C)$$

$$p^* \equiv 1 + p + p;p + p;p;p + \dots$$

# NetKAT expression $p^*$ is encoding network's behavior

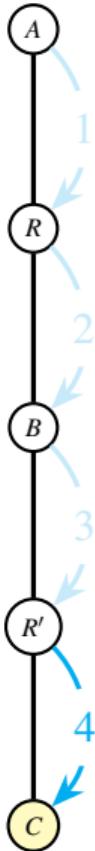


test

$$p \equiv (\text{SW} = A; \text{SW} \leftarrow R) + (\text{SW} = R; \text{SW} \leftarrow B) + (\text{SW} = B; \text{SW} \leftarrow R') + (\text{SW} = R'; \text{SW} \leftarrow C)$$

$$p^* \equiv 1 + p + p; p + p; p; p + \dots$$

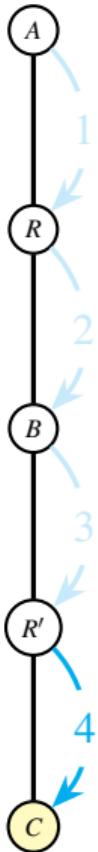
# NetKAT expression $p^*$ is encoding network's behavior



test                                  action  
p  $\equiv (\text{SW} = A; \text{SW} \leftarrow R) + (\text{SW} = R; \text{SW} \leftarrow B) +$   
 $(\text{SW} = B; \text{SW} \leftarrow R') + (\text{SW} = R'; \text{SW} \leftarrow C)$

$$p^* \equiv 1 + p + p; p + p; p; p + \dots$$

# NetKAT expression $p^*$ is encoding network's behavior

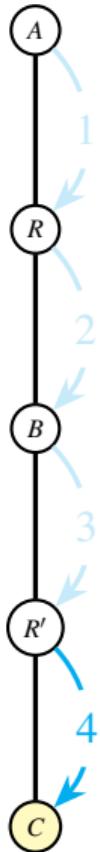


test                                                          action  
$$p \equiv (\text{SW} = A; \text{SW} \leftarrow R) + (\text{SW} = R; \text{SW} \leftarrow B) +$$
$$(\text{SW} = B; \text{SW} \leftarrow R') + (\text{SW} = R'; \text{SW} \leftarrow C)$$

$$p^* \equiv 1 + p + p; p + p; p; p + \dots$$

sequential composition

# NetKAT expression $p^*$ is encoding network's behavior



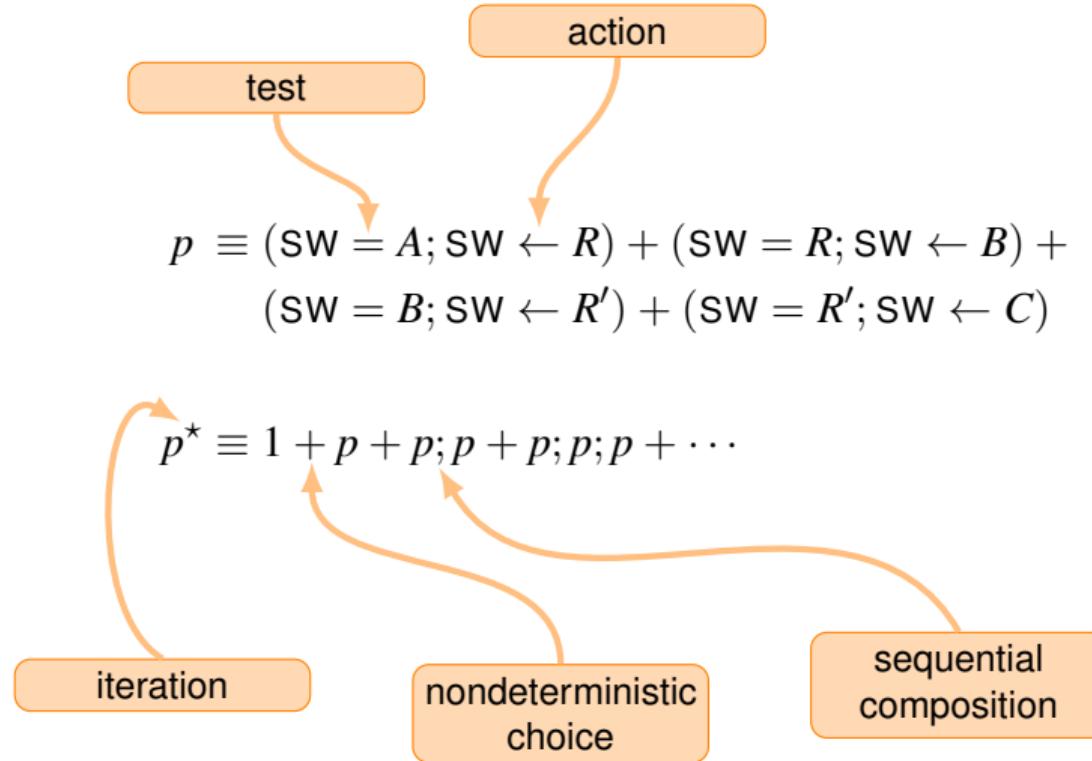
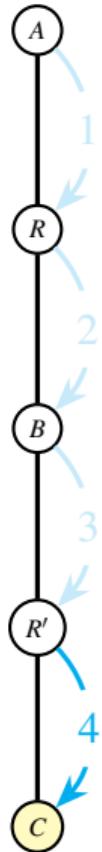
action  
test  
 $p \equiv (\text{SW} = A; \text{SW} \leftarrow R) + (\text{SW} = R; \text{SW} \leftarrow B) + (\text{SW} = B; \text{SW} \leftarrow R') + (\text{SW} = R'; \text{SW} \leftarrow C)$

$$p^* \equiv 1 + p + p; p + p; p; p + \dots$$

nondeterministic choice

sequential composition

# NetKAT expression $p^*$ is encoding network's behavior

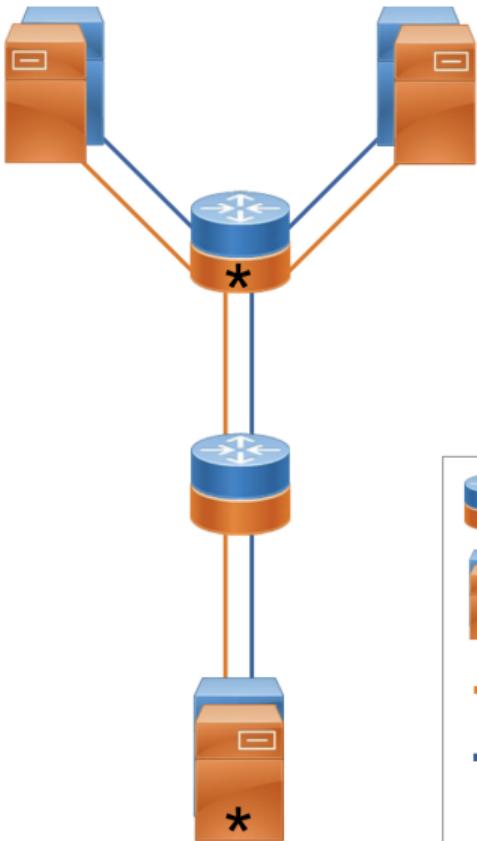




# Bell pair: maximally entangled quantum bits

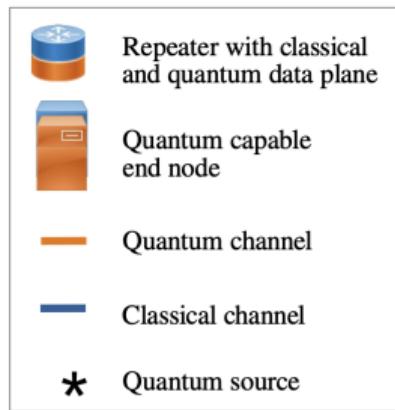
- fundamental unit in quantum networks
- consists of two quantum bits (qubits):  $R \sim B$  is a Bell pair distributed between nodes  $R$  and  $B$
- no headers: control information needs to be sent via separate classical channels





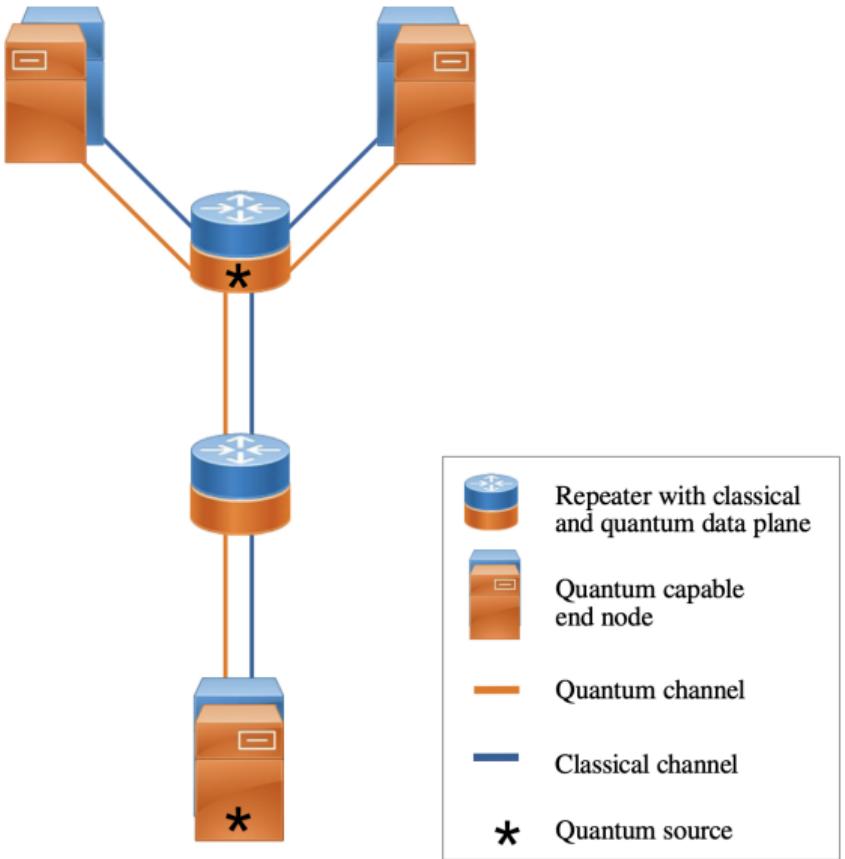
**Quantum network:**  
providing *communication services* to distributed quantum applications

1



<sup>1</sup>W. Kozlowski and S. Wehner: *Towards Large-Scale Quantum Networks*. NANOCOM (2019)

<sup>2</sup>IRTF, QIRG: *Architectural Principles for a Quantum Internet*. RFC 9340 (2023)



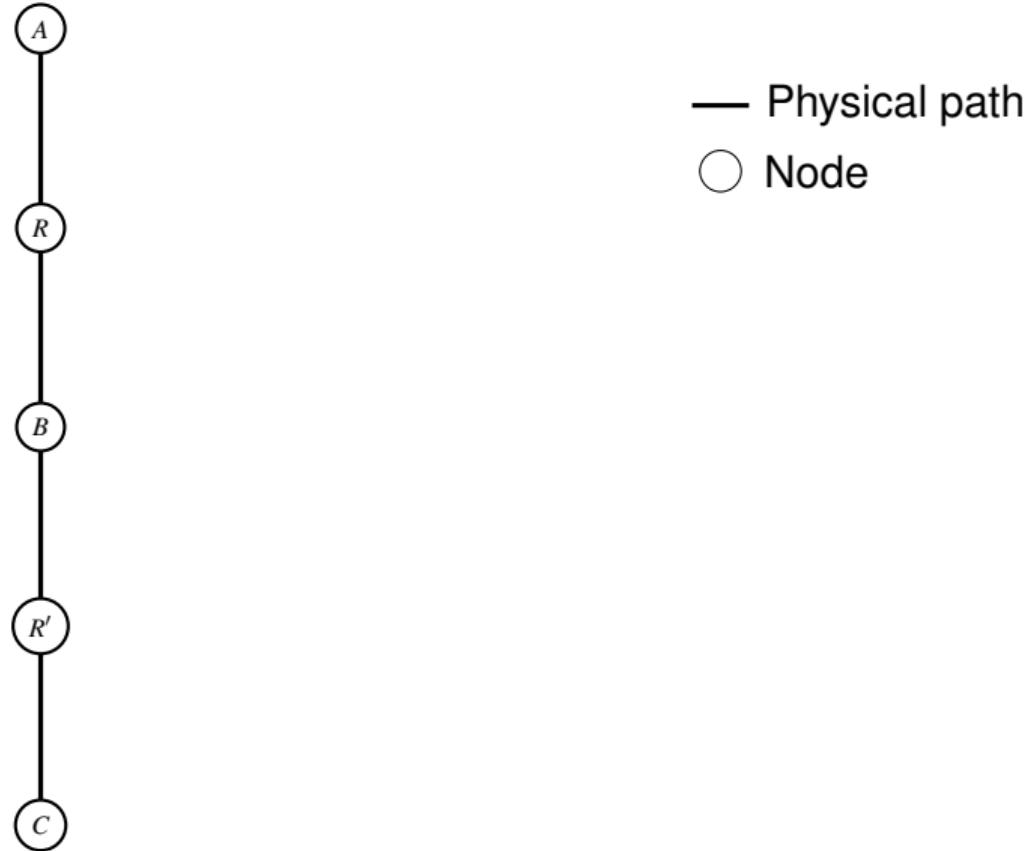
**Quantum network:**  
providing *communication services* to distributed quantum applications

end-to-end Bell pair distribution<sup>2</sup>

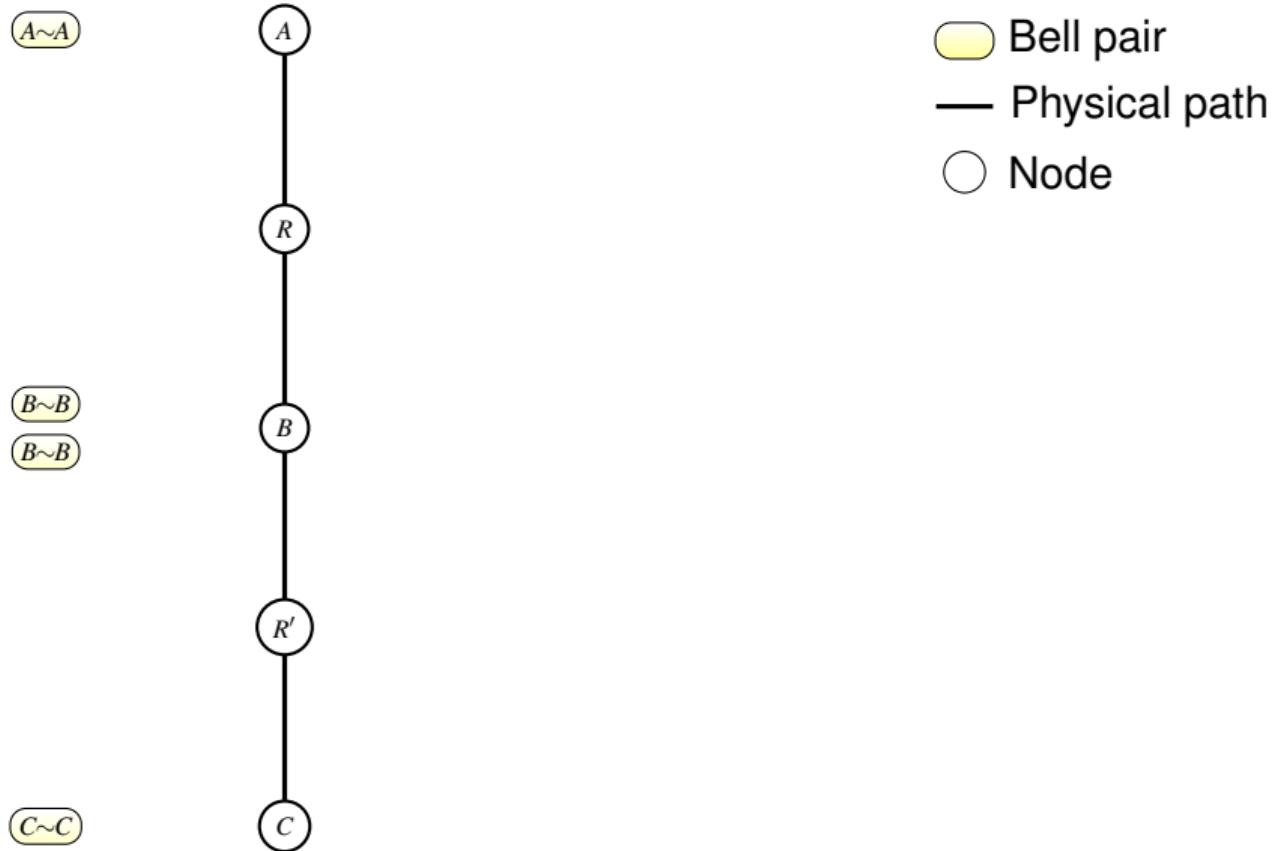
<sup>1</sup>W. Kozlowski and S. Wehner: *Towards Large-Scale Quantum Networks*. NANOCOM (2019)

<sup>2</sup>IRTF, QIRG: *Architectural Principles for a Quantum Internet*. RFC 9340 (2023)

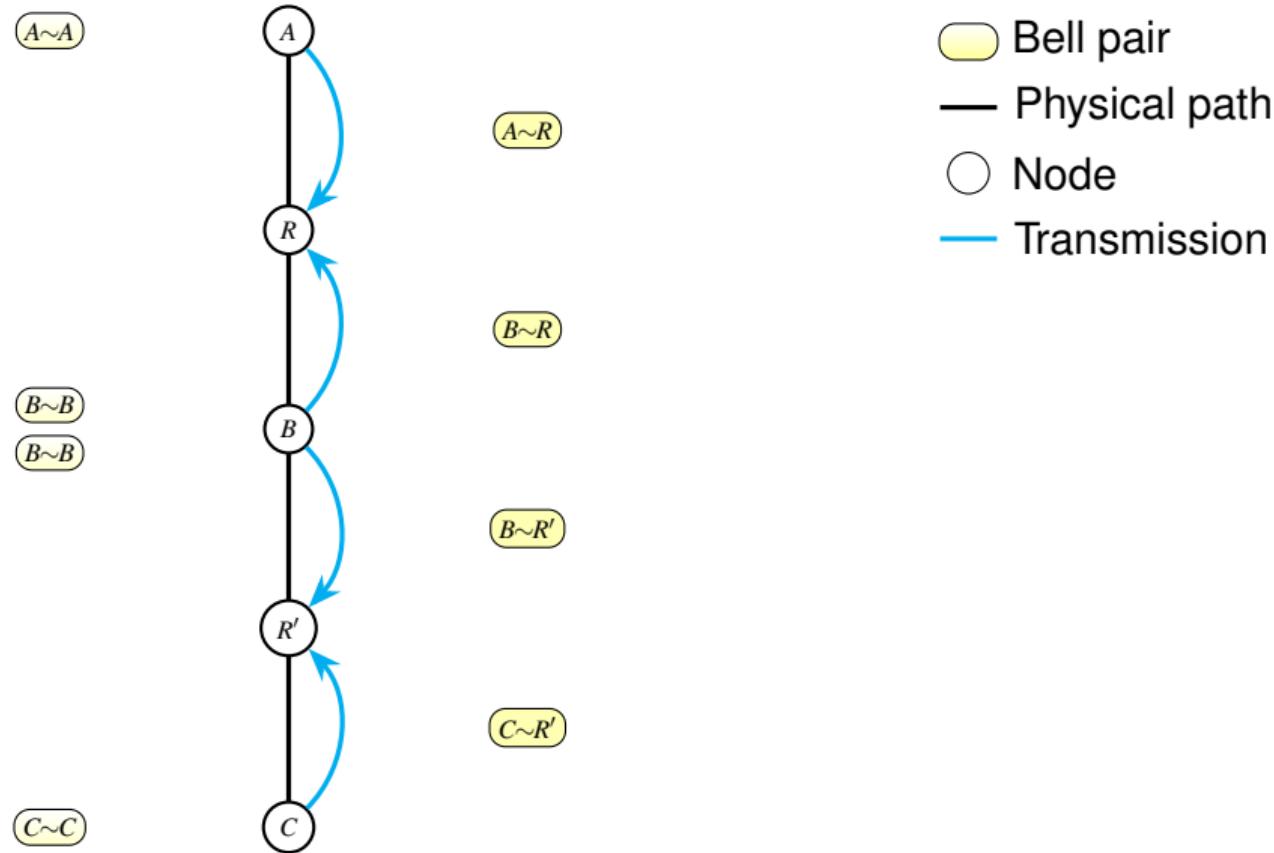
## Bell pair generation



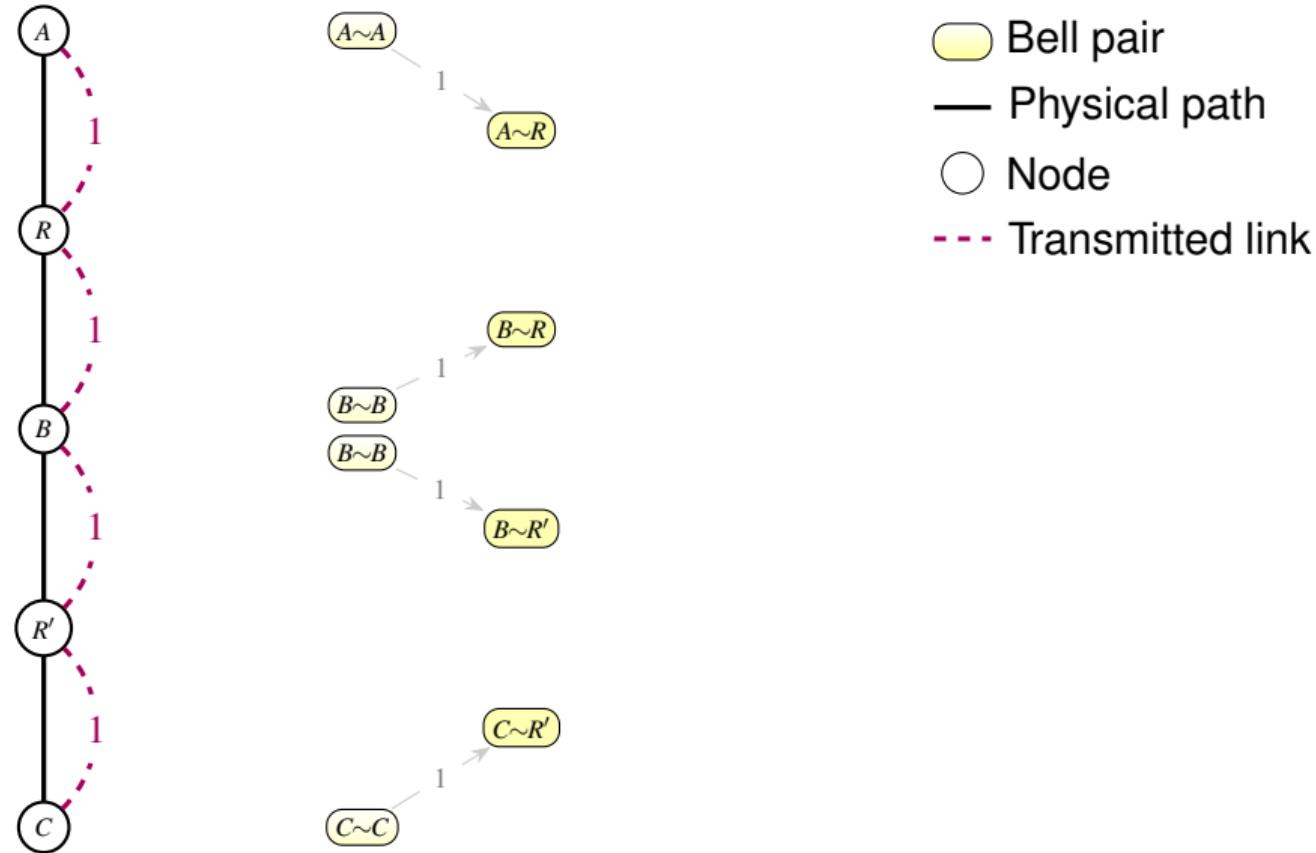
# Bell pair generation: Protocol I



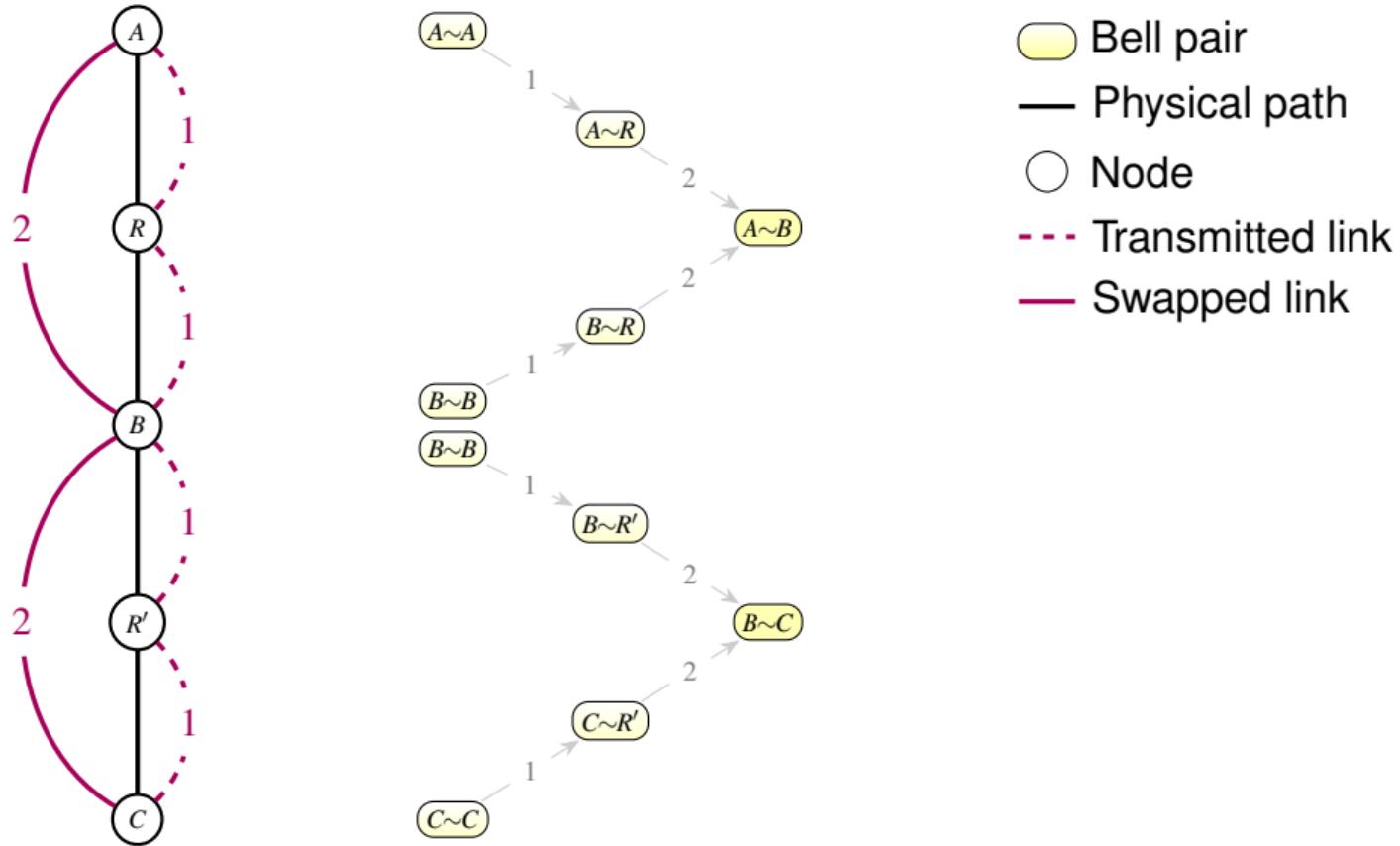
# Bell pair generation: Protocol I



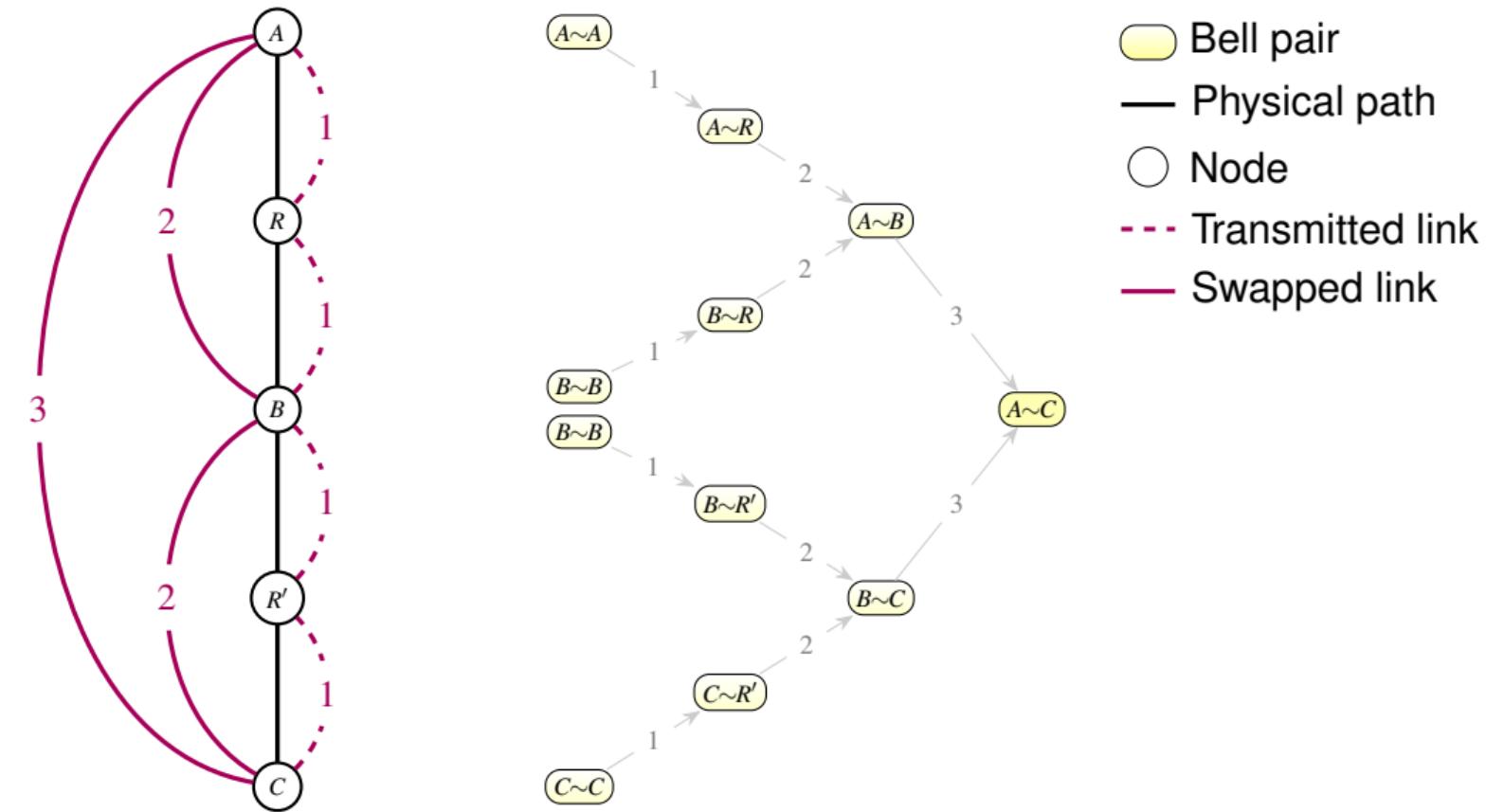
## Bell pair generation: Protocol I, round 1



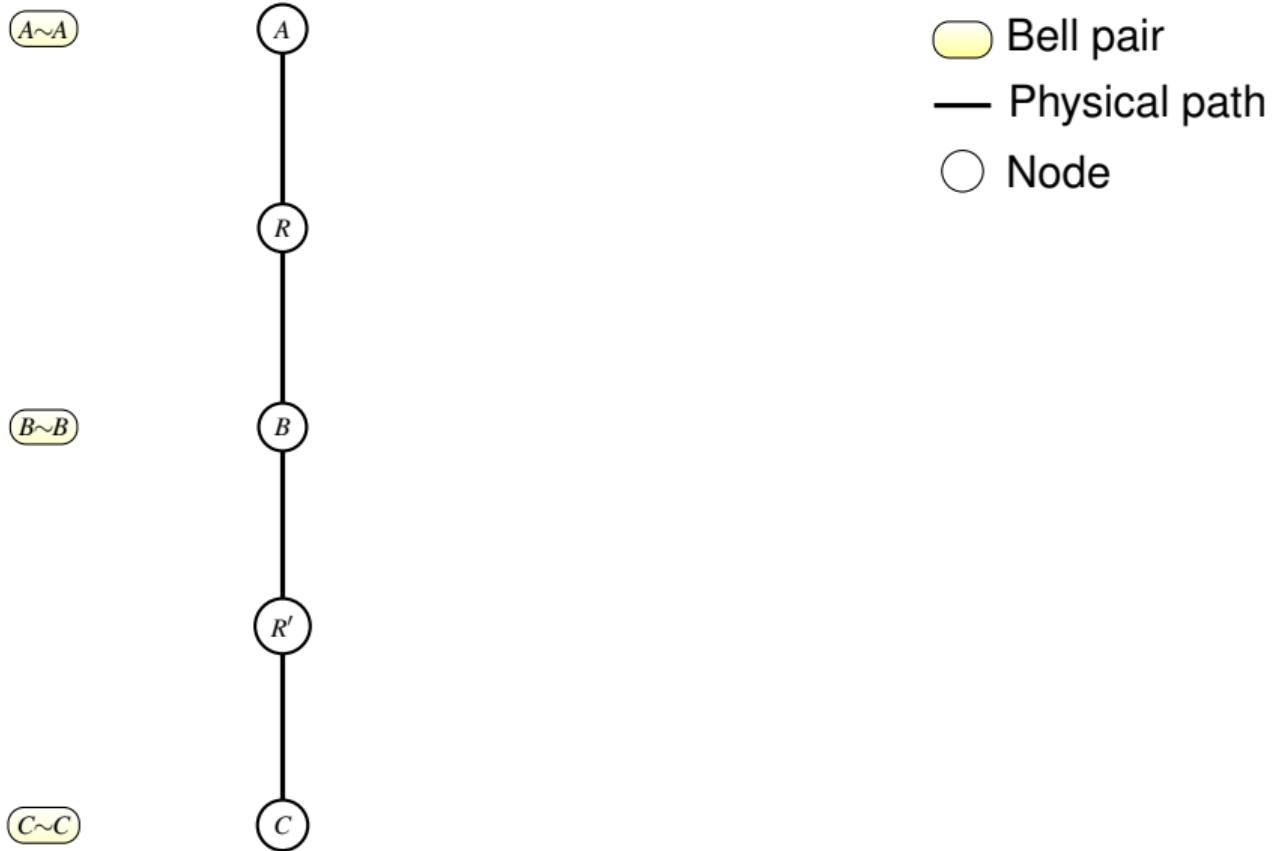
## Bell pair generation: Protocol I, round 2



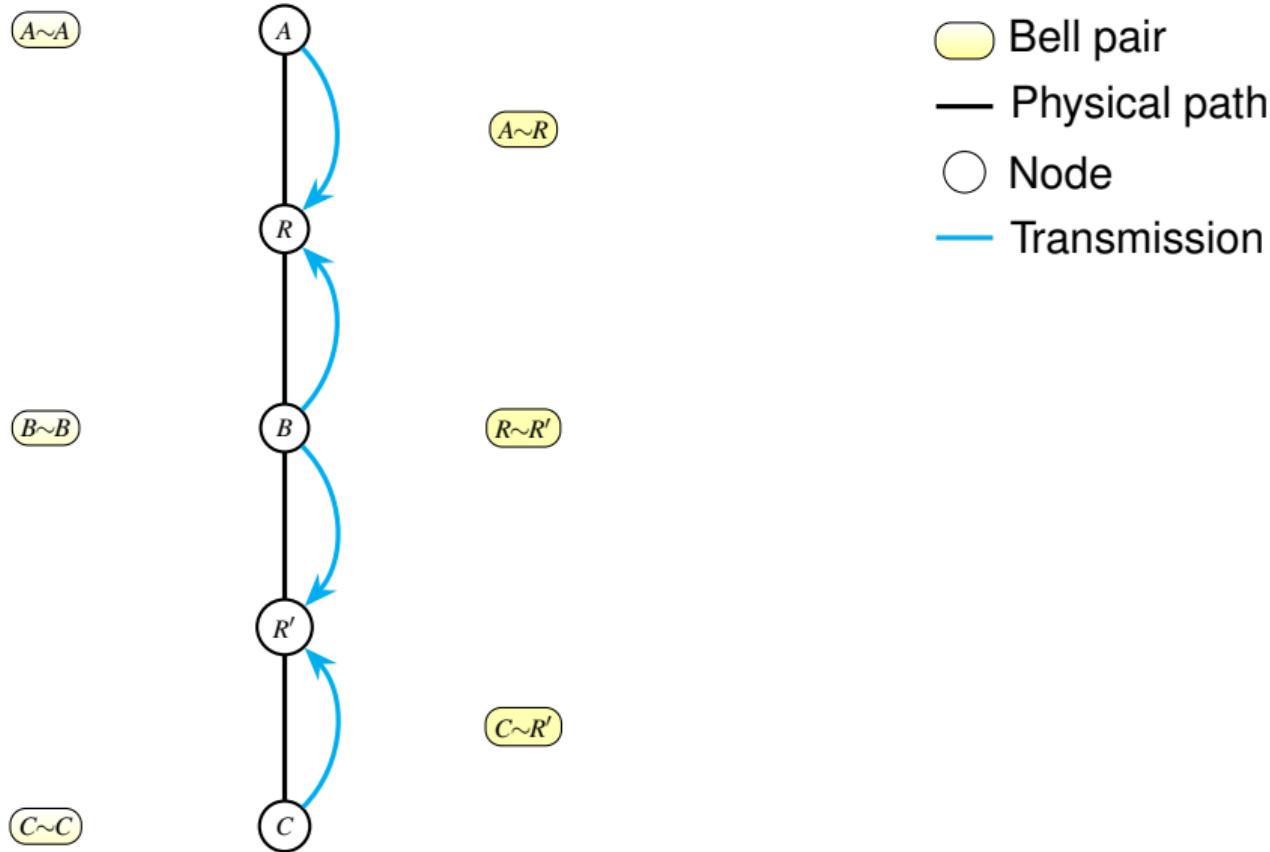
## Bell pair generation: Protocol I, round 3



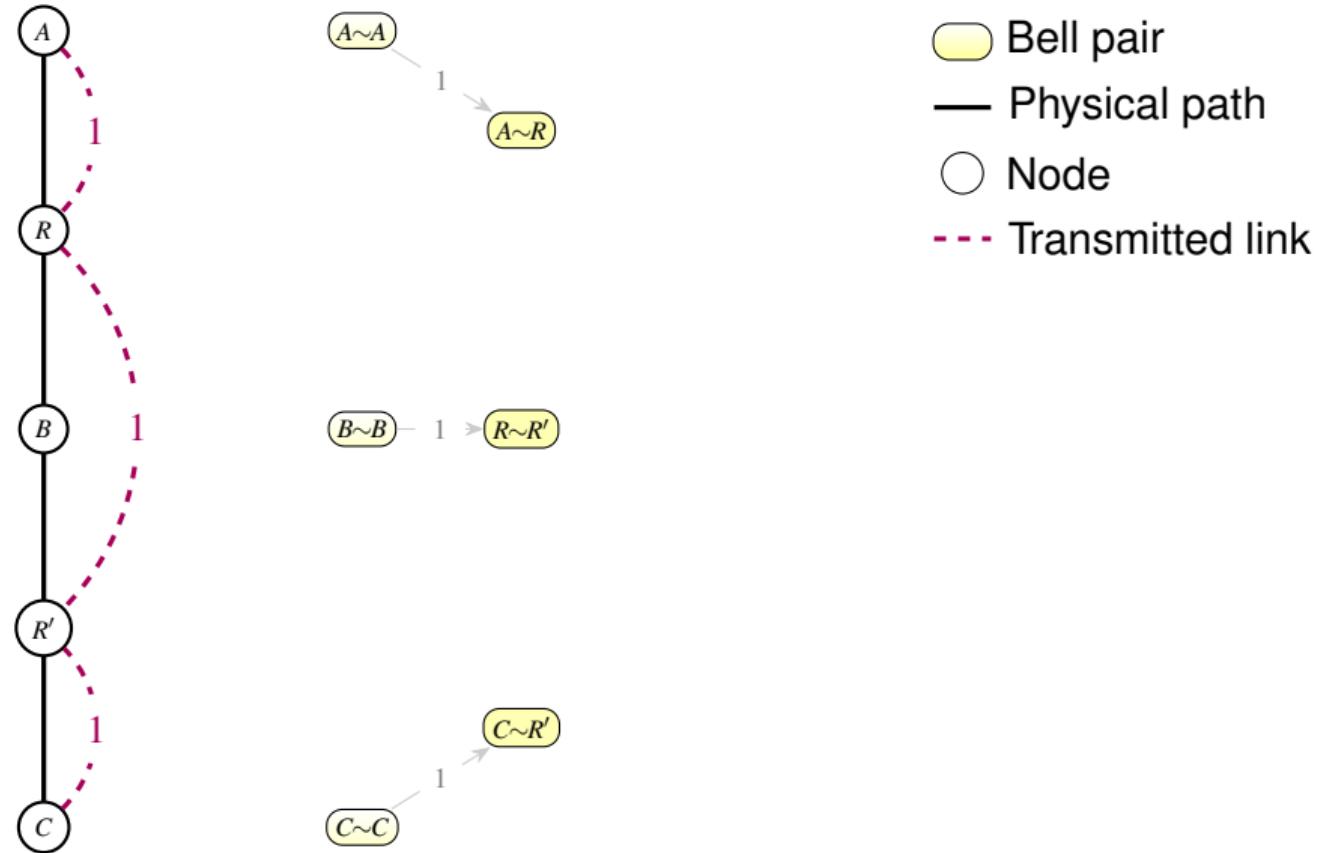
## Bell pair generation: Protocol II



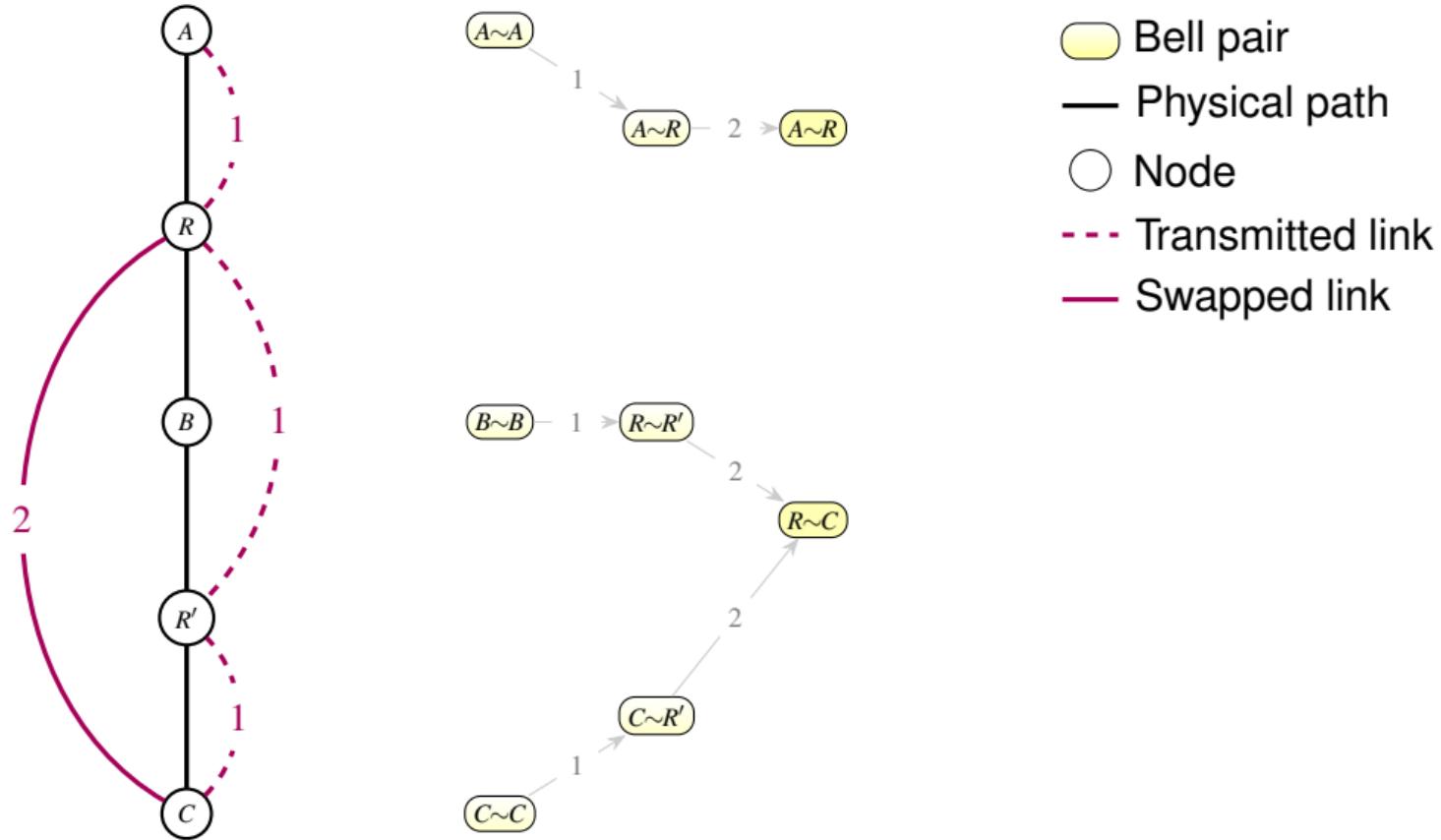
## Bell pair generation: Protocol II



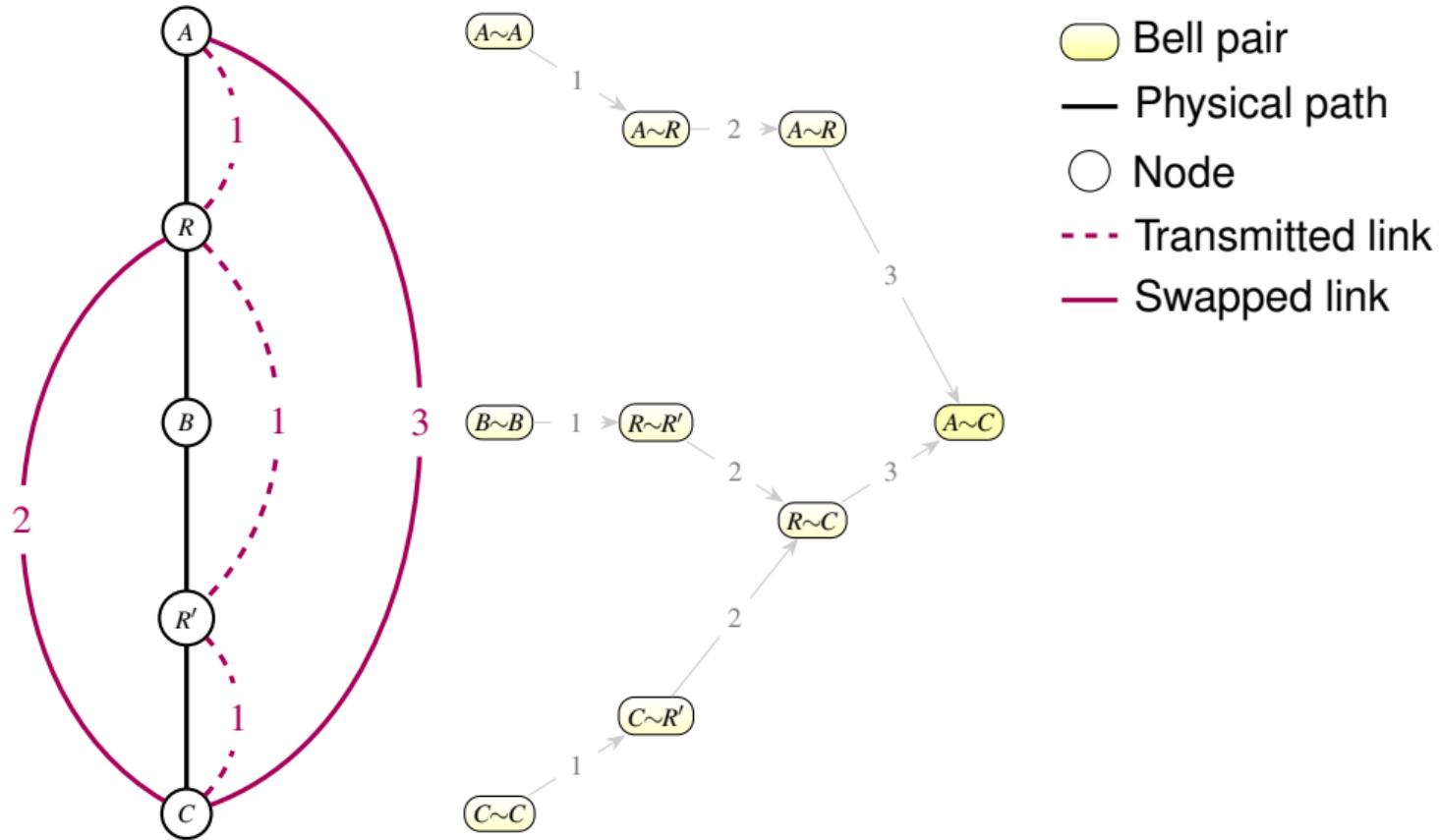
## Bell pair generation: Protocol II, round 1



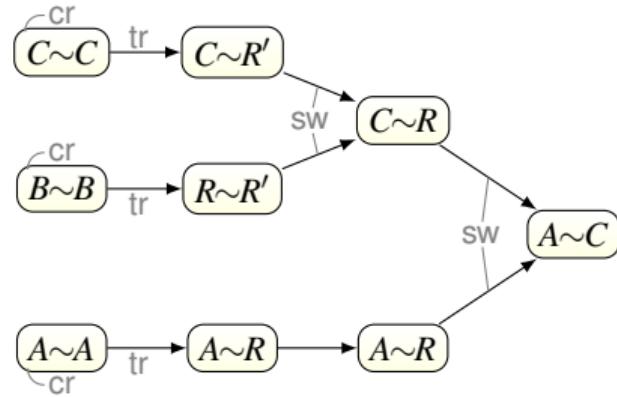
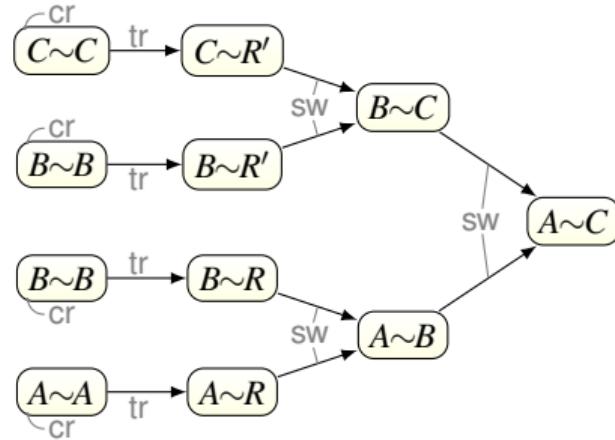
## Bell pair generation: Protocol II, round 2



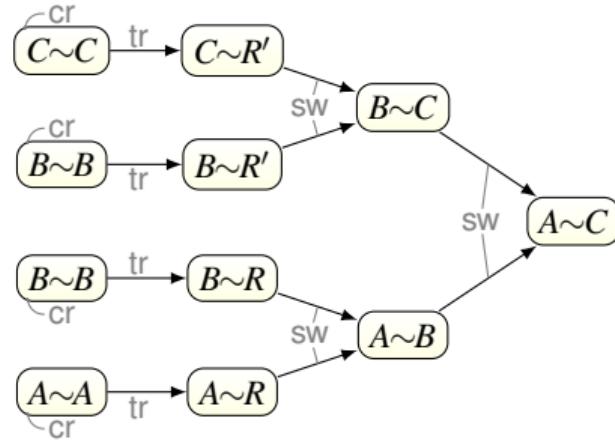
## Bell pair generation: Protocol II, round 3



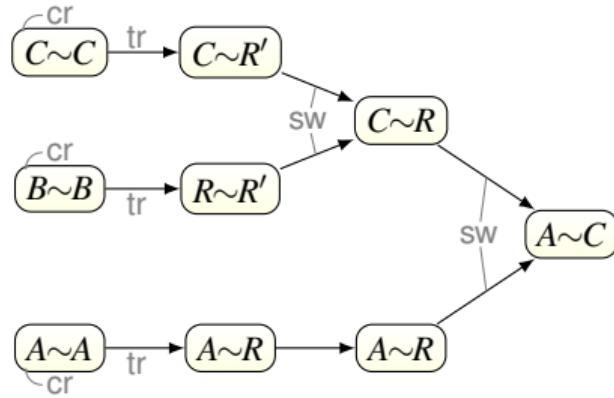
# Specification: Protocol I and Protocol II generating $A \sim C$



# Specification: Protocol I and Protocol II generating $A \sim C$



$(\text{cr}\langle A \rangle \parallel \text{cr}\langle B \rangle \parallel \text{cr}\langle B \rangle \parallel \text{cr}\langle C \rangle);$   
 $(\text{tr}\langle A \rightarrow A \sim R \rangle \parallel \text{tr}\langle B \rightarrow B \sim R \rangle \parallel \text{tr}\langle B \rightarrow B \sim R' \rangle \parallel \text{tr}\langle C \rightarrow C \sim R' \rangle);$   
 $(\text{sw}\langle A \sim B @ R \rangle \parallel \text{sw}\langle B \sim C @ R' \rangle);$   
 $\text{sw}\langle A \sim C @ B \rangle$



$(\text{cr}\langle A \rangle \parallel \text{cr}\langle B \rangle \parallel \text{cr}\langle C \rangle);$   
 $(\text{tr}\langle A \rightarrow A \sim R \rangle \parallel \text{tr}\langle B \rightarrow R \sim R' \rangle \parallel \text{tr}\langle C \rightarrow C \sim R' \rangle);$   
 $\text{sw}\langle C \sim R @ R' \rangle;$   
 $\text{sw}\langle A \sim C @ R \rangle$

## Basic action $r \triangleright o$

$$r \triangleright o: \mathcal{M}(\text{BP}) \rightarrow \mathcal{M}(\text{BP}) \times \mathcal{M}(\text{BP})$$

$$a \mapsto \begin{cases} \textcolor{green}{o} \bowtie \textcolor{blue}{a \setminus r} & \text{if } \textcolor{red}{r} \subseteq a \\ \emptyset \bowtie \textcolor{blue}{a} & \text{otherwise} \end{cases}$$

---

$\bowtie$ : pair of multisets of Bell pairs

## Basic action $r \triangleright o$

required  
Bell pairs

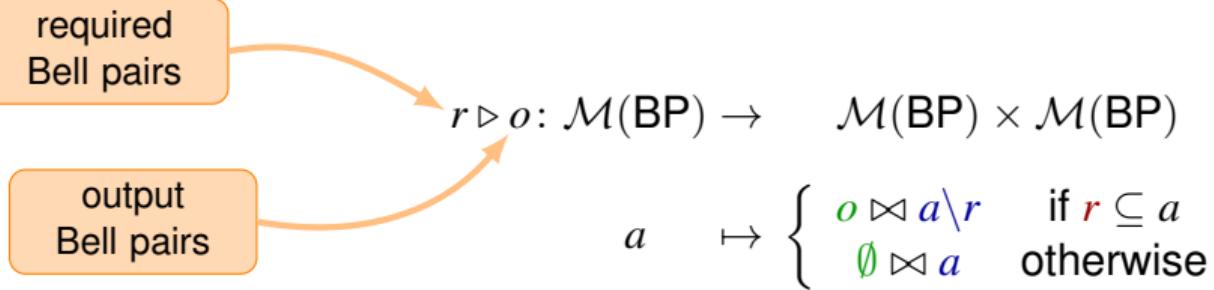
$$r \triangleright o: \mathcal{M}(\text{BP}) \rightarrow \mathcal{M}(\text{BP}) \times \mathcal{M}(\text{BP})$$

$$a \mapsto \begin{cases} o \bowtie a \setminus r & \text{if } r \subseteq a \\ \emptyset \bowtie a & \text{otherwise} \end{cases}$$

---

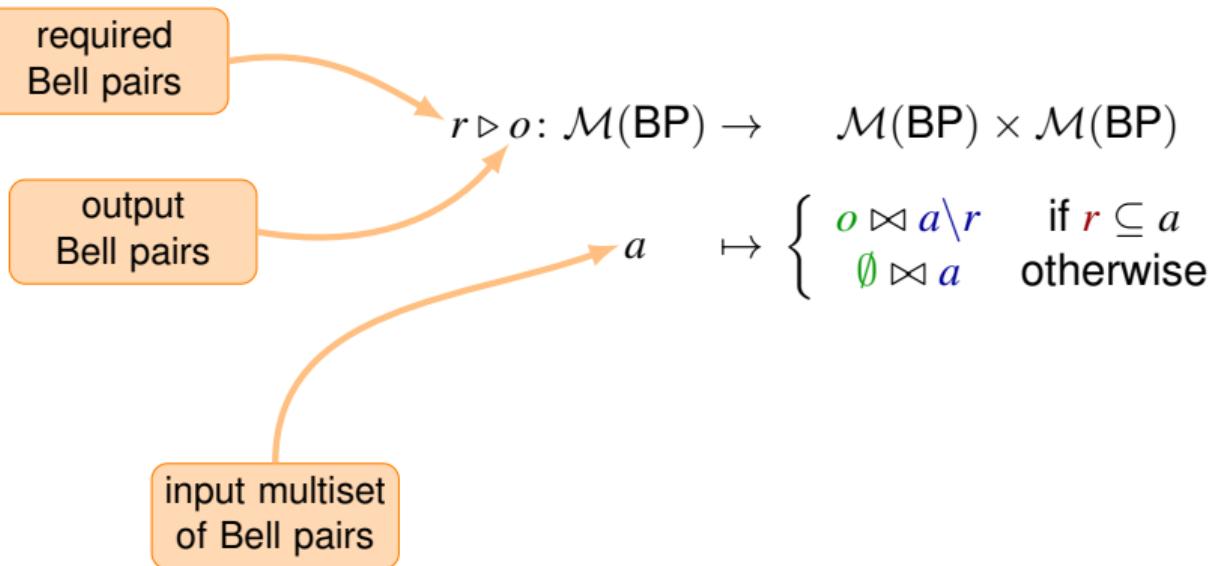
$\bowtie$ : pair of multisets of Bell pairs

## Basic action $r \triangleright o$



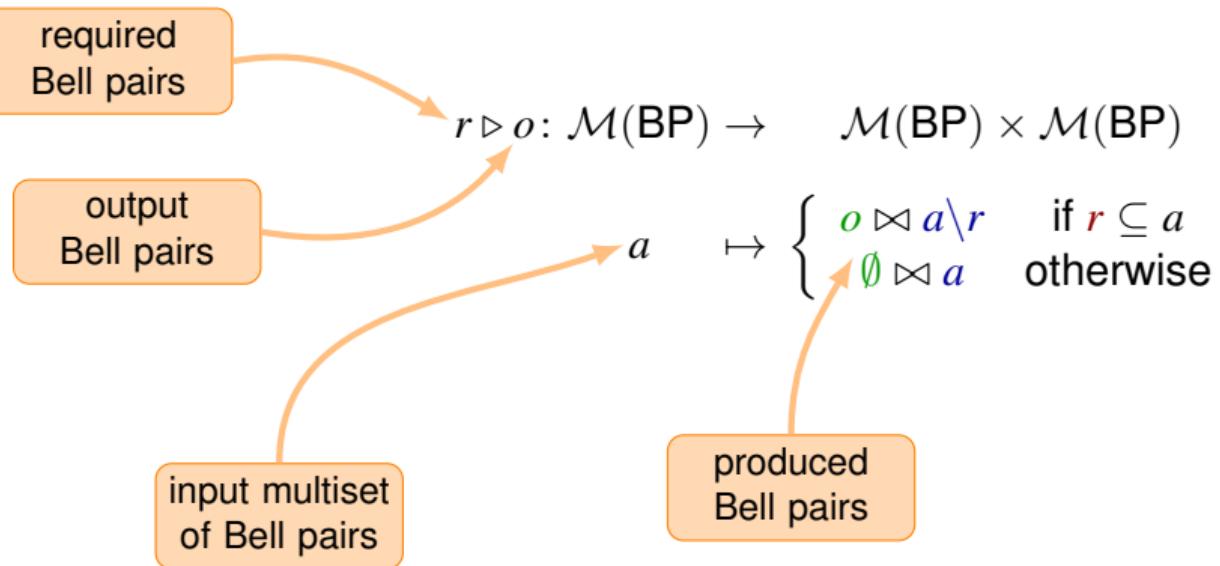
$\bowtie$ : pair of multisets of Bell pairs

## Basic action $r \triangleright o$



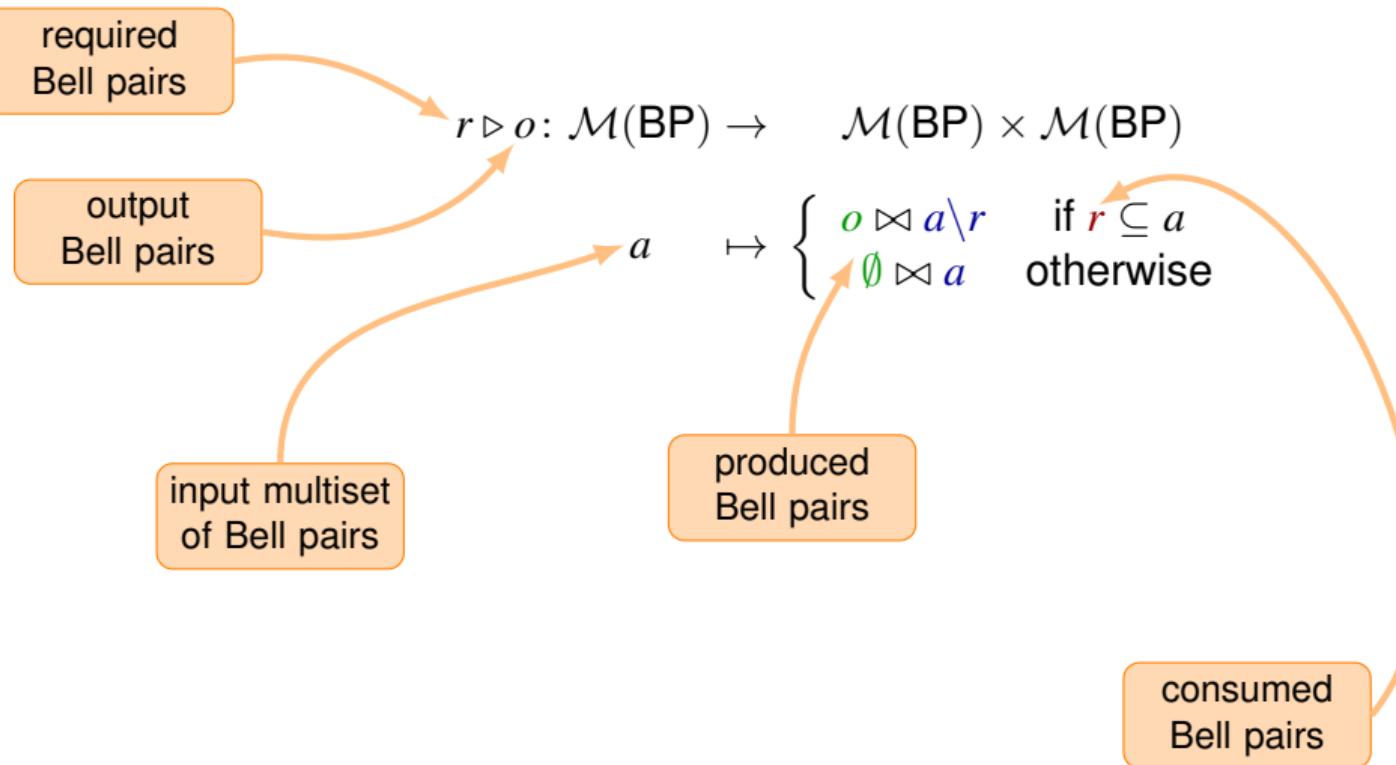
$\bowtie$ : pair of multisets of Bell pairs

## Basic action $r \triangleright o$



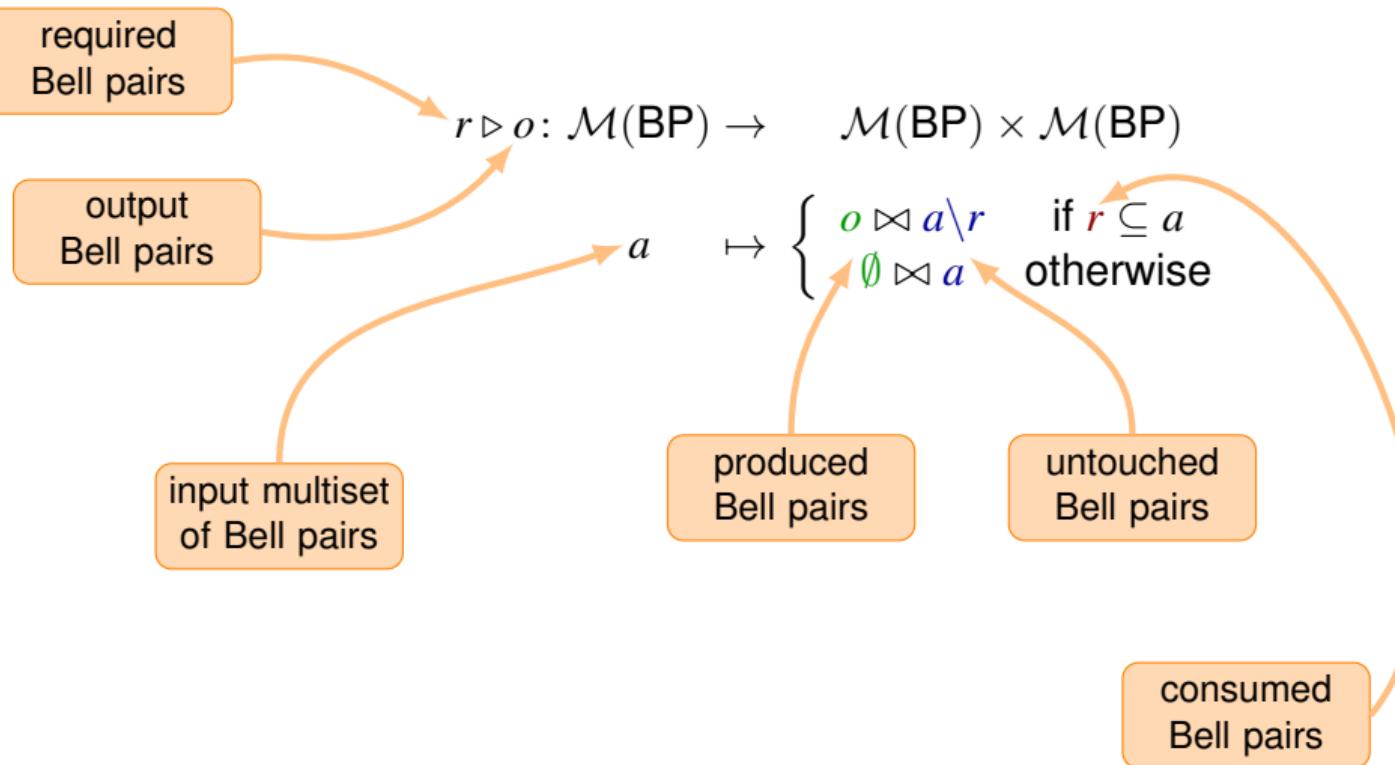
$\bowtie$ : pair of multisets of Bell pairs

## Basic action $r \triangleright o$



$\bowtie$ : pair of multisets of Bell pairs

## Basic action $r \triangleright o$



$\bowtie$ : pair of multisets of Bell pairs

# Basic action

$$r \triangleright o : \mathcal{M}(\text{BP}) \rightarrow \mathcal{M}(\text{BP}) \times \mathcal{M}(\text{BP})$$
$$a \quad \mapsto \begin{cases} o \bowtie a \setminus r & \text{if } r \subseteq a \\ \emptyset \bowtie a & \text{otherwise} \end{cases}$$

## Swap action $\text{sw}\langle A \sim B @ R \rangle$

$\{\!\{A \sim R, B \sim R\}\!\} \triangleright \{\!\{A \sim B\}\!\}$  acting on  $a = \{\!\{A \sim R, A \sim R, B \sim R, B \sim R, A \sim B\}\!\}$

$A \sim R$      $B \sim R$      $A \sim R$      $B \sim R$      $A \sim B$

---

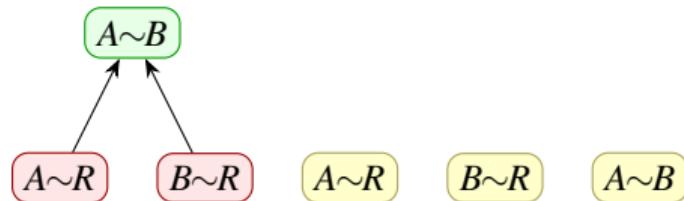
### Input Bell pairs

# Basic action

$$r \triangleright o : \mathcal{M}(\text{BP}) \rightarrow \mathcal{M}(\text{BP}) \times \mathcal{M}(\text{BP})$$
$$a \mapsto \begin{cases} o \bowtie a \setminus r & \text{if } r \subseteq a \\ \emptyset \bowtie a & \text{otherwise} \end{cases}$$

## Swap action $\text{sw}\langle A \sim B @ R \rangle$

$\{\!A\!\sim\!R, B\!\sim\!R\}\! \triangleright \{\!A\!\sim\!B\}$  acting on  $a = \{A\!\sim\!R, A\!\sim\!R, B\!\sim\!R, B\!\sim\!R, A\!\sim\!B\}$



Bell pairs: input and consumed, produced

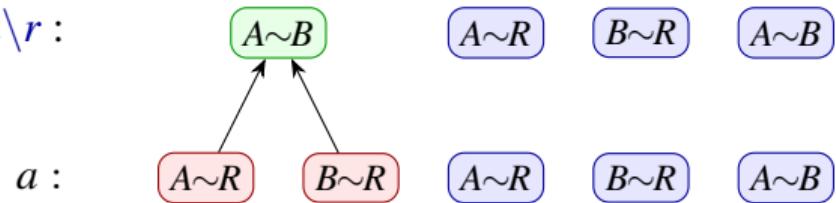
# Basic action

$$r \triangleright o : \mathcal{M}(\text{BP}) \rightarrow \mathcal{M}(\text{BP}) \times \mathcal{M}(\text{BP})$$
$$a \quad \mapsto \begin{cases} o \bowtie a \setminus r & \text{if } r \subseteq a \\ \emptyset \bowtie a & \text{otherwise} \end{cases}$$

## Swap action $\text{sw}\langle A \sim B @ R \rangle$

$\{\{A \sim R, B \sim R\}\} \triangleright \{\{A \sim B\}\}$  acting on  $a = \{\{A \sim R, A \sim R, B \sim R, B \sim R, A \sim B\}\}$

$o \bowtie a \setminus r :$



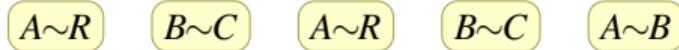
Bell pairs: consumed, produced and untouched

## Basic action

$$r \triangleright o : \mathcal{M}(\text{BP}) \rightarrow \mathcal{M}(\text{BP}) \times \mathcal{M}(\text{BP})$$
$$a \quad \mapsto \begin{cases} o \bowtie a \setminus r & \text{if } r \subseteq a \\ \emptyset \bowtie a & \text{otherwise} \end{cases}$$

### Swap action $\text{sw}\langle A \sim B @ R \rangle$

$\{\!\{A \sim R, B \sim R\}\!\} \triangleright \{\!\{A \sim B\}\!\}$  trying to act on  $a = \{\!\{A \sim R, A \sim R, B \sim C, B \sim C, A \sim B\}\!\}$



---

### Input Bell pairs

# Basic action

$$r \triangleright o : \mathcal{M}(\text{BP}) \rightarrow \mathcal{M}(\text{BP}) \times \mathcal{M}(\text{BP})$$
$$a \quad \mapsto \begin{cases} o \bowtie a \setminus r & \text{if } r \subseteq a \\ \emptyset \bowtie a & \text{otherwise} \end{cases}$$

## Swap action $\text{sw}\langle A \sim B @ R \rangle$

$\{\!\{A \sim R, B \sim R\}\!\} \triangleright \{\!\{A \sim B\}\!\}$  trying to act on  $a = \{A \sim R, A \sim R, B \sim C, B \sim C, A \sim B\}$



$A \sim R$     $B \sim C$     $A \sim R$     $B \sim C$     $A \sim B$

## Input Bell pairs

## Basic action

$$r \triangleright o : \mathcal{M}(\text{BP}) \rightarrow \mathcal{M}(\text{BP}) \times \mathcal{M}(\text{BP})$$
$$a \quad \mapsto \begin{cases} o \bowtie a \setminus r & \text{if } r \subseteq a \\ \emptyset \bowtie a & \text{otherwise} \end{cases}$$

### Swap action $\text{sw}\langle A \sim B @ R \rangle$

$\{\!\{A \sim R, B \sim R\}\!} \triangleright \{\!\{A \sim B\}\!$  trying to act on  $a = \{\!\{A \sim R, A \sim R, B \sim C, B \sim C, A \sim B\}\!$

$$\emptyset \bowtie a : \quad \boxed{A \sim R} \quad \boxed{B \sim C} \quad \boxed{A \sim R} \quad \boxed{B \sim C} \quad \boxed{A \sim B}$$

$$a : \quad \boxed{A \sim R} \quad \boxed{B \sim C} \quad \boxed{A \sim R} \quad \boxed{B \sim C} \quad \boxed{A \sim B}$$

Bell pairs: consumed, produced and untouched

## Basic actions

swap	$\text{sw}\langle A \sim B @ C \rangle \triangleq \{\{A \sim C, B \sim C\} \triangleright \{A \sim B\}$
transmit	$\text{tr}\langle A \rightarrow B \sim C \rangle \triangleq \{\{A \sim A\} \triangleright \{B \sim C\}$
create	$\text{cr}\langle A \rangle \triangleq \emptyset \triangleright \{A \sim A\}$
wait	$\text{wait}\langle r \rangle \triangleq r \triangleright r$
fail	$\text{fail}\langle r \rangle \triangleq r \triangleright \emptyset$

## Basic actions

swap	$\text{sw}\langle A \sim B @ C \rangle \triangleq \{\{A \sim C, B \sim C\} \triangleright \{A \sim B\}$
transmit	$\text{tr}\langle A \rightarrow B \sim C \rangle \triangleq \{\{A \sim A\} \triangleright \{B \sim C\}$
create	$\text{cr}\langle A \rangle \triangleq \emptyset \triangleright \{A \sim A\}$
wait	$\text{wait}\langle r \rangle \triangleq r \triangleright r$
fail	$\text{fail}\langle r \rangle \triangleq r \triangleright \emptyset$

## Probabilistic basic actions

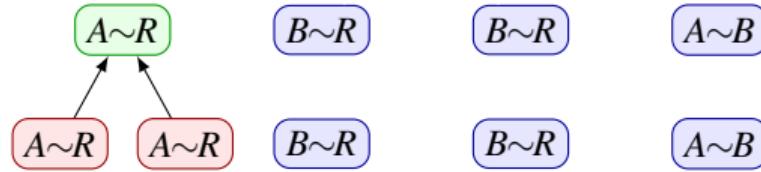
$$r \triangleright o + r \triangleright \emptyset \triangleq r \triangleright o + \text{fail}\langle r \rangle$$

# Probabilistic basic actions

Example (Distillation is inherently probabilistic.)

$$\text{di}\langle A \sim R \rangle \triangleq \{\{A \sim R, A \sim R\} \triangleright \{A \sim R\} + \{A \sim R, A \sim R\} \triangleright \emptyset\}$$

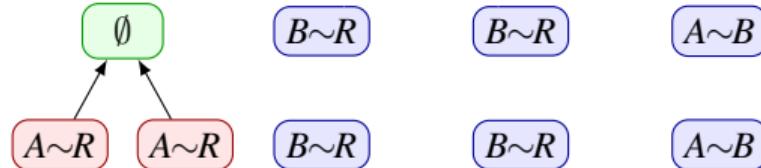
succeed :



input  $a$  :

fail :

input  $a$  :

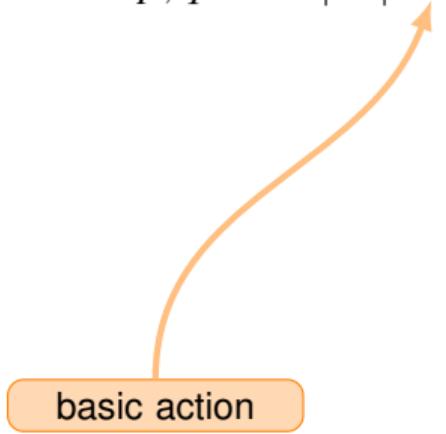


Bell pairs: **consumed**, **produced** and **untouched**

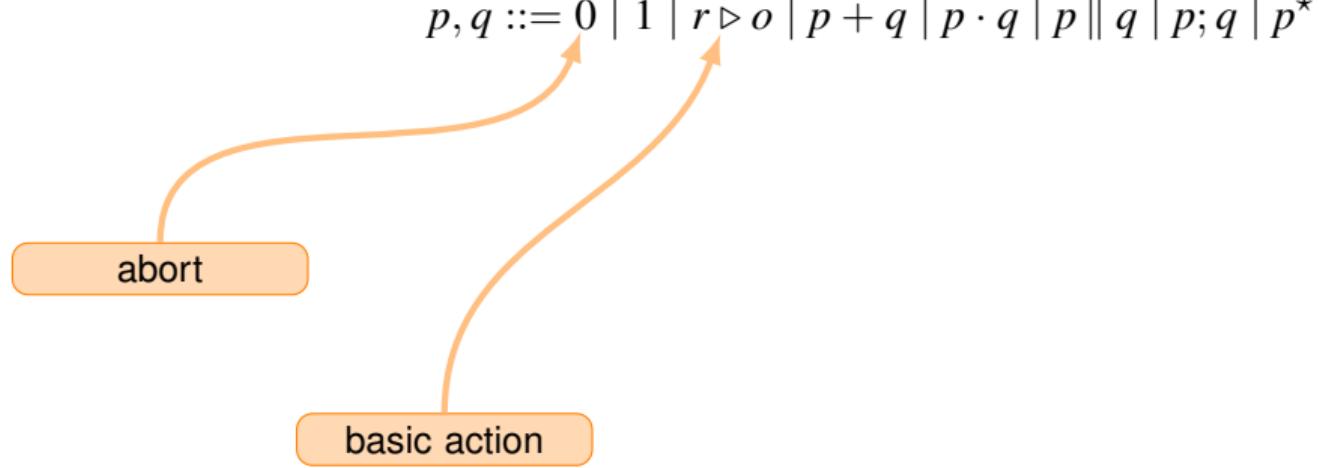
# Protocols

$$p, q ::= 0 \mid 1 \mid r \triangleright o \mid p + q \mid p \cdot q \mid p \parallel q \mid p; q \mid p^*$$

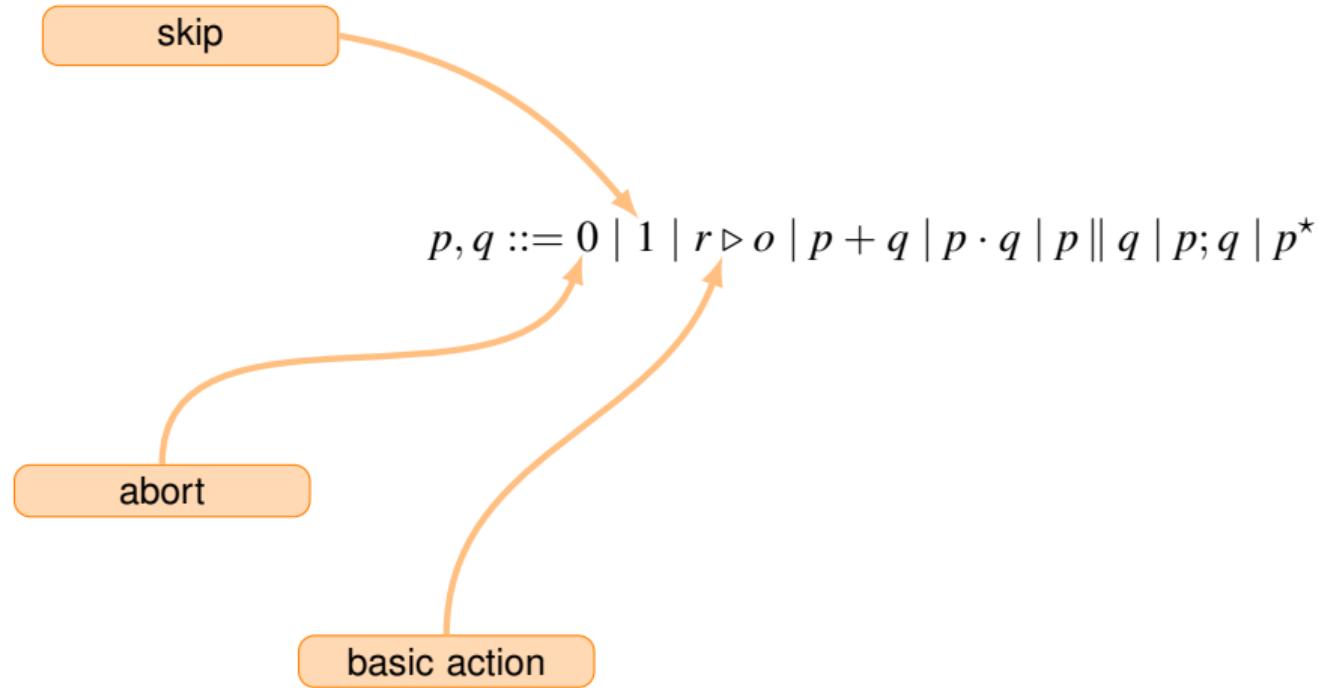
# Protocols

$$p, q ::= 0 \mid 1 \mid r \triangleright o \mid p + q \mid p \cdot q \mid p \parallel q \mid p; q \mid p^*$$


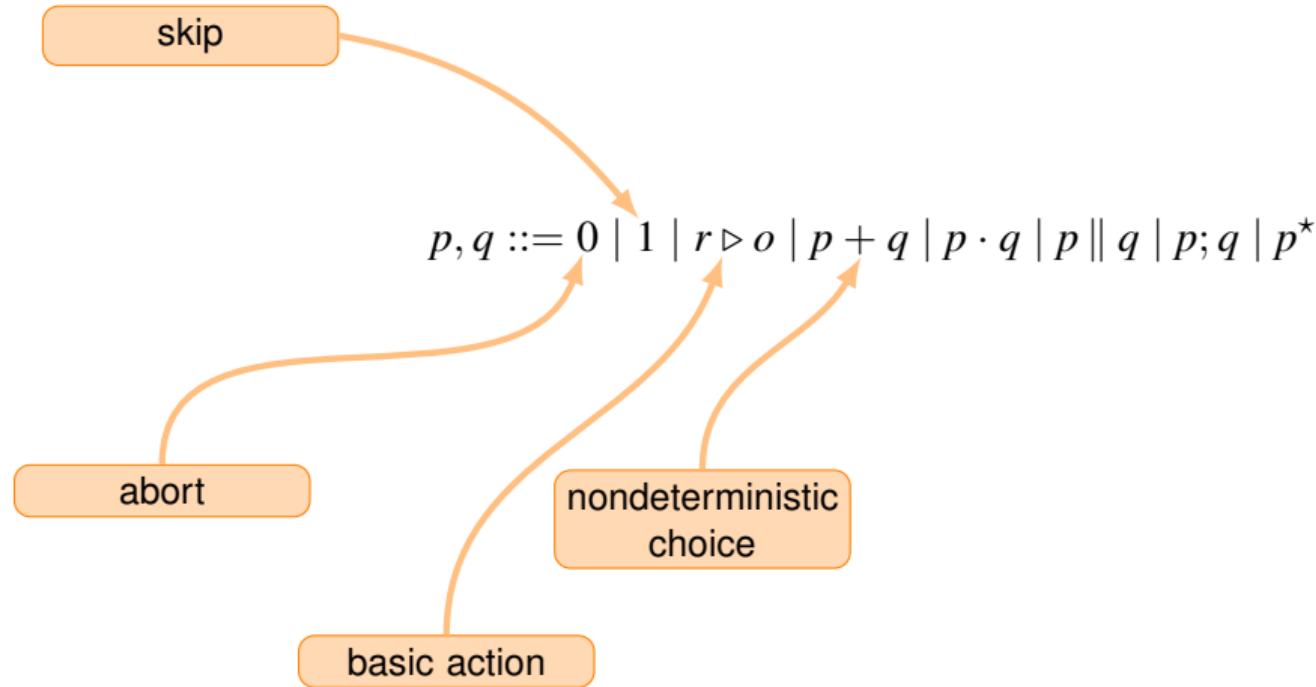
# Protocols

$$p, q ::= 0 \mid 1 \mid r \triangleright o \mid p + q \mid p \cdot q \mid p \parallel q \mid p; q \mid p^*$$


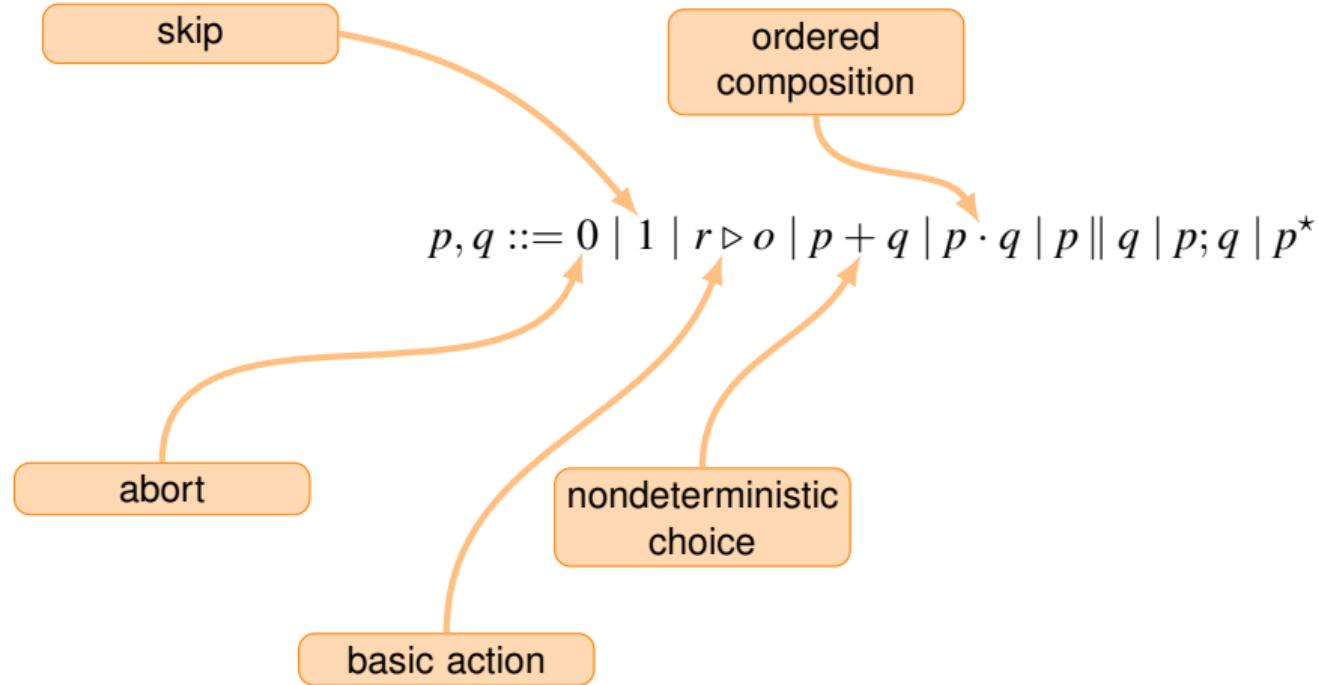
# Protocols



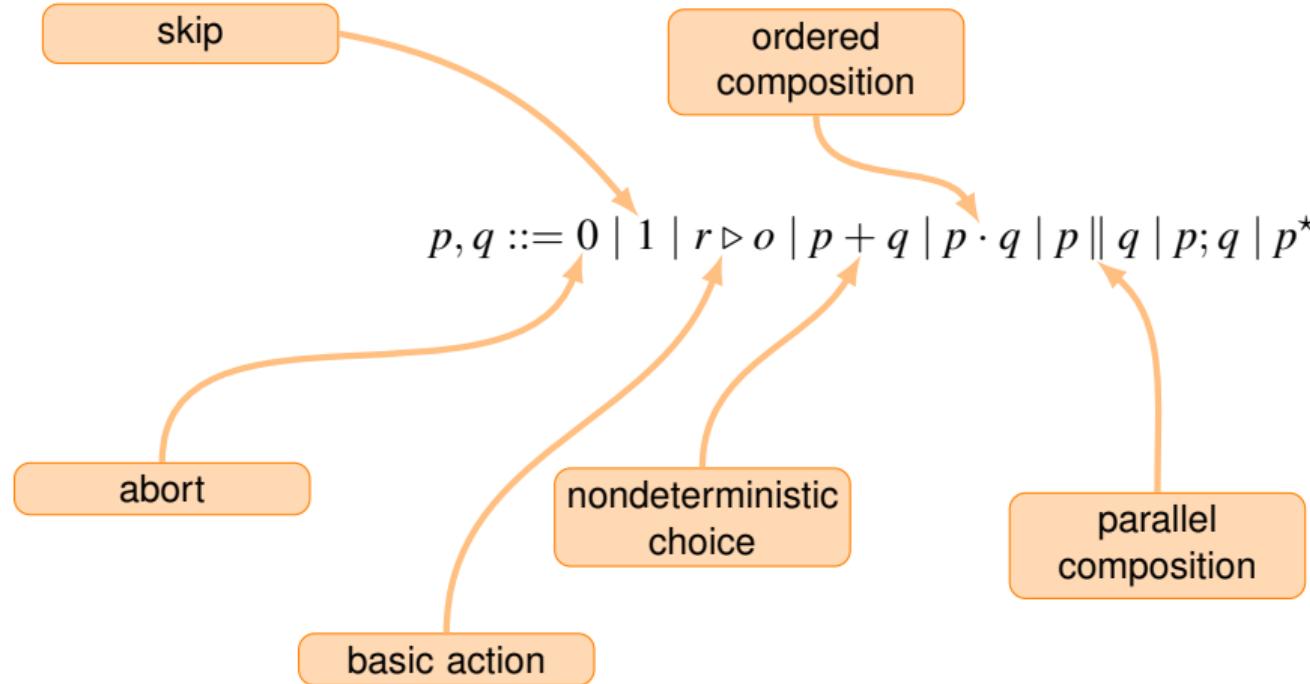
# Protocols



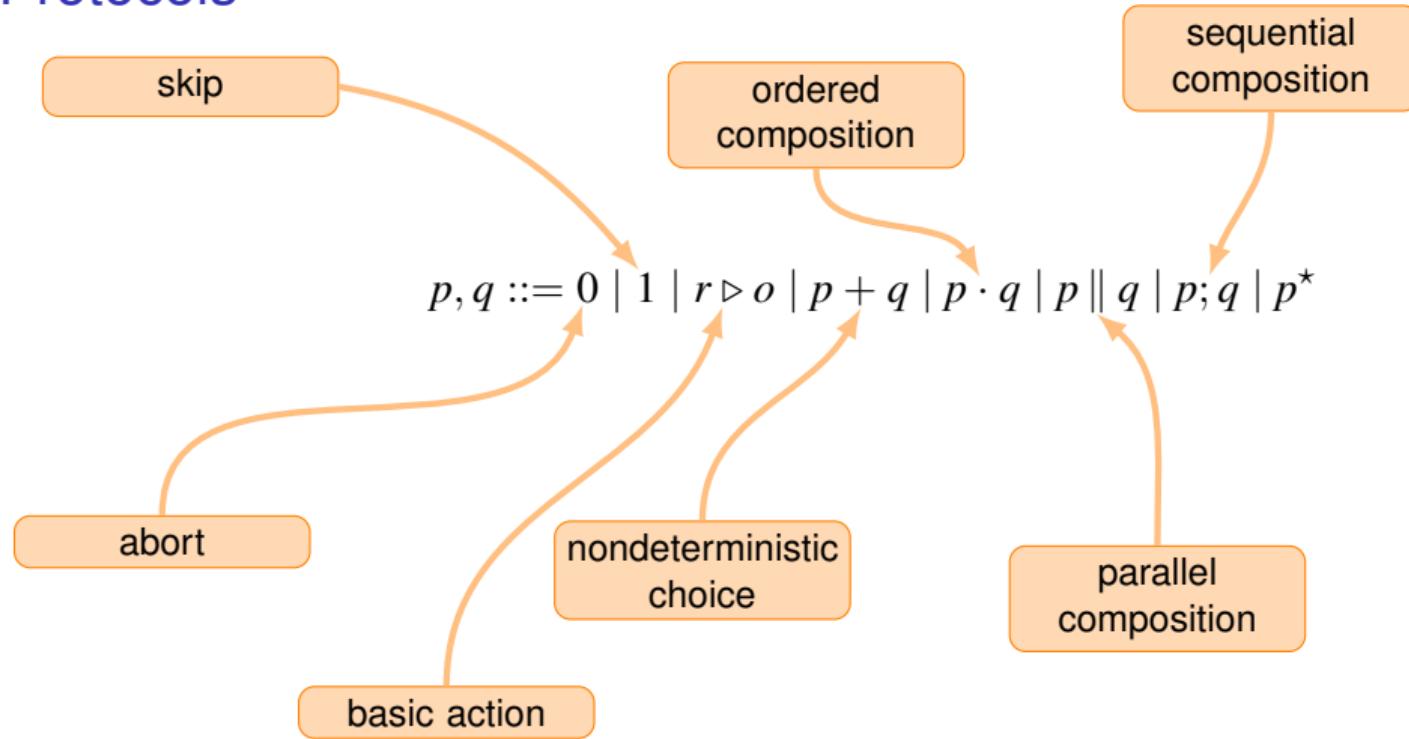
# Protocols



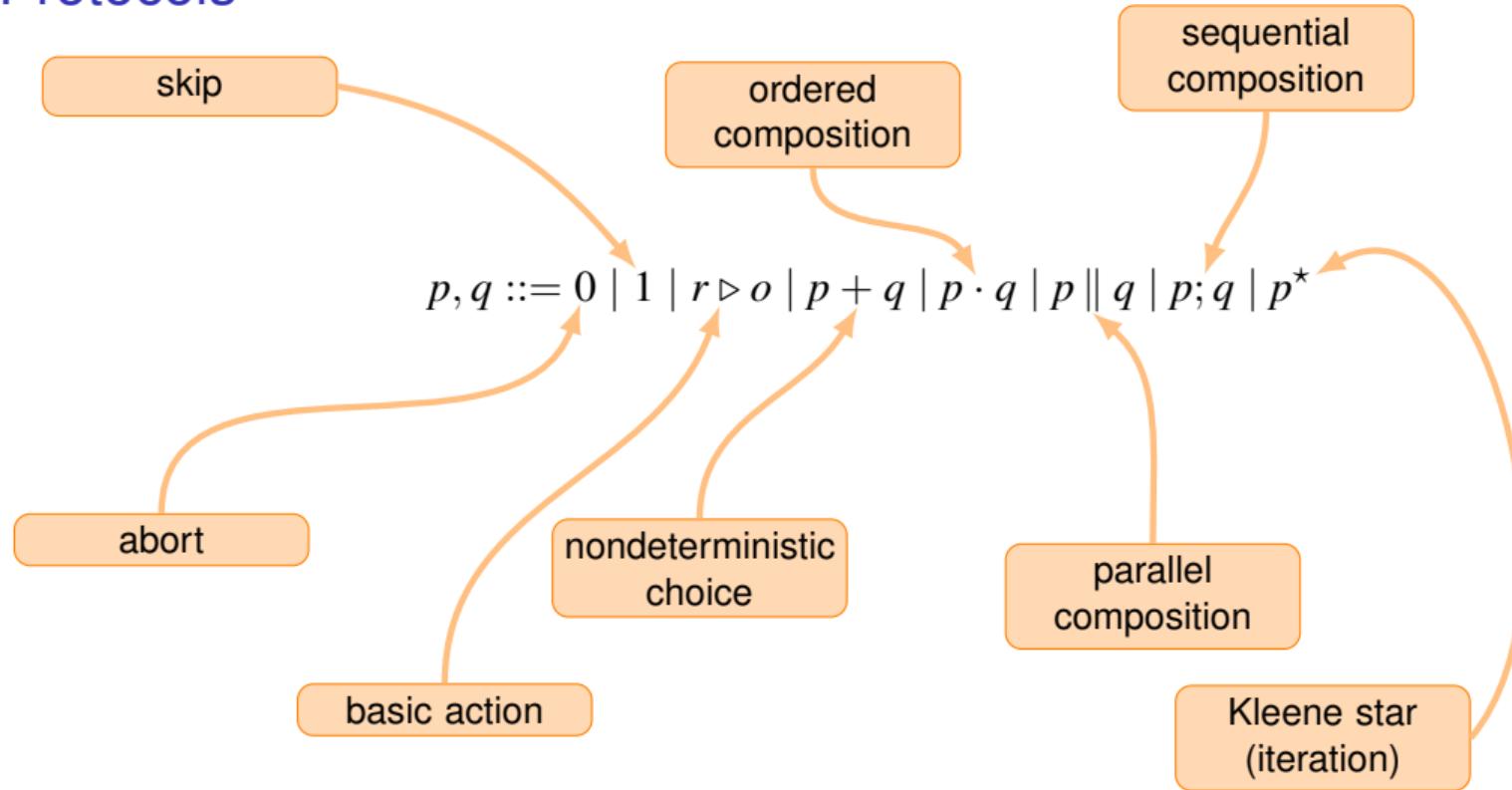
# Protocols



# Protocols



# Protocols



# Single round protocols

## Parallel composition

$\text{sw}\langle A \sim B @ R \rangle \parallel \text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \parallel \text{sw}\langle B \sim C @ R' \rangle$  acts on  $\{A \sim R, B \sim R, B \sim R, B \sim R', C \sim R', A \sim B\}$

$A \sim R$      $B \sim R$      $B \sim R$      $B \sim R'$      $C \sim R'$      $A \sim B$

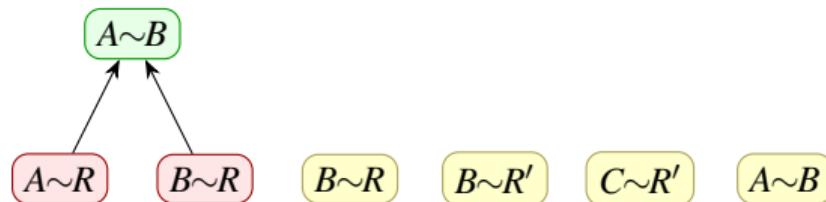
---

Input Bell pairs

# Single round protocols

## Parallel composition

$\text{sw}\langle A \sim B @ R \rangle \| \text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \| \text{sw}\langle B \sim C @ R' \rangle$  acts on  $\{\underline{A \sim R}, \underline{B \sim R}, B \sim R, B \sim R', C \sim R', A \sim B\}$

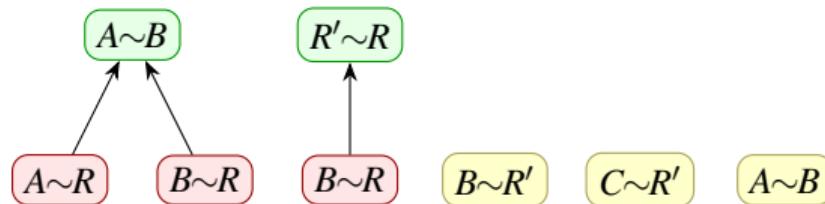


Bell pairs: **input** and **consumed**, **produced**

# Single round protocols

## Parallel composition

$\text{sw}\langle A \sim B @ R \rangle \parallel \text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \parallel \text{sw}\langle B \sim C @ R' \rangle$  acts on  $\{A \sim R, B \sim R, \underline{B \sim R}, B \sim R', C \sim R', A \sim B\}$

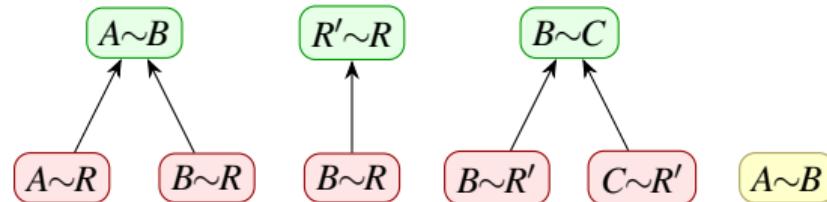


Bell pairs: **input** and **consumed**, **produced**

# Single round protocols

## Parallel composition

$\text{sw}\langle A \sim B @ R \rangle \parallel \text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \parallel \text{sw}\langle B \sim C @ R' \rangle$  acts on  $\{A \sim R, B \sim R, B \sim R, \underline{B \sim R'}, C \sim R', A \sim B\}$

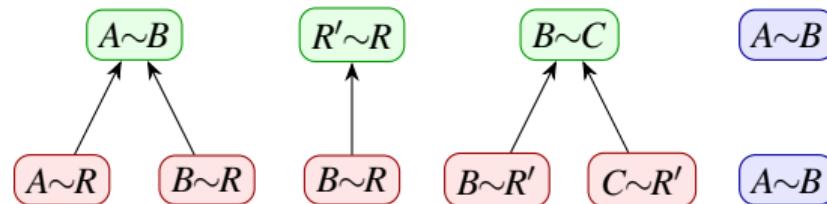


Bell pairs: input and consumed, produced

# Single round protocols

## Parallel composition

$\text{sw}\langle A \sim B @ R \rangle \parallel \text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \parallel \text{sw}\langle B \sim C @ R' \rangle$  acts on  $\{A \sim R, B \sim R, B \sim R, B \sim R', C \sim R', A \sim B\}$

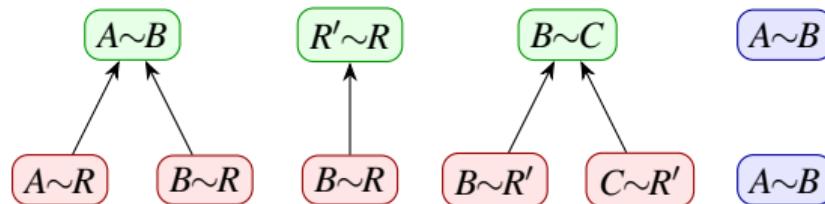


Bell pairs: **consumed**, **produced** and **untouched**

# Single round protocols

## Parallel composition

$\text{sw}\langle A \sim B @ R \rangle \parallel \text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \parallel \text{sw}\langle B \sim C @ R' \rangle$  acts on  $\{A \sim R, B \sim R, B \sim R, B \sim R', C \sim R', A \sim B\}$



The order of basic actions is independent for this input multiset, thus:

$$\text{sw}\langle A \sim B @ R \rangle \cdot \text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \cdot \text{sw}\langle B \sim C @ R' \rangle = \text{sw}\langle A \sim B @ R \rangle \parallel \text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \parallel \text{sw}\langle B \sim C @ R' \rangle$$

---

Bell pairs: **consumed**, **produced** and **untouched**

# Parallel composition vs. ordered composition

$\mathbf{sw}\langle A \sim B @ R \rangle \| \mathbf{tr}\langle B \sim R \rightarrow R' \sim R \rangle \| \mathbf{sw}\langle B \sim C @ R' \rangle$

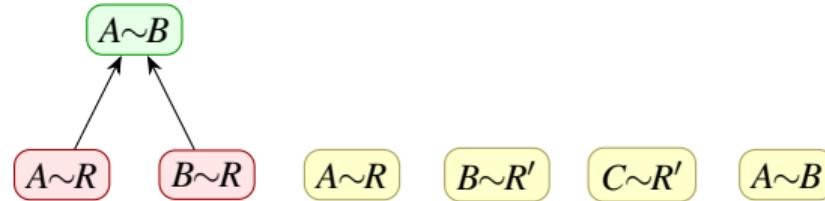
$A \sim R$      $B \sim R$      $A \sim R$      $B \sim R'$      $C \sim R'$      $A \sim B$

---

Bell pairs: **input** and **consumed**, **produced**

# Parallel composition vs. ordered composition

$\text{sw} \langle A \sim B @ R \rangle \cdot \text{tr} \langle B \sim R \rightarrow R' \sim R \rangle \cdot \text{sw} \langle B \sim C @ R' \rangle$

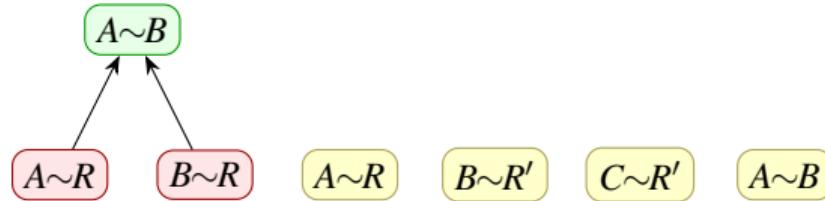


---

Bell pairs: **input** and **consumed**, **produced**

# Parallel composition vs. ordered composition

$\text{sw}\langle A \sim B @ R \rangle \cdot \text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \cdot \text{sw}\langle B \sim C @ R' \rangle$

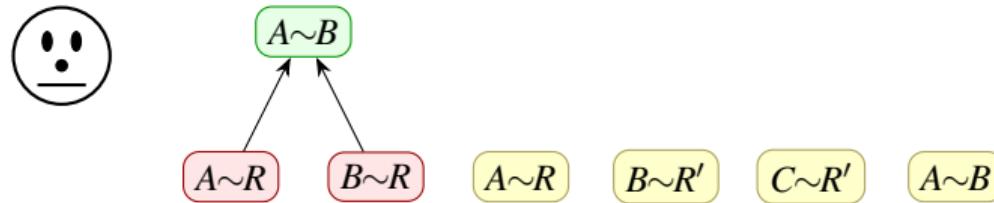


---

Bell pairs: **input** and **consumed**, **produced**

# Parallel composition vs. ordered composition

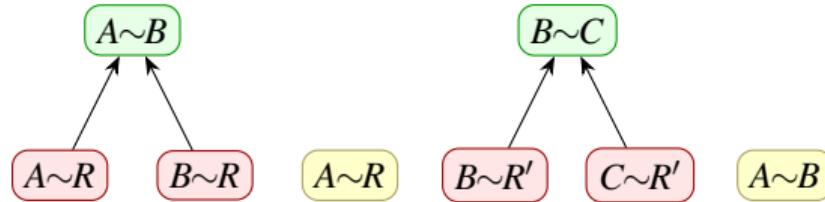
$$\text{sw} \langle A \sim B @ R \rangle \cdot \text{tr} \langle B \sim R \rightarrow R' \sim R \rangle \cdot \text{sw} \langle B \sim C @ R' \rangle$$



Bell pairs: input and consumed, produced

# Parallel composition vs. ordered composition

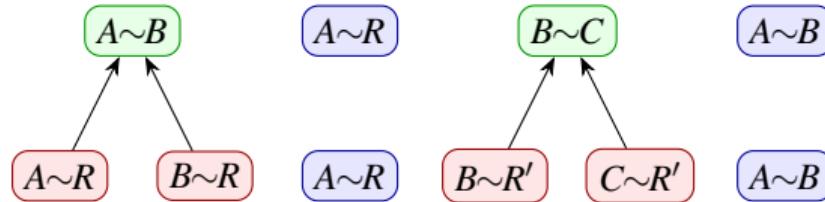
$$\text{sw} \langle A \sim B @ R \rangle \cdot \text{tr} \langle B \sim R \rightarrow R' \sim R \rangle \cdot \text{sw} \langle B \sim C @ R' \rangle$$



<sup>2</sup>Bell pairs: input and consumed, produced.

# Parallel composition vs. ordered composition

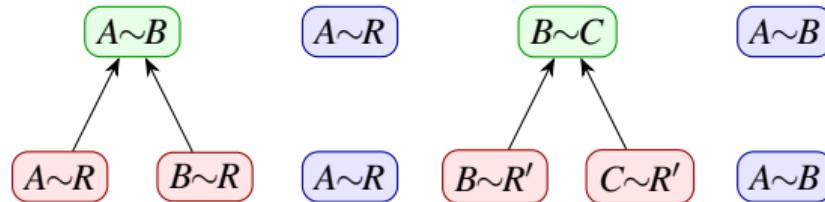
$$\text{sw}\langle A \sim B @ R \rangle \cdot \text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \cdot \text{sw}\langle B \sim C @ R' \rangle$$



Bell pairs: consumed, produced and untouched

# Parallel composition vs. ordered composition

$\text{sw}\langle A \sim B @ R \rangle \cdot \text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \cdot \text{sw}\langle B \sim C @ R' \rangle$



$\text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \cdot \text{sw}\langle A \sim B @ R \rangle \cdot \text{sw}\langle B \sim C @ R' \rangle$

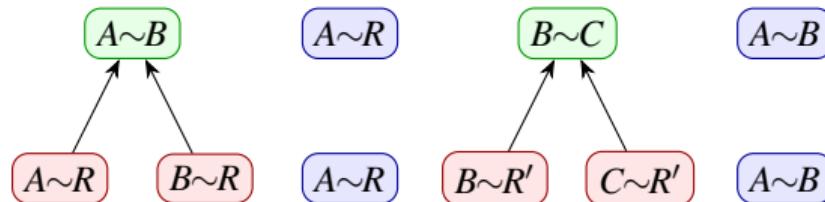


---

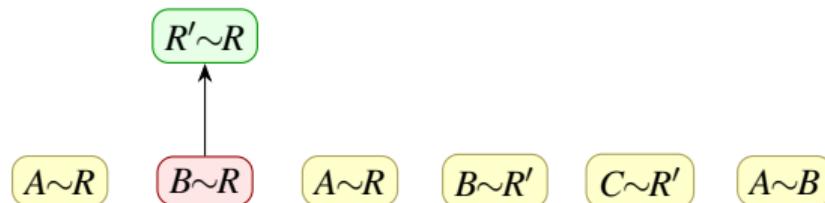
Bell pairs: **consumed**, **produced** and **untouched**

# Parallel composition vs. ordered composition

$$\text{sw}\langle A \sim B @ R \rangle \cdot \text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \cdot \text{sw}\langle B \sim C @ R' \rangle$$



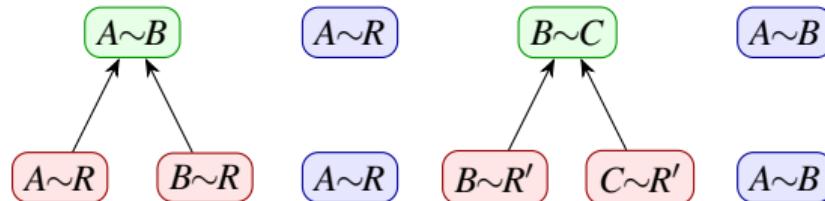
$$\text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \cdot \text{sw}\langle A \sim B @ R \rangle \cdot \text{sw}\langle B \sim C @ R' \rangle$$



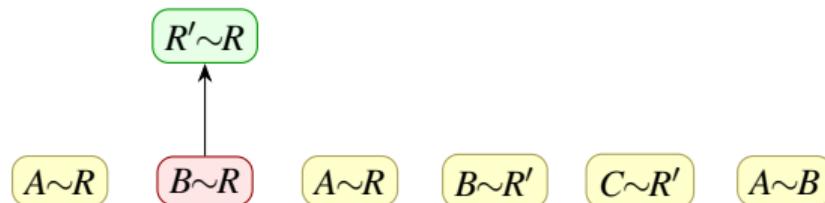
Bell pairs: **consumed**, **produced** and **untouched**

# Parallel composition vs. ordered composition

$\text{sw}\langle A \sim B @ R \rangle \cdot \text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \cdot \text{sw}\langle B \sim C @ R' \rangle$



$\text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \cdot \text{sw}\langle A \sim B @ R \rangle \cdot \text{sw}\langle B \sim C @ R' \rangle$

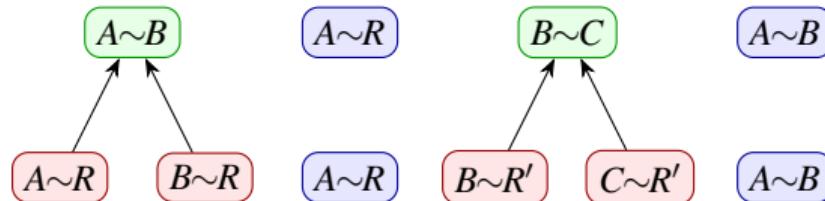


---

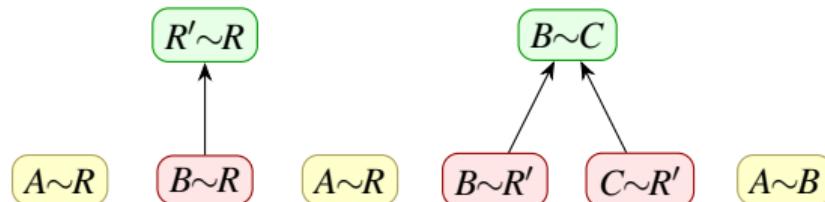
Bell pairs: **consumed**, **produced** and **untouched**

# Parallel composition vs. ordered composition

$$\text{sw}\langle A \sim B @ R \rangle \cdot \text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \cdot \text{sw}\langle B \sim C @ R' \rangle$$



$$\text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \cdot \text{sw}\langle A \sim B @ R \rangle \cdot \text{sw}\langle B \sim C @ R' \rangle$$

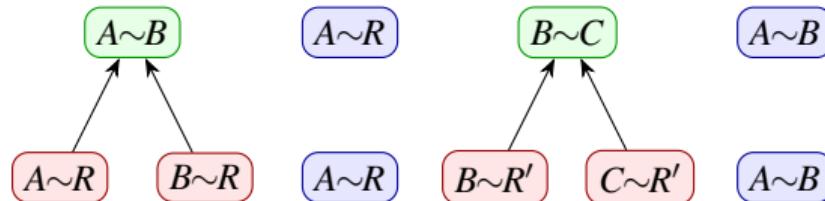


---

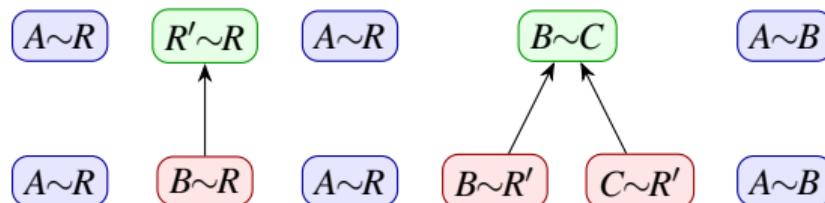
Bell pairs: **consumed**, **produced** and **untouched**

# Parallel composition vs. ordered composition

$$\text{sw}\langle A \sim B @ R \rangle \cdot \text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \cdot \text{sw}\langle B \sim C @ R' \rangle$$



$$\text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \cdot \text{sw}\langle A \sim B @ R \rangle \cdot \text{sw}\langle B \sim C @ R' \rangle$$

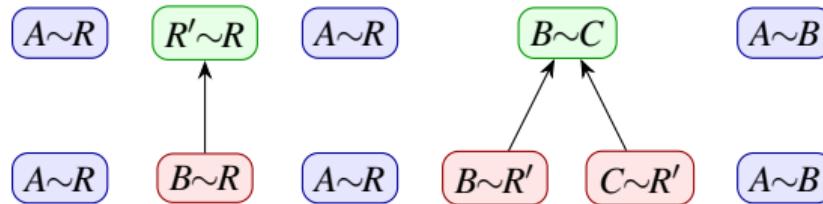
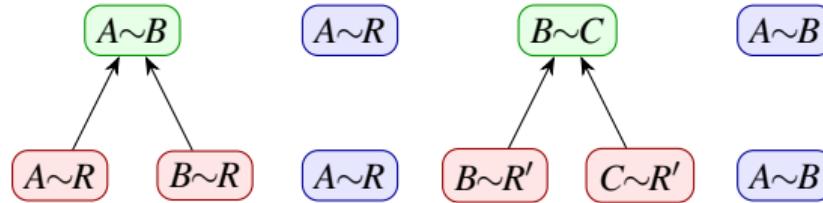


---

Bell pairs: **consumed**, **produced** and **untouched**

# Parallel composition vs. ordered composition

$\text{sw}\langle A \sim B @ R \rangle \| \text{tr}\langle B \sim R \rightarrow R' \sim R \rangle \| \text{sw}\langle B \sim C @ R' \rangle$



---

Bell pairs: **consumed**, **produced** and **untouched**

## Verification - properties with a classical analogue

- *Reachability.* Will the execution of our protocol generate  $A \sim C$ ?

## Verification - properties with a classical analogue

- *Reachability.* Will the execution of our protocol generate  $A \sim C$ ?
- *Waypoint Correctness.* Does our protocol ever perform a swap at node  $B$ ?

## Verification - properties with a classical analogue

- *Reachability.* Will the execution of our protocol generate  $A \sim C$ ?
- *Waypoint Correctness.* Does our protocol ever perform a swap at node  $B$ ?
- *Traffic (Protocol) Isolation.* Can we prove non-interference in a composition of Protocols I and II?

## Verification - properties with a classical analogue

- *Reachability.* Will the execution of our protocol generate  $A \sim C$ ?
- *Waypoint Correctness.* Does our protocol ever perform a swap at node  $B$ ?
- *Traffic (Protocol) Isolation.* Can we prove non-interference in a composition of Protocols I and II?
- *Compilation.* Can we ensure correct compilation to individual devices?

## Verification - properties specific to quantum

- *Resource Utilization.* What is the number of required memory locations and communication qubits? For how many rounds must a Bell pair wait in the memory?

## Verification - properties specific to quantum

- *Resource Utilization.* What is the number of required memory locations and communication qubits? For how many rounds must a Bell pair wait in the memory?
- *Quality of Service.* Do the generated Bell pairs have the required fidelity or capacity?

## Verification - properties specific to quantum

- *Resource Utilization.* What is the number of required memory locations and communication qubits? For how many rounds must a Bell pair wait in the memory?
- *Quality of Service.* Do the generated Bell pairs have the required fidelity or capacity?
- *Compilation.* Can we minimize the number of accesses to the network global state?

# Verification - properties specific to quantum

- *Resource Utilization.* What is the number of required memory locations and communication qubits? For how many rounds must a Bell pair wait in the memory?
- *Quality of Service.* Do the generated Bell pairs have the required fidelity or capacity?
- *Compilation.* Can we minimize the number of accesses to the network global state?

Thank you!

