Formal models of software-defined networks

Feng, H.

Formal Models of Software-Defined Networks

Hui Feng

Cover design: Hui Feng

Formal Models of Software-Defined Networks

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Leiden,
op gezag van rector magnificus prof.dr.ir. H. Bijl,
volgens besluit van het college voor promoties
te verdedigen op dinsdag 3 december 2024
klokke 10:00 uur

door

Hui Feng
geboren te Handan, China
in 1993

Promotor: Prof.dr. M.M.Bonsangue

Promotiecomissie: Prof.dr. L. Barbosa (University of Minho, Portugal)
Prof.dr. M. Sirjani (Mälardalen University, Sweden)
Prof.dr. F. de Boer
Prof.dr. J. Kleijn
Prof.dr. A. Plaat
Dr. Q. Chen

Hui Feng was financially supported through the China Scholarship Council (CSC) to participate in the PhD program at Leiden University. Grant number 201706350092.

The research in this thesis was performed at Leiden Institute of Advanced Computer Science at Leiden University, under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

To my parents.

"Fear is the main source of superstition, and one of the main sources of cruelty.
To conquer fear is the beginning of wisdom."

Bertrand Russell, 1872-1970

Contents

Chapter 1

Introduction

In the field of modern networking, Software-Defined Networks (SDNs) have emerged as a novel paradigm transforming the way we conceptualize, deploy, and manage computer network infrastructures. Defined by their programmability, flexibility, and centralized control, SDNs represent a departure from traditional networking approaches, offering a more dynamic and agile framework to meet the evolving demands of today's digital systems.

The evolution of networking paradigms leading to SDNs can be traced back to the limitations of traditional network architectures [111]. Conventional networks are characterized by their rigid and hardware-centric nature making them hard to adapt to the rapidly changing requirements of modern applications and services. The emergence of virtualization technologies, coupled with the growing complexity of network management tasks made clear the need for more scalable, efficient, and programmable networks.

SDNs address these challenges by decoupling the control plane from the data plane and centralizing network intelligence and management in software-based controllers [42]. Unlike traditional networks where control mechanisms are decentralized and embedded within network switches and routers, SDN logically centralizes the control and thus simplifies network programming. Specifically, SDN concentrates network control within one or more SDN controllers, enabling greater programmability and automation, and facilitating dynamic provisioning, configuration, and optimization of network resources. Moreover, the introduction of separated layers of abstraction simplifies network management tasks and promotes interoperability across heterogeneous networking environments.

The main defining feature of a Software-Defined Network is the centralized network control and management functions, which enhance orchestration and policy enforcement across the network. This means that tasks like configuring and managing network devices can be done more efficiently from a central controller rather than locally on switches. SDN architectures are designed to be programmable. They expose APIs and interfaces that allow for the programmable control of network behavior. This flexibility enables networks to adapt to specific application requirements more effectively. Additionally, SDNs abstract the underlying network infrastructure separating the logical network from its physical implementation, offering more flexibility and scalability [53, 52, 42].

In terms of benefits, SDNs offer numerous advantages. They enable agility and better scalability by decoupling network control from hardware. This leads to greater flexibility and resource utilization as well as enabling rapid deployment [52], as network policies can be updated and enforced across the network almost in real-time, with no need for manual and error-prone configuration of individual devices. SDNs also contribute to improved performance and reliability by allowing for the automation of many routine network management tasks. Finally, SDNs enhance network security by providing centralized visibility and control over network traffic. This enables more effective threat detection, mitigation, and policy enforcement, safeguarding against cyber threats and breaches [83].

Although widely applied with success in many large networks, SDNs present a multitude of challenges inherent to the complexity of SDN architectures themselves. As SDNs decouple the control plane from the data plane and centralize network intelligence, they introduce new layers of abstraction and dependencies, which can lead to increased complexity in design, implementation, and management. The dynamic and highly concurrent nature of SDNs amplifies the challenge of ensuring the correctness and reliability of the controllers. This can result in configuration errors, inconsistencies, and vulnerabilities that undermine the reliability and security of the network.

Current methods for verifying and validating SDN configurations often rely on manual testing, ad-hoc approaches, and proprietary tools, which are not scalable or comprehensive enough to address the complexities of modern SDN deployments. As a result, there is a pressing need for rigorous formal methods that can provide mathematical rigor and systematic approaches to analyzing, designing, and verifying SDN systems [98, 101].

1.1 Formal methods in SDN

Emphasizing the need for rigorous formal methods is essential in addressing these challenges effectively. Formal methods offer a systematic approach to modeling, analyzing, and reasoning about SDN behaviors and properties. By leveraging mathematical techniques such as formal specification languages, model checking, and theorem proving, formal methods can provide rigorous guarantees of correctness, reliability, and security in SDNs.

Formal models applied to SDN not only enhance our understanding of SDN systems but also facilitate precise analysis. Controllers of SDNs are essentially software entities, faults in a controller could lead the entire network to a catastrophic situation. Since networks are critical infrastructures, rigorous validation is the only way to ensure that network policies are correctly implemented before deployment and do not lead to unintended consequences. Techniques such as model checking and theorem proving enable exhaustive exploration of system behaviors under various abstractions, ensuring that SDN deployments used, for example, in data centers, enterprise networks, and cloud computing adhere to specified requirements and constraints.

The potential benefits of employing formal methods in SDN infrastructures are manifold. Increased reliability is achieved through rigorous verification and validation of SDN configurations, reducing the likelihood of configuration errors, protocol conflicts, and performance bottlenecks. VeriCon [14] is a tool designed for the verification of network configurations and behaviors in SDN environments. It provides a comprehensive platform that checks whether a given network configuration satisfies certain safety and correctness properties. VeriCon employs a combination of symbolic execution and model-checking techniques to verify properties such as security policies, reachability, and the absence of loops across all possible packet transmissions in the network. Enhanced security is realized by systematically analyzing and verifying security properties, ensuring that SDN deployments are resilient to cyber threats, attacks, and intrusions [86, 103]. Finally, improved maintainability is attained by precise modeling, enabling a deeper understanding of SDN systems and their behaviors, more efficient troubleshooting, debugging, and evolution of SDN infrastructures over time. As a modeling language designed for concurrent object-oriented systems, the Abstract Behavioral Specification (ABS) language focuses on simulating and verifying network behaviors. ABS tool suite [2] is a collection of tools designed to analyze and verify

SDN, includes capabilities for model checking, simulation, and deductive verification, providing a robust environment for testing and verifying the correctness and performance of SDN configurations and policies.

In the context of SDNs, existing model-checking tools can be applied to verify various properties and behaviors of SDN architectures, protocols, and deployments. Examples include TLA+ [72], a high-level language for modeling programs and systems, especially for concurrent and distributed ones. They can be checked for logical consistency and correctness by writing precise specifications. Its model checker TLC [112] can handle a subclass of TLA+ specifications that is suitable for the behavior of a single switch in a SDN [61]. The Model checker SPIN [48] uses Promela as its input modeling language, and the full Linear Temporal Logic (LTL) as a specification language. We will see in this thesis how SPIN can be used for modeling SDNs. Alloy [56] is a formal design language and a model-checking tool, it uses the Alloy for specifying models and properties, and Alloy's specification language is based on a subset of first-order logic with relational operators. It has been used to formally model the OpenFlow switch verifying a great part of the properties of an OpenFlow switch [94].

Both PRISM [70] and UPPAAL [16] have been utilized for verifying quantitative properties of SDNs such as performance, timing, reliability, and resource utilization. PRISM is a probabilistic model checker supporting Probabilistic Computation Tree Logic and Continuous Stochastic Logic. Although it is primarily designed for probabilistic systems, it has been used to analyze performance, reliability, Quality of Service (QoS), and security aspects of SDN deployments. UPPAAL supports temporal logic specifications expressed in Timed Computational Tree Logic (TCTL). It is specialized in real-time systems and has been employed for timing analysis, performance evaluation, resource allocation, and scheduling and synchronization analysis in SDNs.

SDN testing strategies can increase our trust in SDN deployments ultimately improving the reliability and robustness of SDN systems. However, the complexity and diversity of SDN environments make it challenging to create comprehensive test scenarios that accurately reflect real-world conditions. Additionally, the dynamic nature of SDN infrastructures introduces uncertainties and variability that may not be fully captured in traditional testing approaches, leading to potential gaps in test coverage and effectiveness [87].

1.2 Research objectives

In this thesis, we explore the use of the coordination language Reo as a model of SDNs. Reo [5] is a coordination language designed for specifying and orchestrating the behavior of reactive and distributed systems. It provides a powerful and visually appealing framework for expressing complex coordination patterns among system components, facilitating the modeling of communication, synchronization, and interaction protocols in a concise and modular manner.

Reo is characterized by its formal semantics given in terms of constraint automata [13]. It comes equipped with several tools for the rigorous analysis and verification of system specifications, ensuring correctness and reliability. With support for hierarchical composition and modularity, Reo promotes the reuse and maintainability of coordination patterns, facilitating the development and evolution of distributed systems.

Our choice for Reo is justified by the easy and intuitive visual representation via user-defined channels and their composition. Reo offers a versatile tool for designing and coordinating reactive systems, with applications ranging from network protocols and distributed algorithms to software architectures and concurrent processes.

While Reo was not explicitly developed for SDN, we investigate in this thesis if its principles and capabilities could potentially be applied to model and verify certain aspects of SDN architectures, leading to our first research question:

Research question 1. *Can the coordination language Reo and its semantic model of constraint automata be used as a formal model of SDNs?*

Because Reo was primarily designed for modeling reactive systems and distributed protocols, we found that certain features and concepts in SDN, such as switch flow tables and specific communication messages between switches and controllers, are not well-suited for being directly modeled using Reo. This motivated us to investigate Chapter 3 for a non-trivial extension of Reo with variables while maintaining the idea behind the synchronization operation of the original Reo language.

Our Reo model of SDN switches and controllers together with their network topology includes an abstraction of the OpenFlow communication protocol. While OpenFlow is not the same as an SDN, it enables the programmability and centralized control of its network devices, such as switches and routers. It defines

a standardized interface between the control plane and the data plane of network devices, allowing a centralized SDN controller to dynamically manage the forwarding behavior of network switches and routers [42]. Our Reo model of the OpenFlow protocol serves as a foundational study providing a formal model for implementing and orchestrating network control in the programmable and dynamic environments of SDNs.

The advantage of using Reo as a formal semantics, of OpenFlow-based architectures of SDNs is the modularity of the approach and the possible leverage of verification tools for SDNs. Reo is supported by various tools and frameworks for simulation, model checking, and verification, however, that cannot be directly used with the extension of Reo we considered for SDNs. This brings us to our second research question:

Research question 2. *Can we use existing model checkers to verify properties of SDNs modeled by Reo?*

We answer this question in Chapter 4, where we show how to generate automatically Promela code from symbolic constraint automata, the semantic model of the extension of Reo we use for SDNs. Promela (PROcess MEta LAnguage) is a process modeling language specifically designed for modeling and verifying concurrent and distributed systems using SPIN (Simple Promela Interpreter), a powerful model-checking tool [48]. With SPIN, Promela models can be analyzed for various linear time temporal properties, such as deadlock freedom, liveliness, and safety, through exhaustive state space exploration. SPIN provides an interface for specifying Promela models and defining verification properties, allowing a systematic validation of the correctness of complex concurrent systems.

By modeling the SDN control plane and data plane in Reo and automatically obtaining a Promela code, we can use the SPIN model checking tool to explore the SDN model and verify properties. Another high-level domain-specific language for expressing and reasoning about network policies and behaviors in Software-Defined Networking (SDN) environments is NetKAT [4]. NetKAT is inspired by Kleene algebra, a mathematical framework for describing the behavior of regular expressions, and is extended to the domain of network policies. In NetKAT, network policies are expressed as compositions of basic primitives, such as packet filters, forwarding actions, and logical operators, allowing for the specification of complex routing and forwarding behaviors in a concise and modular manner.

Similar to our Reo model, NetKAT provides formal semantics and mathe-

matical foundations for reasoning about network policies, enabling verification techniques such as equational reasoning and model checking to ensure correctness and consistency. Since both Reo and NetKAT use automata-based semantics, a natural question is whether the two models can be unified in a single framework:

Research question 3. *Can we extend the automata-based model of NetKAT to be stateful and allow concurrency in a way similar to Reo?*

In Chapter 5 we extend NetKAT with ports, which basically are shared variables used by processes to communicate with each other. Differently from a traditional variable, reading a value from a port is destructive, while writing is only possible if the port is empty. The addition of ports to NetKAT and a simple modification of its automata model are enough to be comparable with the Reo model. In particular, the extension of NetKAT with ports allows for concurrency, it is stateful, and it is backward compatible with the original NetKAT model.

So far, the automata model of Reo (and NetKAT) we introduced has been mainly geared toward the verification of SDNs. However, another important aspect of SDN is causality, particularly in understanding and managing network events, behaviors, and dependencies. In SDNs, where the control plane is centralized and programmable, causality helps determine the order of events and actions that influence network behavior. Understanding causality enables SDN controllers to make informed decisions about network policies, routing, and resource allocation based on the sequence of events and their causal relationships. Moreover, causality is crucial for troubleshooting and debugging network issues, as it helps identify the root causes of anomalies or failures by tracing the sequence of events leading to the observed behavior. Our final research question is related to causality:

Research question 4. *Can we use Reo or NetKAT and their associated automata-based models for avoiding hazard events in SDN using causality?*

We partially answer this question in Chapter 6, where we investigate an ordinary automata model incorporating causality into the design of the safety properties so that if a hazard event may occur then not only we must avoid it, but we also offer an alternative sequence of action that does not cause that event to happen. While the automata we consider are simpler than the one used as the model for SDNs via either NetKAT or Reo, we see this result as a first step toward

causality reasoning for SDNs.

1.3 Publications

In this section, we present all publications upon which our research presented in this thesis is based. They constitute the primary source of our theoretical research and methodological framework.

Hui Feng, Farhad Arbab and Marcello Bonsangue. (2019). A Reo Model of Software Defined Networks. In: Ait-Ameur, Y., Qin, S. (eds) Formal Methods and Software Engineering. ICFEM 2019. Lecture Notes in Computer Science, vol 11852. Springer, Cham. https://doi.org/10.1007/978-3-030-32409-4_5

In this paper, we use Reo to model SDN switches and controllers, according to the OpenFlow protocol specification. We use a similar abstract syntax and semantics of OpenFlow messages to model the interactions between the control plane and data plane in an OpenFlow-based SDN architecture. We also implement the data flow between different controllers and switches. The formal semantics of Reo is given by a novel version of constraint automata with memory, that we called symbolic constraint automata. Most of the content of Chapters 3 and 4 is based on this paper.

Hui Feng, Marcello Bonsangue and Benjamin Lion. (2022). From symbolic constraint automata to Promela. Journal of Logical and Algebraic Methods in Programming. vol 128. 100794. https://doi:10.1016/j.jlamp.2022.100794

In this paper, we implement an automatic translation from symbolic constraint automata to Promela code, the programming modeling language of the model checker SPIN. The translation enables the analysis of packet forwarding according to OpenFlow controllers, The focus is on verifying the functional properties of the SDN model, such as reachability, consistency, and correctness of network policies and configurations using linear temporal logic. As expected, the model suffers from the state explosion problem, and more research needs to be done on the scalability of our approach. This could involve simplifying the OpenFlow protocol to reduce the state space to be analyzed. On the positive side, our translation is not only for SND models but for any Reo circuit (with

and without variables). The main results of this paper are presented in Chapter 5.

Hui Feng and Marcello Bonsangue. (2024). Concurrent NetKAT with ports. In: Juw Won Park and Adam Przybyłek and Hossain Shahriar. (eds) SAC'24: Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing. Association for Computing Machinery. https://doi.org/10.1145/3605098.3636048

The starting point of this paper is the original definition of NetKAT automata, which we extend so to support concurrency. The main idea is to use NetKAT automata as a model of propositional Hoare logic for concurrent processes communicating via shared ports.

Marcello Bonsangue, Georgiana Caltais, Hui Feng, and Hünkar Can Tunç (2022). A Language-Based Causal Model for Safety. In: Yamine Aït-Ameur, Florin Crăciun. (eds) Theoretical Aspects of Software Engineering. TASE 2022. Lecture Notes in Computer Science, vol 13299. Springer, Cham. https://doi.org/10.1007/978-3-031-10363-6_20

This paper is a first step towards a notion of causality for SDNs. Using ordinary automata we implemented an algorithm for finding and removing hazards specified as regular expressions. Since Reo can be modeled by NetKat automata with concurrency, causality analysis for SDN should be relatively easy to reach. By understanding the causal links between network events and configuration changes, and resulting outcomes, causality analysis could be used for diagnosing performance bottlenecks and troubleshooting issues helping in identifying the root causes of anomalies and failures. The content of this paper forms the basis for Chapter 7.

1.4 Overview

In this thesis, we applied Reo and constraint automata to formally model SDN architectures based on OpenFlow, We translated our formal model into Promela, so to allow for the use of state-of-the-art model-checking tools. More semantically, we extend the automata semantics of NetKAT to handle concurrent components communicating via shared ports. The latter model is exactly our Reo semantic model of SDNs. Finally, we took a few first steps into a causal analysis for SDNs.

The thesis is organized as follows: In Chapter 2, we introduce the main concepts of software-defined networking (SDN), paying specific attention to the three-layer framework and OpenFlow protocol. Consequently, in Chapter 3 we present the coordination language Reo and its formal semantics. Throughout this thesis, Reo is our main modeling language for the analysis of SDNs. The focus is on composition methods at the semantic level. Subsequently, in Chapter 4 we address **research question 1** by constructing a Reo model that faithfully encapsulates the data plane and control plane of an SDN, including the modeling of individual OpenFlow switches and their interconnecting channels. In Chapter 5, we propose a compositional translation from symbolic constraint automata to Promela, aligning with **research question 2**. In Chapter 6 we address the limitations of NetKAT in handling concurrency and relating it to Reo, thus resolving **research question 3**. Finally, Chapter 7 addresses **research question 4**, wherein we introduce an automata-based causal model where hazards are expressed in terms of regular expressions.

Chapter 2

Software-defined Networks

In this chapter, we briefly introduce software-defined networks (SDNs) starting from the idea behind SDNs and their benefits over traditional computer networks. We also discuss a few protocols and concentrate on OpenFlow, a protocol that enables the centralized control of network switches and allows for programmable networking capabilities.

2.1 Traditional networking vs. SDN

Traditional computer network architectures are based on the seven layers of the Open Systems Interconnection (OSI) model: the physical layer, the data link layer, the network layer, the transport layer, the session layer, the presentation layer, and the application layer. More specifically these layers describe how data is transferred and received over a network. Each layer is accountable for carrying out particular tasks related to sending and receiving data, whereas, for a message to be delivered and received, all of the layers must be utilized.

The seven layers above provide a conceptual framework for understanding network communication[20]. When considering the functional components and responsibilities we then classify network components depending on the control logic, the forwarding functionality, and where applications are executed. In general, the primary function of a network is to ensure the transport of packets via routing decisions and packet forwarding. The actual forwarding part of the network is called the data plane, whereas the part controlling how this has to be executed is called the control plane. In a traditional network, the control and

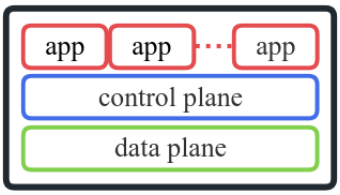


Figure 2.1: Structure of the switch in a traditional network

data planes are closely tied together with the application plane, see Figure 2.1, as they are in charge of packet forwarding, building, and maintaining the routing tables. In fact, within the OSI model[102], switches are typically in the data link layer, and are responsible for forwarding packets by using physical addresses that identify devices on a network, the so-called MAC addresses. But switches can also have routing functionality and are instead positioned in the network layer. As such, if the behavior of this network needs to be changed the switches have to update their local control planes accordingly. This is even more difficult when more and more network devices are employed since the control plane is highly distributed without a global view of the network. As a result, it is very difficult to program network-wide decisions and even more difficult to verify their correspondence against global requirements.

The solution of an SDN architecture is to separate the data plane from the control plane and the application plane[62], moving the latter two from the switches to the network as shown in Figure 2.2. The advantage is a much simpler way to regulate the network behaviour allowing for more flexible and dynamic network management, as well as improved scalability and security. Since the data plane is only responsible for packet forwarding, modifying the networks can now be done at the application level, using software, thus explaining the name "software-defined" networking.

In an SDN, the control plane consists of a few controllers that are responsible for managing a large part of a network. Each controller can be configured to decide how to transfer packets over the network based on a variety of factors, such as traffic patterns, security policies, and service-level agreements.

One of the key benefits of an SDN is that it allows for more efficient use of network resources because each controller can make routing decisions in real-time, without the need to stop the forwarding process of the switches, for example by reconfiguring the network logical structure in response to changing conditions,

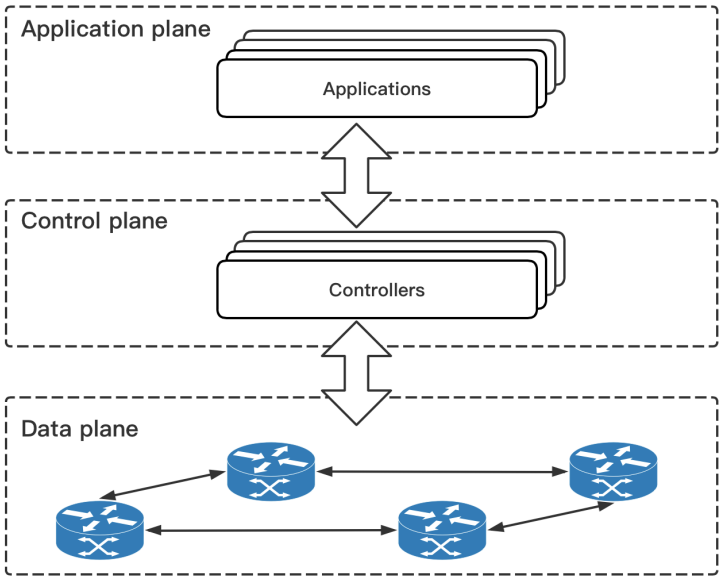


Figure 2.2: SDN architecture

such as nodes or hardware link failures.

2.2 Protocols for SDNs

The architecture offered by SDNs simplifies the design and deployment of network management tasks: the control plane is a logically centralized controller that gathers information from the data plane and provides a global view to applications running on top of the controller. These applications make packet routing decisions based on the global view and distribute these decisions to the data plane via the control plane. There is a northbound interface (NBI) between the application plane and the controller for permitting a specific entity of a network to communicate with a higher-level entity, the interface is on the upper side of the controller called the "northbound" interface. Different from the southbound interfaces (SBIs) between controllers and the switches, NBIs are offered by the controllers themselves and are vendor-dependent as there is no open standard NBI.

On the contrary, the SBI supports communication between the control planes and data plane and is typically subject to OpenFlow [82], a specific communications protocol developed by the Open Networking Foundation (ONF). While

OpenFlow is the most famous communications protocol for SDNs, it is not the only one. For instance, other popular protocols for the SBI of an SDN include:

- the Simple Network Management Protocol (SNMP), an Internet standard protocol used for managing and monitoring network devices, allowing for the collection of information and the control of network elements in a managed network [115] [79];
- the Forwarding and Control Element Separation (ForCES) protocol that separates the control plane from the data plane in network devices beyond switches using a master-slave mechanism, and allowing for centralized control and management of forwarding elements [45];
- the Path Computation Element (PCE) protocol, a communication protocol used to enable centralized path computation and optimization in multi-domain or multi-layer networks, moving this functionality from the routes operating systems to the control plane [105];
- the Border Gateway Protocol (BGP), used to exchange routing and reachability information between autonomous systems (AS) on the Internet, enabling the interconnection and dynamic routing decisions among different networks [91, 73].
- the NETwork CONFiguration Protocol (NETCONF), based on the Yet Another Next Generation (YANG), used for remote network device management and configuration, providing a programmable interface for network automation and orchestration [31].

Each of these protocols has its own set of features, capabilities, and use cases and all can be used in an SDN architecture without competing with each other. In this thesis, we only concentrate on the OpenFlow protocol.

2.3 The OpenFlow protocol

Next, we present an informal introduction to the basics of the OpenFlow protocol, following the specifications released in [39]. We first recall the definition of an SDN packet. Then we describe the three types of OpenFlow messages, from where they flow, and how they are used to change the network behaviors.

OpenFlow is a communications protocol that is used to control the flow of data in SDN. It provides a standard way for SDN controllers to communicate with switches and viceversa [97, 82, 52, 64, 17, 51].

2.3.1 SDN packets

A packet is a unit of data that carries information across a network, encapsulating the control information necessary for its transmission. Besides data, it consists of a header containing fields storing, for example, source and destination addresses, and protocol information. The example below shows two packets, both with a header containing information about the `tcp` and `ethernet` destination address of the packet:

`tcp_dst`:22, `eth_dst`:11	`data`: d1

`tcp_dst`:23, `eth_dst`:11	`data`: d2

2.3.2 OpenFlow flow tables and flow entries

In an SDN architecture, each switch in the data plane has a number of ports that are used to receive and forward packets. Switches are connected to at least one controller, from which they may receive or which they may send OpenFlow messages to. The main task of a switch is to forward packets to other switches.

When a packet arrives at a port of a switch, the SDN controller, which has a global view of the network, determines how the packet should be handled based on its header information. The process of forwarding a packet involves forwarding rules stored in a flow table. The table consists of an ordered set of pairs (b, a), where b is a Boolean condition on the values in the header fields of a packet (the so-called matching criteria) and a is the corresponding action to be executed on the matching packet. The order of the matching-action pairs gives priority to the application of the matching condition. There are basically three types of actions: *forwarding* a packet to one or more ports of the switch, *dropping* a packet, and *updating* a field of a packet with some value.

For example, the leftmost packet above matches the first rule of the table below, and it is forwarded to the output ports 3 and 4. The rightmost packet however matches only the last rule and it is forwarded to port 1 after its field `tcp_dst` is updated to 22.

Matching Condition	Action
`tcp_dst`:22	`Forward`[3, 4]
`tcp_dst`:23, `eth_dst`:12	`drop`
`true`	`tcp_dst` := 22; `Forward`[1]

2.3.3 OpenFlow messages

Controllers and switches communicate through three types of messages. A `PktIn` message is from a switch to a controller. It includes a packet with all its header fields and data and some additional information such as the port from where the packet entered into the switch. Typically a `PktIn` message would be processed by the controller to trigger an update of the flow tables.

A `PktOut` message is from a controller to a switch. It consists of a packet together with a flow table action to be executed by the switch. This way a packet need not pass through the flow table but is, for example, immediately forwarded to other switches. The `PktOut` message is used to implement a one-time redirection of a packet to a different path after a decision of the controller.

The flow table of a switch is updated by a `FlowMod` message, and it is sent by a controller to the switch. Each `FlowMod` message consists of a `ModType` t (`Add`, `Remove`, `Modify`), a matching condition b and an action a. If $t =$ `Add` then the pair (b, a) is added on top of the table (having thus highest priority), while if $t =$ `Modify` then the first pair in the flow table (b', a') with b implying b' is substituted with the new pair (b, a). In the remaining case when $t =$ `Remove`, the first pair in the flow table (b', a') with b implying b' is removed from the table. In this case, the action a does not play any role and therefore can be considered empty. The `FlowMod` message allows a controller to add or remove rules to the flow table, enabling the programmatic control of the behavior of the network. Note that the action of an entry in the flow table is executed only when a package arrives matching the respective conditions.

Those three types of messages plus dedicated packets to communicate data allow controllers to gather information about the network and manage it.

2.3.4 Evaluating the OpenFlow protocol

OpenFlow enables centralized control over network devices, allowing for a global view of the network and centralized control decisions. This facilitates dynamic and flexible network management and it enables programmatic control

by SDN controllers [38, 93]. The controller can be programmed to reconfigure the network automatically in response to changing the network status. This can help to improve the network's overall performance, reliability, and scalability.

OpenFlow primarily focuses on the communication between the control plane and data plane of network switches. The centralized control and programmability introduced by OpenFlow can also pose vulnerability and security challenges [107] [10]. Also, the centralized control model of OpenFlow can become a scalability bottleneck, and distributing the management of switches among several controllers may introduce race conditions. Finally, the matching criteria of a flow table are limited by the fields of a packet header. For example, predicates on the action result are not allowed, limiting some traffic patterns or specialized requirements.

2.4 Two examples

Consider the scenario presented in Figure 2.3 in which Switch 1 forwards all packets received from port A to port C, and sends packets received from port B to port Q as a `PktIn` messages for the controller. The controller in this example has a local view of the topology of the network. In particular, it is aware that ports A and B are connected to the host devices with IP addresses 192.168.0.1 and 192.168.0.2, respectively, while port C is connected to the host device with IP address 192.168.1.1. Further, the switch uses port P for incoming `PktOut` and `FlowMod` messages and port Q for outgoing `PktIn` messages.

The first time a packet is received by Switch 1, by using the OpenFlow protocol the controller will receive a `PktIn` message containing the packet and may, for example, decide to update the flow table of the Switch by sending a `FlowMod` message so that the action of the second line of the flow table is updated to forwarding a packet to port C. By sending also `PktOut` message with the original packet and the new action, the controller guarantees that no placket gets lost.

A second example is given by a controller connected to four switches as shown in Figure 2.4. Assume all switches have initially an empty flow table. When switch 1 receives a packet at port A, since the packet does not match any entries of the flow table, the switch sends it as a `PktIn` message to the controller. The controller calculates a path to the destination device (say a device with IP number 192.168.1.1) and it returns a `FlowMod` message to all switches involved in the path so that the packet is forwarded along ports B, C, D, E, F, G and H in the path

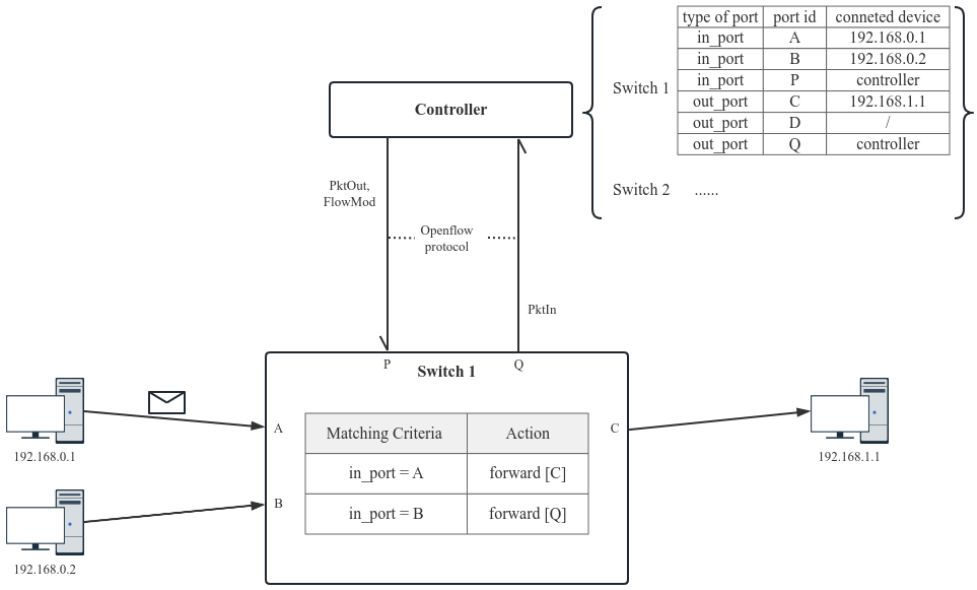


Figure 2.3: A single switch SDN network

(see Figure 2.4). Subsequently, the controller sends a `PktOut` message to switch 1 only, containing the original packet and the action for forwarding it to port B. As the packet moves along the path to its destination, the flow table of the other switches may or may not have received the `FlowMod` message for updating the table. If yes the packet is forwarded from one switch to another until it arrives at its destination.

However, because of an eventual race condition, if one of the switches along the path has not yet received the `FlowMod` message for updating the table, when the packet arrives it will be forwarded again to the controller inside a `PktIn` message, and the controller will have to send it back via a `PktOut` message with the corresponding forwarding action. However, once the first packet arrives at its destination, it can be guaranteed that the next packet that arrives at port A of Switch 1 will go directly to its destination without the need for a controller to intervene.

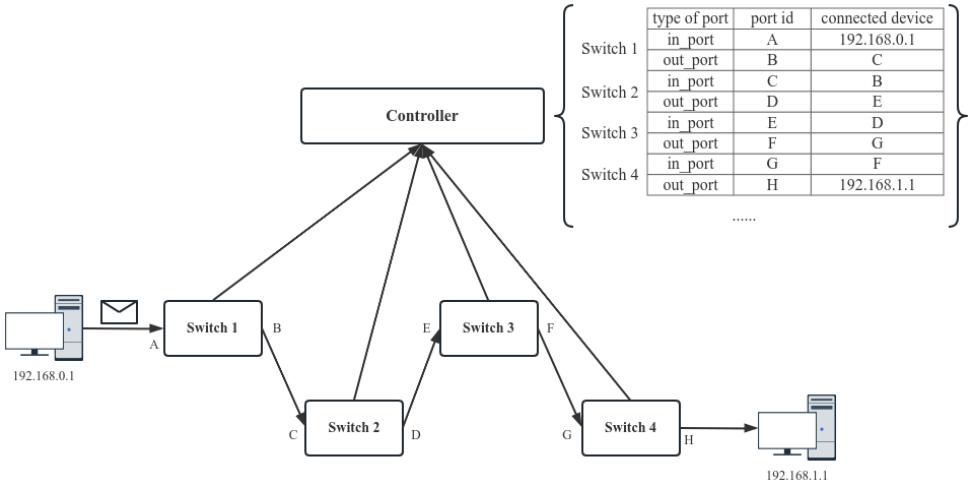


Figure 2.4: A multi-switch SDN network

Chapter 3

Reo and constraint automata

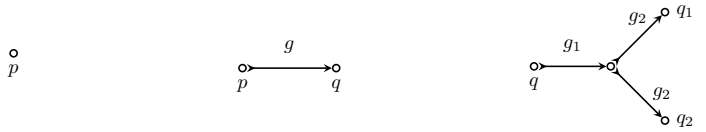
After having introduced SDNs in Chapter 2, we now turn to Reo, a coordination language that we will use in the next chapter to model SDNs. The advantage of using Reo comes from its intuitive graphical syntax that comes equipped with precise automata-based semantics. The emphasis in Reo is on data and synchronization constraints, expressed by connectors, determining their behavior and importance during composition. In fact, Reo has already been used for modeling and analyzing a wide range of systems, including communication protocols [59], workflow systems [85], and control systems [9].

3.1 A short introduction to Reo

Reo is a coordination language for the compositional construction of component connectors [5]. Connectors in Reo are modeled as directed graphs describing the way data flows through a system. The nodes of the graph are called ports and can be used to connect a connector to other connectors. The behavior of a Reo system is determined by the way these ports are connected together, as well as the constraints that are imposed on the data flowing through the system. A port that is used exclusively as the source of edges is called an input port and represents an interface through which the connector receives data. Dually, a port that is only the target of the edges of a connector is called an output port (or sink), and represents an interface through which the connector offers data to the environment. Ports that are both source and target of some edges of the connector are "hidden" to the environment and have a 'merge-replicate' behavior: it

accepts data from one (chosen non-deterministically) of the target edges of the connector and immediately sends it to all edges with the hidden port as a source. Isolated ports do not have any behavior by themselves. Note that, being an input or an output port is a property that depends on the connector, and in fact, when composing connectors, the same port can be input for one and output for the other.

Figure 3.1 shows the graphical representation of three simple Reo connectors: (a) is an isolated Reo port p, (b) is a connector representing a channel where the data flows from the input port p to the output port q under the constraints g, and (c) is a connector with an unnamed internal port that enables the flow of data from q to both q_1 and q_2 under some data constraints. Besides data constraints, connectors specify also synchronization constraints that are visualized by using two different types of edges: synchronous and asynchronous. Specifically, all the connectors in this example use synchronous edges implying that flow between the input and output ports of the connector is logically happening at the same time, i.e. they synchronize.



(a) Port (b) Single channel connector (c) Multiple channel connector

Figure 3.1: Port, channel, and connector

Synchronization constraints in Reo are strict, meaning that they impose synchronization between ports connected by synchronous edges, and nothing else. No flow among other input and output ports belonging to a connector is allowed. This is of importance when two connectors are connected, as data is only allowed to flow between the common input and output ports, as well as on input or output ports that belong to one but not to the other connector.

3.2 Constraint automata

Constraint automata are a formalism to describe the behavior of Reo connectors and their composition [13]. Constraint automata can be thought of as conceptual generalizations of finite state automata where data and synchroniza-

tion constraints influence which transitions are enabled on a given state.

We assume given a finite set $\mathbb{D}$ (ranged over by d) denoting all possible data that can be sent and received by Reo connectors, and a finite set $\mathbb{P}$ of ports names ranged over by p, q. Here $p \in \mathbb{P}$ is a port of a connector (a different concept than a port of a switch in an SDN). While later we will distinguish between input and output ports, for simplicity, and following the original presentation [13] we do not consider this difference here. Given a non-empty subset $N \subseteq \mathbb{P}$ of ports, we define the set $DC(N)$ by the following grammar::

$$g ::= true \mid p = d \mid g_1 \wedge g_2 \mid \neg g\,.$$

Here $p = d$ is the basic constraint imposing the data d to be available at port $p \in N$. Basic constraints can be composed using the usual Boolean operation. Since $\mathbb{D}$ is a finite set, we sometimes use $p = q$ to denote the constraint $\bigwedge_{d \in Data}((p = d) \wedge (q = d))$. As usual, we write $p \neq d$ for $\neg(p = d)$.

Definition 3.1 ([13]). *A constraint automaton is a tuple $(S, N, \longrightarrow, S_0)$ where*

- *S is a finite set of states,*
- *$N \subseteq \mathbb{P}$ is a finite set of ports,*
- *$\longrightarrow \subseteq S \times (N \times DC(N)) \times S$ is a transition relation such that $s \xrightarrow{P,g} s'$ implies $P \neq \emptyset$ and $g \in DC(P)$, and*
- *$S_0 \subseteq S$ is the set of initial states.*

A transition $s_1 \xrightarrow{P,g} s_2$ is enabled in state s_1 when all ports in P have data available (in the case of inputs) or no data (in the case of outputs). In this case, the automaton moves to state s_2 if all data constraints imposed by g are satisfied.

For example, the three automata corresponding to the connectors of Figure 3.1 are defined as follows

(a) $A_1 = (\{s_1\}, \{p\}, \emptyset, \{s_1\})$;

(b) $A_2 = (\{s_2\}, \{p, q\}, \longrightarrow_2, \{s_2\})$ with $s_2 \xrightarrow{\{p,q\},\, g}_2 s_2$; and

(c) $A_3 = (\{s_3\}, \{q, q_1, q_2\}, \longrightarrow_3, \{s_3\})$ with $s_3 \xrightarrow{\{q,q_1,q_2\},\, g_1 \wedge g_2}_3 s_3$.

The semantics of a constraint automaton describe how it behaves and evolves over time. It is defined in terms of Timed Data Streams (TDS) [7] which associate with each port an infinite sequence of events representing the data flowing through the port and the time when this occurs. The transitions of the automaton then

define the constraints that the data must satisfy, whereas synchronization relates to the time of the data flow at the synchronizing ports.

The composition of two constraint automata is a fundamental operation in the formal modeling and analysis of systems using Reo. It describes how data flow events of the two automata should align with each other.

Definition 3.2. *The product of the two constraint automata* $A_1 = (S_1, N_1, \longrightarrow_1, S_{1,0})$ *and* $A_2 = (S_2, N_2, \longrightarrow_2, S_{2,0})$ *with disjoint sets of states* S_1 *and* S_2*, is defined as the automaton*

$$A_1 \bowtie A_2 = (S, N, \longrightarrow, S_0)$$

where $S = S_1 \times S_2$*,* $N = N_1 \cup N_2$*,* $S_0 = S_{1,0} \times S_{2,0}$ *and* $\longrightarrow$ *is defined by the following three rules:*

$$\frac{s_1 \xrightarrow{P_1, g_1}_1 t_1 \quad s_2 \xrightarrow{P_2, g_2}_2 t_2 \;\; \textit{and} \;\; P_1 \cap N_2 = P_2 \cap N_1}{\langle s_1, s_2 \rangle \xrightarrow{P_1 \cup P_2,\, g_1 \wedge g_2} \langle t_1, t_2 \rangle}$$

$$\frac{s_1 \xrightarrow{P_1, g_1}_1 t_1 \;\; \textit{and} \;\; P_1 \cap N_2 = \emptyset}{\langle s_1, s_2 \rangle \xrightarrow{P_1, g_1} \langle t_1, s_2 \rangle} \quad \textit{and} \quad \frac{s_2 \xrightarrow{P_2, g_2}_2 t_2 \;\; \textit{and} \;\; P_2 \cap N_1 = \emptyset}{\langle s_1, s_2 \rangle \xrightarrow{P_2, g_2} \langle s_1, t_2 \rangle}$$

Similar to the product automaton the above composition combines the states of the two automata into pairs to represent the possible configurations of the composed system. Common ports at each transition are matched based on their names. This step ensures that the time of flow events of one automaton coincides with the time of flow events of the other automaton. Also, the data coincides, as data constraints of both automata must be valid. Data flow at disjoint ports remains unchanged and happens in an interleaving fashion.

For example, the automata A_2 and A_3 above share only the port q. When composing them, the behaviors at the other ports are unchanged, but the flow of data at the common port q must be the same. The resulting automaton is $A_1 \bowtie A_2 = (\{\langle s_2, s_3 \rangle\}, \{p, q, q_1, q_2\}, \longrightarrow, \{\langle s_2, s_3 \rangle\})$ with only one transition, namely:

$$\langle s_2, s_3 \rangle \xrightarrow{\{p,q,q_1,q_2\},\, g \wedge g_1 \wedge g_2} \langle s_2, s_3 \rangle .$$

Note that all four ports synchronize and that data in p must satisfy the same constraints $g \wedge g_1 \wedge g_2$ as data at all other ports because it flows from p to q_1 and q_2 passing through q. Formally there is no direction in this flow, and all

ports are still open to the environment, while it may be desirable for port q to be restricted to further communication with the environment after a composition. We will consider an extension addressing both points in the next section.

3.2.1 Constraint automata with memory

In many systems, the current behavior may depend on past events. Constraint automata lack the ability to retain information about past inputs or data, which can influence the current behavior of components. While this capability could be recovered by adding more states to the automaton in the case of finite data, the presentation of the automaton's behavior would enormously suffer and the number of states would increase exponentially. Moreover, in the presence of infinite data adding states would not suffice. By incorporating memory, an automaton gains the ability to become stateful by storing data in local internal variables. And these variables can be utilized in constraints just as input or output ports in ordinary constraints automata.

Next, we extend constraint automata by adding local variables to store data as recently studied in [57]. Except for the existential quantifier in constraints that could be easily added, the main difference with the approach below is a neater treatment of the values in the memory before and after the execution of a transition.

In addition to the set $\mathbb{D}$ of data and $\mathbb{P}$ of ports as used in constraint automata, we now assume another disjoint finite set $\mathbb{M}$ of memory cells ranged over by m. Further, let $\mathcal{F}$ be a set of function symbols and $\mathcal{P}$ a set of predicate symbols. Each predicate symbol and each function symbol comes with an arity, the number of arguments it expects. A term is defined as follows:

$$t ::= d \mid p \mid m \mid m^{\bullet} \mid f(t, ..., t)$$

The idea is that p denotes the value at the port p, m evaluates the value stored in the memory cell before the execution of a transition, and $m^{\bullet}$ to the value immediately after the execution of the transition. Evaluation of functions is as usual. Terms are used in constraints that are defined by the following predicate formulas:

$$\phi ::= \top \mid p = t \mid m = t \mid m^{\bullet} = t \mid P(t, ..., t) \mid \phi \wedge \phi \mid \neg\phi$$

The constraint $p = t$ denotes the equality between the value passing through the port p, and the value obtained by evaluating the term t; $m = t$ is the equality between the value stored in the memory m before evaluating the constraint and the value denoted by t; and $m^\bullet = t$ is the equality between the value stored in the memory m immediately after the evaluation of the constraint and the value denoted by t. The others are just the usual constraints.

In order to define the satisfaction of constraints, we assume the existence of a function $\hat{f}{:}\mathbb{D}^n \to \mathbb{D}$ for each $f \in \mathcal{F}$ of arity n, and a subset $\hat{P} \subseteq \mathbb{D}^m$ for each predicate symbol $P \in \mathcal{P}$ of arity m. For fixed sets of input ports $I \subseteq \mathbb{P}$, output ports $O \subseteq \mathbb{P}$ and hidden ports $H \subseteq \mathbb{P}$, the evaluation of constraint is defined by using the function $\alpha{:}I \cup O \cup \mathbb{M} \to \mathbb{D}_\perp$, and an environment $\eta{:}H \to \mathbb{D}_\perp$ assigning values to hidden ports. α is used for the visible components of a Reo connector. Here $\alpha(p)$ represents the value passing through port p unless $\alpha(p) = \perp$ denotes the absence of data flow through port p. Similarly $\alpha(m)$ denotes the value stored in the memory cell m.

We denote by At the set of all atoms α. Note that $m^\bullet$ is not a part of an atom, because it refers to the value of m after the evaluation of a transition. Therefore we need pairs of atoms, one for the current values stored in memory cells, and another for storing the side effect of an evaluation, i.e., the value of a memory cell after the evaluation. Evaluation of guards is defined inductively as follows:

$$
\begin{array}{llll}
\alpha_1\alpha_2 & \models_\eta & \top & \\
\alpha_1\alpha_2 & \models_\eta & p = t & \text{iff} \quad \alpha_1(p) = [\![t]\!]^\eta_{\alpha_1\alpha_2} \\
\alpha_1\alpha_2 & \models_\eta & m = t & \text{iff} \quad \alpha_1(m) = [\![t]\!]^\eta_{\alpha_1\alpha_2} \\
\alpha_1\alpha_2 & \models_\eta & m^\bullet = t & \text{iff} \quad \alpha_2(m) = [\![t]\!]^\eta_{\alpha_1\alpha_2} \\
\alpha_1\alpha_2 & \models_\eta & P(t_1, ..., t_n) & \text{iff} \quad \langle [\![t_1]\!]^\eta_{\alpha_1\alpha_2}, ..., [\![t_n]\!]^\eta_{\alpha_1\alpha_2} \rangle \in \hat{P} \\
\alpha_1\alpha_2 & \models_\eta & \phi_1 \wedge \phi_2 & \text{iff} \quad \alpha_1\alpha_2 \models_\eta \phi_1 \text{ and } \alpha_1\alpha_2 \models_\eta \phi_2 \\
\alpha_1\alpha_2 & \models_\eta & \neg\phi & \text{iff} \quad \alpha_1\alpha_2 \not\models_\eta \phi
\end{array}
$$

Finally, we define the evaluation of a guard without hidden ports as follows:

$$\alpha_1\alpha_2 \models \phi \text{ if and only if there is } \eta \text{ such that } \alpha_1\alpha_2 \models_\eta \phi.$$

Here $[\![t]\!]^{\eta}_{\alpha_1\alpha_2}$ denotes the value of the term t and is defined inductively by:

$$\begin{aligned}
[\![d]\!]^{\eta}_{\alpha_1\alpha_2} &= d \\
[\![p]\!]^{\eta}_{\alpha_1\alpha_2} &= \begin{cases} \alpha_1(p) & \text{if } p \in I \cup O \\ \eta(p) & \text{if } p \in H \end{cases} \\
[\![m]\!]^{\eta}_{\alpha_1\alpha_2} &= \alpha_1(m) \\
[\![m^{\bullet}]\!]^{\eta}_{\alpha_1\alpha_2} &= \alpha_2(m) \\
[\![f(t_1, ..., t_n)]\!]^{\eta}_{\alpha_1\alpha_2} &= \hat{P}([\![t_1]\!]^{\eta}_{\alpha_1\alpha_2}, ..., [\![t_n]\!]^{\eta}_{\alpha_1\alpha_2})
\end{aligned}$$

We are now ready for the definition of constraint automata with memory cells describing operationally the behavior of a Reo connector.

Definition 3.3. *A constraint automaton with memory is a tuple* $(S, I, O, H, M, \longrightarrow, S_0)$ *where* S *is a finite set of states with* $S_0 \subseteq S$ *the set of initial states,* $I, O, H \subseteq \mathbb{P}$ *are sets of ports known by the automaton,* $M \subseteq \mathbb{M}$ *is the set of memory cells, and* $\longrightarrow$ *is a transition relation with* $s \xrightarrow{N,\phi} s'$ *denoting a transition from a state* s *to* s' *synchronizing a set of ports* $N \subseteq I \cup O \cup H$ *under the data constraint* ϕ*. We assume that the ports appearing in* ϕ *are a subset of* N *and the memory cells occurring in* ϕ *are a subset of* M*.*

An *execution* of a constraint automaton is described by means of infinite strings [55] in At^{ω}. An infinite string $\alpha \cdot w$ is an execution starting from the state s, denoted by $\alpha \cdot w \in E(s)$ if and only if there is a transition $s \xrightarrow{N,\phi} s'$ such that the following three conditions hold:

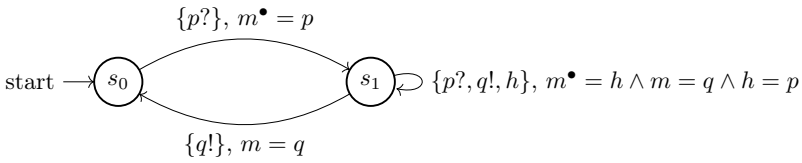
1. $\forall p \in I \cup O$, $p \notin N$ iff $\alpha(p) = \perp$;

2. $w = \alpha' \cdot w'$ and $\alpha\alpha' \models \phi$;

3. $w \in E(s')$

By the above definition, a constraint of a transition $s \xrightarrow{N,\phi} s'$ is evaluated in an execution $\alpha \cdot w$ starting from a state s with respect to its first two atoms. Furthermore, only the input and output ports in N fire, meaning that a value passes through them as recorded by α, and the rest of the string w is an execution of the target state s'.

Consider the following constraint automaton:

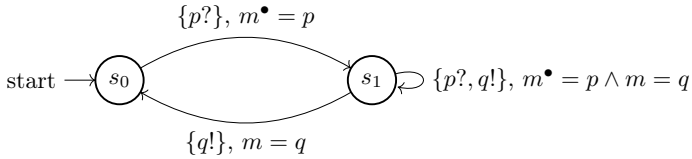


Here the suffixes "?" and "!" on a port are just syntactic means for indicating which ports belong to I and O respectively. The unmarked ports belong to H. An example of an execution of the above automaton starting from s_0 is the infinite string:

$$[p = 1, q = \perp, m = 22] \cdot [p = 3, q = 1, m = 1] \cdot [p = 5, q = 3, m = 3] \cdot$$
$$[p = \perp, q = 5, m = 5] \cdot [p = 7, q = \perp, m = 33] \cdot \ldots$$

Note that the value of the memory of the second element of the string is equal to the value at port p of the first element, and the value of port q of the second element. Similarly for the value of p in the second element and the value of q and the memory m in the third element.

The above automaton has the same executions from the initial state as the following automaton without hidden ports.



While in general it is not always possible to remove all hidden ports without modifying the set of executions, for simplicity and when there is no problem, in the sequel we will simplify a constraint automaton by removing hidden ports and obtaining an automaton with the same structure (states and transitions) and the same executions from its initial state.

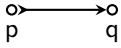
The language of a constraint automaton consists of the projection with respect to the input and output ports of all executions starting from the initial state. It represents the behavior of the automaton as visible from the environment. As such, only input and output ports are visible, but not hidden ports or memory cells. For example, the language accepted by the above two constraint automata is the same (i.e., they are language equivalent) and it includes the following infinite

string

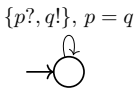
$$[p = 1, q = \perp] \cdot [p = 3, q = 1] \cdot [p = 5, q = 3] \cdot [p = \perp, q = 5] \cdots .$$

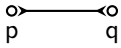
3.2.2 Basic and more complex connectors

In the following table, we associate constraint automata with memory to a few basic Reo connectors here described by means of their usual graphical representation [5, 65].

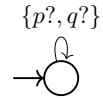


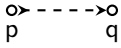
The *synchronous connector* accepts data from its input port p, and it passes synchronously to its output port q.



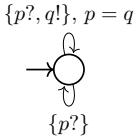


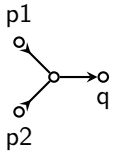
The *synchronous drain* has two input ports p and q, from which it accepts any data, but only when the two ports can be synchronized. The data received as input is not important, only ports' synchronization matters.



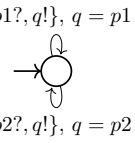


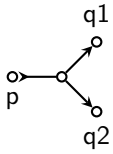
The *lossy synchronous connector* accepts data from its input port p, it either passes them synchronously to its output port q, or loses them in the channel without any reason.



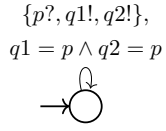


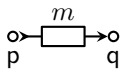
The *non-deterministic merger* receives data from either its input ports $p1$ or $p2$ and sends it to the output port q synchronously. If data is available at both input ports, only one of them is chosen non-deterministically.



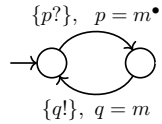


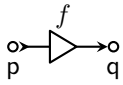
The *replicator connector* receives data from its input port p and replicates it to both output ports $q1$ and $q2$.



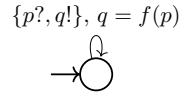


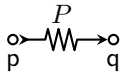
The *FIFO1 connector* receives data from the input port p if the internal buffer m is empty. The data is stored in the buffer, which can only contain at most one data item. When m is full its content flows to the output port q and it becomes empty. The behavior of a similar connector with dots inside the box is represented by the automaton with the other state as the starting state.



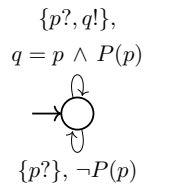


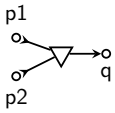
The *transformer connector* applies a function f to a data item received through its input port p, and synchronously offers the data resulting from evaluating $f(p)$ to it output port q.



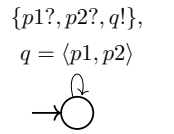


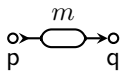
The pattern of a *filter connector* $P \subseteq Data$ specifies the type of data items that can be transmitted through the channel. Any value $d \in P$ is accepted through its input port p if the output port q can simultaneously dispense d; all data items that do not satisfy P are always accepted at the input p but they are immediately lost.



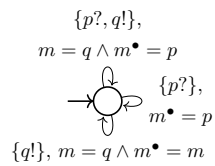


The *pairing connector* accepts two data items through its input ports $p1$ and $p2$, and synchronously output their pairing thorugh the port q.





The *variable connector* output the content of m at the port q and update m with the data at the input p. If only the input port is enabled then update m with the data received, while if only the output port q is enabled, then it sends the value of m to q and does not change m.



Note that the pairing connector is similar to a transformer but with two inputs

and using as a function the pairing $\langle -, - \rangle$. In all automata in the table, we assume that the ports known by each automaton are those used in the connectors.

We are now ready to extend the product of constraint automata given in the previous section so as to consider different types of ports, internal local memory, and restrictions of the ports used during synchronization. As before, the join operation guarantees synchronization and the same data flow at common ports but leaves the behavior at ports unknown to one of the automata unchanged.

Definition 3.4. *Assume* $S_1 \cap S_2 = \emptyset$, $M_1 \cap M_2 = \emptyset$, $H_1 \cap (I_2 \cup O_2 \cup H_2) = \emptyset$ *and* $H_2 \cap (I_1 \cup O_1 \cup H_1) = \emptyset$. *The product of the two constraint automata* $A_1 = (S_1, I_1, O_1,$
$H_1, M_1, \longrightarrow_1, S_{1,0})$ *and* $A_2 = (S_2, I_2, O_2, H_2, M_2, \longrightarrow_2, S_{2,0})$ *is defined as the following automaton:*

$$A_1 \bowtie A_2 = (S, I, O, H, M, \longrightarrow, S_0)$$

where $S = S_1 \times S_2$, $S_0 = S_{1,0} \times S_{2,0}$, $M = M_1 \cup M_2$ $I = (I_1 - O_2) \cup (I_2 - O_1)$, $O = (O_1 - I_2) \cup (O_2 - I_1)$, $H = (I_1 \cap O_2) \cup (I_2 \cap O_1) \cup H_1 \cup H_2$, *and* $\longrightarrow$ *is defined by the following rules:*

$$\frac{s_1 \xrightarrow{N_1, \phi_1}_1 t_1 \quad s_2 \xrightarrow{N_2, \phi_2}_2 t_2 \;\; \textit{and} \;\; Prt_1 \cap N_2 = Prt_2 \cap N_1}{\langle s_1, s_2 \rangle \xrightarrow{N_1 \cup N_2,\, \phi_1 \wedge \phi_2} \langle t_1, t_2 \rangle}$$

$$\frac{s_1 \xrightarrow{N_1, \phi_1}_1 t_1 \;\; \textit{and} \;\; Prt_2 \cap N_1 = \emptyset}{\langle s_1, s_2 \rangle \xrightarrow{N_1,\, \phi_1} \langle t_1, s_2 \rangle} \quad \textit{and} \quad \frac{s_2 \xrightarrow{N_2, \phi_2}_2 t_2 \;\; \textit{and} \;\; Prt_1 \cap N_2 = \emptyset}{\langle s_1, s_2 \rangle \xrightarrow{N_1,\, \phi_1} \langle s_1, t_2 \rangle}$$

Here $Prt_1 = I_1 \cup O_1$, *and* $Prt_2 = I_2 \cup O_2$.

Example 1. Figure 3.2 shows an example of a composition of a *variable* (on the left) on ports $\{A?, B!\}$ with a *FIFO1* connector (second automata from the left) acting on port $\{B?, C!\}$. The result is a new automaton with B as the hidden port (the last automaton), which however is the language equivalent to the automaton of a non-deterministic merger (in Figure 3.3 on the right) on ports $\{A?, C!\}$. Initially, *variable* stores a value of m_1, and $FIFO1$ starts with $m_2 = \perp$.

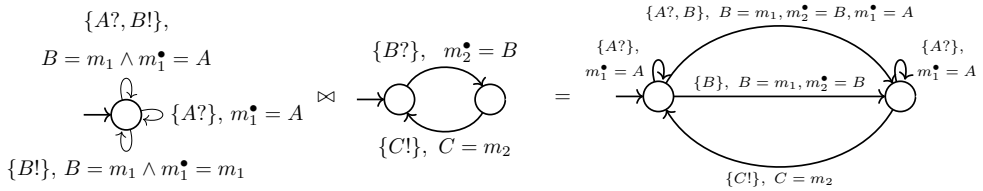


Figure 3.2: The composition of two automata

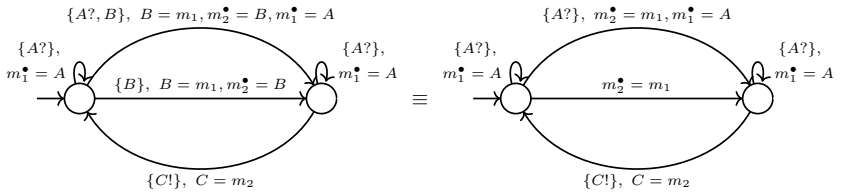


Figure 3.3: Removing hidden ports

Note that the port B is a hidden port in the resulting automaton because it is an output port of one connector and an input port of the other. It is not hard to see that the join operation is associative and commutative. The full conjunction process is as follows.

Assume the first automaton is named A_1, where the state in A_1 is 1, the transition above is $\mathfrak{t}_1$, the right transition is $\mathfrak{t}_2$, and the transition below is $\mathfrak{t}_3$. The second automaton is called A_2, the first state from left is named 2, the second state from left is 3, the transition from state 2 to state 3 is $\mathfrak{t}_4$, the transition from state 3 to state 2 is $\mathfrak{t}_5$, then we have the following information:

For A_1:

$I_1 = \{A\}$, $O_1 = \{B\}$, $Q_1 = \{1\}$, $q_1 = 1$, $Prt_1 = \{A, B\}$, $H = \emptyset$, M_1: $m_1 = v$

the transition $\mathfrak{t}_1$: $1 \xrightarrow{N_1, \phi_1} 1$, $N_1 = \{A, B\}$, ϕ_1: $B = m_1 \wedge m_1^\bullet = A$

the transition $\mathfrak{t}_2$: $1 \xrightarrow{N_2, \phi_2} 1$, $N_2 = \{A\}$, ϕ_2:$m_1^\bullet = A$

the transition $\mathfrak{t}_3$: $1 \xrightarrow{N_3, \phi_3} 1$, $N_3 = \{B, \}$, ϕ_3: $B = m_1 \wedge m_1^\bullet = m_1$

For A_2:

$I_2 = \{B\}$, $O_2 = \{C\}$, $Q_2 = \{2, 3\}$, $q_2 = 2$, $Prt_2 = \{B, C\}$, $H = \emptyset$, M_2: $m_2 = \perp$

the transition $\mathfrak{t}_4$: $2 \xrightarrow{N_4,\phi_4} 3$, $N_4 = \{B\}$, ϕ_4: $m_2^\bullet = B$
the transition $\mathfrak{t}_5$: $3 \xrightarrow{N_5,\phi_5} 2$, $N_5 = \{C\}$, ϕ_5: $C = m_2$

Applying to Definition 3.4, the composition of A_1 and A_2 is $(Q, I, O, H, M, \longrightarrow, q_0)$, where

$$Q = Q_1 \times Q_2 = \{\langle 1,2\rangle, \langle 1,3\rangle\},$$
$$I = (I_1 - O_2) \cup (I_2 - O_1) = (\{A\} - \{C\}) \cup (\{B\} - \{B\}) = \{A\} \cup \emptyset = \{A\},$$
$$O = (O_1 - I_2) \cup (O_2 - I_1) = (\{B\} - \{B\}) \cup (\{C\} - \{A\}) = \emptyset \cup \{C\} = \{C\},$$
$$H = (I_1 \cap O_2) \cup (I_2 \cap O_1) \cup H_1 \cup H_2 = (\{A\} \cap \{C\}) \cup (\{B\} \cap \{B\}) \cup \emptyset \cup \emptyset = \{B\},$$
$$M = [m_1 = v, m_2 = \perp].$$

The transitions are defined as follows:

▶ $\mathfrak{t}_1$ join with $\mathfrak{t}_4$:
$\because N_1 \cap Prt_2 = N_4 \cap Prt_1 = \{B\}$,
$\therefore \mathfrak{t}_1$ join with $\mathfrak{t}_4$: $\langle 1,2\rangle \xrightarrow{N_6,\phi_6} \langle 1,3\rangle$,
N_6 $(i.e., N_1 \cup N_4) = \{A, B\}$, $\phi_6 = \phi_1 \wedge \phi_4$: $B = m_1 \wedge m_1^\bullet = A \wedge m_2^\bullet = B$
▶ $\mathfrak{t}_1$ join with $\mathfrak{t}_5$:
$\because N_1 \cap Prt_2 = \{B\}$, $N_5 \cap Prt_1 = \emptyset$,
$\therefore \mathfrak{t}_1$ join with $\mathfrak{t}_5$: $\langle 1,3\rangle \xrightarrow{N_5,\phi_5} \langle 1,2\rangle$, $N_5 = \{C\}$, ϕ_5: $C = m_2$
▶ $\mathfrak{t}_2$ join with $\mathfrak{t}_4$:
$\because N_2 \cap Prt_2 = \emptyset$, $N_4 \cap Prt_1 = \{B\}$,
$\therefore \mathfrak{t}_2$ join with $\mathfrak{t}_4$: $\langle 1,2\rangle \xrightarrow{N_2,\phi_2} \langle 1,2\rangle$, $N_2 = \{A\}$, ϕ_2: $m_1^\bullet = A$
▶ $\mathfrak{t}_2$ join with $\mathfrak{t}_5$:
$\because N_2 \cap Prt_2 = \emptyset$, $N_5 \cap Prt_1 = \emptyset$,
$\therefore \mathfrak{t}_2$ join with $\mathfrak{t}_5$: $\langle 1,3\rangle \xrightarrow{N_2,\phi_2} \langle 1,3\rangle$, $N_2 = \{A\}, \phi_2$: $m_1^\bullet = A$,
$\langle 1,3\rangle \xrightarrow{N_5,\phi_5} \langle 1,2\rangle$, $N_5 = \{C\}, \phi_5$: $C = m_2$
▶ $\mathfrak{t}_3$ join with $\mathfrak{t}_4$:
$\because N_3 \cap Prt_2 = N_4 \cap Prt_1 = \{B\}$,
$\therefore \mathfrak{t}_3$ join with $\mathfrak{t}_4$: $\langle 1,2\rangle \xrightarrow{N_7,\phi_7} \langle 1,3\rangle$,
$N_7(i.e., N_3 \cup N_4) = \{B\}$, ϕ_7:$m_2^\bullet = B \wedge B = m_1 \wedge m_1^\bullet = m_1$
▶ $\mathfrak{t}_3$ join with $\mathfrak{t}_5$:

$\because N_3 \cap Prt_2 = \{B\},\ N_5 \cap Prt_1 = \emptyset,$
$\therefore \mathtt{t}_3$ join with $\mathtt{t}_5$: $\langle 1,3\rangle \xrightarrow{N_5,\phi_5} \langle 1,2\rangle,\ N_5 = \{C\},\ \phi_5 \colon C = m_2$

The list above has seven transitions after the conjunction, with regards to some transitions that are overlapping, we keep one of them and remove the others. For instance, the second transition of the list $A_1 \bowtie A_2$ appears in the fourth and last one, we only keep one transition as the result. Port B is removed in the final automaton since it becomes the hidden port.

Example 2. As another more complex example, in Figure 3.4 we introduce a three-input sequencer that regulates the flow of data from the input ports $p1, p2$, and $p3$, in sequential order, one after the other. Similar sequencers can be defined for any number of ports [41]. The connector is obtained by properly composing three synchronous drain connectors (connecting $p1$ with i_2, $p2$ with j_2, and $p3$ with k_2), one non-deterministic merger (connecting j_1 and j_3 with $j2$) two replicators (connecting i_1 with i_2 and i_3, and k_1 with k_2 and k_3), two FIFO1 connectors (one from i_3 to j_1, and another from j_3 to k_1), and finally a FIFO1 connector from k_3 to i_1 with a buffer initially storing a token data.

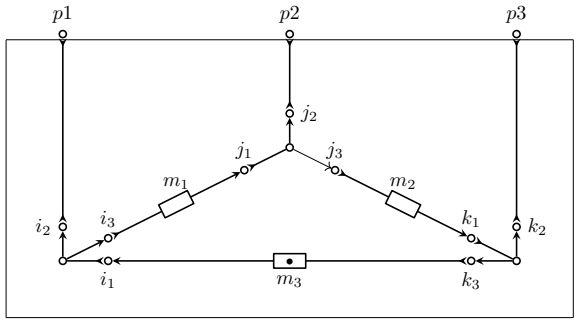
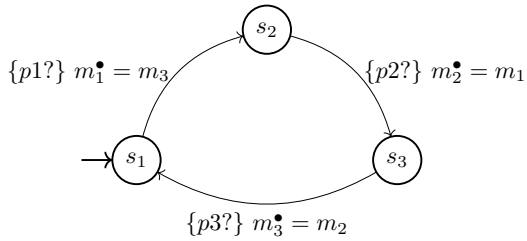


Figure 3.4: A three-input sequencer

Starting from the constraint automata of the basic connectors we have given above, and composing them according to Figure 3.4, we obtain the following three-state constraint automaton, where we have removed all hidden ports.


s_2
$\{p1?\}\ m_1^\bullet = m_3$
$\{p2?\}\ m_2^\bullet = m_1$
s_1
s_3
$\{p3?\}\ m_3^\bullet = m_2$

Chapter 4

A Reo Model of SDNs

In this chapter, we present a formal model of SDNs based on the Reo language. Using Reo we regard components in an SDN as constraints imposed on the interactions among the parties engaged in the processing of network packets. Starting with a small set of simple constraints, we obtain a declarative description of switches in the data plane as well as controllers in the control plane. The composition of these components is supported through other simpler connectors which give a global description of the topology of the network. Using the constraint automata semantics of Reo, the result is a compact finite state model for SDN particularly suited for formal verification, a direction that we will explore in the next chapter.

To scale up to handle large networks, our resulting SDN model is compositional in the sense that the meaning of the entire computer network is obtained by composing that of the individual models of the switches, network topology, and controllers. Furthermore, the resulting model is independent of the (possibly infinite) sequences of packets traversing the network.

4.1 Modeling the data plane

To begin with, we describe the switches of the data plane as Reo circuits, and we translate them into their corresponding constraint automaton. Then, we describe two examples of controllers managing a simple network with two switches. The goal is to send packets from one host to another. We conclude by combining the automata of these two layers with a network topology.

In the context of software-defined networking, a packet refers to a discrete unit of data that is transmitted over a network. It contains the actual data being sent, along with a header that contains the control information necessary for its routing and handling within the SDN infrastructure. We see a packet as a record $\pi : Fields \to Data$ assigning fields from a finite set of $Fields$ to data in $Data$. We denote a packet by $\pi = [f_0 = d_0, f_1 = d_1, ..., f_n = d_n]$, and use the notation $\pi.f$ to denote the value of the field f of the packet π.

We abstract from the concrete information contained in a packet, such as the source and destination IP addresses, protocol information, and source and destination ports. To cater to the latter, we assume that the set $Fields$ includes a field IPt for storing the identity of the input port of the switch where the packet is received and OPt for the output port of the switch where the packet is forwarded. This information is crucial for making forwarding decisions within the SDN network.

4.1.1 The Reo connector of a switch

As packets traverse the network, SDN switches perform specific actions on each packet based on its header information, such as forwarding the packet to specific ports or modifying its headers. Controllers can dynamically program these actions based on their global view of the network. Figure 4.1 introduces the Reo connector representing a switch with an interface consisting of input ports $\{P_0, P_1, ..., P_n\}$ and output ports $\{Q_0, Q_1, ..., Q_m\}$. Here both n and m are greater than or equal to 0 so that a switch has always at least two ports: P_0 and Q_0. Port P_0 is used to receive messages from the controller (or controllers) supervising the switch, whereas port Q_0 is meant for sending packets to the controller. All other ports are connected to other switches or open to the environment for communication with hosts. The input ports receive packets, and the output ports send packets.

We can describe the behavior of the Reo connector representing a switch using three scenarios.

1. The first one is when a packet π is received from a host or another switch. In this case, the input port receiving the packet is P_i for some $1 \leq i \leq n$. The transformer `AddIpt_i` of the channel connected to P_i assign $\pi.IPt$ to i and outputs to node A a triple $(FlowMsg, \pi, \emptyset)$. The first component of the triple is the tag $FlowMsg$ indicating that π is an ordinary network packet with no side effect on the flow table. The last component is the subset of

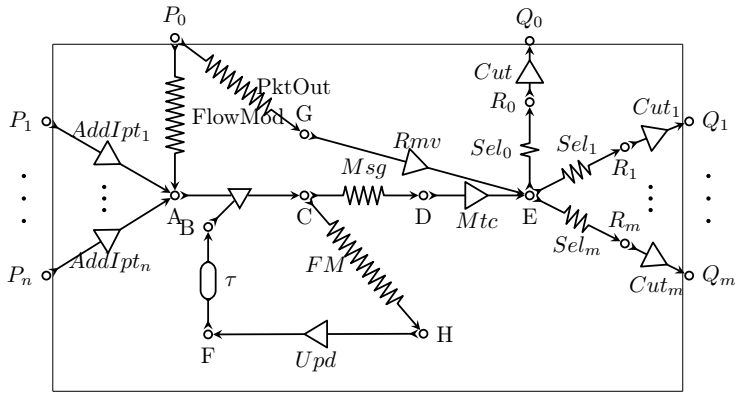


Figure 4.1: The Reo circuit of a switch

output ports of the switch where the packet needs to be forwarded. The above triple is paired with the current flow table stored in τ and received by the filters FM and Msg. These filters check the first component of the triple. In our case, only the filter Msg will succeed, and will pass the triple $(FlowMsg, \pi, \emptyset)$ together with the table τ to the transformer Mtc via node D. This transformer matches the packet π against the table τ, executes the corresponding field assignment modifying π into a new packet π' and outputs the pair (π', F) to node E. Here the set F contains all output ports where the packet π' needs to be forwarded, according to the action of the matching pair in the flow table τ.

The filters Sel_i regulate the forwarding by outputting the pair (π', F) to node R_i if $i \in F$. Note that the same pair may be duplicated to many nodes, and in case $F = \emptyset$ it will be dropped. Also, If $0 \in F$ then the packet is forwarded to the controller. From the node R_i the transformer Cut_i receiving as input the pair (π', F) will output the packet π', removing the information about the forwarding ports.

2. The second situation is when a $PktOut$ message from the controller is received at the input port P_0. A $PktOut$ message is a triple $\langle FlowMsg, \pi, F\rangle$ consisting of a tag $FlowMsg$ as in the previous case, a packet pi and a set of output ports F where π needs to be forwarded. Only the filter $PktOut$ lets this triple flow to the node G, where a transformer receives it, removes the tag, and outputs the pair (π, F) to node E. The selection and forwarding of π to each port in F are as before.

3. The third and last situation is when a $FlowMod$ message from the controller is received at the input port P_0. Also in this case it consists of a triple $\langle t, B, A\rangle$, but unlike the previous cases, this message is meant to update the table stored in τ. More specifically, B is a Boolean condition on $Fields$ matching the pair of τ to be updated, and A is the action for field updating and packet forwarding. The tag t can be either add, $remove$ or $modify$ to add (B, A) on top of table τ, remove the first pair (b, a) of τ with b implying B, or to modify the first pair (b, a) of τ with b implying B into the new pair (b, A). Note that in the case of $t = remove$, the action A does not play any role.

   Of the two filters with input at P_0 only the filter $FlowMod$ will succeed, so the triple $\langle t, B, A\rangle$ can be paired with the current flow table τ and reach node C. Here the filter Msg will fail but FM will succeed, passing all $\langle t, B, A\rangle$ and τ to the transformer Upd. This transformer will update the table τ as described in the triple $\langle t, B, A\rangle$, and will output a new table τ'. The latter is stored as the new current table by the variable channel with input node F.

4.1.2 Constraint automata for switches

While the Reo circuit of a switch may look complicated, its actual constraint automaton is rather simple. It consists of only one state (because all channels used have one single state) and three types of transitions (see Fig. 4.2).

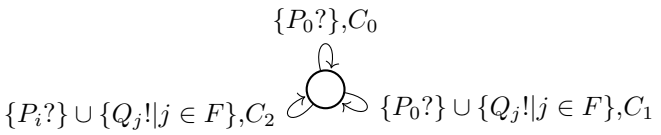


Figure 4.2: Constraint automaton of a switch

The conditions C_0, C_1 and C_2 are:

1. C_0: $P_0 = \langle t, B, A\rangle \wedge t \neq Msg \wedge \tau^\bullet = Upd(\langle \tau, P_0\rangle)$;
2. C_1: $P_0 = \langle Msg, \pi, F\rangle \wedge \bigwedge_{j\in F} Q_j = \pi$;
3. C_2: $Mtc(\langle \tau, \langle Msg, \pi[i/Ipt], \emptyset\rangle\rangle) = \langle \pi', F\rangle \wedge \tau^\bullet = \tau \wedge \bigwedge_{j\in F} Q_j = \pi'$.

Condition C_0 specifies when a $FlowMod$ message is received by a switch so that the flow table is updated. Transitions labeled by condition C_1 or C_2 depend

on the subset of output ports F received as input from P_0 or assigned after a matching action. This means there is a concrete transition for each possible subset of the output ports, but only one will eventually be chosen. Condition C_1 concerns $FlowMsg$ messages received by a controller, while condition C_2 defines the handling of a packet received from a host or another switch.

If we assume that in a switch the number of input ports is n and that the number of output ports is m, then the resulting constraint automata will have one state and $1 + 2^m + (n-1) * 2^m$ transitions.

Each switch in the data plane can be considered as a Reo connector interacting with others only via its input and output ports, while all other nodes and memory cells of the components are hidden. For example, while too large to depict here, the constraint automaton of the data plane composed of two simple switches connected by a synchronous channel as described in Figure 4.3 consists of one state, two memory cells (one for each switch flow table) and 26 transitions, which can be generated using automated tools [6].

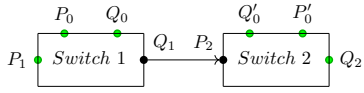


Figure 4.3: Two connected switches

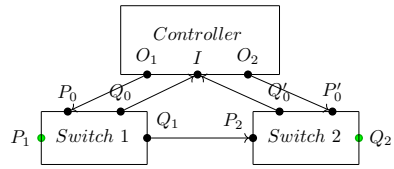


Figure 4.4: A controller and two switches

4.2 Modeling the control plane

The SDN control plane contains a set of controllers. Controllers are typically programmed using various programming languages to define network policies by handling events and configuring flow rules, e.g., matching criteria and actions. Each controller behaves as a reactive system, responding to `PktIn` messages received from switches by sending back either `PktOut` or `FlowMod` messages. We abstract from any full-fledged controller programming language and assume they are specified as Reo connectors, and thus with a behavior described using constraint automata. Input ports and output ports represent the connection of a controller with the switches under its control. Controllers can communicate with each other to allow for synchronization. Figure 4.4 shows a simple example of a controller with two switches.

We proceed with an example. The controller described in Figure 4.5 guarantees a flow of messages from the host connected to port P_1 to the host connected to port Q_2. It is connected to two switches through the output ports O_1 and O_2, and receives `PktIn` messages from the two switches via the input ports I. By chasing the circuit we see that the controller updates the flow table of both switches every time a new packet is received by switch 1 that does not match any condition of the table. The topmost sequencer regulates first the sending of a `FlowMod` message to O_1, then to O_2 and finally, it allows for the sending of the corresponding `PktOut` to O_1.

The second controller shown in Figure 4.6, has a similar specification: it allows for flowing a packet from P_1 to Q_2, but each time it reacts to incoming `PktIn` messages by updating the flow tables of both switches without waiting to receive a `PktIn` message from the second switch.

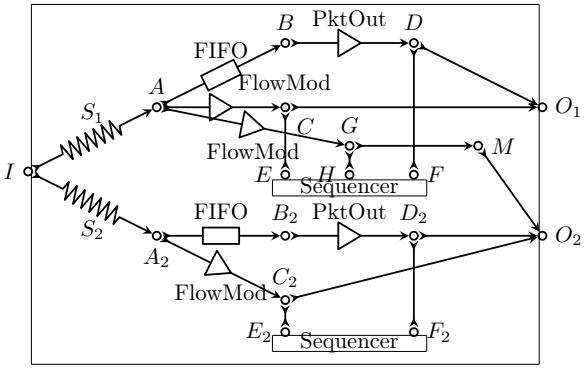


Figure 4.5: Reo circuit of controller 1

We combine the constraint automata of each controller, of the network topology, and all the switches to get a complete model of an SDN. Typically, the rate of a controller to receive messages from a switch is higher than the time needed by the controller itself to process the message and react accordingly. Therefore we use a **Queue** channel between the output ports of each switch and the input port of the controller (instead of synchronous channels $\{Q_0, I\}$ and $\{Q_0', I\}$ in Fig. 4.4). For the connections between switches and from controller to switches we use synchronous channels, but, of course, other channels with delay could be easily used instead. The Reo queue connector and its associated constraint automata with memory are described below.

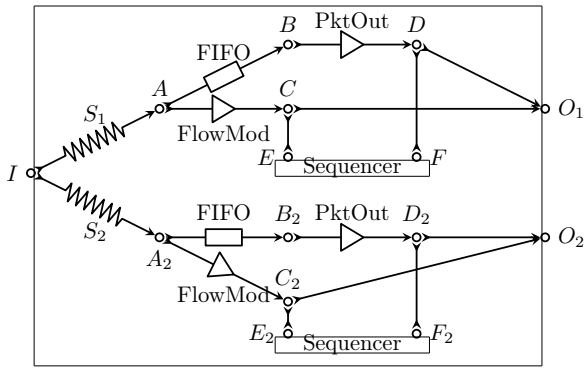
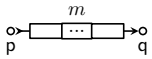
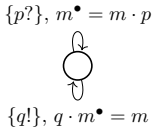


Figure 4.6: Reo circuit of controller 2



The Reo *Queue* connector behaves as a FIFO1, but it has an unbounded internal buffer m. As such, data can always be received from the input port p and stored in the buffer. If the buffer is non-empty then the first element received by p flows from the buffer to the output port q.



Both Reo circuits of the controllers described in the example above guarantee packets flowing from one host to another, but they are implemented differently and their automata are language-distinguishable. For example, when controller 1 receives a `PktIn` message, it sends a `FlowMod` message to switch one and another `FlowMod` to switch two so that a packet can pass the second switch directly without needing to wait for the table to be updated. Controller 2 however, every time receives a `PktIn` message, sends a `FlowMod` message only to the switch from which it received the message, with a consequence the updating of the flow tables of each switch happens only when a packet passes through it.

The constraint automata for controllers in Figure 4.5 and Figure 4.6 are shown in Figure 4.7 and Figure 4.8, respectively. Both automata start from the initial state 1 and always move to the state 2 when they receive a `PktIn` message from the first switch. Receiving a `PktIn` message from the second switch changes the state to 5 to the first controller and 4 to the second controller. They send either one or two `FlowMod` messages to update the table of one or two switches, and

then both send `PktOut` messages to let the packet continue its flowing to the host. After that, both automata move back to the initial state and are ready to react to new incoming messages.

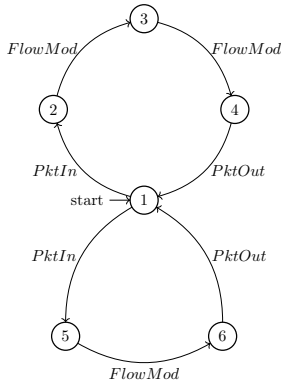


Figure 4.7: Constraint automaton for controller 1

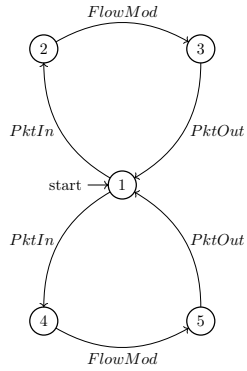


Figure 4.8: Constraint automaton for controller 2

4.3 SDN models for two controller algorithms

In this section, we combine the data plane and control plane (as in 4.1 and 4.2) together with the channel *Queue*. Since we have two different models of the controller (in Figure 4.5 and Figure 4.6), below shows two algorithms for each controller.

Network :

```
        Swith: S1, S2
        Controller: C1
        Connection: S1.Q1 -> S2.P2
        In_port: S1.P1
        Out_port: S2.Q2

Controller 1:
        Def PktIn(pkt){
        IF  pkt.inport = In_port(S1)
            Out = Out1; SwtP = Q1;
            Match = "tcp_dst = pkt.tcp_dst";
            Action1 = "Fwd(Q1)";
            Action2 = "Fwd(Q2)";
            FlowMod (<Add, Match, Action1>) to Out1;
            FlowMod (<Add, Match, Action2>) to Out2;
        ELSE
            Out = Out2; SwtP = Q2;
            Match = "tcp_dst = pkt.tcp_dst";
            Action1 = "Fwd(SwtP)";
            FlowMod (<Add, Match, Action1>) to Out;
        FI
        }

        PktIn (pkt);
        PktOut (Msg, pkt, Action1) to Out
```

The constraint automaton of the whole SDN model by compiling Controller 1 is in Figure 4.9. In the automaton, each state has five loop transitions with conditions:

1. $\{P_1\}$, packet enter into the Queue of switch 1;
2. $\{P_1\}$, packet enter into the Queue of switch 2;
3. $\{P_1\}$, packet drop in the switch 1;
4. $\{P_1\}$, packet drop in the switch 2;
5. $\{P_1, Q_2\}$, packet pass through the switch 1 and switch 2.

The initial state 1 chooses either switch 1 or switch 2 for receiving `PktIn` message, then the controller sends a `FlowMod` message to the chosen switch for installing a certain rule in it, after sending another `FlowMod` message to the alternative switch, the controller sends a `PktOut` message to the chosen switch. For

```
Network:
        Swith: S1, S2
        Controller: C1
        Connection: S1.Q1 -> S2.P2
        In_port: S1.P1
        Out_port: S2.Q2

Controller 2:
        Def PktIn(pkt){
        IF  pkt.inport = In_port(S1)
            Out = Out1; SwtP = Q1;
        ELSE
            Out = Out2; SwtP = Q2;
        FI
        }

        PktIn (pkt);
        Match = "tcp_dst = pkt.tcp_dst";
        Action = "Fwd(SwtP)";
        FlowMod (<Add, Match, Action>) to Out;
        PktOut (Msg, pkt, Action) to Out
```

example, if the first `PktIn` message comes from switch 1, the specific conditions of this automaton are:

1. `PktIn`: $q_1 = q_1^{\bullet} \cdot PktIn$;
2. `FlowMod`: $\tau_1^{\bullet} = Upd(\langle \tau_1, \langle t, b, Act_1 \rangle \rangle), t \neq Msg$;
3. `FlowMod`: $\tau_2^{\bullet} = Upd(\langle \tau_2, \langle t, b, Act_2 \rangle \rangle), t \neq Msg$;
4. `PktOut`: $Q_2 = \pi \wedge PktOut = \langle Msg, \pi, Q_2 \rangle$;

To note q_1 means the memory of Queue in switch 1, τ_1 means the flow table in switch 1, τ_2 means the flow table in switch 2, q_2 means the memory of Queue in switch 2. If the first `PktIn` message comes from switch 2, the specific conditions of this automata are:

1. `PktIn`: $q_2 = q_2^{\bullet} \cdot PktIn$;
2. `FlowMod`: $\tau_2^{\bullet} = Upd(\langle \tau_2, \langle t, b, Act_2 \rangle \rangle), t \neq Msg$;
3. `PktOut`: $Q_2 = \pi \wedge PktOut = \langle Msg, \pi, Q_2 \rangle$.

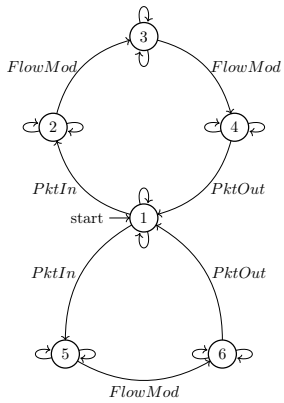


Figure 4.9: The automatons for the SDN with controller 1

The constraint automaton of the whole SDN model by applying controller 2 is in Figure 4.10.

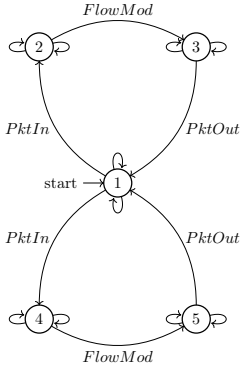


Figure 4.10: The automatons for the SDN with controller 2

Similar to the first algorithm, each state in the Fig 4.8 has five loop transitions, the conditions of these loops are the same, the difference is controller only installs one rule in the flow table of the switch each time when it receives a `PktIn` message. If the first `PktIn` message comes from switch 1, the specific conditions of this automaton are:

1. `PktIn`: $q_1 = q_1^{\bullet} \cdot PktIn$;
2. `FlowMod`: $\tau_1^{\bullet} = Upd(\langle \tau_1, \langle t, b, Act_1 \rangle \rangle), t \neq Msg$;
3. `PktOut`: $PktOut = \langle Msg, \pi, SwtP \rangle$;

If the first `PktIn` message comes from switch 2, the specific conditions of this automaton are:

1. `PktIn`: $q_2 = q_2^\bullet \cdot PktIn$;
2. `FlowMod`: $\tau_2^\bullet = Upd(\langle \tau_2, \langle t, b, Act_2 \rangle \rangle), t \neq Msg$;
3. `PktOut`: $Q_2 = \pi \wedge PktOut = \langle Msg, \pi, Q_2 \rangle$.

It is easy to see these two algorithms are very similar, both in program and automata. However they are distinguishable in the language, e.g., for the loop above in these two automata, algorithm 1 has two $FlowMod$s in sequence, but algorithm 2 has only one $FlowMod$ between $PktIn$ and $PktOut$.

4.4 Related work

The recent interest in the application of formal methods to software-defined networks started with VeriCon [14], an interactive verification system based on first-order logic to model admissible network topologies and network invariants. Similar to our model is a finite state machine model of SDN introduced in [113]. In this work model checking is possible via a translation to binary decision diagrams, under a similar assumption to ours: controllers are described as finite-state machines. Our approach however is based on a declarative description of both controllers, switches, and network topology as a Reo circuit, that is automatically, and compositionally, translated into a finite automaton.

Different than our declarative approach, [2] proposes an actor-based modeling to verify concurrent features of SDN via the ABS tool suite. The use of automata in our work instead of actors makes it easier to specify real-time and other quantitative properties of SDN. We do not explore this direction in this paper, but we leave it for future work. Variation of regular expressions has been very successful in modeling network programming languages [90, 100, 4]. In particular, NetKAT offers a sound and complete algebraic reasoning system with an interesting coalgebraic decision procedure. However, NetKAT only models a stateless snapshot of the data plane traversed by a single packet. There is no update of flow tables and no multiple packages are possible. Also, TLA+ [72] has been used to model the behavior of SDN but in a very restrictive manner allowing only a single switch [61].

Formal models are used not only to verify properties of an SDN such as consistency of flow tables, violation of safety policies, or forwarding loops, but also for

finding flaws in security protocols using CSP and the model checker PAT [110].

4.5 Conclusion

In this chapter, we presented a Reo model of SDN, based on a novel semantics for constraint automata with memory, recently studied in [57]. The difference is in a neater treatment of the values in the memory before and after the execution of a transition. The model is stateful and allows concurrency at the level of controllers but also at the level of the packets. The model can immediately be used for verification of quantitative and qualitative properties of SDN, such as consistency of flow tables, violation of safety policies, or forwarding loops. In the next chapter, we show how this model can be used for verification through a translation of Reo into the language Promela of the model checker Spin. In the future, we plan to verify these properties by using tools like ReoLive [29], or mCRL2 [65], which are part of the Reo framework [89]. Another line of research easily supported by our model is the development of simulation and visualization tools for packets flowing into the network.

Chapter 5

Implementing Reo into Promela

In this chapter, we study a subclass of constraint automata with local variables. The fragment denotes an executable subset of constraint automata for which synchronization and data constraints are expressed in an imperative guarded command style, instead of a relational style as in ordinary constraint automata. To demonstrate the executability property, we provide a translation scheme from symbolic constraint automata to Promela, the language of the model checker Spin. As a proof of concept, we model in Reo a software-defined network circuit, translate it into Promela, and use the Spin model checker to verify that our model satisfies some basic temporal properties.

5.1 A short introduction to Promela

Promela is a formal language widely used to specify concurrent systems and is supported by Spin, a Linear Temporal Logic (LTL) model checker [48, 49]. Theoretically, any LTL formula φ can be converted into a Büchi automaton [40, 104], as well as the negative of φ. To verify if a system satisfies an LTL property φ, we first construct an automaton A for the system, then compute the synchronous product of the Büchi automaton $BA(\neg\varphi)$ and A. If the language of this product is empty, then we called the original system to satisfy φ, otherwise φ is not satisfied with the system, and the counter-examples will be provided, too.

A Promela program is composed of a set of processes, each processes run concurrently and interact through shared channels. Both synchronous and asynchronous communication between processes is supported by its constructs since the modeling and analysis of processes, communication channels, and synchronization primitives are granted. For synchronous communication, a channel works in a rendezvous mode with no buffer (zero capacity), for asynchronous communication, channels work as a Fifo buffer with a non-zero user-specified capacity. All of these features make Promela suitable for verifying the correctness of complex systems, including hardware and software.

One of the most powerful model checkers for Promela is SPIN, SPIN can exhaustively explore all possible system behaviors and check various properties, such as safety, liveness, and temporal logic properties automatically. In a nutshell, each Promela process is transformed by Spin into a finite state automaton, processes are then synchronized into a single system automaton. Similarly, linear temporal properties expressed in the usual LTL syntax, are transformed into finite state automata. The automata representing both the Promela program and the LTL properties are exploited by Spin to verify and assert satisfaction of the properties [48, 49].

The syntax of LTL are:

$$\top \mid \neg\varphi \mid a \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid X\varphi \mid \varphi_1 U \varphi_2 \mid \Diamond\varphi \mid \Box\varphi$$

The property φ could be **true**, **false**, an **atom**, **or**, **and**, **next**, **until**, **eventually**, **always**. The semantics are:

$\alpha \models \top$	iff	α is true
$\alpha \models \neg\varphi$	iff	$\alpha \not\models \varphi$
$\alpha \models a$	iff	$a \in \alpha[0]$
$\alpha \models \varphi_1 \vee \varphi_2$	iff	$\alpha \models \varphi_1$ or $\alpha \models \varphi_2$
$\alpha \models \varphi_1 \wedge \varphi_2$	iff	$\alpha \models \varphi_1$ and $\alpha \models \varphi_2$
$\alpha \models X\varphi$	iff	$\text{suffix}(\alpha, 1) \models \varphi$
$\alpha \models \varphi_1 U \varphi_2$	iff	$\exists j \geq 0$, $\text{suffix}(\alpha, j) \models \varphi_2$ and $\text{suffix}(\alpha, i) \models \varphi_1$, $\forall 0 \leq i < j$
$\alpha \models \Diamond\varphi$	iff	$\exists i \geq 0$, $\text{suffix}(\alpha, i) \models \varphi$
$\alpha \models \Box\varphi$	iff	$\forall i \geq 0$, $\text{suffix}(\alpha, i) \models \varphi$

where $\alpha = \alpha[0]\alpha[1]\alpha[2]\ldots$ is an infinite sequence of states.

5.2 Symbolic constraint automata

In this section, we give an idea of symbolic constraint automata, and show how the composition method of Reo applies to the symbolic constraint automata.

We have seen in Chapter 3 that Reo circuits are usually specified using constraint automata [13]. In that case, each transition is labeled by synchronization and relational data constraints. To enhance the expressive power, we have considered an extension of constraint automata with memories. In both cases data constraints are declarative, and serve more as a specification than an implementation of an executable Reo circuits. In this section, we introduce a subset of constraint automata with transitions labeled by symbolic guarded actions. We show that, under some simple consistency conditions, symbolic constraint automata can be implemented and systematically translated into the Promela language. In fact, most of the original basic Reo connectors can be modeled as symbolic constraint automata, with only exceptions filter channels with predicates that constraint output ports and transformer channels that update input ports.

The basic building blocks of a symbolic constraint automaton include a finite set $\mathbb{D}$ of data ranged over by d, and a set $\mathbb{V}$ of variables, ranged over by x, y and z. We use variables to denote Reo ports shared between a connector and its environment, with different read and write permissions. An input port is a variable that the environment writes to (i.e., *put* a value into it), and from which a Reo connector destructively reads (i.e., *take* a value from it). Symmetrically, an output port is a variable that a connector writes to (i.e., *put*) and from which the environment destructively reads (i.e., *take*). Local variables are internal to a channel and can be used to store data. An assignment of variables to values is a function $\sigma{:}\mathbb{V} \to \mathbb{D}$. We range over input, output, and local variables by i, o, and v, respectively.

To abstract from concrete actions, we use function symbols (ranged over by f) and predicate symbols (ranged over by P). As usual, each function symbol f comes equipped with an arity and coarity, i.e. the number of arguments it expects and it returns, respectively. Similarly, predicate symbols come with an arity. We assume the natural interpretation of such function and predicate symbols, as executable function on $\mathbb{D}^n \to \mathbb{D}^m$ and as a decidable subset of $\mathbb{D}^n$, respectively. To simplify the notation, we identify predicate and function symbols with their interpretations. Here, n is the arity and m the coarity. Syntactic substitution in a term t of every occurrence of variable x for a term t_x is denoted as usual by

$t[t_x/x]$.

Terms of symbolic constraint automata are defined by the grammar:

$$t ::= d \mid x \mid f(\bar{t})$$

Terms denote tuples of data values. Here the (local or port) variable x denotes the data value it stores, while $\bar{t}$ is a shorthand notation for a finite sequence of terms $t_1, \ldots, t_n$, and $f(\bar{t})$ represents a tuple of values resulting from the computation f when executed with input values $\bar{t}$. The size of these tuples depends on the arity and coarity of f.

A *guarded action* α consists of a predicate and an assignment,

$$P(\bar{x}) \to \bar{y} := t\,.$$

We call $P(\bar{x})$ the guard of the guarded action α, and $\bar{y} := t$ the action that is executed when the guard is true. In general, we refer to the guard of a guarded action α by $g(\alpha)$, and the assignment at the right-hand side of α by $a(\alpha)$. We implicitly assume that the size of $\bar{y}$ corresponds to the coarity of t. To avoid problems with simultaneous assignments, and without loss of generality, only different variables may occur on the left-hand side of the assignment in an action. This way, for $z \in \bar{y}$, we can denote by $a(\alpha)_z$ the z-projection of the tuple of values resulting from evaluating the term $a(\alpha)$.

Given finite subsets I, O, and V of $\mathbb{V}$ denoting some input, output, and local variables, respectively, let $Act(I, O, V)$ be the set of actions α such that all variables occurring in $g(\alpha)$ are in $I \cup V$, all variables on the left-hand side of the assignment in $a(\alpha)$ are in $O \cup V$, and all variables on the right-hand side of the assignment in $a(\alpha)$ are either in V or occurring in $g(\alpha)$. The idea is that a guard $g(\alpha)$ constrains what value input may take, based on the current value of its local variables and pending values supplied by the environment. If the guard holds then the right-hand side of $a(\alpha)$ can be satisfied using values from the local variables and values given by the environment on its inputs. The result is assigned to the variables on the left-hand side of $a(\alpha)$ changing the local store and communicating the result of the computation to the environment via the output variables. Since output ports are only used to communicate a value to the environment, we assume no occurrence of them on the guard $g(\alpha)$ and in the term on the right-hand side of the assignment $a(\alpha)$. Dually, since input ports receive values only from the environment, we assume no occurrence of them on the left-hand side

of the assignment $a(\alpha)$. We denote by $I(\alpha)$ the set of all input ports occurring in α. Similarly, we denote by $O(\alpha)$ and $V(\alpha)$ the sets of output ports and local variables, respectively occurring on the left-hand side of the assignment in α. All input ports occurring in $a(\alpha)$ must appear in the input of the guard $g(\alpha)$. The test $x = x$ denotes a guard that is true, and an assignment $x := x$ denotes the skip action. We often do not write them in a transition unless the context requires it.

For example, the guarded action $(i \leq v \to (o, v) := [i, i])$ is an action in $Act(I, O, V)$, with $I = \{i\}, O = \{o\}$ and $V = \{v\}$. But the guarded action $(i \leq v \to (o, i) := [i, o])$ is not because i and o appear both on the left-hand side and on the right-hand side of the action, respectively. Here $[-, -]$ is the pairing function, with arity 2 and coarity 2, mapping two inputs into their corresponding pair of values.

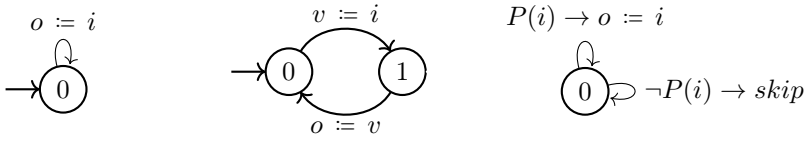
Definition 5.1. *A symbolic constraint automaton is a tuple* $(Q, q_0, I, O, V, \longrightarrow)$, *where*

- Q *is a finite set of states including the initial state* q_0,
- $I \subseteq \mathbb{V}$ *is a finite set of input ports,* $O \subseteq \mathbb{V}$ *is a finite set of output ports,* $V \subseteq \mathbb{V}$ *is a finite set of local variables such that they are mutually disjoint, i.e.,* $I \cap O = I \cap V = O \cap V = \emptyset$,
- $\longrightarrow$ *is a transition relation between states and labeled by actions in* $Act(I, O, V)$.

A transition $q \xrightarrow{\alpha} q'$ denotes the possibility of executing the action α from the state q and moving to the state q'. In order for the actual execution of α to take place, the guard of the action α must hold upon evaluation in the current state. To simplify the notation, we will not write guards of actions that are always true. Note that, differently from [57] and Chapter 3, we do not need to specify pre- and post-values of a local variable, as our actions are imperative and thus the order is implicitly given by the assignment operator.

In Figure 5.1, we show three symbolic constraint automata. The one on the left has no internal state, and the data received at the input port i is synchronously passed to the output o. This connector corresponds to the *synchronous channel* in Reo. The one in the middle has three variables: one input variable i, one output variable o, and an internal variable v. The automaton has two states: state 0 to indicate that the internal variable can be rewritten (i.e., the buffer is empty), and

state 1 to indicate that it cannot (i.e., the buffer is full). The connector assigns to v the value taken from i if it is in the empty state 0, and puts to the port o the value from v if it is in the full state 1. The two states symbolic constraint automaton is simpler than the equivalent single-state automaton (see Figure 5.2), as it avoids guards on the internal variable and the use of an extra special value $\perp$ to indicate that a variable is 'empty'. This connector corresponds to the *Fifo1 channel* in Reo. Finally, the rightmost automaton has a non-trivial guard labeling each of its two transitions. If the predicate P holds when a value is available at an input port i, then the connector behaves like a synchronous connector and passes the input value to the output port o. Otherwise, $\neg P$ holds on the value of i and the value is taken from i and lost, meaning that the component waiting for synchronization on port i is released.



(a) Synch connector (b) Fifo 1 connector (c) Filter connector

Figure 5.1: Three examples of symbolic constraint automata

An execution of a symbolic constraint automaton $(Q, q_0, I, O, V, \longrightarrow)$ is given in terms of an infinite sequence $(\sigma_i)_{i\in\mathbb{N}}$ of assignments of values to variables for which we can find an infinite sequence of states $(q_i)_{i\in\mathbb{N}}$ starting from the initial state q_0 and such that for all $n \geq 0$ there is a transition $q_n \xrightarrow{\alpha} q_{n+1}$ satisfying the following three conditions:

1. the interpretation of the guard $g(\alpha)$ holds in the assignment σ_n,

2. for all $x \in O(\alpha) \cup V(\alpha)$ the value $\sigma_{n+1}(x)$ is the x-projection of the evaluation of the variables on the left-hand side of the assignment in $a(\alpha)$ in the state σ_n, that is $\sigma_{n+1}(x) = \sigma_n(a(\alpha)_x)$;

3. for all variables not involved in the guarded action α, the value does not change, that is, $\sigma_{n+1}(y) = \sigma_n(y)$ for all $y \in (I \cup O \cup V) \setminus (I(\alpha) \cup O(\alpha) \cup V(\alpha))$.

Consecutive assignments σ_n and σ_{n+1} in a sequence represent the change of the internal and observable state of the system. The above conditions guarantee that

the guard must hold on to the values assigned to input variables in the current state before an action is taken, and that, after the execution of an action, the output variables and the local variables involved in the action change according to the action taken. The last condition guarantees that only variables that occur freely in an action get modified (for example by the environment) when executing that action. Note that after the execution of an action, input variables in $I(\alpha)$ change to a new value assigned by the environment. Dually, the value assigned to an output variable before the execution of an action is presumably taken by the environment before the new output value, assigned by the action, overwrites it. In other words, a transition in symbolic constraint automata, likewise constraint automata, assigns values to its output and local variables while being constrained on input and local variables. Also, the environment is allowed to change the values assigned to other variables not declared in the automaton, i.e., any variable in $\mathbb{V} \setminus (I \cup O \cup V)$. This represents the effect of an independent action executed by the environment in parallel with the automaton.

Local variables are used to store externally unobservable information. Only communication via input and output ports should be observable. Therefore, we define the semantics of a symbolic constraint automaton as the set of all possible executions $(\sigma_i)_{i \in \mathbb{N}}$ defined as above, but projected only on their input and output variables. As usual, two automata are then equivalent if they have the same semantics, i.e. they generate the same set of finite traces of assignments of input and output variables.

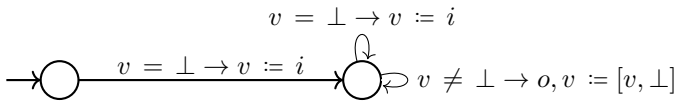


Figure 5.2: An equivalent symbolic constraint automaton for the Fifo 1 connector

Consider the symbolic constraint automaton in Figure 5.1.(b). The following is an example of a sequence of assignments recognized by that automaton:

$$[i = 1, o = 0, v = 0][i = 2, o = 0, v = 1][i = 2, o = 1, v = 1][i = 3, o = 1, v = 2] \cdots$$

Initially, the automaton is in state 0, for example, with value 1 on the input port i. The values of the two other variables do not matter at this point, and can be seen as previous values that remained stored but not accessible. By taking the transition to the state 1, the connector assigns the value of i to the internal variable v. The output port o is

blocked and cannot be changed while executing this transition, while the input port is free and here is assumed to get the value 2 from the environment. When taking the next transition the content of the variable v is put in the port o, the input is blocked and the cycle can start again. For each such a sequence we can find an equivalent sequence for the automaton in Figure 5.2, by using the extra value $\perp$ to check if the buffer is empty

$$[i = 1, o = 0, v = \perp][i = 2, o = 0, v = 1][i = 2, o = 1, v = \perp][i = 3, o = 1, v = 2] \cdots$$

Conversely, for every sequence representing the behavior of the automaton in Figure 5.2 we can find an equivalent sequence of assignments for the automaton in Figure 5.1.(b) by copying the previous value of v instead of $\perp$, and assigning an arbitrary initial value for v. In other words, the two automata are equivalent. Note that the initial state of the automaton in Figure 5.2 forces the connector to start with an empty buffer. Without this initial transition, the two automata would not be equivalent as one could start to output on port o the value stored in v.

The central operation on symbolic constraint automata is synchronization via their shared ports which are input ports for one automaton and output ports for another. Shared ports become internal local variables in the automaton resulting from the composition. No other synchronization by shared variables is allowed, as local variables are only visible within the scope of a connector. Our definition is similar in spirit to that of [57], but, in addition to that work, our symbolic constraint automata could be automatically translated to Promela. The explicit input and output variables, and the guarded command structure on the label impose some prerequisites on the product to avoid inconsistencies. In fact, we define composition only for pairs of symbolic constraint automata A_1 and A_2 such that (1) no local variables are in common, and (2) for every pair of actions α_1 and α_2 of the two automata, they synchronize only on some input ports used by one action and some output ports used by the other, but not on both input and output ports at the same time. More formally, we assume that, for all actions α_1 labeling a transition in A_1 and α_2 labeling a transition in A_2, the following holds

$$I(\alpha_1) \cap O(\alpha_2) \neq \emptyset \Rightarrow O(\alpha_1) \cap I(\alpha_2) = \emptyset$$

or, equivalently,

$$O(\alpha_1) \cap I(\alpha_2) \neq \emptyset \Rightarrow I(\alpha_1) \cap O(\alpha_2) = \emptyset$$

We call two automata with these two properties *consistent*. The intuition behind synchronizing two guarded actions α_1 and α_2 is that their data value should agree on their shared ports so that it can flow from the output of one to the input of the other actions. The above condition together with the fact that the two automata do not share local variables - and thus $V(\alpha_1) \cap V(\alpha_2) = \emptyset$ - in fact impose a causality in the execution of their actions as input is needed to update the internal state and to be passed to output

ports. Next, we define formally the synchronization of two guarded actions α_1 and α_2. Assume $I(\alpha_1) \cap O(\alpha_2) = \bar{u} \neq \emptyset$, and let

$$\alpha_1 = P_1(\bar{x}_1 \cup \bar{u}) \to \bar{y}_1 \coloneqq t_1(\bar{z}_1 \cup \bar{u}) \quad \text{and} \quad \alpha_2 = P_2(\bar{x}_2) \to \bar{y}_2 \cup \bar{u} \coloneqq t_2(\bar{z}_2)\,.$$

where $\bar{x}_1 \cup \bar{u}$ is the sequence of input and local variables occurring in P_1, $\bar{z}_1 \cup \bar{u}$ the sequence of input and local variables occurring in t_1, and $\bar{y}_2 \cup \bar{u}$ the sequence of output and local variables occurring at the left-hand side of the assignment $a(\alpha_2)$. Note that variables in $\bar{u}$ cannot occur in P_2 nor in t_2 as they are output variables for α_2. Similarly, they cannot occur in $\bar{y}_1$, as they are input variables for α_1. By definition of guarded action, they can occur in P_1. Under these circumstances, we can define the synchronization $\alpha_1 \otimes \alpha_2$ as the following guarded action

$$\alpha_1 \otimes \alpha_2 = P_1[t_{2_{\bar{u}}}/\bar{u}] \wedge P_2 \to \bar{y}_1, \bar{y_2} \cup \bar{u} \coloneqq t_1[t_{2_{\bar{u}}}/\bar{u}],\, t_2(\bar{z}_2).$$

Since the guard P_2 does not depend on output variables, we can evaluate it. If it holds we can then assign the values returned by t_2 to the output and local variables at the left-hand side of $a(\alpha_2)$. The values assigned to the shared output values $\bar{u}$ are then used in P_1 as constant replacing the input variables $\bar{u}$. If this predicate holds, then the same substitution is applied to t_1 so that we can compute the assignment. Note that $I(\alpha_1 \otimes \alpha_2) = (I(\alpha_1) \setminus O(\alpha_2)) \cup I(\alpha_2)$, $O(\alpha_1 \otimes \alpha_2) = O(\alpha_1) \cup (O(\alpha_2) \setminus I(\alpha_1))$, and $V(\alpha_1 \otimes \alpha_2) = V(\alpha_1) \cup V(\alpha_2) \cup (I(\alpha_1) \cap O(\alpha_2))$. The definition of $\alpha_1 \otimes \alpha_2$ for the symmetric case when $O(\alpha_1) \cap I(\alpha_2) \neq \emptyset$ is similar.

Definition 5.2. *Without loss of generality, let Q_1 and Q_2 be two disjoint sets of states, and V_1 and V_2 be two disjoint sets of local variables. The composition of two consistent symbolic constraint automata $A_1 = (Q_1, q_1, I_1, O_1, V_1, \longrightarrow_1)$ and $A_2 = (Q_2, q_2, I_2, O_2, V_2, \longrightarrow_2)$ is defined as the automaton $A_1 \bowtie A_2 = (Q, q, I, O, V, \longrightarrow)$ where:*

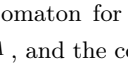
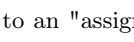
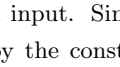
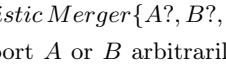
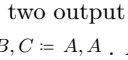
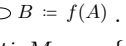
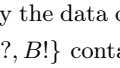
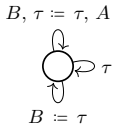
- $Q = Q_1 \times Q_2$,
- $q = \langle q_1, q_2 \rangle$,
- $I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$,
- $O = (O_1 \setminus I_2) \cup (O_2 \setminus I_1)$,
- $V = V_1 \cup V_2 \cup (I_1 \cap O_2) \cup (I_2 \cap O_1)$, *and*
- $\longrightarrow$ *is defined by the following rules:*

$$\frac{q_1 \xrightarrow{\alpha_1}_1 q_1' \text{ and } q_2 \xrightarrow{\alpha_2}_2 q_2' \text{ and } IO(\alpha_1) \cap IO(A_2) = IO(\alpha_2) \cap IO(A_1)}{\langle q_1, q_2 \rangle \xrightarrow{\alpha_1 \otimes \alpha_2} \langle q_1', q_2' \rangle}$$

$$\frac{q_1 \xrightarrow{\alpha_1}_1 q_1' \text{ and } IO(\alpha_1) \cap IO(A_2) = \emptyset}{\langle q_1, q_2 \rangle \xrightarrow{\alpha_1} \langle q_1', q_2 \rangle} \qquad \frac{q_2 \xrightarrow{\alpha_2}_1 q_2' \text{ and } IO(\alpha_2) \cap IO(A_1) = \emptyset}{\langle q_1, q_2 \rangle \xrightarrow{\alpha_2} \langle q_1, q_2' \rangle}$$

where $IO(A_i) = I_i \cup O_i$ *and* $IO(\alpha_i) = I(\alpha_i) \cup O(\alpha_i)$, *for* $i = 1, 2$.

Similar to the join operation on constraint automata the above synchronization operation on symbolic constraint automata synchronizes actions on shared ports and allows independent parallel behavior for actions with no shared ports. The composition of consistent symbolic constraint automata is symmetric and, when defined, associative. Here are some other examples of symbolic constraint automata in below.

The primary distinction between symbolic constraint automata and the constraint automata defined in Chapter 3 lies in the transition label. In the former, synchronization relies on the ports used in the guarded action, whereas the latter explicitly declares which ports must synchronize. Furthermore, symbolic constraint automata are imperative and not declarative, utilizing variables as memory. For example, the constraint automaton for the synchronous connector $Sync\{A?, B!\}$ is denoted as  , and the corresponding symbolic constraint automaton removes the ports and changes the "equal to" to an "assignment"  , explicitly declaring B as an output port and A as an input. Similarly, the Synchronous Drain connector $SyncDrain\{A?, B?\}$ is modeled by the constraint automaton  , ensuring both inputs are lost synchronously when data arrives at both ports. The corresponding symbolic constraint automaton is  , where the guard of the action is always true but involves the ports A and B, while the action is just a "skip." This way, input ports A and B are forced to synchronize, but their received values are lost. The connector $Non-deterministic\,Merger\{A?, B?, C!\}$ has one output port C that receives data from either input port A or B arbitrarily. The symbolic constraint automaton is  . The connector $Replicator\{A?, B!, C!\}$ receives the data from the input port A and copies it into the two output ports B and C. The symbolic constraint automaton representing it is  . A $Transformer\{A?, B!\}$ outputs on port B the data received at input A after applying a function f. The symbolic constraint automaton is similar to the one for the synchronous channel, except for the use of f in the action of its unique transition  . The connector $PairMerger\{A?, B?, C!\}$ looks similar to $Non-deterministicMerger\{A?, B?, C!\}$, except that the output ports C receive an ordered pair formed by the data of A and B, respectively:  . Finally, the connector $Variable\{A?, B!\}$ contains an internal variable τ, that can store input data from A that is available to the output port B. Note that reading from and writing to τ can happen separately or contemporaneously:  .

5.3 From Reo to Promela

Building on the work presented in [77], we translate Reo connectors into Promela programs. We use symbolic constraint automata as a specification of Reo connectors, and use the resulting Promela process as a protocol to coordinate messages exchanged through the ports of other processes.

5.3.1 Implementing Reo ports in Promela

In Promela, a Reo port is expressed as a structure, as shown in Listing 5.1. It contains two Promela channels of capacity one: a data and a trig channel. A port in Reo is directional. We call putter the component that puts an element on the port, and getter the component that gets an element from the port. Operations on a port are blocking unless both a put and a get are performed at the same time. In which case the port *fires*, and the data is forwarded from the putter to the getter.

The data channel in the Promela implementation of a Reo port is used to forward the data message from the putter to the getter, while the trig channel is used to synchronize the putter and the getter. The reason of using two channels of size one instead of a rendezvous channel of size zero is that it is impossible in Promela to query whether a process is currently waiting on a rendezvous channel. We will later see that querying the state of a port is necessary for the protocol to coordinate the boundary processes.

Listing 5.1: definition of a Reo port in Promela

```
1 typedef port {
2     chan data = [1] of {Data};
3     chan trig = [1] of {int}; }
```

As described in Listing 5.2, two actions can be performed on a port: put and take. The function call put(q, a) atomically fills the data channel of q with the datum a, and blocks on the trig channel, waiting to synchronize with the component on the output side of q. The integer variable x is used to get a value from the trig channel, and hence to synchronize to it; the actual value communicated does not matter.

The function call take(q, a) atomically notifies, by outputting on the trig channel, that there is a component willing to take data, and blocks on the data channel, until a datum can be read and stored into the variable a. The integer value of -1 written into the synchronization channel is arbitrary, as trig is used only for signaling.

Listing 5.2: put and take functions

```
1 inline put(q,a) {
2   int x;
3   q.data!a;
4   q.trig?x }
```

```
5
6  inline take(q,a) {
7    q.trig!-1; q.data?a }
```

We describe two temporal properties in Listing 5.3 that reflect the synchronous behavior of a port. We say that a port *fires* whenever a data is exchanged between the putter and the getter. If a port does not fire, it is *silent*. In the case of an implementation of a port with two buffers, the firing property occurs whenever a port has both an input and an output request, i.e. both channels are full. We then know, due to the definition of the put and take operations, that the putter and the getter will be released from blocking on the port, and the getter will get the value from the data channel. We define some macros in Listing 5.3 to encode firing and silent property of port p as an LTL property.

Listing 5.3: Macros for firing of ports

```
1  #define p_fires (
2      !(len(p.data) == 0) && !(len(p.trig) == 0) &&
3      X((len(p.data) == 0) || (len(p.trig) == 0)))
4
5  #define p_silent    (! p_fires)
```

Ports are not typed as input or output, as that depends on the component/connector that uses them. We have seen in the previous section that within a connector, a port used in a guard must be an input port, whereas a port used on the left-hand side of the assignment of an action is an output port.

5.3.2 Implementing Reo connectors in Promela

Next we describe how to implement Reo connectors expressed as symbolic constraint automata in Promela. A symbolic constraint automaton is encoded, in Promela, as a proctype. A Promela proctype has a name, a signature, and a body. The Spin model checker executes each proctype of its main concurrently, while taking into account blocking operations on channels. As we will see, input/output variables of a symbolic constraint automaton lead to shared port channels in Promela between the protocol and the boundary processes.

Let $(Q, q_0, I, O, V, \longrightarrow)$ be a symbolic constraint automaton. Input ports I and output ports O are passed as parameters to the Promela process resulting from the translation. As expected, local variables in V are declared locally to the Promela process, i.e., within theproctype body. For each port $p \in I \cup O$, a new local variable _p is declared so as to store the value taken if p is an input or passed if p is an output. We use mytype as a generic type for input, output, and memory variables.

Each state in Q is encoded as a special value for the state variable. The state variable therefore models the control flow between the states of the automaton. For simplicity,

and without loss of generality, here we assume that $q_0 = 0$ is the initial state and $Q = \{0, 1, \ldots, n\}$.

We use the Promela non-deterministic $\mathsf{do} - \mathsf{od}$ construct to model concurrent applications of transitions of a symbolic constraint automaton. The guard (respectively, the command) of the statements in the $\mathsf{do} - \mathsf{od}$ results from the translation of the guard (respectively, the command) in the action labeling the corresponding transition. As a result, each guard in the $\mathsf{do} - \mathsf{od}$ loop contains a clause that controls that variable state has the value corresponding to the pre-state, and updates the value to the post-state in the command. The Promela language allows for non-destructive reads of channel's value: the operation A.data? < _a> assigns to the variable _a the value stored in channel A.data without actually removing the values from that channel. However, in Promela, this operation cannot be executed in a guard within a $\mathsf{do} - \mathsf{od}$ statement. We circumvent this problem by using a control variable f_i for each transition $t_i \in \longrightarrow$ of the automaton. The variable f_i can take two values. By default, f_i is 1. If the synchronization constraint of the guard of the i-th transition is satisfied but the data constraint is not, then the value of f_i is set to 0. Every transition, if taken successfully, resets all the f_i to 1.

More formally, for a symbolic constraint automaton $(Q, q, I, O, V, \longrightarrow)$ we show in Listing 5.4 its translation to Promela:

Listing 5.4: Promela code generated for a symbolic constraint automaton

```
1  proctype SCA(port P̄){
2  ∀p ∈ P, mytype _p;
3  ∀v ∈ V, mytype v;
4  ∀f ∈ {f1,...fk}. bool f = 1;
5  int state=0;
6  do
7    :: transition 1;
8    ⋮
9    :: transition k;
10 od }
```

Here $Q = \{0, \ldots, n\}$, $q = 0$, $P = I \cup O$, $\overline{\mathsf{v}} = V$, and the remaining overlined variables are just consecutive sequences of them. For every (input or output) port $\mathsf{p} \in \overline{P}$ there is a variable _p associated with it on which we store the value communicated via the port. The control variable state is used to store the current state of the automaton (thus it ranges between 0 and n), and variables $\{\mathsf{f1}, \ldots \mathsf{fk}\}$ are used when evaluating predicates on values received at input ports. Here k is the number of transitions. Let $q \xrightarrow{\alpha} q'$ be the j-th transition of the symbolic constraint automaton, with $1 \leq j \leq k$, and remember that the input ports in α are $I(\alpha) = \{\mathsf{i1}, \ldots, \mathsf{im}\}$ the output ports are $O(\alpha) = \{\mathsf{o1}, \ldots, \mathsf{ol}\}$, and the local variables occurring in α are $\{\mathsf{v1}, \ldots, \mathsf{vh}\}$. Then transition j in the listing above is given by

```
state==q && fj==1 && ⋀_{i∈I(α)} Full(i.data) && ⋀_{o∈O(α)} Full(o.trig)
  -> Atomic { i1.data?<_i1>; ...; im.data?<_im>;
     if
       :: P(i1,...,im,v1,...vh) == True -> take(i1,_i1);...; take(im,_im);
          A(_i1,...,_im,v1,...vh, _o1,...,_ol);
          put(o1,_o1);...; put(ol,_ol);
          state=q';f1=1;...; fk =1;
       :: else -> fj=0;
     fi }
```

where P is a function encoding the guard of α and A is a function encoding the assignment action of α. When the control variables are not used in a transition (for example because there is no predicate on input variables), then it is possible to simplify the generated Promela code by removing such control variables. Similarly, when there is only one state, the variable state can be removed as its value is constant.

We list several examples of translation of symbolic constraint automata to Promela. Listing 5.5 gives the Promela code resulting from the translation of the symbolic constraint automaton in Figure 5.1.(a) that models the *Sync* connector.

Listing 5.5: Promela code generated for a Sync connector

```
proctype Sync(port A; port B){
mytype _a; mytype _b;//internal values in Sync
int state = 0; //initial state
do
:: state==0 && Full(A.data) && Full(B.trig)
   -> Atomic{take(A, _a); _b =_a; put(B,_b); state = 0};
od }
```

Here the parameter port A is the input port of the Sync channel, and port B is the output port. There are no local variables except those associated with the ports, and the control variable state that we will discuss later. The condition full(A.data)&&full(B.trig) is satisfied if and only if there is an ongoing put(A, a) operation on the input port A and an ongoing take(B, x) operation on the output port B. In this case, the Sync process atomically takes the data from port A, stores it in a local variable _a, executes the action of the associated transition of the symbolic constraint automata, i.e. _a = _b, and puts the value in port B. Of course, the single state automaton of the Sync connector could have been modeled by a much simpler Promela process without such an extra variable, but, as for the other connectors, we keep it for generality.

Listing 5.6 shows the Promela code for a two-state symbolic constraint automaton of the Fifo1 Reo connector given in Figure 5.1.(b).

Listing 5.6: Promela code generated for Fifo1 connector

```
proctype Fifo1(port A; port B){
mytype _a; mytype _b; int state = 0;
```

```
3  mytype v; //v is the buffer of the Fifo 1 connector
4  // _a,_b,v are the input, output and local variables, respectively
5  do
6  :: Full(A.data) && state==0
7     -> Atomic{take(A,_a); v=_a; state=1};
8  :: Full(B.trig) && state==1
9     -> Atomic{_b=v; put(B,_b); state=0};
10 od }
```

Besides an input and an output port, we also have a local variable v for the buffer of the Fifo1 connector. The control variable state is initially set to 0, corresponding to the initial state of the automaton.

This time, the do − od loop contains two transitions, one for each transition of the symbolic constraint automaton. One statement corresponds to the transition that moves the control from state = 0 to state = 1 if there is a pending data on the input port A. The value is then taken and assigned to the local variable v. The other statement corresponds to the transition that moves the control from state = 1 to state = 0 when there is a pending request at the output port. In which case, the stored value is forwarded to the output port B.

Next, we show how to translate the Filter connector of Figure 5.1.(c) that, unlike the other two examples, contains two transitions labeled with a non-trivial predicate on the input variable. The Promela code of the Filter connector is presented in Listing 5.7.

Listing 5.7: Promela code generated for Filter connector

```
1  proctype Filter(port A; port B){
2  mytype _a; mytype _b; int state=0; bool f1=1; bool f2=1;
3  do
4  :: state==0 && f1==1 && Full(A.data) && Full(B.trig)
5     -> Atomic{ A.data ? <_a>;
6       if
7         :: P(_a)==True -> take(A,_a); _b=_a; put(B,_b); state=0; f1=1; f2=1;
8         :: else -> f1=0;
9       fi }
10 :: Full(A.data) && state==0 && f2==1
11    -> Atomic{ A.data ? <_a>;
12      if
13        :: P(_a)==False -> take(A,_a); state=0; f1=1; f2=1;
14        :: else -> f2=0;
15      fi }
16 od }
```

Initially the control variables f1 and f2 associated with the two transitions are set to true, meaning that any transition can be potentially selected. As before, the satisfiability of each guard depends on the presence of some data at the input port A and the presence of some signal at the output port B. The predicate P is evaluated only after one of the two statements of the do − od loop is chosen. If true, the action of the transition is

taken, and the two control variables f1 and f2 are reset to true. Otherwise, the value of the associated control variable fi is set to false and the control goes back to the loop statement. In this way the i-th transition associated with fn will not be selected anymore, even if all other predicates in the guard of the statement are true (for $i = 1, 2$), which removes some undesirable livelocks.

We leave the detailed description of the encoding in Promela of a composition operator mimicking that of symbolic constraint automata but just give an example below corresponding to the composition of the Promela code generated for the Fifo1 and the Filter connectors in Listings 5.6 and 5.7, respectively.

Listing 5.8: Promela code generated by composing Filter(port A;port B) with Fifo1(port B;port C)

```
1  proctype FilterFifo1(port A; port C){
2  mytype _a; mytype _c;
3  mytype v; //v is the buffer of the Fifo 1 connector
4  mytype b; //b is a shared port that becomes a local variable
5  int state=0; //there are 1 x 2 states in total
6  bool f1=1; bool f2=1; bool f3=1; //there are 3 transitions in total
7  do
8  :: state==0 && f1==1 && Full(A.data)
9     -> Atomic{ A.data ? <_a>;
10      if
11        :: P(_a)==True -> take(A,_a); _b=_a; v=_b; state=1; f1=1; f2=1; f3=1;
12        :: else -> f1=0;
13      fi }
14 :: state==0 && f2==1 && Full(A.data)
15    -> Atomic{ A.data ? <_a>;
16      if
17        :: P(_a)==False -> take(A,_a); state=0; f1=1; f2=1; f3 =1;
18        :: else -> f2=0;
19      fi }
20 :: state==1 && f3==1 && Full(C.trig)
21    -> Atomic{_c=v; put(C,_c); state=0; f1=1; f2=1; f3 =1;};
22 od }
```

5.3.3 Other Reo connectors in Promela

In this part, we show more examples of standard Reo connectors encoded as symbolic constraint automata and translated to Promela. The symbolic constraint automata of these Reo connectors are shown in the end of Section 5.2, it is important to remark that the control variables f_i do not introduce fairness or priority among the transitions, they only control the flow so that Promela will not choose the same transition again with a guard that has already been evaluated to false. Below in Listing 5.9 shows the translation of SynchronousDrain, which has two input ports A and B, without any output ports. SynchronousDrain process automatically takes two data items from A and B when

the condition full(A.data)&&full(B.data) is satisfied, there is no data item be forwarded to any ports.

Listing 5.9: Promela code generated for Synchronous Drain connector

```
proctype Syncdrain(port A; port B){
mytype _a; mytype _b;
int state = 0;
do
:: Full(A.data) && Full(B.data) -> Atomic{take(A, _a); take(B, _b); state = 0;};
od }
```

The process of Non − deterministicMerger indicates two loops in do − od, it either chooses to forward the data from port A to C when full(A.data)&&full(C.trig) is satisfied, or choose to forward the data from port B to C when full(B.data)&&full(C.trig) is satisfied. The state has not been changed and remains to be state = 0. The code is presented in Listing 5.10.

Listing 5.10: Promela code generated for Non-deterministic Merger connector

```
proctype Merger(port A; port B; port C){
mytype _a; mytype _b; mytype _c;
int state = 0;
do
:: Full(A.data) && Full(C.trig)
   -> Atomic{take(A, _a); _c = _a; put(C, _c); state = 0;};
:: Full(B.data) && Full(C.trig)
   -> Atomic{take(B, _b); _c = _b; put(C, _c); state = 0;};
od }
```

Listing 5.11 denotes the Replicator connector in Promela code, the process executes replication of the input, then forward these two value to output port B and C if full(A.data)&&full(B.trig)&&full(C.trig) is satisfied.

Listing 5.11: Promela code generated for Replicator connector

```
proctype Replicator(port A; port B; port C){
mytype _a; mytype _b; mytype _c;
int state = 0;
do
:: Full(A.data) && Full(B.trig) && Full(C.trig)
   -> Atomic{take(A, _a); _b = _a; _c = _a; put(B, _b); put(C, _c); state = 0;};
od }
```

Transformer process has a user defined inline function f which is distinct from other processes. In Listing 5.12, f has a parameter x where x here indicates the value _a after ongoing operation take(A, _a), then be forwarded to output port B when the condition full(A.data)&&full(B.trig) is satisfied.

Listing 5.12: Promela code generated for Transformer connector

```
proctype Transformer(port A; port B){
mytype _a; mytype _b;
int state = 0;
do
:: Full(A.data) && Full(B.trig)
   -> Atomic{take(A, _a); _b = f(_a); put(B, _b); state = 0;};
od
inline f(x){
% user defined
}
```

The PairMerger connector has a $\otimes$ connected to each channels, which here in Promela implemented by _c = ⟨_a, _b⟩, after taking data from input port A and B, when conditions be satisfied. It sends the result _c to the output port C at the end. The code is in Listing 5.13.

Listing 5.13: Promela code generated for PairMerger connector

```
proctype PairMerger(port A; port B; port C){
mytype _a; mytype _b; mytype _c;
int state=0;
do
:: Full(A.data) && Full(B.data) && Full(C.trig)
   -> Atomic{take(A, _a); take(B, _b); _c = <_a, _b>; put(C, _c); state = 0;};
od }
```

Listing 5.14 supports a three-choice Variable process. The variable τ could be updated by the inputs of A when there is a input in A, output port B receives τ when there is a pending request at B. However, B can only receive the not updated τ in one transition.

Listing 5.14: Promela code generated for Variable connector

```
proctype Variable(port A; port B){
mytype _a; mytype _b; mytype τ;
int state = 0;
do
:: Full(A.data) && Full(B.trig)
   -> Atomic{ take(A, _a); _b = τ; τ = _a; put(B, _b); state = 0;};
:: Full(A.data) -> Atomic{ take(A, _a); τ = _a; state = 0;};
:: Full(B.trig) -> Atomic{ _b = τ; put(B, _b); state = 0;};
od}
```

5.4 A case study: verifying a SDN

In this section, we apply our translation of symbolic constraint automata to Promela on a software defined network model, and use the Spin model checker to verify several

temporal properties. We use the model of software defined networks introduced in [33], where all components of an SDN are represented as Reo connectors, and thus as symbolic constraint automata. The model is stateful and reflects the SDN separation between switches, controllers, and network. For each part, we briefly describe the Promela code obtained from the Reo components, and how we model the basic data flow operations of an SDN: `PktIn, PktOut, FlowMod`.

5.4.1 A Promela SDN model via symbolic constraint automata

According to the model of SDN presented in Chapter 4, we got the generated Promela code for the Reo model of a switch $\mathsf{Switch}(\mathsf{P0}, \mathsf{P1}, \mathsf{P2}, \mathsf{Q0}, \mathsf{Q1}, \mathsf{Q2})$ with two input ports $\mathsf{P1}$ and $\mathsf{P2}$, two output ports $\mathsf{Q1}$ and $\mathsf{Q2}$ as shown in Figure 5.3. As explained in the last chapter, those ports form the interface of the switch with the rest of the network. Additionally, the switch interface is extended with an input port $\mathsf{P0}$ and an output port $\mathsf{Q0}$ that serve to exchange flow messages with the SDN controller that we can model directly in Promela code. Flow messages from the controller $\mathsf{Controller}(\mathsf{P}, \mathsf{Q})$ to the switch are exchanged synchronously, via the synchronous connector $\mathsf{Sync}(\mathsf{P}, \mathsf{P0})$. On the other direction, flow messages are exchanged asynchronously via a queue connector $\mathsf{Queue}(\mathsf{Q0}, \mathsf{Q})$ from the port $\mathsf{Q0}$ of the switch to the port Q of the controller.

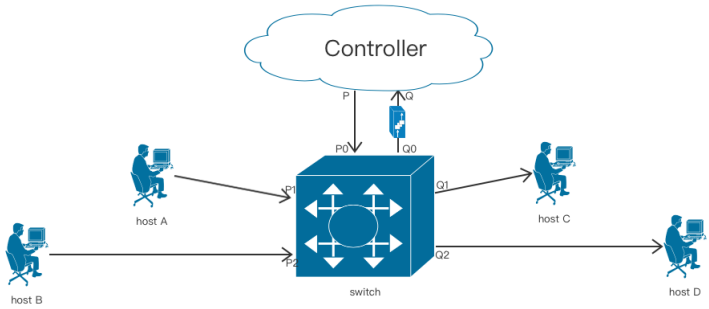


Figure 5.3: A simple example of an SDN architecture

Informally, the symbolic constraint automaton of a switch consists of a single state and few transitions labeled by the following type of guarded actions:

- α_0 is executed when a `FlowMod` message is received from the input port $\mathsf{P0}$. The action here consists in updating the flow table according to the information sent by the controller.
- α_1 is enabled when a `PktOut` message is received by the switch from port $\mathsf{P0}$. The resulting action forwards a packet to a subset (possibly empty) of output ports of

the switch. This subset is contained in the `PktOut` message from the controller.

- α_2 is enabled when one of the input port P1 or P2 of the switch receives a normal packet that is then forwarded to a subset (possibly empty) of output ports according to the current information in the flow table of the switch.

The Promela code of the switch obtained via the translation from a symbolic constraint automaton is presented (in a simplified manner for reason of space) below. The full code and the results of its verification can be found in [32].

```
proctype Switch(port P0, P1, P2, Q0, Q1, Q2){
Message m;   // either FlowMod or PktOut
Packet p;
Flowtable ft_old; ft_new
bool f1=1; f2=1; ...;fn=1;  //control variables
int state=0;    //automaton current state
do
//packet from P1 to Q1: this is a packet-forwarding α2 type of action
:: state==0 && f1==1 && Full(P1.data) && Full(Q1.trig)
    -> atomic{ take(P1,p); Match(ft_old,p,[ Q1 ]); put(Q1,p); f1=1;...fn=1}
//packet from P1 to both Q0 (PktIn message to controller) and Q2:
// this is a α2 type of action
:: state==0 && f2==1 && Full(P1.data) && Full(Q0.trig) && Full(Q1.trig)
    -> atomic{ take(P1,p); Match(ft_old,p,[Q0,Q1]); put(Q0,p);put(Q1,p);
    f1=1;...fn=1 }
//message from P0 to Q1 and Q2: this is an PktOut α1 type of action
:: state==0 && f3==1 && Full(P0.data) && Full(Q1.trig)&& Full(Q2.trig)
   -> atomic{ P0.data?<m>;
     if
      :: m==<p,[Q1,Q2]> -> take(P0,m);put(Q1,p);put(Q2,p); f1=1;...fn=1;
      :: else -> f3=0;
    fi }
//message from P0 to update flow table: this is a FlowMod α0 type of action
:: state==0 && f4==1 && Full(P0.data)
   -> atomic{ P0.data?<m>;
     if
      :: m==<p,ft_new> -> take(P0,p);update(ft_old,ft_new); f1=1;...fn=1;
      :: else -> f4=0;
    fi }
//Similar actions follows here....
...
od}
```

Each transition synchronizes some of the actors in a network (hosts, switches, controllers). Packet forwarding is done on the basis of the result of a function `Match()` of the Reo transformer channel between ports D and E in Figure 4.1. Another non-trivial function used here is `update()`, belonging to the transformer between ports F and E in Figure 4.1 and used to model a `FlowMod` operation.

In our case study as described in Figure 5.3, the network consists of a switch programmed by a controller, where hosts A and B produce packets, and hosts C and D

consume them. We abstract from the specific behaviour of the hosts and model them simply as producers and consumers of messages, respectively.

```
proctype HostA(port A){
packet p1;
atomic{p1.header = 11; p1.ipt = P1; put(A,p1)}
}
```

Host A produces a single packet that is sent to port $P1$. Here we assume a packet contains a header (with information such as the tcp/ip source or destination), and the port of the switch it is supposed to be received directly from the host.

```
proctype HostB(port B){
packet p2;
p2.ipt = P2;
do
  :: atomic{p2.header = 11; put(B,p2)};
  :: atomic{p2.header = 22; put(B,p2)}
od
}
```

The above Promela code for host B is similar to that of A, except that host B repeatedly sends packets with header 11 or 22 to port $P2$. Hosts C and D are consumers that repeatedly execute the *take* action from their ports C and D respectively. Once a packet is received, they update their own local `counter` storing the number of packets with header 11 that they receive. Below we show the Promela pseudocode of host C, that of D is similar.

```
proctype HostC(port C){
packet q1;
int counter=0;
do
  :: atomic{take(C,q1); if q1.header==11 -> counter++;}
od
}
```

Finally we give the Promela code of the controller. The controller takes a packet from port Q if available. If the packet originally passed through port $P1$ then the controller adds a rule in the flow table to forward similar messages (i.e., coming from the same address as in the header) to port $Q1$. Packets with header 22 need to be forwarded to both ports $Q1$ and $Q2$ (thus to hosts C and D). Finally, the following firewall is installed: packets with header 11 that have passed through port $P2$ must be dropped. Note that the controller will insert rules into the flow table of the switch to execute the above commands, and will apply the action itself only the first time a packet does not match any rule in the flow table, i.e., the first time the packet is forwarded to the controller.

Listing 5.15: an example of controller

```
proctype Controller(port Q, P){
Flowtable ft;
Packet p;
Message m;
do
:: Full(Q.data) && (Q.data.ipt==P1) ->
    take(Q, p);
    ft.cond = p.header; ft.action = [Q1];
    //create a rule: if match header then forward to Q1
    m=<p,ft>; put(P, m); //update flow table
    m=<p,[Q1]>; put(P, m); // Forward p to port Q1;
:: Full(Q.data) && (Q.data.header==11) && (Q.data.ipt==P2) ->
//insert a firewall rule: no message from P2
    take(Q,p);
    ft.condition = p.header; f.action = []; // drop package if comes from P2
    m=<p,ft>; put(P,m); //update flow table
:: Full(Q.data) && (Q.data.header==22) ->
    take(Q, p);
    ft.cond = p.header; ft.action = [Q1,Q2];
    //create rule: if match header then forward to Q1 and Q2
    m=<p,ft>; put(P, m); //update flow table
    m=<p,[Q1,Q2]>; put(P, m); // Forward p to port Q1 and Q2;
od
}
```

Port Q of the controller is linked to the output port of the queue connector Queue(Q0, Q) which may store at most 10 packets. This number is reasonable as we do not expect many `PktIn` messages to be forwarded to the Controller.

5.4.2 Verification and simulation

Figure 5.4 describes the scenario for which host A sends a packet with header 11 *after* host B sent a packet with header 22. Here the packet of A will arrive to C by first passing through the controller, and the packet from host B will arrive to both hosts C and D. The LTL properties associated with this scenario are

```
prop1 {[]((p1.header==11 && p1.ipt==P1)→<>(q1.header==11))}
    /* satisfied */
prop2 {[]((p2.header==22)→(<>(q1.header==22)&&<>(q2.header==22)))}
    /* satisfied */
```

Intuitively they says: The message of host A will receive B (no loop holes) and always, if B sends a 22-message, then both eventually C receives a 22-message (but not necessarily the same one), and eventually the same for D. Assuming that only B sends 22-messages then this means that every messages with header 22 received from hosts C or D is originated from host B.

Figure 5.5 describes the scenario when the packet from host A is received at the switch. As the flow table of the switch is empty, the switch forwards the packet to the

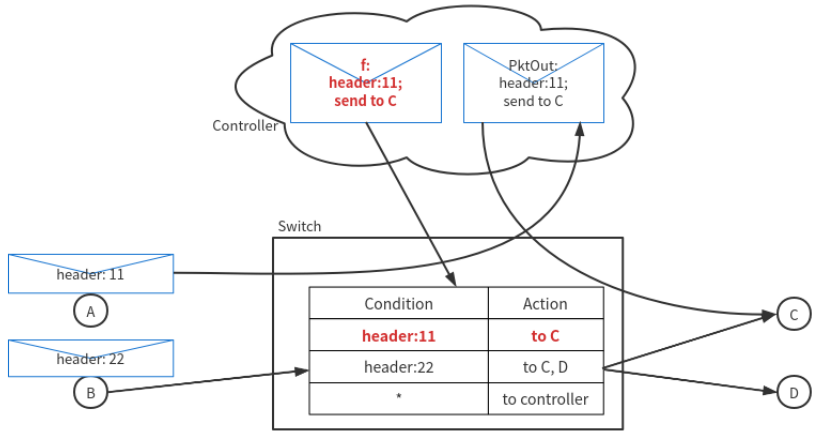


Figure 5.4: Packets from A and B arriving both to C

controller. However, the packet with the same header arriving from B is dropped. Of course, if the packet B arrives at the switch after the updates of the flow table due to the first message from host A, then the packet of B will match the flow table and be redirected to host C, violating the firewall rule. We can verify this formally in Spin via the following LTL property:

```
prop3 {<>(msg1.header==11)&&(<>[](HostC_counter==1))} /* unsatisfied */
```

This property together with the code of the three hosts A, B and C, states that there is a state in the system where a packet with header 11 is received (either from host A or B) and eventually C receive a message with header 11 and no other such a message afterwards.

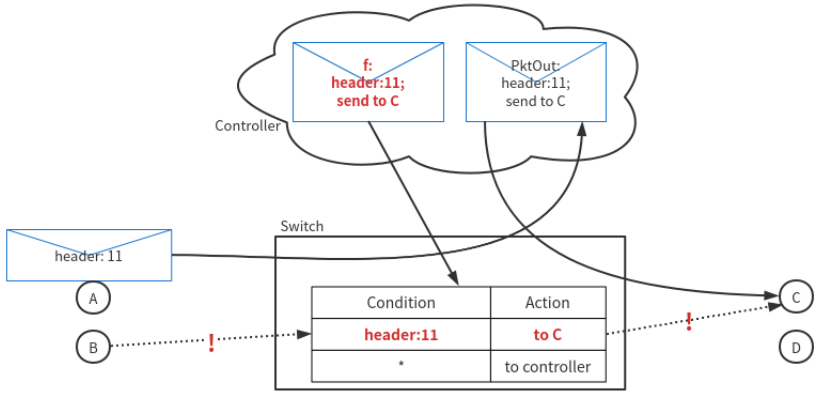


Figure 5.5: Are packets from B with header=11 always dropped?

We use the Spin model checker to verify the three LTL properties, with the following parameters: Extra Compile-Time Directives is set to 20700; the number of hash-functions in Bitstate mode to 5, and the Physical Memory Available to 102400 Mbytes. We used depth-first search with partial order reduction and Bitstate/Supertrace in order to achieve better performance [50]. The results of the verification are shown in Table 5.1.

Property	Errors found	Time usage	Depth reached	States stored	States matched	Transi -tions	State-vector
prop1	0	10.5s	479	214292	660445	1058424	2088 byte
prop2	0	8.64s	479	152057	373096	767167	2088 byte
prop3	2520	14.3s	500	183077	924560	1346688	2184 byte

Table 5.1: Verification results

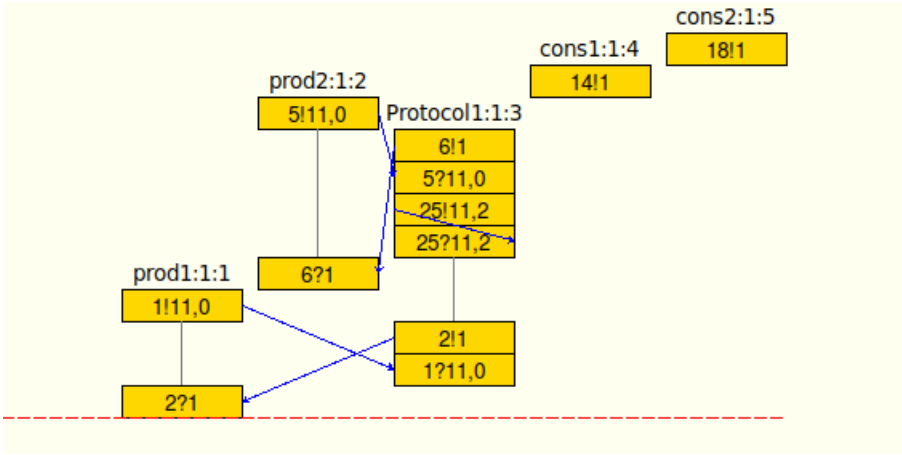


Figure 5.6: Simulation result

We simulate the results of `prop3` with the first founded error written in the ".trail" file, the result of this simulation is shown in Figure 5.6, where we see the packet forwarded by host B (here internally called prod2) to the switch (here called Protocol1). The flow table in the switch is updated successfully (see in action $25!11, 2$) but soon the packet is dropped (see in action $25?11, 2$). Before C (cons1) receives any other packets, the one sent from host A matches the new flow table (header $= 11$) and thus will be dropped (in action $1?11, 0$) instead of being forwarded to C.

5.5 Related work

The first automata based model for Reo connectors appeared in [13] where constraint automata have been introduced. The authors define a product on constraint automata that implements the Reo composition operator on timed-data streams semantics [7]. Since then, several other operational models followed, such as Büchi automata of records [54], guarded automata [19], and Reo automata [99]. Those models extend constraint automata by allowing some context-dependent reasonings. See [58] for an overview of the main operational and denotational models of Reo. Constraint automata with memory and their composition have been thoroughly studied in [57]. Our work is based on a similar model and semantics, but while we aim for finding a subset of it that can be easily implemented, the work in [57] concentrates on the efficient computation of the main composition operator.

Also, model checking of Reo connectors has been a very active area of research. Vereofy is a dedicated model checker developed explicitly to verify linear time and branching time temporal properties of Reo connectors expressed as constraint automata [12]. Vereofy, however does not allow for explicit data to be handled in the automata and properties. For this reason, in [66] the authors encode Reo connectors as communicating processes, and use mCRL2 to check some behavioral properties. The model checker mCRL2 [22], is based on the Algebra of communicating processes [11] with properties expressed as formulas in the modal μ-calculus with strong and branching bisimulation as equivalences as well as strong and weak trace equivalence. UPPAAL is a model checker [16], for linear time temporal property of networks of timed automata [3]. UPPAAL can perform reachability analysis, as well as simulation and error reports. See [28] for a recent use of UPPAAL to verify behavioral properties of real time Reo connectors. Our work differs from previous works on model checking Reo Connector in the following points. First of all, Spin is a data sensitive model checker. Contrary to Vereofy, we can verify temporal properties on connectors that involve data values and local memory. Second, our use of the Spin model checker is designed specifically to verify LTL properties of Reo connectors, contrary to the strong bisimulation equivalence used in proving equalities in the mCRL2 encoding of Reo channels in [66]. Thus is more in line with Vereofy and the semantic basis of Reo that is trace based, without branching properties. In addition, the encoding of Reo described in [66] does not encompass memory. Finally, our translation into Promela differs from the above works in the use of atomic statements in order to enforce synchrony. Of course extension of our work to real time systems and UPPAAL are imaginable, and can be pursuit in the near future. All in all our work extends the literature by providing another tool chain to compile connectors to Promela and use the Spin model checker.

In [53], the authors check network consistency properties in the model of SDN topol-

ogy by using UPPAAL with the goal to detect an inconsistency or verify a flow against real-time properties. An SDN model extended with synchronization barriers is considered in [1]. Their approach is based on encoding an SDN model into the ABS language, and use the SYCO tool to verify properties about safety policies and network loops. [35] uses Flow-LTL to specify the data flow in an extension of Petri nets. Concurrent updates and packet coherence in the network are then checked with the hardware model checker ABC.

5.6 Conclusion

In this chapter, we presented a full automatic translation from symbolic constraint automata to Promela. Symbolic constraint automata are a characterization of constraint automata with memory that can be used to compactly model almost all connectors of the coordination language Reo. In particular, we restrict ourselves to an executable subset of Reo, assuming predicate and actions to be decidable. On the one hand, symbolic constraint automata cannot characterize relational constraints involving, for example, output ports in predicates. On the other hand, our symbolic constraint automata and their Promela translation easily allow for a generalization to lossy connectors by testing the non-presence of a trigger in an output port before executing a lossy transition, as shown, for example, in the following Promela code for a lossy synchronous connector:

```
1 proctype LossySync(port A; port B){
2 mytype _a; mytype _b;//internal values in Sync
3 int state = 0; //initial state
4 do
5 :: state = 0 && Full(A.data) && Full(B.trig) ->
6     Atomic{take(A, _a); _b =_a; put(B,_b); state = 0};
7 :: state = 0 && Full(A.data) && Empty(B.trig) ->
8     Atomic{take(A, _a); state = 0};
9 od }
```

The translation to Promela is interesting in itself as we had to circumvent the problem of Promela not allowing for checking complex predicates on input ports in a guard of a statement. This is however a crucial feature for Reo connectors, that we have taken care of via delayed input and using control variables for recording transitions already taken (and thus guaranteeing not only liveness, but also fairness in the transition selection process).

As a proof of concept, we run our translation for the verification of an SDN model in Spin. The intuitive and modular Promela code is internally translated to a large transition system with more than 200.000 states and 100.000 transitions. It would be interesting to look for an abstraction mechanism at the level of symbolic constraint automata to help reduce the state explosion. An obvious candidate is the combination

of partial order reduction techniques at the symbolic level of the automaton itself.

Chapter 6

Concurrent NetKAT with Ports

In this chapter, we extend the symbolic constraint automata which has already been introduced in chapter 5 to NetKAT [4], therefore NetKAT supports concurrency operation. To accomplish this goal, we define pNetKAT as the language of NetKAT with ports, and pNKA as the NetKAT automata, which has a parallel operator that allows policy communication through the ports. We also define ioNKA as the NetKAT automata with I/O (input/output) ports, which can be transformed from the symbolic constraint automata [34], ioNKA allows composition operation when there are no synchronization causality problems just like the symbolic constraint automata. We proceed as follows. In Section 2 we briefly present NetKAT with a focus on the automata model. While the original model is deterministic, we present also an equivalent but more compact model based on non-deterministic NetKAT automata (NKA). In Section 3 we extend NetKAT protocols with communication actions and concurrency and define closed semantics using non-deterministic NetKAT automata with ports (pNKA). We continue in Section 4 by introducing non-deterministic NetKAT automata with input and output ports (ioNKA) and use them to model NetKAT with ports. We then briefly recall Reo and its symbolic constraint automata semantics and show how to compositionally translate them into NetKAT automata.

6.1 Introduction

The rapid evolution of technology, increasing network traffic, and the need for flexible and scalable computer networks have necessitated a paradigm shift in network manage-

ment. Traditional network architectures use distributed switches to receive and forward packets, each switch consisting of hardware and dedicated control software. Software Defined Networks (SDNs) provide a centralized approach to network control and management by separating the control plane from the data plane [42]. This separation allows for programmability and agility in network configurations, enabling dynamic provisioning of resources, efficient traffic management, and the ability to adapt to changing requirements.

The level of programmability of the software controllers in an SDN to handle traffic flow, routing decisions, and network policies together with the use of protocols such as OpenFlow [82] have generated increasing interest in the academic community to provide a theoretical foundation for understanding the principles, components, and interactions within SDNs. Examples include model-checking to verify controller programs [14, 15, 1, 34], formal models of OpenFlow [60, 33], or some specific part of it, such as the topology discovery mechanism [109] or security protocols [27].

A policy-based approach is taken by NetKAT [4], a process algebraic model that emphasizes the policy-driven nature of SDNs. It consists of an extension of Kleene Algebra with Tests tailored to define high-level policy specification and network components and observe the network behavior from the point of view of a packet [69]. NetKAT, however, is not stateful and does not allow modeling concurrent policies and multiple packets. In this chapter, we present pNetKAT, a conservative extension of NetKAT, allowing multiple concurrent policies to communicate via shared ports. In pNetKAT, ports are treated as shared variables that can be undefined when no communication is possible. We give an operational semantics to pNetKAT using non-deterministic NetKAT automata with a slightly modified acceptance rule that enforces observability only if sequences with successful synchronization steps). Without ports, both syntactically and semantically pNetKAT and NetKAT coincide.

Under the assumption that ports are declared as either input or output, we give another semantics to pNetKAT by refining the acceptance rule of non-deterministic NetKAT automata so to allow for the system to interact with the environment along the input and output ports. The new semantics is an extension of the previous one (and thus the new equivalence is stricter, in general). We show that this model can be used as semantics for the coordination language Reo [5]. from which we can borrow the join composition operator and define it for NetKAT automata with input and output ports.

Unlike other methods, our pNetKAT extension to a stateful and concurrent NetKAT is conservative as it remains in the semantic realm of language equivalence instead of moving to pomset [106] or bisimulation equivalence [24]. The connection with Reo paves the way to a more expressive concurrent NetKAT, with (concurrent, stateful) policies declaring input and output ports (as switches and controllers in SDNs) that can be composed using a join operation (only communication on common ports must

synchronize, while policies using undeclared ports in another process can proceed in parallel).

Related work. Several works are extending NetKAT in different directions. For example, [81] introduces network event structures to model constraints on updates and define an extension of NetKAT policies with mutable state to give semantics to stateful SDN controllers. DyNetKAT [24] is a NetKAT extension with concurrency and a stateful state to model SDNs with dynamic configurations. The extended language is a process algebra with constructs for synchronization, sequential composition, and recursion built on top of NetKAT policies. While DyNetKAT allows for multi-packet behavior, the syntax does not allow for the basic NetKAT "dup" action. Also, the focus is on bisimulation rather than our (and NetKAT) language equivalence, which comes equipped with sound and ground-complete axiomatization.

Staying in the realm of Kleene algebra is the line of works followed by [106], where CNetKAT is introduced as a combination of Kleene algebra with tests, concurrent Kleene algebra, and network operators. The semantics is given in terms of pomset languages and is thus based on true concurrency rather than interleaving.

Besides the work we already mentioned, there are other formal models for SDN closely related to NetKAT that involve concurrency. For example, concurrent NetCore [95] extends NetCore with concurrency, while NetKAT is an extension of NetCore with Kleene star. In terms of tools, SDNRacer [30] checks various concurrency-induced errors in SDNs and precisely captures the asynchronous interaction between controllers and switches.

Constraint automata are the first automata-based model for Reo connectors [13]. Since then, various other operational models have emerged (see [58] for an overview). Relevant to our work here is the extension of constraint automata with memory [57] and the more recent work of symbolic constraint automata [34] that focus on an implementable subset, instead of an efficient computation of the composition operator. In this chapter, we show how to embed symbolic constraint automata into ioNKA. We follow I/O automata [78] and constraint automata [13] by explicitly declaring at the interface the ports that are used as input and output. Transitions in ioNKA, however, are neither action-based nor imperative, but rather declarative using pre- and post-conditions in the style of NetKAT automata.

6.2 NetKAT

In this section, we briefly introduce NetKAT [4], a language for specifying the flow of a packet through a network, and give its semantics in terms of finite automata and languages.

We assume fixed a finite set of fields Fld, say of size k, and a finite set of values Val. A packet π is a record of fields, that is, a function from Fld to Val that we represent by $[f_1 = v_1, \cdots, f_k = v_k]$. Tests for the value stored in a field form the basic building block for the set of predicates $B(Fld)$ defined by the following grammar:

$$a, b ::= 1 \mid 0 \mid f = v \mid a + b \mid a \cdot b \mid \neg a \,.$$

The set of all predicates (modulo the usual equations) forms a Boolean algebra, where $+$ is interpreted as the disjunction, $\cdot$ as the conjunction, and $\neg$ as negation. Further, 1 is the truth predicate, and 0 denotes false. The set At of atoms α, β of the Boolean algebra $B(Fld)$ corresponds to the set of valuations, that is complete conjunctions of basic tests $f = v$ ranging over all fields in Fld. For simplicity, and with a convenient abuse of notation, we denote an atom as a record $\alpha = [f_1 = v_1, \cdots, f_k = v_k]$, allowing us to switch between packets and atoms. The behavior of a packet through the network is specified by policies

$$p, q ::= a \mid f \leftarrow v \mid dup \mid p + q \mid p \cdot q \mid p^* \,.$$

Here a is a predicate in $B(Fld)$, $f \leftarrow v$ is the assignment of the value v to the field f of a packet, $p + q$ is the nondeterministic choice between the policies p and q, $p \cdot q$ specify the sequential composition of two policies, and p^* the iterative execution of a policy p. The predicate 0 denotes failure and 1 is skip. As usual, we will often not write "$\cdot$" in policies. When applied to predicates, "$+$" and "$\cdot$" act as logical disjunction and conjunction operators, respectively.

The behavior of a packet π through the network is specified by a string in $(At \cdot At) \cdot At^*$, denoting a sequence of conditions satisfied by the packet π before and after being forwarded from one switch to another in the network. Syntactically, the forwarding is specified by the action dup, which is thus the only observable action of a policy. The semantics of a policy is then given by the set of all possible behaviors of a packet under that policy. Since this is a regular subset of $(At \cdot At) \cdot At^*$, following [36], we use an automaton to describe it.

Definition 6.1. *A deterministic NetKAT automaton (dNKA) is a tuple* $(S, Fld, \delta, \xi, s_0)$ *where*

- S *is a finite set of states,*
- Fld *is a finite set of fields,*
- $\delta \colon S \times At \times At \to S$ *is a transition map,*
- $\xi \colon S \times At \times At \to 2$ *is an observation map, and*
- $s_0 \in S$ *is a distinguished initial state.*

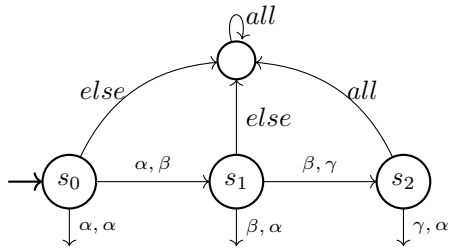


Figure 6.1: An example of a dNKA

Here At is the set of atoms of $B(Fld)$, and 2 is the two-element Boolean set.

Differently from an ordinary automaton, a dNKA uses pre- and post-conditions as labels to specify the execution of an action in a computation. Here $\delta(s, \alpha, \beta) = s'$ denotes a transition from state s to a state s' executed by an action satisfying the pre-condition α and resulting in a post-condition β. Further, the observation map $\xi(s, \alpha, \beta) = 1$ if and only if an action in state s satisfies the pre-condition α, results in the post-condition β, and successfully terminates a computation.

Figure 6.1 shows a dNKA. There are four states but only $\{s_0, s_1, s_2\}$ are accepting computations that end in the pair of atoms labeling the respective vertical down arrows. The state s_0 is the initial state, as marked by an incoming arrow without a source. As usual, labeled arrows between two states represent the transition map. Here we assume only three atoms: α, β, and γ.

The language accepted by a dNKA is a subset of strings in $(At \cdot At) \cdot At^*$ and is defined with the help of the following auxiliary acceptance predicate:

Definition 6.2. *For a dNKA $M = (S, Fld, \delta, \xi, s_0)$, we say that a string $\sigma \in (At \cdot At) \cdot At^*$ is accepted by M if and only if the deterministic acceptance predicate $DAcc(s_0, \sigma)$ holds, where $DAcc$ is defined inductively as follows:*

- $DAcc(s, \alpha\beta) = \xi(s, \alpha, \beta)$,
- $DAcc(s, \alpha\beta \cdot \tau) = DAcc(\delta(s, \alpha, \beta), \beta \cdot \tau)$,

where $s \in S$, $\alpha, \beta \in At$, and $\tau \in At^+$. The language $L_d(M)$ is defined as the set of all strings accepted by M.

The language of the automaton in Figure 6.1 is $\{\alpha\alpha, \alpha\beta\alpha, \alpha\beta\gamma\alpha\}$. In fact, for example, $DAcc(s_0, \alpha\beta\alpha) = DAcc(s_1, \beta\alpha) = \xi(s_1, \beta, \alpha) = 1$.

For a more compact representation of the operational semantics of NetKAT, we use non-deterministic NetKAT automata as introduced in [108].

Definition 6.3. *A non-deterministic NetKAT automaton (NKA) is a tuple* $(S, Fld, \Delta, \Xi, s_0)$*, where*

- S *is a finite set of states;*
- Fld *is a finite set of fields;*
- $\Delta\colon S \times At \times At \to \mathcal{P}(S)$ *is a transition relation;*
- $\Xi\colon S \times At \times At \to 2$ *is an observation map, and*
- $s_0 \in S$ *is a distinguished initial state.*

As before, here At *is the set of atoms of* $B(Fld)$*.*

For example, the sub-automaton defined by restricting the one in Figure 6.1 to the three states s_0, s_1 and s_2 is an NKA.

Having non-determinism is reflected in the definition of the language accepted, which now selects only transitions leading to successful computations.

Definition 6.4. *For an NKA* $N = (S, Fld, \Delta, \Xi, s_0)$*, we say that a string* $\sigma \in (At \cdot At) \cdot At^*$ *is accepted by* N *if and only if the non-deterministic acceptance predicate* $NDAcc(s_0, \sigma)$ *holds, where NDAcc is defined inductively as follows:*

- $NDAcc(s, \alpha\beta) = \Xi(s, \alpha, \beta)$,
- $NDAcc(s, \alpha\beta \cdot \tau) \iff \exists s' \in \Delta(s, \alpha, \beta)\,.\, NDAcc(s', \beta \cdot \tau)$,

where $s \in S$*,* $\alpha, \beta \in At$*, and* $\tau \in At^+$*. The language* $L_{nd}(N)$ *is defined as the set of all strings in* $(At \cdot At) \cdot At^*$ *accepted by* N*.*

Every dNKA can be easily seen as an NKA with a functional transition relation. Conversely, given an NKA, we can construct a dNKA that is language equivalent.

Theorem 1. *For every NKA* N *there exists a dNKA* M *such that* $L_d(M) = L_{nd}(N)$*.*

The result is similar to the powerset construction for ordinary finite automata. In fact, given a NKA $N = (S, Fld, \Delta, \Xi, s_0)$ we can define a dNKA $M = (\mathcal{P}(S), Fld, \delta, \xi, \{s_0\})$ with

- $\xi(X, \alpha, \beta) = 1$ if and only if $\exists s \in X.\Xi(s, \alpha, \beta) = 1$,
- $s \in \delta(X, \alpha, \beta)$ if and only if $\exists s' \in X.s \in \Delta(s', \alpha, \beta)$.

Then, for all $X \subseteq S$, $\alpha, \beta \in At$, and $\sigma \in At^*$ we can prove that $DAcc(X, \alpha\beta \cdot \sigma)$ if and only if there exists $s \in X$ such that $NDAcc(s, \alpha\beta \cdot \sigma)$. Note that the above language equivalence does not hold if Δ and Ξ would take as input general Boolean predicates instead of atoms.

In Table 6.1 we give the operational semantics of NetKAT policies in terms of an NKA. States of the automaton are policies themselves, that we consider modulo associativity, idempotency, and commutativity of the "+" operation to guarantee local finiteness. A state represents (an equivalence class of) what still needs to be executed.

Table 6.1: Operational semantics of NetKAT

$dup \xrightarrow{\alpha,\alpha} \alpha$	$\dfrac{\alpha \leq a}{a{\downarrow_{(\alpha,\,\alpha)}}}$	$\dfrac{\beta \leq (f=v)}{f \leftarrow v{\downarrow_{(\alpha,\,\beta)}}}$
$\dfrac{p_1 \xrightarrow{\alpha,\beta} p}{p_1 + p_2 \xrightarrow{\alpha,\beta} p}$		$\dfrac{p_1{\downarrow_{(\alpha,\,\beta)}}}{p_1 + p_2{\downarrow_{(\alpha,\,\beta)}}}$
$\dfrac{p_1 \xrightarrow{\alpha,\beta} p}{p_1 \cdot p_2 \xrightarrow{\alpha,\beta} p \cdot p_2}$		$\dfrac{p_1{\downarrow_{(\alpha,\,\beta)}} \quad p_2{\downarrow_{(\beta,\,\gamma)}}}{p_1 \cdot p_2{\downarrow_{(\alpha,\,\gamma)}}}$
$\dfrac{p_1{\downarrow_{(\alpha,\,\beta)}} \quad p_2 \xrightarrow{\beta,\gamma} p}{p_1 \cdot p_2 \xrightarrow{\alpha,\gamma} p}$		
$\dfrac{p \xrightarrow{\alpha,\beta} p_1}{p^* \xrightarrow{\alpha,\beta} p_1 \cdot p^*}$		$p^*{\downarrow_{(\alpha,\,\alpha)}}$
$\dfrac{p{\downarrow_{(\alpha,\,\beta)}} \quad p^* \xrightarrow{\beta,\gamma} p_1}{p^* \xrightarrow{\alpha,\gamma} p_1}$		$\dfrac{p{\downarrow_{(\alpha,\,\beta)}} \quad p^*{\downarrow_{(\beta,\,\gamma)}}}{p^*{\downarrow_{(\alpha,\,\gamma)}}}$

We have two types of rules: those specifying transitions (on the left-hand side of Table 6.1), and those for observations, specifying the accepting states (on the right-hand side). Intuitively, the behavior of a policy is to guide a given packet into a network. This is described by the assignment of values to the fields to record, for example, where the packet is, where it has to go, and other information. Policies filter out executions via predicates. The basic transition step of a policy is given only by the execution of a *dup* action. Predicate evaluations and field assignments are evaluated locally in the current state. A policy execution may terminate in an accepting state (as specified on the right-hand side of Table 6.1) or may diverge in an infinite computation (via the transition rules of p^*) and not be observed. Note that since we consider states modulo associativity, commutativity, and idempotency of the "+" operation, there is no need for symmetric rules for the "+" for both the transition and the observation relation.

For a given policy p, in [36] a dNKA $M(p)$ is constructed using syntactic derivatives. Similarly, Let $N(p)$ denote the NKA constructed using the rules in Table 6.1, with as initial state (the equivalence class of) p. We then have the automata $M(p)$ and $N(p)$ accept the same language [108].

6.3 NetKAT with ports

Next, we extend NetKAT protocols with a parallel operator and allow policies to communicate via ports. A port x is a shared variable between two processes that can be updated with a value v by an output operation $x!v$ and can be destructively read by an input operation $x?f$ which stores the communicated value into a field f. Unlike a variable, however, a port may be undefined, here denoted by the symbol $\perp$ that we assume is not a value in Val. Intuitively, a port x is undefined, i.e. $x = \perp$, if it can be used by an output operation. Dually, input on a port x can only take place if x is not undefined, i.e. $\neg(x = \perp)$ that, as usual, we denote by $x \neq \perp$. In other words, we see an output $x!v$ as the atomic execution of the guarded command $x = \perp \cdot x \leftarrow v$, whereas an input $x?f$ can be seen as the atomic execution of the guarded command $x \neq \perp \cdot f \leftarrow x \cdot x \leftarrow \perp$. Here we use the assignment $f \leftarrow x$ of a variable to a field, which is just an abbreviation for the protocol $\Sigma_{v \in Val}(x = v \cdot f \leftarrow v)$ because Val is assumed to be finite. Communication of two parallel protocols via a port x in an undefined state is then the atomic execution of an output command on x followed by an input on x, resulting in the command

$$(x = \perp \cdot x \leftarrow v) \cdot (x \neq \perp \cdot f \leftarrow v \cdot x \leftarrow \perp)$$

which, because is executed atomically, can be thought of as equivalent to $x = \perp \cdot f \leftarrow v \cdot x \leftarrow \perp$.

Formally, we assume a finite set of variables Var partitioned in a set of fields Fld and a set of ports Prt. As for NetKAT, fields are ranged over by f, while ports are by x. All variables can store values from Val but only ports can be undefined, which we denote with $\perp \notin Val$. The set of predicates $B(Var)$ extends those of NetKAT by allowing basic tests on all variables, including ports, as defined by the grammar

$$a, b ::= 1 \mid 0 \mid f = v \mid x = v \mid x = \perp \mid a + b \mid a \cdot b \mid \neg a \,,$$

where, $f \in Fld$, $x \in Prt$, and $v \in Val$. We use $f = x$ as a shorthand for the test $\Sigma_{v \in Val} x = v \cdot f = v$. This is well defined because the set Val is finite. The behavior of a packet in pNetKAT through a network subject to several communicating parallel policies is specified by the following grammar that extends the one of NetKAT with communication actions and a parallel operator:

$$p, q ::= a \mid f \leftarrow v \mid dup \mid x?f \mid x!v \mid p + q \mid p \cdot q \mid p||q \mid p^* \,.$$

As discussed above, here $x?f$ is an input action that is executed only when the port x has a value available that is assigned immediately to the field f. The output action $x!v$ is executed if the port x is not busy (there is no value) and makes available the value

v at the port. Note that only fields can be assigned directly by policies, whereas ports can change values only through successful communications. Policies can be executed in parallel via the operator "||". Parallel policies executing an input, respectively an output, action on the same port synchronize.

The operational semantics of pNetKAT are given in terms of NKA as presented in Definition 6.3. The only addition to the rules given in Table 6.1 is the transition and observation map for input and output actions and for the parallel composition of policies. The extra rules are presented next.

Input and output actions are, like *dup*, primitive actions that have a transition step and do not terminate for any observable pairs of atoms:

$$\frac{\alpha \leq (x = v) \quad \beta = \alpha[\bot\, /x][v/f]}{x?f \xrightarrow{\alpha,\beta} \beta} \qquad \frac{\alpha \leq (x = \bot) \quad \beta = \alpha[v/x]}{x!v \xrightarrow{\alpha,\beta} \beta}$$

The conditions in the premises of the two rules express the precondition and postcondition of the input and output, respectively, as we already discussed. Here $\alpha[v/x]$ ($\alpha[v/f]$) is the atom assigning a port x to v (a field f to v, respectively) and all other variables are as in α.

The transition relation of the parallel composition $p_1||p_2$ of two policies p_1 and p_2 is described by three types of rules, namely: synchronization, interleaving, and termination. When they occur in parallel, an input and an output action on the same ports synchronize:

$$\frac{p_1 \xrightarrow{\alpha_1,\beta_1} p \quad p_2 \xrightarrow{\alpha_2,\beta_2} q}{p_1||p_2 \xrightarrow{\alpha,\beta} p||q} \qquad \frac{p_1 \xrightarrow{\alpha_1,\beta_1} p \quad p_2 \xrightarrow{\alpha_2,\beta_2} q}{p_2||p_1 \xrightarrow{\alpha,\beta} q||p}$$

under the condition that there is a port $x \in \mathit{Prt}$ and a field $f \in \mathit{Fld}$ such that $\alpha(x) = \beta(x) = \alpha_1(x) = \beta_2(x) = \bot$ and $\beta_1(x) = \alpha_2(x) = \beta_2(f)$, whereas for all other variables $y \in \mathit{Var}$ different from x, $\alpha_1(y) = \alpha_2(y) = \alpha(y)$ and $\beta_1(y) = \beta_2(y) = \beta(y)$. The above condition says that the pair (α_1, β_1) describes the output of the value v on a port x, that is received and assigned to field f by the input action specified by (α_2, β_2). For all other variables, the preconditions and the postconditions of all transitions involved do not change.

If the transition of a policy does not have a visible effect on the state of a port, then when in parallel with any other policy it can proceed in an interleaving fashion:

$$\frac{p_1 \xrightarrow{\alpha,\beta} p}{p_1||p_2 \xrightarrow{\alpha,\beta} p||p_2} \qquad \frac{p_1 \xrightarrow{\alpha,\beta} p}{p_2||p_1 \xrightarrow{\alpha,\beta} p_2||p}$$

where $\alpha(x) = \beta(x)$ for all port $x \in Prt$. Note that, the above symmetric rules in combination with the synchronization rules imply that there cannot be multiparty synchronization.

Similar to the shuffle of languages, if a policy p_1 terminates when in parallel with another policy p_2, then p_2 can continue alone from the postcondition observed at the termination of p_1:

$$\frac{p_1{\downarrow_{(\alpha,\,\beta)}} \quad p_2 \xrightarrow{\beta,\gamma} p}{p_1||p_2 \xrightarrow{\alpha,\gamma} p} \qquad \frac{p_1{\downarrow_{(\alpha,\,\beta)}} \quad p_2 \xrightarrow{\beta,\gamma} p}{p_2||p_1 \xrightarrow{\alpha,\gamma} p} \qquad \frac{p_1{\downarrow_{(\alpha,\,\beta)}} \quad p_2{\downarrow_{(\alpha,\,\beta)}}}{p_1||p_2{\downarrow_{(\alpha,\,\beta)}}}$$

Generally, the parallel composition of two policies does not terminate immediately, as it may involve input and output actions. However, if no communication action is involved, then it terminates observing the pair (α, β) if both policies do the same. Note that this means inconsistent policies cannot terminate successfully, as they both act atomically on the same packet.

As in the previous section, we denote by $N(p)$ the NKA constructed using the rules in Table 1 and the above ones for the parallel composition, with as states equivalence classes of policies modulo commutativity and associativity of both "+" and "||", and idempotency of only "+", and with as initial state (the equivalence class of) p. To enforce synchronization, we impose that ports are undefined at all times in every accepted string (a condition satisfied by the synchronization step but not by the postcondition of an open output and a precondition of an open input). We thus refine the acceptance predicate for NKA with ports (thus, pNetKAT) as follows:

Definition 6.5. *Let At be the set of atoms of the Boolean algebra $B(Var)$. For an NKA (with ports) $N = (S, Var, \Delta, \Xi, s_0)$, we say that a string $\sigma \in (At \cdot At) \cdot At^*$ is accepted by N if and only if the predicate $PAcc(s_0, \sigma)$ holds, where $PAcc$ is defined inductively as follows:*

- $PAcc(s, \alpha\beta) \iff \Xi(s, \alpha, \beta)$ *and* $\forall x \in Prt.\alpha(x) = \beta(x) = \bot$,
- $PAcc(s, \alpha\beta\sigma) \iff \exists s' \in \Delta(s, \alpha, \beta)\,.\,PAcc(s', \beta\sigma)$ *and* $\forall x \in Prt.\alpha(x) = \bot$,

where $s \in S$, $\alpha, \beta \in At$, and $\sigma \in At^+$. The language $L_p(N)$ is defined as the set of all strings in $(At \cdot At) \cdot At^*$ accepted by N. We refer to NKA with PAcc predicates as $pNKA$.

Because of the symmetry in the rules of the parallel composition, we have that "||" is a commutative and associative operator. It is not idempotent in general, except for policies with no occurrences of *dup*, input, or output actions. For example $a||b = a \cdot b$ and $f \leftarrow v||f \leftarrow v = f \leftarrow v$.

If there are no ports in Var (i.e. $Var = Fld$) then they do not appear in atoms in At. In this case, the definition of $PAcc$ coincides with the usual definition $NDAcc$

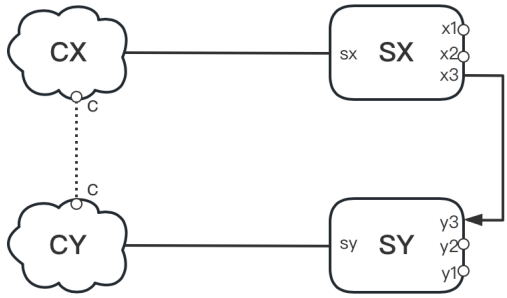


Figure 6.2: A SDN with two switches and two controllers

of accepted strings for an NKA. Note that because ports are undefined in every atom occurring in a string accepted by *PAcc* ports can be removed (or added) to an NKA without changing its language equivalence. Using the Kleene theorem for NetKAT [36], we can relate (non-compositionally) pNetKAT with NetKAT:

Theorem 2. *For every pNetKAT policy p there is a NetKAT policy q such $L_{nd}(N(q))$ is equal to $L_p(N(p))$ after removing the ports from every atom.*

This implies that for every process in pNetKAT, we can find an 'equivalent' process in NetKAT, basically by compiling parallel processes into interleaved ones if no open communication is involved and transforming synchronizations into assignments. In other words, the semantics of pNetKAT is closed, meaning that it does not allow any external communication after the system is defined. In the next section, we define an open semantics that allows for the synchronization of several ports at the same time.

We conclude this section with an example adapted from [24] and sketched in Figure 6.2. Two switches SX and SY have 3 ports each: $x1, x2, x3$ and $y1, y2, y3$, respectively. Their behavior depends on their current flow table and it is described by the following set of policies:

$$\begin{aligned} SX_0 &= 0 & SY_0 &= 0 \\ SX_1 &= (f = x1) \cdot f \leftarrow x3 & SY_1 &= (f = y3) \cdot f \leftarrow y1 \\ SX_2 &= (f = x2) \cdot f \leftarrow x3 & SY_2 &= (f = y3) \cdot f \leftarrow y2\,, \end{aligned}$$

where f is a field of a packet that records the last passed port. The switches are linked through ports $x3$ and $y3$:

$$L = (f = x3) \cdot dup \cdot f \leftarrow y3\,.$$

Under the flow tables SX_1 and SY_1, for example, a packet that arrives at port $x1$ of

switch SX is forwarded to port $x3$. The latter is linked to port $y3$ of switch SY, which forwards the packet to port $y1$. Note the role of the *dup* action to record that a packet moves from one switch to another.

Each switch is linked with a controller via the ports sx and sy. CX is the controller of switch SX and CY of switch SY. The two controllers are concurrently acting on their switch by updating their flow tables. The task of the two controllers is to guarantee that incoming packets at port $x1$ arrive at port $y1$ and incoming packets at port $x2$ arrive at port $y2$. No mixing of flow is allowed. To avoid race conditions, the controllers have to synchronize and guarantee a proper order of execution of their concurrent behaviors:

$$\begin{aligned} CX &= (f = x1) \cdot (sx!0 \cdot c!1 \cdot c?g \cdot sx!1) \\ &\quad + (f = x2) \cdot (sx!0 \cdot c!2 \cdot c?g \cdot sx!2) \\ CY &= c?g \cdot ((g = 1 \cdot sy!1) + (g = 2 \cdot sy!2)) \cdot c!0\,. \end{aligned}$$

Here sx and sy are the ports connecting the controllers to their controlled switches. When sending the flow message 0, 1, or 2, the flow table will be updated accordingly. The two controllers use port c to synchronize each other and pass the information about which flow table they have updated. While waiting for the update of the flow table of switch SY, the switch SX first drops all incoming packets, and only after SY is updated then SX accept packets from the correct port.

The behavior of the entire network is given by

$$(ft1 \leftarrow 0) \cdot (ft2 \leftarrow 0) \cdot (N^* || CX || CY)^*\,,$$

where

$$N = \Sigma_{j=0}^{2}(ft1 = j) \cdot SX_j + \Sigma_{j=0}^{2}(ft2 = j) \cdot SY_j + L + sx?t1 + sy?t2$$

Initially, both switches start with empty flow tables that are updated when a controller sends a flow message to its switch via the port sx or sy, respectively.

6.4 NetKAT automata with I/O ports

In the previous section, we used NKA for giving a closed semantics of our concurrent policy language pNetKAT using the acceptance predicate *PAcc* that takes into account ports. Next, we consider NetKAT automata for open concurrent systems and use them as a model of pNetKAT.

To begin with, we partition the set of ports *Prt* into input ports *IPrt* and output ports *OPrt*. Together with the disjoint set of fields *Fld* they form a finite set of variables *Var*. Input ports are ranged over by i and output ports by o. As before, all variables can store values from *Val* but only input and output ports can be undefined, which we

denote with $\perp \notin Val$. Intuitively, an input port i of a connector is enabled if it contains a value different from $\perp$ so that this value is ready to be taken by the connector when synchronizing on i with the environment that puts the value in it. Dually, an output port o of a connector is undefined (i.e., $o = \perp$) when the port o is ready to receive a value from the connector and synchronizes with the environment when it will read from o.

We use input and output ports to define a novel operational behavior of NKA by an acceptance predicate that, differently from *PAcc*, does not enforce synchronization and leaves the system open to communication instead of closing it in the style of [21].

Definition 6.6. *Let At be the set of atoms of the Boolean predicates $B(Var)$, where $Var = IPrt \cup OPrt \cup Fld$. For an NKA $N = (S, Val, \Delta, \xi, s_0)$ with atoms At involving input and output ports, we say that a string $\sigma \in (At \times At)^+$ is accepted by N if and only if the predicate $IOAcc(s_0, \sigma)$ holds, where IOAcc is defined inductively as follows:*

- $IOAcc(s, (\alpha, \beta)) \iff \Xi(s, \alpha, \beta)$,
- $IOAcc(s, (\alpha, \beta)\sigma) \iff \exists s' \in \Delta(s, \alpha, \beta)\,.\, IOAcc(s', \sigma)$ *and* $\beta \rhd head(\sigma)$,

where $s \in Q$, $\alpha, \beta \in At$, $\sigma \in (At \times At)^+$. The language $L_{IO}(N)$ is defined as the set of all strings in $(At \times At)^+$ accepted by N. We refer to NKA with IOAcc predicates as $ioNKA$

A pair (α, β) in a string accepted above represents the pre/post condition of an action executed by a component. In between two pairs, the environment can communicate with the components and change the values at its ports. We formalize this using the $\rhd$ predicates. In fact, for every string in $(At \times At)^+$, we define $head((\alpha, \beta)\sigma) = \alpha$, and for every two atoms α and β we say that the predicate $\beta \rhd \alpha$ holds if and only if:

a. local variables cannot be modified by the environment, i.e., $\beta(f) = \alpha(f)$ for every field $f \in Fld$;

b. the environment can put a value to an input port only if the port is not already enabled, i.e. either $\beta(i) = \alpha(i)$ or $\beta(i) = \perp$;

c. the environment can take a value from an output port only if there is one, i.e., either $\beta(o) = \alpha(o)$ or $\alpha(o) = \perp$.

Here we see β as the postcondition of an action, and α as the precondition of the next action both to be executed by the component, or, dually, they are the pre- and postcondition of actions executed by the environment. The conditions on the second and third items above allow the environment to communicate with a component only through input ports that are not enabled and output ports that contain values. As such the semantics of a component caters to all possible interactions with the environment and is open. For example, if a component executes an action ending in a postcondition

$[f = 1, i = \bot, o = 3]$ then the environment could assign a value to the input port i so that at the next step the component would start with a precondition $[f = 1, i = 2, o = 3]$. Alternatively, the environment could take the value from the output port o and put a value in the input variable i resulting in the next step component precondition $[f = 1, i = 2, o = \bot]$. However, the environment could never change the value of the field f as it is local to the component.

The set of input and output ports used by a pair (α, β) is defined by

$$\begin{aligned} I(\alpha,\beta) &= \{i \in \mathit{IPrt} \mid \alpha(i) \neq \beta(i) = \bot\} \text{ and} \\ O(\alpha,\beta) &= \{o \in \mathit{OPrt} \mid \beta(o) \neq \alpha(o) = \bot\}\,. \end{aligned}$$

The above reflects the fact that an input port must be enabled in the precondition and is available for communication after the value has been taken, and dually for an output port.

In the absence of input and output ports, the condition on the first item ensures that for any two consecutive pairs $(\alpha_1, \beta_1)(\alpha_2, \beta_2)$ occurring in an accepted string, the postcondition β_1 is equal to the precondition α_2. In this case, we can transform a strings $\sigma \in (\mathit{At} \times \mathit{At})^+$ into essentially equal strings in $t(\sigma) \in (\mathit{At} \cdot \mathit{At}) \cdot \mathit{At}^*$ as follows:

$$t((\alpha,\beta)) = \alpha\beta \quad t((\alpha,\beta)\sigma) = \alpha \cdot t(\sigma)\,.$$

The transformation t unifies the subsequent postcondition and precondition because they are equal. The inverse t^{-1} of t maps strings in $(\mathit{At} \cdot \mathit{At}) \cdot \mathit{At}^*$ into strings in $(\mathit{At} \times \mathit{At})^+$ by equating subsequent postcondition and precondition:

$$t^{-1}(\alpha\beta) = (\alpha,\beta) \quad t^{-1}(\alpha\beta\sigma) = (\alpha,\beta) \cdot t^{-1}(\beta\sigma)\,.$$

Here $\sigma \in \mathit{At}^+$ and α, β are atoms in $B(\mathit{Var})$, with $\mathit{Var} = \mathit{IPrt} \cup \mathit{OPrt} \cup \mathit{Fld}$.

Theorem 3. *For every NKA automaton with no (input and output) ports, $\mathit{IOAcc}(s, \sigma) = \mathit{PAcc}(s, t(\sigma)) = \mathit{NDAcc}(s, t(\sigma))$, for any state s and string $\sigma \in (\mathit{At} \times \mathit{At})^+$.*

Proof. The proof follows immediately by induction on the length of σ using the fact that there are no ports and the definitions of the predicate *IOAcc*, $\mathit{PAcc}(s, t(\sigma))$, and *NDAcc*, as well as the definition of the transformation t. □

In other words, the predicate *IOAcc* is a conservative extension of *NDAcc* in the context of NetKAT automata when there are no ports. However, if we assume $\mathit{Prt} = \mathit{IPrt} \cup \mathit{Oprt}$ and $\mathit{Var} = \mathit{Prt} \cup \mathit{Fld}$ so that atoms in $B(\mathit{Var})$ are of the correct type for both predicates *PAcc* and *IOAcc*, we then have the following result.

Theorem 4. *Let $\mathit{Var} = \mathit{Prt} \cup \mathit{Fld}$ and $\mathit{Prt} = \mathit{IPrt} \cup \mathit{Oprt}$ and $(S, \mathit{Var}, \Delta, \Xi, s_0)$ be a NKA. For every string $\sigma \in (\mathit{At} \cdot \mathit{At}) \cdot \mathit{At}^*$ where At is the set of atoms of $B(\mathit{Var})$ and*

$s \in S$ if the predicate $PAcc(s, \sigma)$ holds then also $IOAcc(s, t^{-1}(\sigma))$ holds.

Proof. The proof is by induction on the length of $\sigma \in (At \cdot At) \cdot At^*$. For the base case, assume $\sigma = \alpha\beta$. We then have

$$\begin{array}{rcll} PAcc(s, \alpha\beta) & \Longrightarrow & \Xi(s, \alpha, \beta) & \text{Definition of } PAcc \\ & \Longleftrightarrow & IOAcc(s, (\alpha, \beta)) & \text{Definition of } IOAcc \\ & \Longleftrightarrow & IOAcc(s, t^{-1}(\alpha\beta)) & \text{Definition of } t^{-1}. \end{array}$$

Assume now $\sigma' = \gamma\sigma''$ where $\gamma \in At$ and $\sigma'' \in At^* >$ Consider the induction step with the string $\sigma = \alpha\beta\sigma'$. First of all, note that $t^{-1}(\alpha\beta\sigma') = (\alpha, \beta) \cdot t^{-1}(\beta\sigma')$ and similarly, $t^{-1}(\beta\sigma') = (\beta, \gamma) \cdot t^{-1}(\sigma'')$. Thus $\beta = head(t^{-1}(\beta\sigma'))$. We have

$$\begin{array}{rcll} PAcc(s, \alpha\beta\sigma') & \Longrightarrow & \exists s' \in \Delta(s, \alpha, \beta).PAcc(s, \beta\sigma') & \text{Def. of } PAcc \\ & \Longrightarrow & \exists s' \in \Delta(s, \alpha, \beta).IOAcc(s, t^{-1}(\beta\sigma')) & \text{Ind. hypothesis} \\ & \Longleftrightarrow & \exists s' \in \Delta(s, \alpha, \beta).IOAcc(s, t^{-1}(\beta\sigma')) \;\&\; \beta \rhd \beta & \text{Def. of } \rhd \\ & \Longleftrightarrow & \exists s' \in \Delta(s, \alpha, \beta).IOAcc(s, t^{-1}(\beta\sigma')) \;\&\; \beta \rhd head(t^{-1}(\beta\sigma')) & \\ & \Longleftrightarrow & IOAcc(s, (\alpha\,\beta) \cdot t^{-1}(\beta\sigma'))) & \text{Def. of } IOAcc \\ & \Longleftrightarrow & IOAcc(s, t^{-1}(\alpha\beta)\sigma') & \text{Def. of } t^{-1}. \end{array}$$

□

As a consequence of the above, we have that if two policies of $pNetKAT$ are language equivalent with respect to the $IOAcc$ then they are also language equivalent for $PAcc$. The converse is, in general, not true, meaning that the equivalence generated by $pNKA$ is coarser than that of $ioNKA$.

6.4.1 From symbolic constraint automata to ioNKA

Next, we show that NetKAT automata can be used to express the semantics of the coordination language Reo [5] too. We use symbolic constraint automata as a semantic model of Reo connectors as presented in Chapter 5, in Definition 5.1 with the addition of accepting states to consider only finite executions.

Definition 6.7. *A symbolic constraint automaton with accepting states (SCA) is a tuple $(S, s_0, I, O, F, \longrightarrow, A)$ such that $(S, s_0, I, O, F, \longrightarrow)$ is an ordinary symbolic constraint automaton and $A \subseteq S$ is a set of accepting states.*

An execution of a symbolic constraint automata with accepting states is defined as in Chapter 5 with the obvious adaptation to finite sequences.

Given a guarded action $\phi(\bar{x}, \bar{y}) = P(\bar{x}) \to \bar{y} := a(\bar{x})$ and atoms α, β assigning values to all variables (and possibly $\perp$ to some input or output ports) we denote by $P(\alpha)$ the evaluation of $P(\bar{x})$ where all occurrences of (free) variables $z \in \bar{x}$ are substituted with

$\alpha(z) \in \mathit{Val}$. Similarly, we denote by $a(\alpha)$ the list of values obtained by evaluating a when all variables $z \in \bar{x}$ get value $\alpha(z) \in \mathit{Val}$. Finally, we say that the Hoare triple $\{\alpha\}\phi\{\beta\}$ holds if

- ϕ is executable under α, that is $\alpha(i) \neq \bot$ for all input ports $i \in \bar{x}$ and $\alpha(o) = \bot$ for all output port $o \in \bar{y}$.
- α is a precondition of ϕ enabling its guard, that is $\alpha \leq P(\alpha)$; and
- β is a postcondition of ϕ changing only the variables in $\bar{y}$ and consuming the value from all input ports in $\bar{x}$, that is $\alpha[a(\alpha)/\bar{y}, \bar{\bot}/\bar{i}] \leq \beta$

where $\alpha[\bar{v}/\bar{y}, \bar{\bot}/\bar{i}]$ is the atom mapping variables in $\bar{y}$ to the respective values in $\bar{v}$, enabling input ports in x to receive values, and remaining unchanged otherwise.

Pre and postconditions of guarded actions are used to construct an ioNKA from a symbolic constraint automaton

Definition 6.8. *A SCA with accepting states $(S, s_0, I, O, F, \longrightarrow, A)$ can be transformed into a ioNKA $(S, Var\,\Delta, \Xi, s_0)$ with $Var = I \cup O \cup F$ and*

- $s' \in \Delta(s, \alpha, \beta)$ *if and only if* $s \xrightarrow{\phi} s'$ *and* $\{\alpha\}\phi\{\beta\}$*;*
- $\Xi(s, \alpha, \beta)$ *if and only if* $s \xrightarrow{\phi} s' \in A$ *and* $\{\alpha\}\phi\{\beta\}$*.*

Here α and β are atoms in $B(Var)$.

Consider, for example, the symbolic constraint automaton in Figure 5.1.(b) of a Fifo1 connector. The corresponding NetKAT automaton has the following transition and observation maps:

$$\Delta(0, \alpha, \beta) = \{1\} \qquad \Delta(1, \alpha', \beta') = \{0\}\,, \text{ and } \Xi(1, \alpha', \beta')$$

for any atom $\alpha \leq i = v, \beta \leq (i = \bot \cdot f = v)$, $\alpha' \leq (o = \bot \cdot f = u)$ and $\beta' \leq (o = u \cdot f = u)$. A string accepted by this automaton is, for example, $([i = v, o = \bot, f = u], [i = \bot, o = \bot, f = v]) \cdot ([i = \bot, o = \bot, f = v], [i = \bot, o = v, f = v])$.

As another example, the ioNKA obtained from the symbolic constraint automaton in Figure 5.1.(c) denoting a filter connector has the following transition and observation maps:

$$\Delta(0, \alpha, \beta) = \{0\} \quad \Delta(0, \alpha', \beta') = \{0\}\,, \text{ and } \Xi(0, \alpha, \beta) \quad \Xi(0, \alpha', \beta')$$

for any atom $\alpha \leq i = v \in P(v)$, $\beta \leq (i = \bot \cdot o = v)$, $\alpha' \leq i = v \notin P(v)$, and $\beta' \leq (i = \bot)$.

Correctness of the translation from symbolic constraint automata to NKA with respect to the following notion of bisimulation is immediate by construction. However, this bisimulation relation will become more interesting when proving the correctness of the parallel composition of two automata.

Definition 6.9. *Given a symbolic constraint automaton with accepting states $C = (S, s_0, I, O, F, \longrightarrow, A)$ and an NKA $N = (Q, V\,\Delta, \Xi, q_0)$ with $V = I \cup O \cup F$, we say that a binary relation $R \subseteq S \times T$ is a bisimulation if $(s_0, q_0) \in R$ and whenever $(s, q) \in R$ then*

- *for all $s \xrightarrow{\phi} s'$ and $\{\alpha\}\phi\{\beta\}$ there exists $q' \in \Delta(q, \alpha, \beta)$ such that $(s', q') \in R$;*
- *for all $q' \in \Delta(q, \alpha, \beta)$ there exists $s \xrightarrow{\phi} s'$ such that $\{\alpha\}\phi\{\beta\}$ and $(s', q') \in R$;*
- *$\Xi(q, \alpha, \beta)$ holds for all $s \xrightarrow{\phi} s' \in A$ and $\{\alpha\}\phi\{\beta\}$;*
- *for all $\Xi(q, \alpha, \beta)$ there exists $s \xrightarrow{\phi} s'$ such that $\{\alpha\}\phi\{\beta\}$ and $s' \in A$.*

Transitions with guarded actions must be matched by transitions with all pre and postconditions of those actions, and vice-versa, every pair of pre and postconditions must be related to at least one guarded action. Note that if two states q and q' of an ioNKA are language equivalent with respect to *IOAcc*, and a state s of an SCA is bisimilar to q then s is bisimilar to q' too, where bisimilarity is the largest bisimulation between an SCA and a ioNKA.

6.4.2 Composing ioNKA

We conclude this section with a very brief presentation of a composition operator between NetKAT automata with input and output ports inspired by the one used in Reo [13]. The idea is that the two automata synchronize via all (and only) the shared ports that are input for one automaton and output port for another. To avoid broadcasting, shared ports become local fields. No other synchronization is allowed, as all fields are only visible within the scope of an automaton. The composition is defined only when no causality problem can arise when the input and output ports of two automata are synchronized in the same step.

Definition 6.10. *Let $N_1 = (S_1, V_1\,, \Delta_1, \Xi_1, s_1)$ and $N_2 = (S_2, V_2, \Delta_2, \Xi_2, s_2)$ be two non-deterministic NetKAT automata with $V_i = I_i \cup O_i \cup F_i$ for $i = 1, 2$ such that F_1 and F_2 are disjoint sets of fields in Fld. Assume that for every pair of (α_1, β_1) and (α_2, β_2) and state s_1 and s_2 such that either $\Delta_1(s_1, \alpha_1, \beta_1) \neq \emptyset$ and $\Delta_2(s_2, \alpha_2, \beta_2) \neq \emptyset$ or both $\Xi_1(s_1, \alpha_1, \beta_1)$ and $\Xi_2(s_2, \alpha_2, \beta_2)$ holds, the two automata synchronize only on the input ports used by one and output ports used by the other, but not on both input and output ports at the same time, that is*

$$I(\alpha_1, \beta_1) \cap O(\alpha_2, \beta_2) \neq \emptyset \Rightarrow O(\alpha_1, \beta_1) \cap I(\alpha_2, \beta_2) = \emptyset\,.$$

Then the composition $N_1 \bowtie N_2$ is defined as the ioNKA $(S, V\,\Delta, \Xi, s_0)$ where:

- *$S = S_1 \times S_2$;*
- *$s_0 = \langle s_1, s_2 \rangle$;*

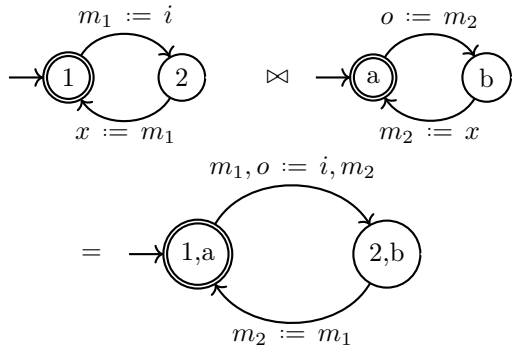


Figure 6.3: Symbolic constraint automata for Fifo 1 composed with Fifofull

- $V = I \cup O \cup F$ *with* $I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$, $O = (O_1 \setminus I_2) \cup (O_2 \setminus I_1)$, *and* $F = F_1 \cup F_2 \cup (I_1 \cap O_2) \cup (I_2 \cap O_1)$;
- $\langle s', t' \rangle \in \Delta(\langle s, t \rangle, \alpha, \beta)$ *if* $s' \in \Delta_1(s, \alpha_1, \beta_1)$ *and* $t' \in \Delta_2(t, \alpha_2, \beta_2)$ *such that if* $x \in I_1 \cap O_2$ *then* $\alpha_1(x) = \beta_2(x) \neq \bot$, *and if* $x \in O_1 \cap I_2$ *then* $\alpha_2(x) = \beta_1(x) \neq \bot$;
- $\Xi(\langle s, t \rangle, \alpha, \beta)$ *holds if both* $\Xi_1(s, \alpha_1, \beta_1)$ *and* $\Xi_2(t, \alpha_2, \beta_2)$ *hold, such that if* $x \in I_1 \cap O_2$ *then* $\alpha_1(x) = \beta_2(x) \neq \bot$, *and if* $x \in O_1 \cap I_2$ *then* $\alpha_1(x) = \beta_2(x) \neq \bot$,

where, in the last two items, for all $i \in I, o \in O$ *and* $f \in F$,

$$\alpha(i) = \begin{cases} \alpha_1(i) & \text{if } i \in I_1 \setminus O_2 \\ \alpha_2(i) & \text{if } i \in I_2 \setminus O_1 \end{cases} \qquad \beta(i) = \begin{cases} \beta_1(i) & \text{if } i \in I_1 \setminus O_2 \\ \beta_2(i) & \text{if } i \in I_2 \setminus O_1 \end{cases}$$

$$\alpha(o) = \begin{cases} \alpha_1(o) & \text{if } o \in O_1 \setminus I_2 \\ \alpha_2(o) & \text{if } o \in O_2 \setminus I_1 \end{cases} \qquad \beta(o) = \begin{cases} \beta_1(o) & \text{if } o \in O_1 \setminus I_2 \\ \beta_2(o) & \text{if } o \in O_2 \setminus I_1 \end{cases}$$

$$\alpha(f) = \begin{cases} \alpha_1(f) & \text{if } f \in F_1 \cup (I_1 \cap O_2) \\ \alpha_2(f) & \text{if } f \in F_2 \cup (I_2 \cap O_1) \end{cases} \qquad \beta(f) = \begin{cases} \beta_1(f) & \text{if } o \in F_1 \cup (O_1 \cap I_2) \\ \beta_2(f) & \text{if } o \in F_2 \cup (O_2 \cap I_1) \end{cases}$$

The above operation is congruence with respect to language equivalence as defined in Definition 6.6 and is correct with respect to the parallel operator for symbolic constraint automata as given in [34] in the sense that if there is a bisimulation relation between two symbolic constraint automata and two ioNKA then we can find a bisimulation between their respective parallel composition.

As an example, we show the composition of two SCA constraints automata, one representing a FIFO buffer of size 1 taking values from the input port i, buffering the field m_1 and outputting the buffered value at the port x, and the other similar but with input port x output port o and starting with a full buffer m_2 instead of the empty m_1.

The two symbolic constraint automata are described at the top of Figure 6.3, while their composition is the SCA depicted at the bottom. We concentrate on the synchronization of the transition execution of the action $m_1 := i$ with that executing the action

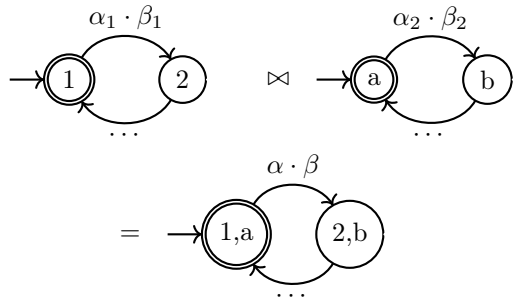


Figure 6.4: pNKA for Fifo 1 composed with Fifofull

$o := m_2$. They are implemented in the ioNKA in Figure 6.4, where $\alpha_1 = [i = v_1,\, x = v_2,\, m_1 = v_0]$, $\beta_1 = [i = \bot,\, x = v_2,\, m_1 = v_1]$, $\alpha_2 = [x = u_1,\, o = \bot,\, m_2 = u_2]$, and $\beta_2 = [x = u_1,\, o = u_2,\, m_2 = u_2]$. Here v_1 is the data received as input by the first connector and u_2 the one output by the second connector, while v_2 and u_1 are values (possibly bottom) already present at the output and input port of the two connectors, respectively. Following the definition we get the following sets of "used" ports:

$$I(\alpha_1, \beta_1) = \{i\} \qquad O(\alpha_1, \beta_1) = \emptyset\,,$$
$$I(\alpha_2, \beta_2) = \emptyset \qquad O(\alpha_2, \beta_2) = \{o\}\,.$$

The resulting composition of the above transitions results in the precondition $\alpha = [i = v_1,\, x = v_2,\, o = \bot,\, m_1 = v_0,\, m_2 = u_2]$, and postcondition $\beta = [i = \bot,\, x = v_2,\, o = u_2,\, m_1 = v_1,\, m_2 = u_2]$, where x becomes a local field. Note that if we create a loop and let the port $o = i$ in the second SCA then we have a problem of causality and the composition cannot take place. The problem could be solved by inserting e.g., a (synchronous) connector between o and i.

We leave it as future work the extension of the syntax of pNetKAT with an explicit declaration of input and output ports for each policy, that can be combined with the join operation $\bowtie$ as defined above.

6.5 Conclusion and future work

We extended NetKAT with concurrency and communication via shared ports. We followed two semantics lines using non-deterministic constraints automata: one observing successful synchronization only, and another allowing interaction with the environment. In both cases, communication by ports played an important role, and the second one can be used as a compositional model of the Reo coordination language too.

We focussed on the operational semantics and compositionality. A possible next step

is the study of axiomatizations of our two extensions. From a more practical point of view, we could use our work on model checking Reo with SPIN [34] to obtain a model checker for concurrent NetKAT. An orthogonal extension is to combine concurrency with stacks to model VLANs [108].

Chapter 7

Towards causality reasoning for SCA

In this chapter, we introduce an NFA causal model based on counterfactuals, inspired by the seminal works on causal analysis by Halpern and Pearl, adapted to finite automata models and with safety properties defined by regular expressions [26]. The latter encodes undesired execution traces. We devise a framework that computes actual causes, or minimal traces that lead to states enabling hazardous behaviors. Furthermore, our framework exploits counterfactual information and identifies modalities to steer causal executions toward alternative safe ones. This can provide systems engineers with valuable data for actual debugging and fixing erroneous behaviors. Our framework employs standard algorithms from automata theory, thus paving the way to further generalizations from finite automata to richer structures like probabilistic, KAT, and NetKAT automata. The ultimate goal is to extend the framework to symbolic constraint automata [34], so as to be applied for causal reasoning on SDNs, for example by using our Reo model presented in Chapter 4.

7.1 Introduction

Causal models and associated causal inference machinery are precious tools for the interpretation and explanation of systems failures. Current testing and verification frameworks such as equivalence checking, for instance, assess whether or not systems comply with their specifications, and at most will produce a counterexample in case the system fails. Causal analysis, instead, plays an important role in explaining complex phenomena that are actual sources of hazards by adding, for example, additional information to

counterexamples on how to avoid the hazard.

A notion of causality often embraced and adopted by computer scientists was introduced by Halpern and Pearl in their seminal works [47, 46]. Their causal model encodes complex logical structures of multiple events that contribute to undesired effects, or hazards. In essence, the model is based on the so-called alternative worlds, originally proposed by Lewis [76]. In short, Lewis assumes the existence of worlds satisfying a sufficiency condition, where both the cause and the effect occur, and other worlds satisfying a necessity condition, in which neither the cause nor the effect occurs. This enables formulating the counterfactual argument, which defines a first condition to be satisfied by a cause, namely: when the presumed cause does not occur, the effect will not occur either. More complex aspects such as redundancy and preemption are also captured by the causal model in [47, 46]. For intuition, redundancy refers to simultaneous events that play the same role in enabling an undesired effect. Orthogonally, preemption refers to subsequent events that have the same power to enable the effect. In both cases, the counterfactual test alone cannot determine the actual cause. Last, but not least, causes in the spirit of [46] comply with a minimality requirement which guarantees that only the relevant set of causal events is identified.

Related work. Over time, several notions of causality have been proposed, each of which is tailored to the type of the system under analysis, and associated correctness specifications. Of particular interest for this chapter are the works in [74, 23, 25]. The aforementioned results propose trace-based adoptions of causality á la Halpern and Pearl, applicable to automata models. These, in combination with model checking-based methodologies, enabled computing causes for the violation of safety and liveness properties in Kripke structures and labeled transition systems, for instance.

Our work is closely related to the contribution in [25]. Given an automaton model, the naive goal is to identify the shortest sequence of actions that enable the effect, i.e., that can bring the system into a hazardous state. These are called “causal traces". Note that, in contrast with the often tedious counterexamples identified by model-checkers, the minimality of causal traces implies concise descriptions of systems faults. Thus, causal traces encode essential information for systems engineers, for instance, and they can serve as a debugging aid. As previously stated, in the spirit of Halpern and Pearl, our definition of causality imposes a sufficiency condition: namely, whenever a causal trace is executed, the effect is reached as well. However, important information on how to avoid/fix hazardous behaviors can be extracted based on the aforementioned set of alternative worlds (or traces in our model), that do not lead to an undesired effect. Hence, we designed our causal model in the spirit of the counterfactual criterion of Lewis and identified modalities to avoid hazardous scenarios. Similarly to [74, 23, 25], we call these escape options – “events causal by their non-occurrence". This information

can be exploited to steer an execution towards an alternative safe one, with immediate applicability in synthesizing schedulers, for instance.

A rich body of work successfully exploited the counterfactual argument for fault analysis and debugging techniques. Examples related to counterexample explanation in model checking are the works in [44, 43, 92], for instance. In [43] the authors propose a framework for understanding errors in ANSI C programs, based on distance metrics for program executions. In [44] the cause describing the error includes the identification of source code fragments crucial to distinguishing success from failure and differences in invariants between failing and non-failing runs. Distance criteria have also been exploited in [92], in combination with the so-called nearest neighbor queries to perform fault localization. The why-because-analysis in [71] was used to reason about aviation accidents, in a framework where Lamport's Temporal Logic of Actions (TLA) described both the behavior of a system, the (history of) hazards and the sequence of the states leading to an accident. The work in [114] provides a comprehensive approach to systematic debugging including, among others, delta debugging – a technique for isolating minimal input to reproduce an error.

For finer notions of causal dependencies that distinguish between interleaving and true concurrency, for instance, we refer to event structures [8, 88]. Nevertheless, in our work, we adhere to the approaches in [74, 23, 25], and do not take into consideration the order of events along execution traces.

Our contributions. We propose a shifting from the bisimulation setting presented in [25] to a trace-based setting in the context of regular languages and automata theory. The benefits are multifold. For instance, the paradigm change facilitates the application of more standard algorithms from automata theory, in contrast with the rather ad-hoc procedures in [74, 23, 25]. Furthermore, the current framework enables the use of an expressive logic for defining safety properties in terms of regular expressions (or automata), instead of the ordinary Hennessy-Milner logic. The language-based approach to causality enables representing both hazards and causal explanations in terms of automata – a format better accepted by engineers. In addition, in this chapter, we use regular languages (or full regular expressions including Kleene-star) to encode the non-occurrence of events. Previous related works such as [74, 23, 25] can only provide finite sets of runs steering an execution towards an alternative safe one. Orthogonal to the aforementioned results, the current approach entails a "may" semantics of causality, instead of "must"; nevertheless, we believe that the approach can be easily modified to cater to the "must" version. Besides, in contrast with the results in [25], steering executions are guaranteed not to jump over hazardous states by simply concatenating sequences causal by their non-occurrence and the causal trace. The ultimate goal of the current work is to generalize from finite automata to richer structures like probabilistic automata and NetKAT

automata [4, 37].

Structure of the chapter. In Section 7.2 we provide an overview of regular languages and associated automata theory aspects. A running example is introduced in Section 7.3. Section 7.4 defines the language-based model of causality, whereas in Section 7.5 we show how to compute actual causes and safe computations. In Section 7.6 we provide an experimental evaluation of our method and in Section 7.7 we discuss how our model can be extended with tests and assignments. Section 7.8 concludes our work.

7.2 Preliminaries

In this section, we recall a few basic facts about regular languages, finite automata, and regular expressions [80].

Let A be a finite set of actions that we refer to as an *alphabet*. A *word* or *string* over A is a finite sequence $a_1 \dots a_n$ of elements from A. We denote by ε the empty word, i.e. the sequence of length 0, and write A^* to denote the set of (possibly empty) words over A. A *language* L is just a subset of words, that is $L \subseteq A^*$. We call a word w' to be a *prefix* of a word w whenever $w = w'w''$. A word w' is said to be a *sub-word* of a word w, if w' is obtained by deleting one or more elements of A at some not necessarily adjacent positions in w. We denote by $sub(w)$ the set of all sub-words of w. Note that $sub(\varepsilon) = \emptyset$. Also, $\varepsilon \in sub(w)$ but $w \notin sub(w)$ for every non empty word w.

A *finite automaton* (FA) is a 5-tuple $M = (S, A, i, \longrightarrow, F)$, where S is a finite set of states, $i \in S$ is the initial state, $F \subseteq S$ is the set of accepting states and $\longrightarrow \subseteq S \times A \times S$ is the transition relation. For simplicity, we write $s \xrightarrow{a} t$ whenever $(s, a, t) \in \longrightarrow$. A transition relation is called deterministic if for all $s \in S$ and $a \in A$ if $s \xrightarrow{a} t_1$ and $s \xrightarrow{a} t_2$ then $t_1 = t_2$.

A string $w \in A^*$ is accepted by an automaton M from a state s if either (1) $w = \varepsilon$ and $s \in F$, or (2) $w = aw'$ and there exist $s \xrightarrow{a} t$ such that w' is accepted by M from the state t. The *language accepted* by a FA M is the set $L(M) = \{w \in A^* \mid M \text{ accepts } w \text{ from } i\}$. Since for every FA M, we can build an FA N with a deterministic transition relation such that $L(M) = L(N)$, without loss of generality we will consider only finite automata with a deterministic transition relation.

A language L over the alphabet A is said to be *regular* if there exists a finite automaton M accepting it, that is $L(M) = L$. The class of all regular languages is closed under union, intersection, concatenation, complement, and Kleene star. Here language union and intersections are the usual set-theoretic operations, whereas concatenation of two languages L_1 and L_2 is given by the set $L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$. Finally, for a language L, its Kleene star closure is defined by $L^* = \bigcup_{n \in \mathbb{N}} L^n$ where $L^0 = \{\varepsilon\}$ and $L^{n+1} = L \cdot L^n$ for all $n \in \mathbb{N}$, thus denoting the concatenation of a language with

itself a finite number of time.

In this chapter, we are interested in system communicating by message passing, and thus we will always assume that the alphabet A is partitioned in three disjoint subsets A_I, A_O, and A_P of input, output, and private actions, respectively. Notationally, for $a \in A$, we write $a?$ if a is an input action in A_I and $a!$ if a is an output action in A_O, and use no markings for private actions in A_P. We use σ to denote an action that can be either input, output, or private.

Let A and B be two alphabets with disjoint private actions, and assume the set P is disjoint from Q. Given two finite automata $M = (P, A, i, \longrightarrow_M, E)$ and $N = (Q, B, j, \longrightarrow_N, F)$ their *parallel composition* is defined by the finite automaton $M \mid\mid N = (P \times Q, \Sigma, \langle i, j\rangle, \longrightarrow, E \times F)$ where $\Sigma_I = (A_I \setminus B_O) \cup (B_I \setminus A_O)$, $\Sigma_O = (A_O \setminus B_I) \cup (B_O \setminus A_I)$, $\Sigma_P = (A_P \cup B_P) \cup (A_I \cap B_O) \cup (A_O \cap B_I)$, and $\longrightarrow$ is the least transition relation such that

$$\frac{p \xrightarrow{\sigma}_M p' \quad \sigma \notin B}{\langle p, q\rangle \xrightarrow{\sigma} \langle p', q\rangle} \qquad \frac{q \xrightarrow{\sigma}_N q' \quad \sigma \notin A}{\langle p, q\rangle \xrightarrow{\sigma} \langle p, q'\rangle}$$

$$\frac{p \xrightarrow{a?}_M p' \quad q \xrightarrow{a!}_N q'}{\langle p, q\rangle \xrightarrow{a} \langle p', q'\rangle} \qquad \frac{p \xrightarrow{a!}_M p' \quad q \xrightarrow{a?}_N q'}{\langle p, q\rangle \xrightarrow{a} \langle p', q'\rangle}$$

The topmost rules are about either private actions that are not affected by the other automaton or communication actions that do not involve the other automaton. The two rules at the bottom are about complementary communication actions $a!$ and $a?$ that are synchronized resulting in the private action a. Note that when $A = B$ with $A_I = B_O$, $A_O = B_I$, and $A_P = B_P = \emptyset$ then parallel composition reduces to the product automata where all actions synchronize. In the case A is completely disjoint from B then parallel composition results in the so-called shuffle product. Other variations of synchronization products could be defined similarly, including multi-process synchronization, hiding of successful communication, value passing synchronization (for a finite value domain), and synchronization parameterized by a finite subset of actions.

For the characterization of the parallel composition of two languages, we need first to introduce the *projection* function. Given two alphabets A_1 and A_2 we define the projection $\pi_i{:}(A_1 \cup A_2)^* \to A_i^*$ by $\pi_i(\varepsilon) = \varepsilon$, and $\pi_i(\sigma \cdot w) = \sigma \cdot \pi_i(x)$ if $\sigma \in A_i$, and $\pi_i(w)$ otherwise. Because projections are surjective functions they have inverse π_i^{-1} returning the set of strings that are projected into a given one. More precisely, we define the *inverse projection* by $\pi_i^{-1}(w) = \{x \in (A_1 \cup A_2)^* \mid \pi_i(x) = w\}$ for every $w \in A_i^*$. Projections and their inverses can extended to languages by applying them to all the strings in the language. In general we have that $\pi_i(\pi_i^{-1}(L)) = L$ but for the converse it only holds that $L \subseteq \pi_i^{-1}(\pi_i(L))$. Note that if two alphabets A_1 and A_2 have disjoint private actions and we partition $A_1 \cup A_2$ as in the alphabet of the parallel composition of two automata, then projections will assign private actions of $A_1 \cup A_2$ to either private, input or output actions in A_i unambiguously. Similarly, inverse projections assign private

actions to private actions but may assign input and output actions to private ones.

The *parallel composition* of two languages $L_1 \subseteq A_1^*$ and $L_2 \subseteq A_2^*$ is the language $L_1 \mid\mid L_2$ on the alphabet $A_1 \cup A_2$ defined as $\pi_1^{-1}(L_1) \cap \pi_2^{-1}(L_2)$. The intersection takes care that dual communication actions will be synchronized, and that disjoint private events will be shuffled with the others. As expected, we have that $L(M_1 \mid\mid M_2) = L(M_1) \mid\mid L(M_2)$, implying that regular languages are closed under parallel composition [96].

We conclude this section by introducing extended regular expressions, that we may use as alternative syntax to FAs to reason about causality in complex systems composed of several components potentially communicating with each other.

Given an alphabet A including communication actions, *extended regular expressions* are given by the following grammar:

$$e ::= 0 \mid 1 \mid a \mid a? \mid a! \mid e\,;e \mid e + e \mid e \mid\mid e \mid e^*\,, \tag{7.1}$$

where $a \in A_P$, $a?$ implies $a \in A_I$, and $a!$ implies $a \in A_O$. In process theoretic terms, 0 denotes no behavior, and 1 denotes a terminating process. The further building blocks of processes are (communication) actions. Processes can be composed sequentially, non-deterministically, in parallel, or can loop a finite number of times. Communication between process terms is performed based on synchronizations between opposite communication actions, that play thus a sender, respectively, receiver role. In the sequel, we often use A as shorthand for the regular expression obtained by the finite set of every action in A, and $\neg a$ as a shorthand for the set of every action in A except a. Note that in general, we could extend negation to all regular expressions, as regular languages are closed under complement.

Ordinary regular expressions are expressions without any parallel composition. Except for the parallel composition, we assume that an action cannot be used as input and output in the same 'sequential' expression, i.e., regular expression with no occurrence of the $\mid\mid$ operator. With this mild restriction, we can associate each regular expression e a language $L(e)$ inductively as follows:

$$\begin{array}{lll} L(0)=\emptyset & L(e_1\,;e_2)=L(e_1)\cdot L(e_2) & \\ L(1)=\{\varepsilon\} & L(e_1+e_2)=L(e_1)\cup L(e_2) & L(e^*)=L(e)^* \\ L(a)=\{a\} & L(e_1 \mid\mid e_2)=L(e_1) \mid\mid L(e_2) & \end{array}$$

It is well known [63] that the language of an ordinary regular expression is regular. The same holds for our extended regular expressions, as we have seen that regular languages are closed under parallel composition. This implies that for every (extended) regular expression e there exists an automaton M such that $L(e) = L(M)$. We will not describe the construction here as it is outside the scope of this chapter.

7.3 A Railway crossing Example

In this section, we recall the railway crossing example from [25] and adapt it to our present setting. The example consists of a car, a train, and a gate of a crossing that communicates with the train. The gate can communicate its status of being closed ($Gc!$) or open ($Go!$). The status changes to closed only after the gate receives a message from the train that is approaching the crossing ($Ta?$), and it can change to open only after it receives the message that the train leaves the crossing ($Tl?$). The behavior of the gate is described by the following regular expression:

$$G = (Go!^* ; (1 + Ta? ; Gc!^* ; Tl?))^* .$$

When a train is approaching the crossing, it sends a message ($Ta!$). After that, it will enter the crossing (Tc) and then send a message informing its departure from the crossing ($Tl!$). This behavior is described by the following regular expression:

$$T = Ta! ; Tc ; Tl! .$$

Finally, a car can approach the crossing (Ca), wait as long as the gate is closed ($Gc?$), eventually observe the gate being open ($Go?$), and only then it may enter the crossing (Cc) and leave the crossing afterward (Cl). The regular expression encoding is given by:

$$C = Ca ; Gc?^* ; Go? ; Cc ; Cl .$$

The FAs corresponding to the above three regular expressions are illustrated in Figure 7.1. Note that the car can enter the crossing only after the gate is open, whereas the gate enters the state of being open only after a train signals its departure.

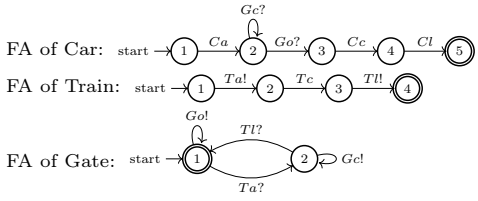


Figure 7.1: The Car, Train, and Gate as FAs

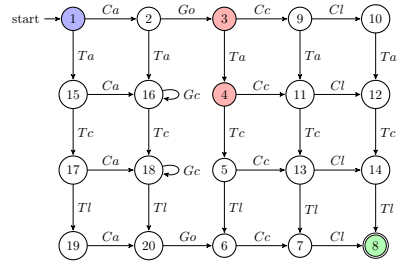


Figure 7.2: The Railway System as a FA

In Figure 7.2 we see the automaton describing the railway system that results from the parallel composition of the three regular expressions: $C \mid\mid T \mid\mid G$ where, for simplicity, we renamed the states. For example, the initial state ① corresponds to the state $\langle 1, 1, 1 \rangle$

and the only accepting state is ⑧ corresponding to $\langle 5, 4, 1\rangle$. The red states ③ and ④ will be used in the next section as examples of states leading to a hazard situation: a car entering the crossing and not leaving it before the train enters the crossing too.

7.4 A Language-based causal model

In this section, we introduce a notion of causality with respect to a so-called *hazard*, or *effect* expressed in terms of regular expressions. The current causal framework is inspired by the model introduced in [25] and massaged into the setting of FAs to use trace semantics instead of bisimulation, and define different system properties in terms of regular expressions (such as reachability) instead of the ordinary Hennessy-Milner logic.

In short, a hazard is a regular language specified by a regular expression e (or the corresponding automaton). It is said to occur in a FA M representing our model whenever there is a finite (and possibly empty) string $c = a_0 \dots a_n$ in M such that after c we may observe the hazard, that is, $L(c\,;e) \cap L(M) \neq \emptyset$. In this case, we say that c may enable the hazard e in M. Additional conditions that have to be satisfied by c, such as minimality and non-occurrence of events, are formalized in Definition 7.1.

For an intuition, consider the railway crossing example of the previous section. A hazardous situation can happen whenever both the train and the car enter the crossing, and none of them leaves the crossing before the other one enters it. The regular expression encoding this hazard is:

$$e = (Cc\,;(\neg Cl)^*\,;Tc + Tc\,;(\neg Tl)^*\,;Cc)\,;A^* \tag{7.2}$$

Note that the hazard situation can terminate with any string in A^*. This is to guarantee that after a trace c enables e, their concatenation will contain behaviors accepted by the automaton, and thus the hazard is observed. It is straightforward to see that in the FA in Fig. 7.1 it is possible to reach the above hazard with the string $c_1 = Ca\,Go$ leading to the state ③, but also with the string $c_2 = Ca\,Go\,Ta$ leading to the state ④. The intersection of the language of the hazard e with that of the automaton M starting from either state ③ or ④ instead of ① is non-empty. Furthermore, state ③ and ④ are both reachable from the initial state ①.

We may say that c_1 does a better job at describing the relevant sequence of actions that, if triggered, lead to a hazard because it is a *minimal* sequence enabling it. Moreover, we see that it is possible to avoid the hazard by "decorating" the string c_1 with the strings Ta, $Tc\,Tl$ and, respectively, $Cc\,Cl$. This can result, for instance, in the string $w = Ta\,\underline{Ca}\,Tc\,Tl\,\underline{Go}\,Cc\,Cl$ which does not lead to a hazard. Sequences such as Ta, $Tc\,Tl$ and $Cc\,Cl$ are called *causal by non-occurrence* in works such as [23, 25]. Non-

occurrence is essential for describing how certain dangerous situations, if controllable, can be avoided within a system. This concept plays an important role in our definition of causality.

As formalized in Definition 7.1, the non-occurrence of events is captured in terms of the so-called *computations* [25]. The latter are strings in a regular language, typically denoted by π, built on top of a string $c = a_0 \dots a_n$, and "decorated" with strings $d_0^i, \dots, d_{n+1}^i$, with $i \in I$, where I is a finite set of integers, such that:

$$w \in \pi \quad \Rightarrow \quad w = d_0^i a_0 d_1^i \cdots a_n d_{n+1}^i \,.$$

Intuitively, given a trace c that enables a hazard, strings in π describe all the alternative runs (such as w above) that execute all actions in c and avoid the hazard. The only requirement is that all strings specified by π are observable executions of M; i.e., for a given FA M, $\pi \subseteq L(M)$. Notice that π being a regular language means that it can be expressed as a regular expression r, and because all strings in π contain c as subword, we have $r = \Sigma_{j,k} r_k^j$ with $r_k^j = r_0^j; a_0; r_1^j; \dots a_k; r_{k+1}^j$ for some finite indexes j and k and regular expressions r_{k+1}^j. For simplicity, we sometimes write r instead of π.

The next definition formally introduces *decorated causes* for an FA M with respect to a hazard e.

Definition 7.1 (Causality for FAs)**.** *Let $M = (S, A, s_0, \to, F)$ be a FA, e be a regular expression over A, denoting a hazard, and $c \in A^*$. We say that the computation π built on top of c, with $\pi \subseteq L(M)$, is a* decorated cause *of the hazard e if*

AAC1: *The string c may enable e – $L(c\,;e) \cap L(M) \neq \emptyset$*

AAC2.1: *If the effect e is not observed then it has not been caused by c –*
$\forall w \in L(M) \setminus L(A^*\,;e) : (L(w\,;e) \cap L(M) = \emptyset) \Rightarrow (c \notin sub(w) \vee w \in \pi)$.

AAC2.2: *Strings of π are safe, i.e., they do not cause the effect e –*
$\forall w \in \pi : w \notin L(A^*\,;e) \wedge (L(w\,;e) \cap L(M) = \emptyset)$

AAC3: *Minimality –*
for all $c' \in sub(c)$ there is no computation π' built on top of c' with $\pi' \subseteq L(M)$, that satisfies **AAC1**–**AAC2.2** *with respect to the string c' and the hazard e.*

We call c as above a causal trace *and sometimes write $Cause_c(e, M)$ to denote the corresponding decorated cause π. We let $Causes(e, M)$ be the union of all $Cause_c(e, M)$.*

Intuitively, **AAC1** identifies a scenario where the string c enables the hazard e in M. Note that **AAC1** entails a "may" semantics of causality, instead of "must", as c does not always have to lead to e. Catering for the "must" version requires modifying **AAC1** to $L(c\,;e) \subseteq L(M)$. **AAC2.1** is a necessity condition according to which, if a word w cannot enable e, then either w does not contain the causal trace c (meaning it

is an execution bringing not to the hazard), or it has been decorated with events that eliminate the possibility of executing the hazard. Note that **AAC2.1** can be equivalently expressed (by modus tollens) as a sufficiency condition stating that a string w enables the hazard e whenever the causal trace is contained in w but it is not decorated with elements causal by their non-occurrence that would avoid the execution of the hazard:

$$\forall w \in L(M) \setminus L(A^*\,;e) : (c \in sub(w) \wedge w \notin \pi) \Rightarrow (L(w\,;e) \cap L(M) \neq \emptyset)$$

AAC2.2 requires causal traces decorated with events causal by their non-occurrence to avoid the hazard. Furthermore, note that c itself cannot be a safe computation in π, because otherwise **AAC2.2** would contradict **AAC1** . Observe that **AAC2.2** is reminiscent of the traditional counterfactual criterion of Lewis, as it allows us to test the dependence of e on c under certain contingencies encoded, in our case, in terms of non-occurrence of events. We refer to [47] for more insight on the so-called *structural contingencies*. **AAC3** is the minimality condition that requires considering decorated causes entailed by the shortest causal traces c satisfying **AAC1** –**AAC2.2** .

We conclude the section with a few examples intended to clarify certain aspects of the above definition and the differences with the work [25]. To begin with, we illustrate the role played by loops in the decorations of computations.

Example 1. Consider the automaton M_1 in Figure 7.3 and let the hazard be expressed by the regular expression $e = c\,;A^*$, meaning that we have to avoid executing action c.

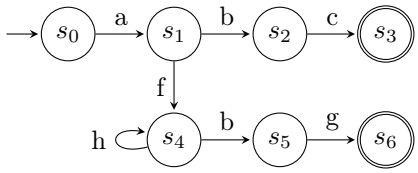


Figure 7.3: Automaton M_1

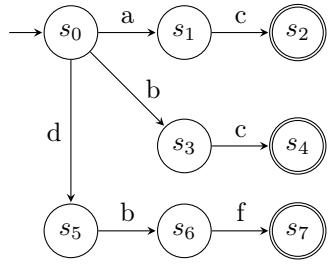


Figure 7.4: Automaton M_2

Clearly, the string $a\,b$ is a possible cause for the hazard. Hence, $Cause_{ab}(e, M_1)$ for this example can be encoded via the regular expression: $a\,;f\,;h^*\,;b\,;g$. Note that as a result of considering the decorations as regular expressions, all finite repetitions of the loop are conveniently represented with the Kleene star operator. The work in [25] handles loops in the decorations by unfolding the loop only a finite number of times specified a priori, hence, only the string afh^nbg would be describing hazard avoidance, for all $n \leq k$ and some fixed k.

In the second example, we consider the case when there are no possible decorations

to steer a causal trace away from its hazard.

Example 2. Consider the automaton M_2 in Figure 7.4 and let the hazard be as before expressed by the regular expression $e = c\,;A^*$.

In this example, there are two possible causal traces, namely, a and b. There are no possible decorations for the causal trace a to make it avoid the hazard, whereas, there exists a decoration for the causal trace b with $Cause_b(e, M_2) = d\,;b\,;f$. Whenever there are no computations π satisfying Definition 7.1 for e in M w.r.t. a trace c, we say that the hazard e, if enabled by c, is *unavoidable* in M.

In the above two examples, there was no actual difference if we had used c as a hazard instead of the regular expression $c\,;A^*$. In the next example, we show an FA where the two expressions entail different decorated causes.

Example 3. Consider the automaton M_3 in Figure 7.5 and the hazards $e = c\,;A^*$ and $e' = c$. For both hazards, ab is the causal trace, but

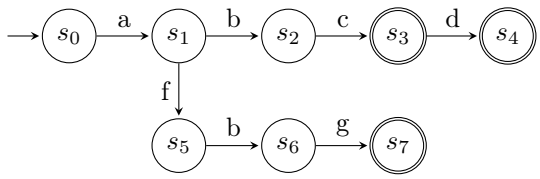


Figure 7.5: Example 3

$$\begin{aligned} Cause_{ab}(e, M_3) &= a\,;f\,;b\,;g \\ Cause_{ab}(e', M_3) &= a\,;f\,;b\,;g \;+\; a\,;b\,;c\,;d \end{aligned}$$

Observe that the string $abcd$ is considered safe (i.e., avoids the hazard) according to $Cause_{ab}(e', M_3)$ but is not considered safe in $Causes_{ab}(e, M_3)$, where the string $afbg$ is considered safe in both cases. This is different than the usual notion of safety (modeled as in e and thus forbidding any possible continuation after the hazard) as e' allows to overpass the hazard if the system does not stop there. The expression e' asserts that the trace cannot halt with the action c. Accordingly, both $abcd$ and $afbg$ are valid strings that satisfy this condition and thus avoid the hazard e'. On the other hand, the expression e asserts that the action c followed by any possible sequence of actions (i.e., in A^*) constitutes a violation, hence, the action c cannot be observed at any point in execution. Therefore, only $afbg$ is a valid execution that will avoid the hazard e. It is essentially not possible to define properties similar to e with the approach in [25], as they allow jumping over a hazardous state while executing strings in π.

7.5 Computing causes

Given a FA $M = (S, A, i, \longrightarrow, F)$ and an effect specified by a regular expression e on A, we show an algorithm for computing the set $Causes(e, M)$ using standard operations on automata and graphs. The algorithm first computes the set of loop-free traces that lead to the hazard e. Then, for each one of them, it determines the associated computation satisfying conditions **AAC2.1** – **AAC2.2** in Definition 7.1. The union of all such computations will give a first approximation of the set $Causes(e, M)$. We will then show below how to obtain precisely the set $Causes(e, M)$ by requiring the minimality condition **AAC3** in Definition 7.1.

Algorithm 1: Computing Causes

Input: A FA $M = (S, A, i, \longrightarrow, F)$, an effect e.
Output: The set of decorated causes $Causes(e, M)$.

(1) Compute the set of traces that lead to e by following the steps:

(1.1) For all $s \in S$, construct the FA $P_s = (S, A, s, \longrightarrow, F)$ and compute the following intersection:
$L(P'_s) = L(P_s) \cap L(e)$.

(1.2) Construct the automaton $P = (S, A, i, \longrightarrow, F')$ where
$F' = \{s \mid L(P'_s) \neq \emptyset\}$.

(1.3) Compute all simple paths from the intial states i and a final state $f \in F$ in P.

(1.4) Let $CausalTraces$ be the set of all strings in $L(P)$ labeling the paths computed in (1.3).

(2) For all $c = a_0 \ldots a_n \in CausalTraces$, compute $Cause_c$(e, M) by :

$$(L(A^*;a_0;A^*;\ldots;A^*;a_n;A^*) \setminus \{c\}) \cap (L(M) \setminus (L(A^*;e) \cup L(P)))$$

(3) Return the union of all the languages computed in step (2) as Causes(e, M).

Next, we discuss the underlying ideas behind the certain steps of Algorithm 1 and then provide a proof of correctness for the algorithm. We first compute all traces that enable e by constructing in steps (1.1) and (1.2) the automaton P that accepts exactly all traces in M possibly causing the effect e. The only difference between the automata P and M is their set of final states. The procedure for constructing P first involves constructing a set of automata P_s, for all the states s of the automaton M, such that

s is the initial state in P_s and accepts strings of the language of the hazard e. If the intersection of $L(P_s)$ with $L(e)$ is non-empty, then the corresponding state is considered as a final state in the automaton P (step (1.2)). As a result, the strings in $L(P)$ are exactly those strings bringing M to a state where the hazard is activated. For our railway crossing example in Section 7.3 with the hazard given by the regular expression in (7.2), the automaton P would be the one in Figure 7.2 with states ③ and ④ as the only final states.

In step (1.3) we compute $CausalTraces$ as the subset of strings accepted by P via a simple path starting from the initial state and ending in a final state. These paths correspond to the set of loop-free traces that lead to the hazard e. While this condition does not guarantee minimality (see discussion below) it already reduces the set of possibly causal traces to a finite set. In general, $L(P)$ will be infinite, if it involves a loop in the automaton.

For each of the above finitely many causal traces, in step (2), we compute the set of associated computations. For a given possibly causal trace c, this is done by subtracting all the traces that enable the effect (i.e., $L(P)$) and all the traces that observe the effect (i.e., $L(A^*;e)$) from $L(M)$ and then take the intersection of the resulting language with the language resulted from c decorated with non-occurrence in all possible ways. Note that the intersection computed in step (2) may be empty, meaning that the hazard e is unavoidable when executing the actions of c. For our running example in Section 7.3, the possible causal traces computed by the algorithm are $CaGo$ and $CaGoTa$. Examples of strings in the associated computations are $CaGoTaCcClTcTl$ and $CaGoCcClTaTcTl$. Note that the first string avoids the hazard for both possibly causal traces, while the latter is a string that avoids the hazard for $CaGo$.

Finally, the union of the resulting languages in the step (2) of Algorithm 1 is returned as a first approximation of the set of all decorated causes of M for the hazard e. For this set, the following theorem guarantees that conditions **AAC1** – **AAC2.2** hold. However, condition **AAC3** may fail to hold.

Theorem 5. *The computations in $Causes(e, M)$ returned by Algorithm 1 satisfy conditions* **AAC1** – **AAC2.2** *by construction.*

Proof. The set $Causes(e, M)$ returned by Algorithm 1 is obtained as the union of all $Causes_c(e, M)$ for all $c \in CausalTraces$. Elements in this set are obtained in step (1.4). These strings are computed based on the language that the automaton P (constructed in step (1.2)) recognizes. By construction, $x \in L(P)$ implies there is $y \in L(e)$ such that $xy \in L(M)$. Hence $L(x\,;e) \cap L(M) \neq \emptyset$. Since $CausalTraces \subseteq L(P)$, condition **AAC1** holds.

In order to show that **AAC2.1** holds for some $c \in CausalTraces$, take a string x accepted by M that is not in $L(A^*\,;e)$. Assume that $L(x\,;e) \cap L(M) = \emptyset$. Then $x \notin L(P)$

because otherwise, as we have just seen above, there would exist $y \in L(e)$ such that $xy \in L(M)$. Therefore, $x \in L(M) \setminus (L(A^*;e) \cup L(P))$. Because $CausalTraces \subseteq L(P)$, it follows that $x \neq c$ for any possibly causal trace c. We have now two cases: for every $c \in CausalTraces$ either $c \in sub(x)$ or not. In the latter case **AAC2.1** holds. In the other case $c \in sub(x)$ and thus $x \in L(A^*;a_0;A^*;\ldots;A^*;a_n;A^*)$, from which it follows based on step (2) that $x \in Causes_c(e, M)$, and thus **AAC2.1** holds.

It remains to show that **AAC2.2** . For some possible causal trace $c \in CausalTrace$ let $x \in Cause_c(e, M)$. We must show that $x \notin A^*e$ and that $L(x;e) \cap L(M) = \emptyset$. The first part of the conjunction in **AAC2.2** holds because the construction in step (2) $Cause_c(e, M)$ cannot contain strings from $L(A^*;e)$. Similarly, the second part of the conjunction holds because $L(P)$ is subtracted from $L(M)$ in the same step. □

Condition **AAC3** does not necessarily hold for $Cause_c(e, M)$ used by the Algorithm 1. In fact, for possibly causal traces $x, y \in CausalTraces$, if $x \in sub(y)$ then any sub-string of x is also a sub-string of y. In other words, for $a_0 \cdots a_n = x \neq y = b_0 \cdots b_m$ we have

$$L(A^*;a_0;A^*;\ldots;A^*;a_n;A^*) \subseteq L(A^*;b_0;A^*;\ldots;A^*;b_m;A^*) \tag{7.3}$$

By step (2) of Algorithm 1 we thus have that $Causes_x(e, M) \subseteq Causes_y(e, M)$. Note that it must be the case that $m > n$ for $x \in sub(y)$. We can therefore easily compute the smallest sets of safe computations by removing from the set $CausalTraces$ all strings y that have another possible causal trace $x \in CausalTraces$ of smaller length as sub-word. In our running example, the trace $CaGo$ is a sub-word of the other one $CaGoTa$, and indeed, the computation for $CaGoTa$ is included in the computation for $CaGo$ as well. Hence, only the causal trace $CaGo$ satisfies the minimality condition **AAC3** .

7.6 Experimental evaluation

In this section, we provide an experimental evaluation and assess the applicability of our method. We developed a tool prototype implementing our approach and evaluated the time performance by computing the decorated causes on randomly generated FAs with growing size. The implementation is based on Python and closely follows Algorithm 1. The inputs to our tool are an FA and a regular expression which describes the effect on the given FA. The output of our tool is an automaton that characterizes the set of all decorated causes with respect to the given inputs. In our implementation, we utilized the BRICS automaton library [84] for performing standard automaton operations.

We evaluated our tool in the following experimental setting: we generated random FAs by using the libalf [18] framework. In the process of generating FAs, we fixed the size of the alphabet to 5. We then generated over 1000 FAs with an increasing number

of states and achieved a maximum of 300 states. Figure 7.6 shows an example of an FA with 5 states that was generated randomly by libalf. For each generated FA we also randomly computed an effect for which the decorated causes are determined. We fixed the size of the effect length to 3. All the experiments were conducted on a computer running Ubuntu 20.04.3 with an 8-core 1.8GHz Intel i7-10510U processor and 16 GB RAM.

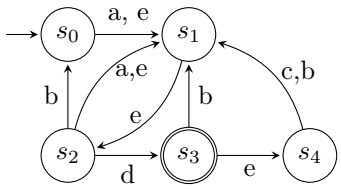


Figure 7.6: Randomly generated FA with 5 states.

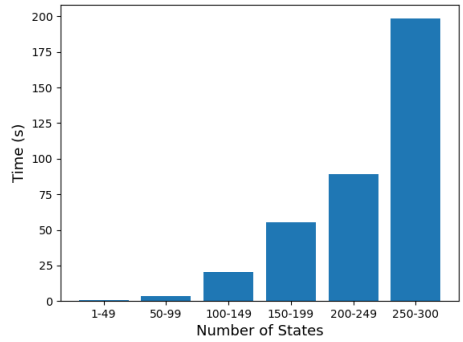


Figure 7.7: Experimental Results

The results of our experiments are displayed in Figure 7.7. We group the randomly generated FAs by their number of states and report the average running times in each group. We only report the times of the experiments in which the decorated causes were not empty. The results indicate that for relatively small FAs with less than 100 states, a result is obtained within 10 seconds. For larger FAs with 250 to 300 states, a result is obtained in 3 minutes on average and within 15 minutes at maximum. We remark that these results are obtained without any attempts to tailor the standard automaton operations to our setting.

	Number of States in the Input FA					
	1-49	**50-99**	**100-149**	**150-199**	**200-249**	**250-300**
# States	71	185	266	422	484	560
# Transitions	236	654	997	1565	1862	2177
# Potential Causes	81	328	10476	21932	44750	73318
# (Minimal) Causes	3	8	10	18	10	22

Table 7.1: Average size of obtained decorated causes.

In Table 7.1 we summarize some information on the automata that recognize the decorated causes returned by the algorithm. Depending on the number of states of the automata given as input, we report the average number of states and transitions of the returned automata, the average number of causes, and the average number of minimal causes obtained. As expected, the size of the automata of the output increased

linearly with that of the input. However, the number of potential causal traces computed increases exponentially. That is not the case for the number of minimal causal traces, as it increases only marginally when the size of the input increases. In fact, in the majority of the cases, the number of minimal causes is less than 5, regardless of the size of the given input automaton.

7.7 Extensions

To illustrate the generality of our causal model we briefly discuss possible extensions to consider the addition of tests and assignments.

Adding tests: KAT The set of regular expressions we considered in (7.1) can be extended with a set B of Boolean tests that we assume generated from a finite set At of atoms, meaning that every $b \in B$ is equivalent modulo the equations of the Boolean algebra to a finite disjunction of atoms in At. This way one can model basic programming constructs, like conditionals, loops, guarded actions, and assertions using tests in B and actions in A.

Kozen [67] showed that the above extensions of regular expressions, called KAT (Kleene algebra with tests) expressions, play the same role with regular sets of guarded strings as ordinary regular languages play for regular expressions. Here a *guarded string* is an ordinary string over the alphabet $A \cup At$, such that the symbols in A alternate with the atoms At. Formally, a guarded language is a subset of $(At \times A)^* \times At$.

A *deterministic KAT automaton* recognizing guarded strings [68] is just a deterministic finite automaton $(S, \Sigma, i, \longrightarrow, F)$ with $\Sigma = At \times A$ and $F \subseteq S \times At$. The only differences are thus the transitions that are now labeled by guarded actions (α, a), and the accepting states, which are now labeled with atoms marking the end of an accepted string. The idea is that an action a is executed only when its guard α (pre-condition) is true, and a string is accepted only in states where the post-condition holds. We say that a guarded string $w \in (At \times A)^* \times At$ is accepted by a KAT automaton M from a state s if either (1) $w = \alpha$ and $(s, \alpha) \in F$, or (2) $w = (\alpha, a)w'$ and there exists $s \xrightarrow{\alpha,a} t$ such that w' is accepted by M from the state t. The *language accepted* by a KAT automaton M is the set $L(M) = \{w \in (At \times A)^* \times At \mid M \text{ accepts } w \text{ from } i\}$.

Our causal model for automata extends naturally to KAT automata by considering hazards e as KAT expressions and causes c as strings in $(At \times A)^*$. Safe computations in M for the hazard e with respect to c are non-empty strings of $L(M)$ satisfying **AAC1** as in Definition 7.1 but with respect to the alphabet $(At \times A)$ instead of A only. Also, the algorithm for computing causes needs no adjustment, but for the way how operations on automata are computed.

Adding assignments: NetKAT NetKAT[4] is a network programming model, which is used for specifying and verifying the packet-processing behavior of software-defined networks. In a nutshell, it is a variation on KAT that considers actions not as abstract elements of an alphabet A but rather as state transformers, like assignments, that are executed when a precondition α is satisfied and modified into a post-condition β.

For a given set of atoms At of a Boolean algebra B, a *deterministic NetKAT automaton* [37] is a deterministic FA $M = (S, \Sigma, i, \longrightarrow, F)$ such that $\sigma = At \times At$ and $F \subseteq S \times (At \times At)$. The transition relation $\longrightarrow$ is thus labeled by pairs of atoms (α, β) and so are the accepting states. The interpretation of these pair of atoms is that they represent pre-conditions and post-conditions of one-step executions.

A string $w \in At \times At \times At^*$ is accepted by M from a state s only when post-conditions match the subsequent pre-condition, meaning that either (1) $w = \alpha\beta$ and $(s, \alpha, \beta) \in F$, or (2) $w = (\alpha\beta)w'$ and there exists $s \xrightarrow{\alpha,\beta} t$ such that $\beta w'$ is accepted by M from the state t. Note that in the last condition is crucial that w' is not the empty string. The *language accepted* by a NetKAT automaton M is the set $L(M) = \{w \in At \times A \times At^* \mid M \text{ accepts } w \text{ from } i\}$.

As for KAT automata, our causal model for automata extends naturally to NetKAT automata too, with hazard represented by NetKAT expressions [37], causes as strings in At^*, and safe computations as strings in $L(M)$ that can be projected into a cause by deleting some atoms and satisfying the rest of the conditions of Definition 7.1.

7.8 Conclusions

In this chapter, we moved the causal model proposed in [25] from labeled transition systems to finite automata to obtain a language-based causal model for safety. The model is in line with the notion of causality described in a logical context in [46] in the sense that a hazard may be observed if and only if it has been caused. Analogously to the alternative worlds of Lewis [75], we also considered decorated causes as alternatives to causes in the sense that they allow executing all actions of a cause interleaved with other actions that guarantee hazard avoidance.

We treated only the case when causes *may* enable a hazard while strings of the decorated causes must avoid it. While it can be interesting to consider a stronger notion of causes as strings c that bring the automaton M to states where the hazard e is inevitable for any of its possible extensions (i.e., by changing **AAC1** to $L(c\,;e) \subseteq L(M)$), such a change would imply that there would be no causes in our railway system example.

We have also presented an algorithm to compute decorated causes, relying only on basic automata-theoretic operations. The algorithms could be improved, using model checking techniques for marking those states in which a hazard is enabled, and search

techniques to find the decorated causes avoiding marked states. Also, it would be interesting to move from automata back to labeled transition systems but remain in a trace setting, with hazards specified as LTL properties.

Finally, we briefly discussed extensions of our work to KAT and NetKAT automata. More work needs to be done here, both to precisely set the definitions and to show the applicability of the method to, for example, find causes of a hazard in a software-defined network.

Chapter 8

Conclusions

With the explosive development of the Internet, traditional network architectures have struggled to meet the increasing demands for network scale, complexity, and dynamism. Software-defined networking (SDN) emerged in response, offering a novel architecture that separates the control plane from the data plane, achieving centralized control and flexibility of the network. SDN introduces a new paradigm for network management, making it more flexible and efficient, and capable of quickly adapting to changing network requirements and application scenarios.

The core advantage of SDN lies in its high programmability, allowing network administrators to directly control network behavior through software interfaces rather than relying on the physical configurations of traditional network devices. This design not only simplifies network configuration and management but also accelerates the deployment of new network services and policies, enabling the network to adapt more swiftly to new business requirements. Moreover, the openness standards and protocols of SDN promote innovation of third-party applications and services, providing possibilities for customized network functions and services.

However, as an innovative technology, SDN also faces several challenges, including network security issues, performance bottlenecks, and compatibility issues with existing network devices and protocols. To overcome these challenges, researchers and engineers are continuously exploring and developing new technologies and methods, such as using formal methods to verify and ensure the correctness of SDN controllers and applications and developing more efficient data plane technologies to enhance network performance.

Classic formal verification methods for Software-Defined Networking (SDN) can be implemented using model checking, a process where the properties of a system are verified against a formal model of the system. The formal model for SDN is indeed complex, as it needs to accurately represent various aspects of the network, including the behavior

of the control plane, the data plane, and the interactions between them. This complexity is further compounded by the dynamic and programmable nature of SDN, which allows network behavior to be modified at runtime through software. Model checking of SDN involves creating a formal model of the network's behavior, including all possible states and transitions that the network can undergo. This model must also incorporate the SDN controller's logic, which dictates how the network's configuration can change in response to different events. The verification process then checks if this model adheres to certain specified properties or invariants, such as connectivity, security policies, or absence of loops, under all possible scenarios.

In this thesis, we apply model checking to check SDN properties via the language Reo and its operational semantics constraint automata. Due to the language provided by Reo accurately describing the coordination and communication protocols in systems, it is more suitable for SDN, which separates the network control plane from the data plane. Also, the Reo is particularly suited to describing concurrent and synchronous processes, it can explicitly represent the synchronization and asynchronous transmission of data flows, as well as dependencies between components. In the following section, we summarize the main research contributions of this thesis and analyze the limitations or challenges that we encountered during the research.

8.1 Main contributions

In Chapter 2, we concretely analyzed the composition structure of SDN and contrasted it with the traditional network architecture by how data is transferred in different layers. By implementing the OpenFlow protocol used in SDN, we described three basic OpenFlow messages that communicated in the control plane and data plane. To formalize this process, we built a Reo model of SDN controllers and switches which answered the first research question:

Research question 1. *Can the coordination language Reo and its semantic model of constraint automata be used as a formal model of SDNs?*

Reo is a powerful tool for designing and reasoning about communication, coordination, and data flow in concurrent systems, which are key aspects of SDNs. As we discussed in Chapter 4, our SDN model is based on Reo circuits which take semantics on a novel definition of constraint automata with memory (defined in Chapter 4). The model is based on the OpenFlow protocol, which is formalized via PktIn, PktOut, and FlowMod messages. A small case study shows the details of the model, highlighting the complicacy of a stateful and concurrent behavior. To be able to use the model for verification purposes we answered the second research question.

Research question 2. *Can we use existing model checkers to verify properties of SDNs modeled by Reo?*

To address this question, we gave a translation of symbolic constraint automata used in our model of SDNs to Promela in Chapter 5. Since Promela can be used for both model checking and simulation by SPIN, our answer to the research question is affirmative, We modeled a simple SDN and verified that a safety property was not satisfied.

While the Reo model of SDNs has typically very few states and many transitions, because of the memory involved, the translation to Promela generates a transition system with a very large number of states, making the verification of a real SDN challenging.

Research question 3. *Can we extend the automata-based model of NetKAT to be stateful and allow concurrency in a way similar to Reo?*

NetKAT was originally designed as a network programming language that uses a formal algebraic approach to describe and reason about network behaviors. It allows executions for packet filtering, modification, and forwarding rules, but it primarily operates under a stateless and sequential paradigm. Without changing the NetKAT automata model, in Chapter 6 we extended NetKAT with concurrent processes that communicate via shared ports. It turns out that the extended model is very similar to our symbolic constraint automata, and therefore can model a large subset of Reo. Our focus was purely semantics, and we left as a future direction an axiomatization for the extended NetKAT.

Research question 4. *Can we use Reo or NetKAT and their associated automata-based models for avoiding hazard events in SDN using causality?*

Another crucial aspect of SDN is causality, in a scenario where packets need to be processed and forwarded through the network, causality ensures that packets are handled correctly, and avoids any hazards that may cause their loss. We gave a partial answer to the above research question in Chapter 7, where we study causality in the context of ordinary automata. We presented an algorithm for detecting hazardous events returning, if possible, an alternative action sequence that will guarantee not to cause the occurrence of the event. Although the type of automata we considered are simpler than those used for modeling SDNs, we consider this result as the first step towards causal reasoning for SDN.

8.2 Future research directions

We conclude by presenting potential avenues for future research that build upon the work we presented in this thesis.

Verification for SDNs models

When answering our second research question we notice a state explosion problem because in Promela, explicit modeling of memory states can rapidly escalate the size of the state space, leading to computational infeasibility for the verification of large-scale SDNs. Transitioning from model checking in Promela to a symbolic model checker could offer a better approach to mitigate the state explosion problem, as it is based on a symbolic representation of data structures and variables, enabling the exploration of large state spaces more efficiently through symbolic manipulation. By abstracting memory operations and representing them symbolically, a symbolic model checker for SDNs could analyze system properties across entire classes of flow tables and messages rather than exhaustively enumerating individual ones. This paradigm shift would allow for a more scalable and tractable analysis of SDN verification in contrast to traditional model checking approaches including ours but also, for instance, ReoLive [29] and mCRL2 [65] as provided by the Reo framework [89].

Optimizing the Promela model

Another strategy aimed at reducing the computational complexity and state space of our Promela model for SDNs could be to simplify the Reo model of SDNs by abstracting away details that are not essential for the verification task at hand. This can involve removing transitions in the Reo model of a switch that we know will not be involved in the verification or simplifying the network structure by using smaller bounds for queues used between a switch and a controller. Additionally, employing partial-order reduction techniques can help reduce the number of explored states by considering only relevant interleavings of concurrent processes. Finally, exploiting symmetry in the model, such as identical processes or state transitions, can further reduce the state space by eliminating redundant exploration.

Formal verification of NetKAT

Our automata model for concurrent NetKAT with ports is a fine-grain description of the dynamics of a system through assertions, in terms of pre- and post-conditions. It would be interesting to see if a combination of symbolic techniques with deductive verification methods could be devised to enable more efficient and scalable verification of network properties.

Causality for SDNs

An obvious direction is to extend the causality framework we presented in Chapter 7 from ordinary automata to symbolic constraint automata and NetKAT automata. While this direction should not present too many technical difficulties, in practice the computational complexity of hazard discovering and avoiding could be intractable. A promising alternative could be combining a formal automata-based model of SDNs with artificial intelligence methods to explore causality within SDN environments. One possible strategy would be to utilize machine learning algorithms, to analyze the causal relationships inferred from (large) subsets of sequences accepted by the automata model to identify causal dependencies between events. Additionally, reinforcement learning algorithms could be utilized to infer causal relationships and optimize execution to avoid hazardous events.

Bibliography

[1] Elvira Albert, Miguel Gomez-Zamalloa, Miguel Isabel, Albert Rubio, Matteo Sammartino, and Alexandra Silva. Actor-based model checking for software-defined networks. *Journal of Logical and Algebraic Methods in Programming*, 118:100617, 2021.

[2] Elvira Albert, Miguel Gómez-Zamalloa, Albert Rubio, Matteo Sammartino, and Alexandra Silva. SDN-actors: Modeling and verification of SDN programs. In *International Symposium on Formal Methods*, pages 550–567. Springer, 2018.

[3] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[4] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. *Acm sigplan notices*, 49(1):113–126, 2014.

[5] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[6] Farhad Arbab, Christian Koehler, Ziyan Maraikar, Young-Joo Moon, and José Proença. Modeling, testing and executing reo connectors with the eclipse coordination tools. *Tool demo session at FACS*, 8, 2008.

[7] Farhad Arbab and JJMM Rutten. A coinductive calculus of component connectors. In *Recent Trends in Algebraic Development Techniques: 16th International Workshop, WADT*, volume 2755, pages 34–55. Springer, 2002.

[8] Youssef Arbach, David S. Karcher, Kirstin Peters, and Uwe Nestmann. Dynamic causality in event structures. *Logical Methods in Computer Science*, 14(1), 2018.

[9] Ebrahim Ardeshir-Larijani, Alireza Farhadi, and Farhad Arbab. Simulation of hybrid reo connectors. In *2020 CSI/CPSSI International Symposium on Real-Time and Embedded Systems and Technologies (RTEST)*, pages 1–10. IEEE, 2020.

[10] Abdelhadi Azzouni, Nguyen Thi Mai Trang, Raouf Boutaba, and Guy Pujolle. Limitations of openflow topology discovery protocol. In *2017 16th annual mediterranean Ad hoc networking workshop (Med-Hoc-Net)*, pages 1–3. IEEE, 2017.

[11] Jos CM Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2-3):131–146, 2005.

[12] Christel Baier, Tobias Blechmann, Joachim Klein, Sascha Klüppelholz, and Wolfgang Leister. Design and verification of systems with exogenous coordination using vereofy. In *Leveraging Applications of Formal Methods, Verification, and Validation: 4th International Symposium on Leveraging Applications, ISoLA 2010, Part II 4*, pages 97–111. Springer, 2010.

[13] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in Reo by constraint automata. *Science of computer programming*, 61(2):75–113, 2006.

[14] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. *SIGPLAN Not.*, 49(6):282–293, 2014.

[15] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on UPPAAL. *Formal methods for the design of real-time systems*, pages 200–236, 2004.

[16] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems*, volume 1066 of *LNCS*, pages 232–243. Springer, 1995.

[17] Kevin Benton, L Jean Camp, and Chris Small. Openflow vulnerability assessment. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 151–152. ACM, 2013.

[18] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R Piegdon. libalf: The automata learning framework. In *International Conference on Computer Aided Verification*, pages 360–364. Springer, 2010.

[19] Marcello Bonsangue, Dave Clarke, and Alexandra Silva. A model of context-dependent component connectors. *Science of Computer Programming*, 77(6):685–706, 2012.

[20] Neil Briscoe. Understanding the osi 7-layer model. *PC Network Advisor*, 120(2):13–15, 2000.

[21] Stephen Brookes. Full abstraction for a shared-variable parallel language. *Information and Computation*, 127(2):145–163, 1996.

[22] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS*, volume 11428 of *LNCS*, pages 21–39. Springer, 2019.

[23] Georgiana Caltais, Sophie Linnea Guetlein, and Stefan Leue. Causality for General LTL-definable Properties. In Bernd Finkbeiner and Samantha Kleinberg, editors, *Proceedings 3rd Workshop on formal reasoning about Causation, Responsibility, and Explanations in Science and Technology*, volume 286 of *EPTCS*, pages 1–15, 2018.

[24] Georgiana Caltais, Hossein Hojjat, Mohammad Reza Mousavi, and Hünkar Can Tunç. DyNetKAT: An Algebra of Dynamic Networks. In *International Conference on Foundations of Software Science and Computation Structures*, pages 184–204. Springer, 2022.

[25] Georgiana Caltais, Mohammad Reza Mousavi, and Hargurbir Singh. Causal reasoning for safety in Hennessy Milner logic. *Fundamenta Informaticae*, 173(2-3):217–251, 2020.

[26] Georgiana Caltais and Can Olmezoglu. Counterfactual causality in networks. *arXiv preprint arXiv:2211.00758*, 2022.

[27] Dexian Chang, Ning Zhu, and Yingjie Yang. Security analysis of sdn access control protocol based on proverif. In *Proceedings of IEEE 3rd International Conference on Civil Aviation Safety and Information Technology (ICCASIT)*, pages 1155–1159. IEEE, 2021.

[28] Guillermina Cledou, José Proença, Bernhard H. C. Sputh, and Eric Verhulst. Hubs for VirtuosoNext: Online verification of real-time coordinators. *Science of Computer Programming*, 203:102566, 2021.

[29] Rúben Cruz and José Proença. Reolive: Analysing connectors in your browser. In *Software Technologies: Applications and Foundations: STAF 2018 Collocated Workshops, Toulouse, France, June 25-29, 2018, Revised Selected Papers*, pages 336–350. Springer, 2018.

[30] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin Vechev. SDNRacer: Concurrency Analysis for Software-Defined Networks. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 402–415. ACM, 2016.

[31] R Enns, M Bjorklund, J Schoenwaelder, and A Bierman. Rfc 6241: Network configuration protocol (netconf), 2011.

[32] Hui Feng. Link to the code and results. https://github.com/githuifeng/Reo2Promela.git.

[33] Hui Feng, Farhad Arbab, and Marcello Bonsangue. A Reo model of Software Defined Networks. In *Formal Methods and Software Engineering: 21st International Conference on Formal Engineering Methods, ICFEM 2019, Shenzhen, China, November 5–9, 2019, Proceedings 21*, pages 69–85. Springer, 2019.

[34] Hui Feng, Marcello Bonsangue, and Benjamin Lion. From symbolic constraint automata to Promela. *Journal of Logical and Algebraic Methods in Programming*, 128:100794, 2022.

[35] Bernd Finkbeiner, Manuel Gieseking, Jesko Hecking-Harbusch, and Ernst-Rüdiger Olderog. Model checking data flows in concurrent network updates (full version). *ArXiv*, abs/1907.11061, 2019.

[36] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A coalgebraic decision procedure for NetKAT. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 343–355. ACM, 2015.

[37] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A coalgebraic decision procedure for NetKAT. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 343–355. ACM, 2015.

[38] Open Networking Foundation. Openflow overview.

[39] Open Networking Foundation. Openflow specification.

[40] Rob Gerth, Doron Peled, Moshe Y Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing and Verification XV: Proceedings of the Fifteenth IFIP WG6. 1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995*, pages 3–18. Springer, 1996.

[41] Fatemeh Ghassemi, Samira Tasharofi, and Marjan Sirjani. Automated mapping of Reo circuits to constraint automata. *Electronic Notes in Theoretical Computer Science*, 159:99–115, 2006.

[42] Paul Göransson and Chuck Black. *Software Defined Networks: A Comprehensive Approach*. Morgan Kaufmann, 2016.

[43] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(3), 2006.

[44] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *Workshop on Software Model Checking (SPIN)*, Lecture Notes in Computer Science 2648, pages 121–135. Springer, 2003.

[45] Evangelos Haleplidis, Jamal Hadi Salim, Joel M Halpern, Susan Hares, Kostas Pentikousis, Kentaro Ogawa, Weiming Wang, Spyros Denazis, and Odysseas Koufopavlou. Network programmability with forces. *IEEE Communications Surveys & Tutorials*, 17(3):1423–1440, 2015.

[46] Joseph Y. Halpern. A Modification of the Halpern-Pearl Definition of Causality. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 3022–3033. AAAI Press, 2015.

[47] Joseph Y. Halpern and Judea Pearl. Causes and explanations: A structural-model approach: Part 1: Causes. In Jack S. Breese and Daphne Koller, editors, *UAI '01: Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence*, pages 194–202. Morgan Kaufmann, 2001.

[48] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[49] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[50] Gerard J Holzmann. An analysis of bitstate hashing. *Formal methods in system design*, 13(3):289–307, 1998.

[51] Fei Hu. *Network Innovation through Openflow and SDN*. Crc Press, 2014.

[52] Fei Hu, Qi Hao, and Ke Bao. A survey on software-defined network and openflow: From concept to implementation. *IEEE Communications Surveys & Tutorials*, 16(4):2181–2206, 2014.

[53] Ali Hussein, Imad H Elhajj, Ali Chehab, and Ayman Kayssi. Sdn verification plane for consistency establishment. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 519–524. IEEE, 2016.

[54] Mohammad Izadi, Marcello Bonsangue, and Dave Clarke. Büchi automata for modeling component connectors. *Software & Systems Modeling*, 10:183–200, 2011.

[55] Mohammad Izadi and Marcello M Bonsangue. Recasting constraint automata into Büchi automata. In *Theoretical Aspects of Computing-ICTAC 2008: 5th International Colloquium, Istanbul, Turkey, September 1-3, 2008. Proceedings 5*, pages 156–170. Springer, 2008.

[56] Daniel Jackson. Alloy: a language and tool for exploring software designs. *Communications of the ACM*, 62(9):66–76, 2019.

[57] Sung-Shik T. Q. Jongmans, Tobias Kappé, and Farhad Arbab. Constraint automata with memory cells and their composition. *Science of Computer Programming*, 146:50–86, 2017.

[58] Sung-Shik TQ Jongmans and Farhad Arbab. Overview of thirty semantic formalisms for reo. *Scientific Annals of Computer Science*, 22(1), 2012.

[59] Sung-Shik TQ Jongmans and Farhad Arbab. Modularizing and specifying protocols among threads. *arXiv preprint arXiv:1302.6333*, 2013.

[60] Miyoung Kang, Eun-Young Kang, Dae-Yon Hwang, Beom-Jin Kim, Ki-Hyuk Nam, Myung-Ki Shin, and Jin-Young Choi. Formal modeling and verification of SDN-openflow. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 481–482. IEEE, 2013.

[61] Young-Mi Kim, Miyoung Kang, and Jin-Young Choi. Formal specification and verification of firewall using TLA+. In *Proceedings of the International Conference on Security and Management (SAM)*, pages 247–251. IEEE, 2017.

[62] Keith Kirkpatrick. Software-defined networking. *Communications of the ACM*, 56(9):16–19, 2013.

[63] Stephen Cole Kleene. Representationof events in nerve nets and finite automata. *CE Shannon and J. McCarthy*, 1951.

[64] Rowan Klöti, Vasileios Kotronis, and Paul Smith. Openflow: A security analysis. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pages 1–6. IEEE, 2013.

[65] Natallia Kokash, Christian Krause, and Erik P de Vink. Data-aware design and verification of service compositions with Reo and mCRL2. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2406–2413. ACM, 2010.

[66] Natallia Kokash, Christian Krause, and Erik P. de Vink. Reo + mCRL2: A framework for model-checking dataflow in service compositions. *Formal Aspects of Computing*, 24(2):187–216, 2012.

[67] Dexter Kozen. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems*, 19(3):427–443, 1997.

[68] Dexter Kozen. Automata on guarded strings and applications. *Matemática Contemporânea*, 24:117–139, 2003.

[69] Dexter Kozen. NetKAT – a formal system for the verification of networks. In Jacques Garrigue, editor, *Programming Languages and Systems*, pages 1–18. Springer, 2014.

[70] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 585–591. Springer, 2011.

[71] Peter Ladkin and Karsten Loer. Analysing Aviation Accidents Using WB-Analysis – an Application of Multimodal Reasoning. In *AAAI Spring Symposium*. AAAI, 1998.

[72] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[73] Petr Lapukhov, Ariff Premji, and Jon Mitchell. Use of bgp for routing in large-scale data centers. Technical report, Internet Engineering Task Force (IETF), 2016.

[74] Florian Leitner-Fischer and Stefan Leue. Causality checking for complex system models. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 248–267. Springer, 2013.

[75] D. Lewis. Causation. *Journal of Philosopy*, 70:556–567, 1973.

[76] D. Lewis. *Counterfactuals*. Blackwell Publishers, 1973.

[77] Benjamin Lion, Samir Chouali, and Farhad Arbab. Compiling protocols to Promela and verifying their LTL properties. In *Proceedings of MODELS 2018 Workshops*, volume 2245 of *CEUR Workshop Proceedings*, pages 31–39. CEUR-WS.org, 2018.

[78] Nancy Lynch. I/o automata: A model for discrete event systems. In *Annual Conference on Information Sciences and Systems*, pages 29–38. Princeton University, Princeton, N.J, 1998.

[79] Mani Prashanth Varma Manthena, Niels LM van Adrichem, Casper van den Broek, and Fernando Kuipers. An sdn-based architecture for network-as-a-service. In *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*, pages 1–5. IEEE, 2015.

[80] John C Martin. *Introduction to Languages and the Theory of Computation*, volume 4. McGraw-Hill NY, 1991.

[81] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Černỳ. Event-driven network programming. *ACM SIGPLAN Notices*, 51(6):369–385, 2016.

[82] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *Computer Communication Review*, 38(2):69–74, 2008.

[83] Dmitrij Melkov and Sarunas Paulikas. Security benefits and drawbacks of software-defined networking. In *2021 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)*, pages 1–4. IEEE, 2021.

[84] Anders Møller. dk.brics.automaton – Finite-State Automata and Regular Expressions for Java, 2021. `http://www.brics.dk/automaton/`.

[85] Seyedeh Elham Mousavi Bafrooi. Specification and implementation of workflow control patterns in reo. Master's thesis, University of Waterloo, 2006.

[86] Nadya El Moussaid, Ahmed Toumanari, and Maryam El Azhari. Security analysis as software-defined security for SDN environment. In *Proceedings of 2017 Fourth International Conference on Software Defined Systems (SDS)*, pages 87–92. IEEE, 2017.

[87] Gilbert N. Nde and Rahamatullah Khondoker. Sdn testing and debugging tools: A survey. In *2016 5th International Conference on Informatics, Electronics and Vision (ICIEV)*, pages 631–635. IEEE, 2016.

[88] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13:85–108, 1981.

[89] José Proença, Dave Clarke, Erik De Vink, and Farhad Arbab. Dreams: A Framework for Distributed Synchronous Coordination. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1510–1515. ACM, 2012.

[90] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. Fattire: Declarative Fault Tolerance for Software-Defined Networks. In *Proceedings of the 2nd ACM SIGCOMM workshop on Hot topics in Software Defined Networking*, pages 109–114. ACM, 2013.

[91] Yakov Rekhter, Tony Li, and Susan Hares. Rfc 4271: A border gateway protocol 4 (bgp-4), 2006.

[92] Manos Renieres and Steven P Reiss. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 30–39. IEEE, 2003.

[93] Justus Rischke and Hani Salah. Software-defined networks. In *Computing in Communication Networks*, pages 107–118. Elsevier, 2020.

[94] Natali Ruchansky and Davide Proserpio. A (not) nice way to verify the openflow switch specification: formal modelling of the openflow switch using alloy. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 527–528. ACM, 2013.

[95] Cole Schlesinger, Michael Greenberg, and David Walker. Concurrent netcore: From policies to pipelines. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 11–24. ACM, 2014.

[96] Hanan Shabana and Mikhail V. Volkov. Optimal synchronization of partial deterministic finite automata. *CoRR*, abs/2002.01045, 2020.

[97] Rob Sherwood, Michael Chan, Adam Covington, Glen Gibb, Mario Flajslik, Nikhil Handigol, Te-Yuan Huang, Peyman Kazemian, Masayoshi Kobayashi, Jad Naous, et al. Carving research slices out of your production networks with openflow. *ACM SIGCOMM Computer Communication Review*, 40(1):129–130, 2010.

[98] Nitin Shukla, Mayank Pandey, and Shashank Srivastava. Formal modeling and verification of software-defined networks: A survey. *Transactions on Emerging Telecommunications Technologies*, 29(5):2082, 2019.

[99] Alexandra Silva. A specification language for reo connectors. In Farhad Arbab and Marjan Sirjani, editors, *Fundamentals of Software Engineering - 4th IPM International Conference, FSEN 2011, Tehran, Iran, April 20-22, 2011, Revised Selected Papers*, volume 7141 of *Lecture Notes in Computer Science*, pages 368–376. Springer, 2011.

[100] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A language for provisioning network resources. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 213–226. ACM, 2014.

[101] Alireza. Souri, Monire Norouzi, Parvaneh Asghari, Amir Masoud Rahmani, and Gazaleh Emadi. A systematic literature review on formal verification of software-defined networks. *Transactions on Emerging Telecommunications Technologies*, 31(2):3788, 2020.

[102] William Stallings. *Handbook of computer-communications standards; Vol. 1: the open systems interconnection (OSI) model and OSI-related standards*. Macmillan Publishing Co., Inc., 1987.

[103] Mevlut Serkan Tok and Mehmet Demirci. Security analysis of SDN controller-based DHCP services and attack mitigation with DHCPguard. *Computers & Security*, 109:102394, 2021.

[104] Moshe Y Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and computation*, 115(1):1–37, 1994.

[105] Jean Philippe Vasseur and Jean Louis Le Roux. Path computation element (pce) communication protocol (pcep). Technical report, Network Working Group, 2009.

[106] Jana Wagemaker, Nate Foster, Tobias Kappé, Dexter Kozen, Jurriaan Rot, and Alexandra Silva. Concurrent NetKAT. In *European Symposium on Programming*, pages 575–602. Springer, Cham, 2022.

[107] Raniyah Wazirali, Rami Ahmad, and Suheib Alhiyari. Sdn-openflow topology discovery: an overview of performance issues. *Applied Sciences*, 11(15):6999, 2021.

[108] Shuangqing Xiang, Marcello Bonsangue, and Huibiao Zhu. Pdnet: A programming language for software-defined networks with vlan. In *International Conference on Formal Engineering Methods*, pages 203–218. Springer, 2019.

[109] Shuangqing Xiang, Huibiao Zhu, Xi Wu, Lili Xiao, Marcello M. Bonsangue, Wanling Xie, and Lei Zhang. Modeling and verifying the topology discovery mechanism

of openflow controllers in software-defined networks using process algebra. *Science of Computer Programming*, 187:102343, 2020.

[110] Shuangqing Xiang, Huibiao Zhu, Lili Xiao, and Wanling Xie. Modeling and Verifying TopoGuard in OpenFlow-Based Software Defined Networks. In *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 84–91. IEEE Computer Society, 2018.

[111] Lily Yang, Todd A. Anderson, Ram Gopal, and Ram Dantu. Forwarding and Control Element Separation (ForCES) Framework. RFC 3746, 2004.

[112] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.

[113] Vladimir A. Zakharov, R. L. Smelyansky, and Eugene V. Chemeritsky. A formal model and verification problems for software defined networks. *Automatic Control and Computer Sciences*, 48(7):398–406, 2014.

[114] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2009.

[115] Yongyue Zhang, Xiangyang Gong, Yannan Hu, Wendong Wang, and Xirong Que. SDNMP: Enabling SDN management using traditional NMS. In *2015 IEEE International Conference on Communication Workshop (ICCW)*, pages 357–362. IEEE, 2015.

Samenvatting

SDN (Software-Defined Networking) vertegenwoordigt een revolutionaire benadering van netwerkarchitectuur die het dynamische en flexibele beheer van netwerkbronnen mogelijk maakt via softwaregebaseerde besturing. Dit wordt bereikt door het besturingslag los te koppelen van het gegevenslag en softwareplatforms te benutten voor programmeerbare besturing. Het besturingslag is de controller, die het hele netwerk beheert, bestuurt en beleid formuleert. Het verzamelt netwerkstatusinformatie, ontwikkelt doorstuurbeleid en instrueert de onderliggende netwerkapparaten via southbound-interfaces (bijv. OpenFlow) over hoe datapakketten moeten worden doorgestuurd. Het gegevenslag bestaat uit verschillende netwerkapparaten (bijv. switches) die verantwoordelijk zijn voor het doorsturen van datapakketten op basis van instructies van het besturingsvlak. Dit proefschrift introduceert het idee van SDN en het OpenFlow-protocol in Hoofdstuk 2 en presenteert vervolgens formele modellen in de volgende hoofdstukken.

Hoofdstuk 3 duikt in de Reo-coördinatietaal en de toepassing ervan bij het modelleren van SDN's. Reo wordt gepresenteerd als een krachtig hulpmiddel voor het specificeren en orkestreren van het gedrag van reactieve en gedistribueerde systemen door het gebruik van connectoren, wat grafiekgebaseerde representaties zijn van gegevensstromen en synchronisatiebeperkingen. Verder word ook de samenstelling van constraint automaten, die de modulaire constructie van complexe systemen mogelijk maken door eenvoudigere componenten te combineren besproken. Verschillende basis-Reo-connectoren worden geïntroduceerd en hun bijbehorende beperkingsautomaten worden beschreven, waarbij wordt gedemonstreerd hoe deze modellen kunnen worden gebruikt om verschillende netwerkscenario's weer te geven en te analyseren. Het hoofdstuk eindigt met een illustratie van hoe Reo en constraint automaten een rigoureus en flexibel raamwerk bieden voor het modelleren van het ingewikkelde gedrag van SDN's, waarmee de basis wordt gelegd voor de formele verificatie en analyse van deze systemen in volgende hoofdstukken.

Het onderzoek in Hoofdstuk 4 presenteert een nieuw Reo-gebaseerd model dat het gedrag van SDN-switches en SDN-controllers nauwkeurig weergeeft. Dit model abstraheert ook het OpenFlow-communicatieprotocol, en biedt een formeel raamwerk voor de

implementatie en analyse van SDN's.

Hoofdstuk 5 gebruikt Promela om het constraint automaten model van SDN te implementeren en het formele model te verifiëren met de SPIN model checker. Om een rigoureuze verificatie van SDN-eigenschappen mogelijk te maken, worden de Reo-modellen vertaald naar Promela, de invoertaal voor de SPIN model checker. Deze vertaling vergemakkelijkt de formele verificatie van essentiële SDN-kenmerken, waaronder bereikbaarheid, consistentie en de juistheid van de netwerk gedrag.

In Hoofdstuk 6 wordt de modellering taal NetKAT uitgebreid. NetKAT is en formele taal voor het specificeren van netwerk gedrag, die hier is uitgebreid om concurrent gedrag te ondersteunen door de introductie van poorten. Deze uitbreiding verbetert de mogelijkheid van NetKAT om stateful en concurrent systemen te modelleren, waardoor het aansluit bij het op Reo gebaseerde formalisme en de toepasbaarheid ervan wordt uitgebreid naar complexere SDN-scenario's.

Om een pioniersrol te vervullen bij de integratie van causaliteitsredeneringen in SDN-modellen, is de aanpak in Hoofdstuk 7 cruciaal voor het diagnosticeren van netwerkanomalieën en het voorkomen van gevaarlijke gebeurtenissen. Hiermee wordt de basis gelegd voor geavanceerdere causaliteitsanalyses in toekomstig werk.

Summary

SDN (Software-Defined Networking) represents a revolutionary approach to network architecture that enables the dynamic and flexible management of network resources through software-based control. It achieves this by decoupling the control plane from the data plane and leveraging software platforms for programmable control. The control plane lies controller, which manages, controls, and formulates policies for the entire network. It collects network state information, develops forwarding policies, and instructs the underlying network devices through southbound interfaces (e.g., OpenFlow) on how to forward data packets. The data plane consists of various network devices (e.g., switches) that are responsible for forwarding data packets based on instructions from the control plane. This thesis introduces the idea of SDN and OpenFlow protocol in Chapter 2, then presents the formal expressions in the following chapters.

Chapter 3 delves into the Reo coordination language and its application in modeling SDNs. Reo is presented as a powerful tool for specifying and orchestrating the behavior of reactive and distributed systems through the use of connectors, which are graph-based representations of data flows and synchronization constraints. Furthermore, it also discusses the composition of constraint automata, which enables the modular construction of complex systems by combining simpler components. Several basic Reo connectors are introduced, and their corresponding constraint automata are described, demonstrating how these models can be used to represent and analyze various networking scenarios. The chapter concludes by illustrating how Reo and constraint automata provide a rigorous and flexible framework for modeling the intricate behaviors of SDNs, laying the groundwork for the formal verification and analysis of these systems in subsequent chapters.

The research in Chapter 4 presents a novel Reo-based model that accurately represents the behavior of SDN switches and controllers. This model also abstracts the OpenFlow communication protocol, providing a formal framework for the implementation and analysis of SDNs.

Chapter 5 uses Promela to implement our constraint automata model of SDN, and verify the formal model by SPIN model checker. To enable rigorous verification of SDN

properties, the Reo models are translated into Promela which is the input language for the SPIN model checker. This translation facilitates the formal verification of essential SDN characteristics, including reachability, consistency, and the correctness of network policies.

Moreover, the exploration in Chapter 6 extends NetKAT, a formal language for specifying network policies, to support concurrency through the introduction of ports. This extension enhances NetKAT's capability to model stateful and concurrent systems, aligning it with the Reo-based formalism and broadening its applicability to more complex SDN scenarios.

To pioneer the integration of causality reasoning into SDN models, the approach in Chapter 7 is crucial for diagnosing network anomalies and preventing hazardous events, laying the groundwork for more sophisticated causality analyses in future work.

Acknowledgements

As I look back on the journey that led to the completion of this doctoral dissertation, I am filled with a sense of accomplishment and pride. But I also recognize that this is just the beginning of a lifelong pursuit of knowledge and understanding. As I embark on the next chapter of my academic career, I carry with me the memories of those who have inspired and supported me. To each one of you, I offer my heartfelt gratitude for your role in shaping who I am today, and for the countless ways in which you have enriched my life.

Foremost, I am deeply grateful to Prof. Marcello Bonsangue for his constant support, invaluable guidance, and patience throughout my research journey. From the very beginning of this project, he has been a constant source of motivation, encouragement, and wisdom. His unwavering belief in my abilities and his constant encouragement helped me to complete this thesis. His insights, feedback, and suggestions have enriched my understanding of the subject matter and helped me refine my research questions and methodology.

I would also like to express my special thanks to Prof. Farhad Arbab for his support and assistance at the beginning of my research. Thanks for explaining me about the core idea of Reo and coordination, which have made me carry out my research work better. I am specially grateful to Dr. Benjamin Lion: I have had so much fun at the CWI working together with you on the coding part and during the many coffee breaks.

I am deeply indebted to Dr. Georgiana Caltais who has helped me during my research studies and gave valuable opinions and guidance on our research work. Thanks for inviting me to visit Konstanz University, where I met Hunkhar and Dang and had a pleasant time working together. I would like to express my gratitude to Prof. Meng Sun of Peking University and Prof. Wanlin Gao from China Agricultural University: thanks for inviting me to present my work and for all those big lunches and dinners.

I would like to express my special thanks to my best friends, Shanshan, Xiaomin, and Xiaofang, who have been my friends since childhood. No matter which country or city I am in, they always encourage and support me remotely. Thank you to my best friend Dr. Li Zhang for providing me with insightful advice all along, it's really fortunate to

have met you. Also, I would like to thank every friend I have made in Leiden for their love and support. Dr. F. Ye provided constant encouragement and support during the challenging times, Dr. J. Bian shared food and all her opinions about work, Xinyuan offered a listening ear and comforting words whenever I needed it, Niklas and Guerino provided practical assistance against the special weather of the Netherlands that allowed me to focus on my work. And thanks to W. Chu, Z. Yang, J. Shi, L. Yang with whom I shared the same office: I really enjoyed the time chatting with you. Thanks to Yuxuan and Shuaiqun for all the unwavering support and caring when I have needs. Thanks to Dr. C. Wang's company during the COVID-19 epidemic, which made that time very interesting. Many thanks to Dr. X. Chen, Dr. W. Chen, Weikang, Feibo, Qianru, et al.

Finally, I would like to express my heartfelt thanks to my family, who have been my constant source of strength and inspiration. To my parents, your unconditional love, sacrifices, and unwavering support have been the foundation upon which I have built my academic pursuits. Your belief in me and your encouragement during the toughest moments have been invaluable. I apologize for the times when my studies may have taken precedence over spending quality time with you, but please know that your teaching and guidance are deeply etched in my heart and will guide me always. I would like to thank my grandfathers, Prof. Feng and M. Zhang, I really appreciate your unconditional support and understanding, and I hope you will have no disease in the other world.

Curriculum Vitae

Hui Feng was born in Handan, China, on March 26, 1993. In 2011, she started her Bsc. studies in Network Engineering and received her Bsc. degree in 2015. After that, she started her Msc. studies in Computer Security and Resilience at the China Agriculture University in Beijing, China, where she obtained her Msc. degree in 2017.

In September 2017, she started her PhD research supported by the China Scholarship Council (CSC No. 201706350092). She worked at the Leiden Institute of Advanced Computer Science (LIACS), Leiden University, the Netherlands, under the supervision of Prof.dr. M.M. Bonsangue and the now emeritus Prof.dr. F. Arbab. Hui Feng's research interests include formal methods, automata theory, and software-defined networks. She has published papers in international journals and conferences, including the Journal of Logical and Algebraic Methods in Programming and the International Conference on Formal Engineering Methods. During her PhD studies, she took courses in communication in science, time management, and scientific conduct, among others.

Publication List

- Hui Feng, Farhad Arbab and Marcello Bonsangue. (2019). *A Reo Model of Software Defined Networks.* In: Ait-Ameur, Y., Qin, S. (eds) *Formal Methods and Software Engineering (ICFEM 2019).* Lecture Notes in Computer Science, vol 11852. Springer. https://doi.org/10.1007/978-3-030-32409-4_5

- Hui Feng and Marcello Bonsangue. (2024). *Concurrent NetKAT with ports.* In: Juw Won Park and Adam Przybyłek and Hossain Shahriar. (eds) *SAC'24: Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing. Association for Computing Machinery.* https://doi.org/10.1145/3605098.3636048

- Marcello Bonsangue, Georgiana Caltais, Hui Feng, and Hünkar Can Tunç (2022). *A Language-Based Causal Model for Safety.* In: Yamine Aït-Ameur, Florin Crăciun. (eds) *Theoretical Aspects of Software Engineering (TASE 2022).* Lecture Notes in Computer Science, vol 13299. Springer. https://doi.org/10.1007/978-3-031-10363-6_20

- Hui Feng, Marcello Bonsangue and Benjamin Lion. (2022). *From symbolic constraint automata to Promela. Journal of Logical and Algebraic Methods in Programming*, volume 128, 100794. https://doi:10.1016/j.jlamp.2022.100794

Propositions

pertaining to the thesis

Formal models of Software-Defined Networks

by Hui Feng

1. Incorporating formal models and languages into software-defined network development enhances overall system reliability by enabling thorough verification of network behavior and policies. [Chapter 1 & 2]

2. The coordination language Reo can be used to model software-defined networks, allowing for compositional and formal specification of network components and their interactions. [Chapter 3 & 4]

3. By translating Reo models into Promela code, developers can utilize the SPIN model checker to perform exhaustive verification of software-defined networks for correctness and reliability. [Chapter 5]

4. Constraint automata can be used to provide extended formal semantics for concurrent NetKAT programs, enabling modeling and analysis of SDN policies. [Chapter 6]

5. OpenFlow is a protocol for controlling network devices, but it is not synonymous with software-defined networking. The latter is a broader paradigm that encompasses various architectures and technologies beyond just protocol-level control.

6. Testing software-defined networks remains difficult due to their dynamic nature and the complex interactions between the control and data planes, which can obscure subtle errors.

7. One of the primary challenges in the formal verification of software-defined networks is ensuring scalability as network size and complexity increase, requiring advanced algorithms and heuristics to manage the computational load.

8. Temporal logic and other formal logic systems can be effectively utilized to specify and verify dynamic properties of software-defined networks, such as flow consistency and packet forwarding correctness.

9. The PhD process, with its intense focus on critical thinking and deep analysis, remains the gold standard for intellectual mastery and is increasingly critical in today's fast-paced, skills-based economy.

10. Engaging with new information has become a practical necessity, demanding constant adaptation to stay relevant. The true challenge lies in knowing what to prioritize and when to disengage.

Hui Feng
Leiden
December 3^{rd}, 2024