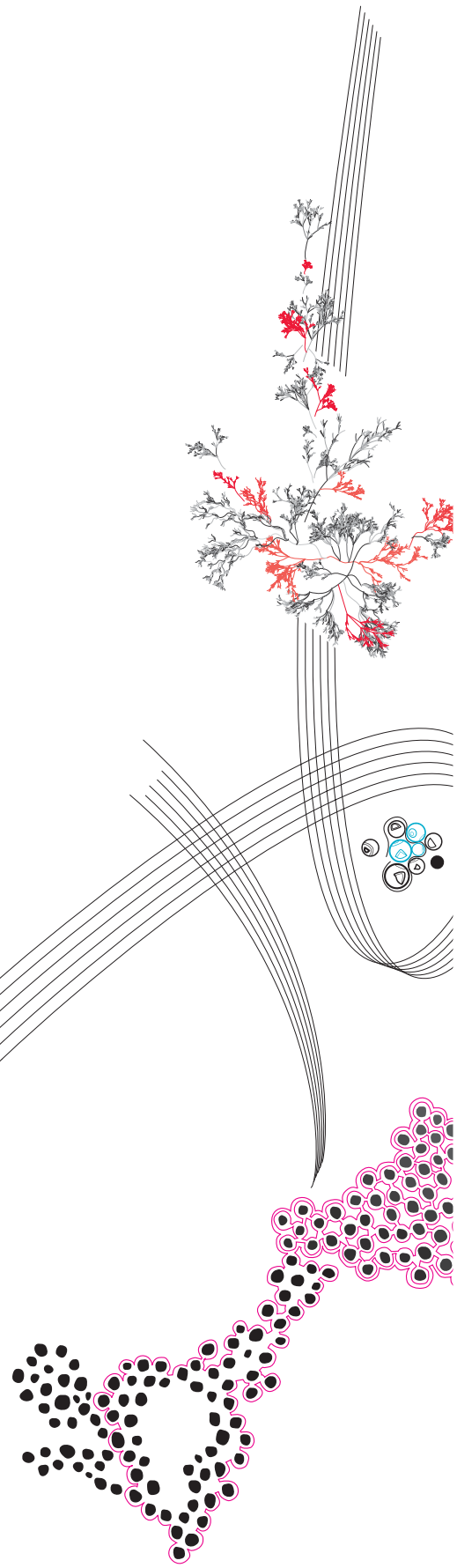


MSc Computer Science
Final Project

Static Analysis of Race Hazards in Software-Defined Networks

Andrei-Ion Covaci

Supervisors
dr. Georgiana Caltais
dr.ir. Alex Chiumento
dr.ir. Vadim Zaytsev

August, 2025

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

Software-Defined Networking (SDN) introduces significant advancements in the computer networks domain but also presents new challenges, including complex concurrency issues. One critical concern in SDNs is the occurrence of race hazards across the communication between network components. These races can adversely affect network performance and stability, making it essential to detect and understand their root causes. Although various SDN verification tools are available, few effectively address race hazards detection across both the control and data planes.

Tracer is one such tool designed to statically detect race hazards in SDNs by leveraging the symbolic semantics of DyNetKAT, a domain-specific language with a robust equational theory. However, Tracer's initial implementation lacks performance benchmarks and thorough validation, limiting its practical utility. This study continues the work laid out for Tracer, making several key contributions to the detection of race hazards in Software-Defined Networks (SDNs). We formalize three distinct types of harmful SDN race hazards and optimize the DyNetKAT symbolic semantics to enable their detection. To model network behavior more effectively, we apply the "One Big Switch" abstraction to the data plane. Additionally, we enhance the communication modeling between the control and data planes in DyNetKAT through a mechanism inspired from the OpenFlow protocol. We also present one of the first performance benchmark for DyNetKAT using a dataset of real world network topologies. Finally, we develop RaceLoom, a new implementation of Tracer capable of detecting harmful SDN race hazards, and benchmark its performance.

Keywords: Formal Verification, Race Condition, Software-Defined Networking, Static Analysis, Symbolic Execution, Kleene Algebra

Chapter 1

Introduction

Software-Defined Networking (SDN) is an emerging paradigm of traditional networking in which the *control plane*, responsible for making decisions about packet processing and traffic routing, is decoupled from the *data plane*, where actual packet forwarding happens. Figure 1.1 depicts this decoupling in SDNs compared to traditional networks. Using a separate control plane enables centralized network management, tackling previous consistency issues of proprietary software and protocols that the networking equipment came with. Network programmability is another advantage introduced through this decoupling that simplifies policy enforcement, network reconfigurability, and evolution by allowing software applications to run on top of the control plane.

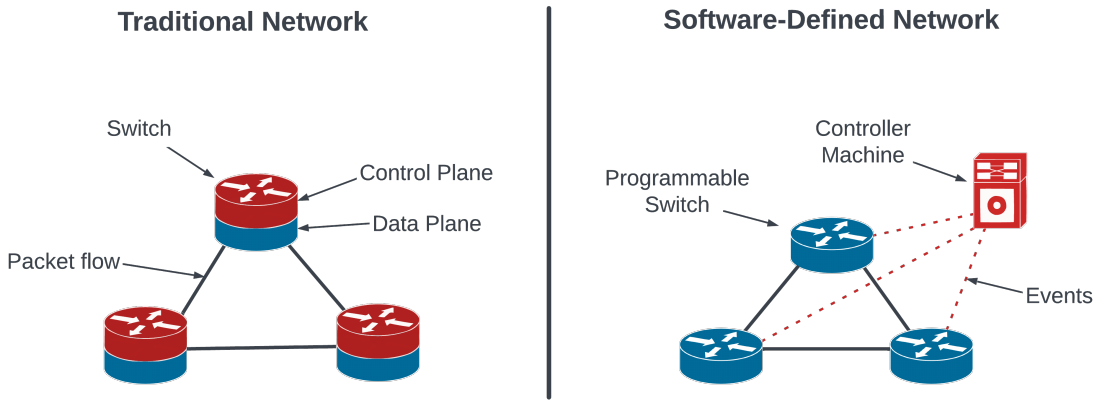


FIGURE 1.1: Traditional network versus software-defined network [13]

The architecture of an SDN is divided into three layers: the management plane, the control plane, and the data plane [28] [30]. In general, they are responsible for: defining the network functionality, e.g. load balancing, enforcing this functionality throughout the network, and performing the packet processing according to the enforced rules, respectively. The communication between these layers is done through a set of Application Programming Interfaces (APIs), called the Northbound Interface between the application and the control plane, and the Southbound Interface between the control and the data plane.

To preserve flexibility and interoperability, communication protocols in SDNs are event-based, e.g., OpenFlow [34], one of the most popular communication protocols for such networks. Apart from network events, i.e. the messages passed between the devices in a

network, other event sources have also been identified. For example, at the controller level, system calls from the operating system or functional calls from the application running on the device also influence the behavior of the controller [6]. This aspect allows SDNs to be viewed as large and complex distributed systems, inheriting their associated problems [28]. However, despite their similarities, SDNs also come with their specific subset and distribution of bugs, of which a good majority are triggered by external calls and network events, such as misconfigurations of the data plane [6]. One of the main causes of these bugs is the concurrent behavior of network devices, which is also a challenge usually encountered in distributed systems [50]. While the internal workings of the devices and the protocols used in SDNs try to prevent harmful concurrency bugs, e.g. by making the controller run as a state machine [6], or design ways for developers to prevent such bugs, e.g. barrier messages in OpenFlow [34], they still appear and have harmful consequences in practice. Some examples include: forwarding loops [17], traffic processing using outdated policies [54], miss-configuration of switches' flow tables [51], or decreased performance of the network due to unnecessary communication on the southbound interface [49].

Many verification approaches and tools have been developed to ensure the reliability of software-defined networks. A 2017 study identified 18 verification tools for SDN [30]. These tools focus on detecting various problems of SDNs, including host reachability, network loops, black holes, but also harmful behaviors caused by concurrency or packet interleaving, and the main methods employed are based on model checking, theorem proving, or runtime verification and debugging. One of the challenges faced by the said tools is to capture the desired behavior of the network to be analyzed. To do this, most of the tools described in [30] use some sort of dynamic analysis, online or offline, where the actual network and/or its input based on real packet traces is examined. An exception to this rule are tools based on model checking, such as [10], [1], or [51], where specific abstractions of the SDN controller software or the communication protocols used are derived to enable the analysis of the desired properties.

To aid in verification and abstract some of the technical details associated with networks, domain-specific languages such as NetKAT [2] have been developed. NetKAT [2] is a framework based on Klenee Algebra with Tests (KAT) that provides unified syntax and semantics to specify, program and reason about network data planes. An important advantage of NetKAT is its solid mathematical foundation, which provides consistent reasoning principles and also a sound and complete equational theory. Using this framework, one can verify multiple network properties, such as packet reachability or traffic isolation, in traditional or software-defined networks. This allowed several NetKAT-based SDN tools to emerge, such as McNetKAT [46], a scalable tool for probabilistic network programs that can reason about program equivalence or networking properties, or KATch [36], a symbolic verifier of network-wide specifications. The main limitation of NetKAT and the specified tools is that they only focus on the behavior of the data plane, without incorporating the dynamic re-configurations of the forwarding devices performed by the control plane. To address this, a new extension of NetKAT was developed named DyNetKAT [9], which introduces the necessary syntax and semantics to capture the behavior of both the data and the control plane, while extending the sound and complete equational theory of NetKAT over these new semantics. In this way, DyNetKAT maintains the mathematical foundation laid by NetKAT, which enables the formal verification of complete SDNs.

Being a new specification language with a strong mathematical base, DyNetKAT opens the possibility to formally analyze various properties of software-defined networks, but not many tools exploit the advantages of this language. In [9], a framework is also introduced

to specify the safety properties of DyNetKAT networks, allowing the reasoning to happen purely in equation form. Then, this is adjusted to allow reachability properties to be checked, and a prototype tool is implemented to verify this type of properties, which can quickly verify large networks with more than 1000 switches.

Tracer [54] is another DyNetKAT-based verification tool that checks the presence of harmful race hazards in SDNs by using Vector Clocks [29] and the symbolic semantics of DyNetKAT [8]. In this context, a harmful SDN race hazard refers to the usage of outdated policies or a misconfiguration of network switches caused by communication delays occurring between the control and data plane. It is worth remarking that Tracer is a *static analysis tool* based on symbolic execution that uses these techniques in a way that is not very common among SDN verification tools. For example, only 4 out of the 18 tools presented in [30] use symbolic execution or static analysis methods other than model checking, and not all consider the dynamic reconfiguration of switches carried out by the control plane. The proof of concept implemented for Tracer shows the potential of the symbolic execution approach, but there is a lack of experiments and performance benchmarks to assess the efficacy of the method used. Furthermore, an analysis of the race conditions reported by the tool is also required as not all concurrent behaviors are necessarily harmful [17] [40]. This also requires an understanding of how harmful race hazards occur and impact a Software-Defined Network.

Although the source code of the prototype developed for Tracer [54] is publicly available, the implementation is not robust enough to be properly evaluated. Among the issues, it was remarked that not all NetKAT operators are supported by the prototype, e.g. the Klenee star from NetKAT [2], and, even on small examples and very limited depth of search, the tool cannot always finish the analysis of the SDN. Hence, a new and more robust implementation is required to evaluate the performance of the proposed approach based on vector clocks. Moreover, a proper evaluation strategy is needed, which we develop using the Topology Zoo [25], a dataset of more than 250 real world network topologies of different sizes and shapes.

Thus, this research aims to continue the work carried out for Tracer [54] on race hazard detection in Software-Defined Networks using DyNetKAT. For this purpose, we focus on the following research questions:

RQ1 How can symbolic execution in DyNetKAT be used to identify harmful race hazards between the control and data plane of Software-Defined Networks?

RQ2: How can a realistic benchmark of DyNetKAT networks be achieved for performance evaluation?

RQ3: What is the impact of a new Tracer implementation on the performance of the tool?

a) What is the most promising suite of frameworks for an efficient implementation of Tracer?

b) How does the performance of the reimplemented prototype evaluate on the RQ2 benchmark?

RQ4: How can race hazard detection in DyNetKAT be improved?

a) What are the bottlenecks of the current approach for race hazard detection in DyNetKAT?

b) How can other methods or techniques be utilized in conjunction with DyNetKAT to address these bottlenecks?

In response to the aforementioned research questions, this study advances the work presented in [54] through the following contributions:

1. Formalization of three types of *harmful* SDN race hazards (related to **RQ1**)
2. Optimized DyNetKAT symbolic semantics for SDN race hazard detection (related to **RQ1** and **RQ4**)
3. DyNetKAT "*Big Switch*" abstraction of the SDN data plane (related to **RQ4**)
4. Improved DyNetKAT modeling of the SDN control and data plane using a communication mechanism inspired from the OpenFlow protocol [34]
5. First real-world inspired performance benchmark for DyNetKAT based on the Topology Zoo [25] (related to **RQ2**)
6. RaceLoom: a new implementation of Tracer [54] for detecting SDN race hazards and providing associated trace explanations that lead to a harmful state (related to **RQ3**)
7. Performance evaluation of RaceLoom (related to **RQ3**)

The rest of the paper is organized as follows.

In chapter 2, we provide background on software-defined networking, concurrency, and domain-specific languages for SDNs, focusing in particular on DyNetKAT. We also review related work on race hazard detection in SDNs and introduce Tracer [54], the tool that forms the basis for our improvements.

Chapter 3 presents our enhanced symbolic framework based on DyNetKAT for detecting race hazards in SDNs, extending the work from [8].

In chapter 4, we describe the implementation of RaceLoom, our improved version of Tracer, which builds on the symbolic framework introduced in the previous chapter.

Chapter 5 evaluates RaceLoom's ability to detect harmful SDN race hazards across three example networks and benchmarks the performance of the tool.

Finally, in chapter 6, we summarize our findings, discuss limitations, and outline directions for future work.

Chapter 2

Background and Related Work

This chapter provides essential background for our study, reviews related work on race hazard detection in SDNs, and introduces the symbolic analysis approach used by Tracer [54].

Section 2.1 introduces the fundamentals of software-defined networking, followed by a discussion of concurrency concepts in section 2.2. Section 2.3 presents domain-specific languages for modeling SDNs, with a focus on NetKAT [2] and DyNetKAT [9]. Finally, section 2.4 explores existing approaches for detecting race hazards in SDNs and ends with a detailed overview of Tracer's symbolic execution approach.

2.1 Software-Defined Networks

In traditional computer networks, each network device was designed to have the control plane, responsible for enforcing the behavior of the network, on top of the forwarding plane, responsible for processing packets [47]. Although this embedding of the two planes was a key factor in improving resilience, it also led to significantly more complex and static networks. In turn, this quickly caused the networks to be less modifiable, adaptable, controllable, and have fewer monitoring capabilities. As a result, it became increasingly challenging to implement various in-path functionalities, such as firewalls or security monitors, which had to be done through an enormous number of middle boxes and specialized components. Apart from the large number of such components, many of them used proprietary software and protocols, hindering the introduction of new network abstractions and increasing the risk of inconsistencies between devices produced by different entities. [28]

To counteract many of these shortcomings, a decoupling between the control and data plane was suggested, under a new paradigm called Software-Defined Networking. In a software-defined network (SDN), the architecture can be visualized as three different layers: the application plane, the control plane, and the data plane [28] [30]. The application plane encompasses the various software applications that implement the desired network functionality, e.g. load balancing. This plane communicates to the control plane what the behavior of the network should be through a set of APIs, typically referred to as the Northbound Interface. Then, to achieve the desired network behavior, the controllers take care of dynamically reconfiguring the devices on the data plane, such as switches, by installing the corresponding flow rules and policies on every device. This is done through another set of APIs called the Southbound Interface. Finally, each device in the data plane

stores one or more flow tables with rules that tell the device what to do with the incoming traffic. Figure 2.1 depicts the basic architecture of an SDN.

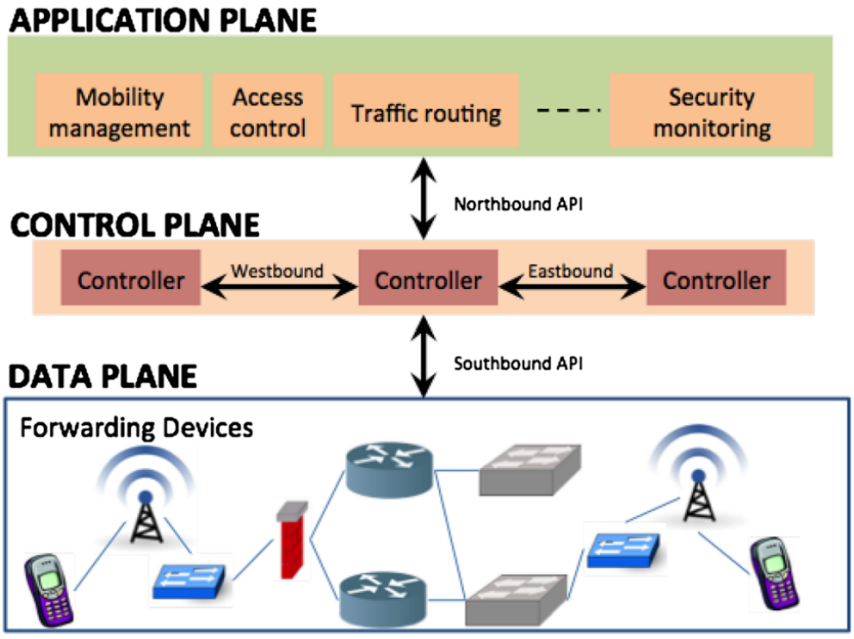


FIGURE 2.1: Basic SDN architecture [30]

By organizing a computer network in this way, it becomes significantly easier to modify and dynamically adapt the network, as the network control is now centralized and various software applications can be run on top of the SDN controllers. Furthermore, the architecture of SDNs come with increased flexibility, allowing new network abstractions to be introduced. Finally, the forwarding decisions of the data plane devices are based on flow rules rather than destination rules, which provides a unified behavior of these devices, such as firewalls, load balancers, etc. [28]

Although this paradigm brings many benefits over traditional networks, it also comes with its own issues and challenges. Only in the control plane, Bhardwaj et al. [6] identified 500 critical bugs in three of the most popular SDN controllers. Concurrency, incorrect controller logic, memory issues, and environment interactions, such as sending and processing network events, are some of the root causes of these bugs. Moreover, SDNs are inherently event-driven, communication protocols being message-based, but also interaction of network devices with the local environment, where events from the operating system or from third-party libraries have to be processed [6]. Due to this aspect, SDNs are typically associated with distributed systems [50], inheriting their challenges.

2.2 Concurrency and Race Conditions

In computer science, a number of actions are said to be concurrent when they are partially ordered, i.e. not all pairs of actions have a strict order in which they are performed, allowing for different sequences of these actions to be observed in different executions of the same system [29]. This behavior is encountered in multithreaded software or computer systems where multiple threads or parts of the system are running in parallel. This typically means an increase in performance, as multiple CPU cores can be leveraged and more actions can be performed in the same amount of time. However, this interleaving of actions makes the

behavior of the system unpredictable, causing possible undesired or harmful effects known as *race conditions*.

A subset of these harmful effects is represented by *data races*, which are usually defined as two or more unordered actions racing to access the same memory location, and at least one of them is a write [18]. It should be remarked that not all data races are race conditions, or vice versa [41]. However, for simplicity, this paper will refer to any interleaving of unordered actions that access the same part of the system, of which at least one attempts to modify the said part, as a race hazard.

Data races manifest differently depending on the concurrency model used. In multithreaded software programs, they act as concurrent accesses to the same location of the underlying memory model of the program, where one access is for writing to memory and the others either read from or also write to memory. A basic example here would consist of two parallel threads that read and assign, respectively, an unprotected shared variable [41]. In event-based systems, data races also manifest in a similar way, the difference being that they are primarily caused by unordered interleaving of events, which are triggered nondeterministically. For example, website events, triggered by user interactions or other external inputs and actions, may lead to unordered reads and writes of / to unprotected shared variables, as the order of these events cannot be guaranteed a priori [40].

In distributed systems [50], where the underlying structure is comprised of multiple computer nodes that are physically spread apart and the communication is message-based, data races manifest as misconfigurations or undesired state changes of the system components. In this context, data races are mainly caused by unordered sending and receiving of messages, and local computations at a given node in the system, triggering violations of the expected event orderings or the atomic processing of event sequences [38]. This may happen in isolation, at one component in the system, or across multiple components, which could potentially affect the global computation performed by the system.

As mentioned above, software-defined networks closely resemble distributed systems, so race hazards, such as data races, also manifest in a similar fashion. Sun et al. [49] identified three types of data race in one of the most popular SDN communication protocols, OpenFlow [34]. They can occur locally, at the controller, where multiple threads race with each other to send messages to the same switch, or between multiple devices, where network delays lead to data races between a controller and a switch, or between two switches. For example, a controller-switch data race occurs at a switch when the processing of a packet races with a flow table installation sent by the controller. Vinarskii et al. [51] further adapt these data race types for extended finite input / output automata models inspired by popular SDN specifications used in practice. They define two types of data race: *input / output races*, which occur when a model has to nondeterministically accept an input or produce an output, such as a switch installing a new policy and processing a packet concurrently, and *intra-channel races* where the order of messages sent across a channel is changed due to network delays, such as a controller installing and immediately deleting the same policy on a switch. In [17], El-Hassany et al. derive a commutativity model from the OpenFlow switch specification that describes when two events or actions are allowed to commute, i.e., regardless of the order in which they are processed or performed, the final result is still the same. Then, they introduce *commutativity races*, which are defined as a sequence of actions that are not ordered according to the derived commutativity model.

Regardless of their type, not all race hazards mentioned are necessarily harmful, e.g. any two racing events that obey the commutativity rules of the model in [17] can be considered a

harmless race hazard in that context. However, when harmful race hazards occur, they may have serious consequences on the hosts in the network, on the traffic that flows through the network, or on the SDN itself. For example, an SDN race hazard could lead to a forwarding loop, causing traffic delays and drops in user connections due to incorrect packet forwarding to the right server replicas [17]. Or in [53], Xu et al. present a novel attack that exploits harmful data races caused by network events sent to the SDN controller to manipulate its internal state, allowing the attacker to steal privacy information or trigger Denial of Service attacks [44] on the controller.

2.3 Domain-Specific Languages for SDNs

Many Domain Specific Languages (DSLs) exist for modeling SDNs, their main advantage being a high abstraction level that strips away various SDN details, such as the communication protocol [7], and moves the focus of programmers on network modeling. Some popular examples include Frenetic [20], a high-level programming language for distributed switches, Pyretic [42], a Python programming platform based on many of the concepts introduced by Frenetic, NetCore [37], a declarative language for specifying packet forwarding policies, P4 [7], a protocol- and target-independent language for specifying packet processing behavior of a switch, or Merlin [48], a declarative, high-level specification language for modeling networks and their resources constraints.

Relevant for verification purposes, few of these DSLs offer some support for formal reasoning, such as NetCore [37], but are often limited in terms of the network properties they can verify. To fill this gap, NetKAT [2] was introduced as a network modeling language for packet forwarding policies based on Kleene Algebra with Tests (KAT) [27]. Being Kleene complete and providing a sound and complete set of axioms over its semantics, NetKAT enables formal mathematical reasoning of the SDN data plane. This allows for the verification of various properties with respect to a given network, such as packet reachability or traffic isolation, or to check the equivalence of NetKAT programs [2].

Syntax

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

Semantics

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

$$[\![1]\!]\ h \triangleq \{h\}$$

$$[\![0]\!]\ h \triangleq \{\}$$

$$[\![f = n]\!]\ (pk{::}h) \triangleq \begin{cases} \{pk{::}h\} & \text{if } pk.f = n \\ \{\} & \text{otherwise} \end{cases}$$

$$[\![\neg a]\!]\ h \triangleq \{h\} \setminus ([\![a]\!]\ h)$$

$$[\![f \leftarrow n]\!]\ (pk{::}h) \triangleq \{pk[f := n]{::}h\}$$

$$[\![p + q]\!]\ h \triangleq [\![p]\!]\ h \cup [\![q]\!]\ h$$

$$[\![p \cdot q]\!]\ h \triangleq ([\![p]\!] \bullet [\![q]\!])\ h$$

$$[\![p^*]\!]\ h \triangleq \bigcup_{i \in \mathbb{N}} F^i\ h$$

where $F^0\ h \triangleq \{h\}$ and $F^{i+1}\ h \triangleq ([\![p]\!] \bullet F^i)\ h$

$$[\![\mathsf{dup}]\!]\ (pk{::}h) \triangleq \{pk{::}(pk{::}h)\}$$

FIGURE 2.2: Syntax and semantics of NetKAT [2]

For a better understanding of how forwarding policies can be encoded using NetKAT, consider the syntax and semantics of the language shown in figure 2.2, and the simple topology depicted in figure 2.3, consisting of two hosts exchanging packets over one switch.

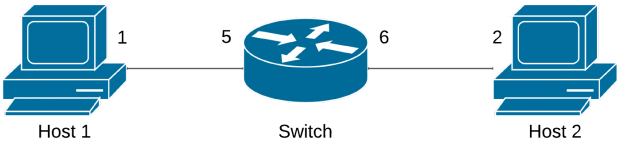


FIGURE 2.3: Two hosts exchanging packets over a switch

In NetKAT, packets are represented as records comprising various *fields*, each assigned specific *values*. When a switch processes a packet, it appends the packet to a list of previously processed packets, known as a *history*. NetKAT facilitates the expression of packet-processing behavior through a combination of logical predicates and policies. The language includes:

- *Predicates*: These serve as filters and encompass constants such as true and false, field tests ($f = v$), which assert the value of a specific field, logical negation , conjunction, and disjunction.
- *Policies*: These define packet-processing behaviors and include filters, which are the composition of predicates, field assignments ($f \leftarrow v$), duplication of the current packet, union, sequential composition, and repetition, which represents zero or more repetitions of a policy.

Semantic-wise, each NetKAT predicate and policy is expressed as a function $[\![-]\!]$ that takes a history and outputs a set of histories, which can be empty. It must be remarked that these functions are only concerned with the first packet in the given history. For example, the semantic rule for tests is defined by the function $[\![f = n]\!](pk :: h)$, which outputs a set containing the input history if the first packet pk in that history has a field f with value n, or an empty set otherwise. With these in mind, the example from figure 2.3 can be encoded in NetKAT as follows:

$$\begin{aligned} SW \quad \triangleq \quad & ((typ = SSH) \cdot (port = 5) \cdot (port \leftarrow 2)) + \\ & ((typ = SSH) \cdot (port = 6) \cdot (port \leftarrow 1)) \end{aligned} \tag{2.1}$$

where *port* and *typ* are the fields of the packets flowing through the network denoting the type of packet and its current port. Intuitively, the encoding states that whenever the switch encounters a packet of type *SSH* with port value *5* it should redirect it to Host2 by assigning its current port field to value *2*. If it encounters an SSH packet with port value *6*, the switch should redirect it to Host1 in a similar fashion. Any other packet that fails the type and port tests is dropped.

With the small network example encoded in NetKAT, one can now formally reason about different network properties in a pure equational fashion. For example, to check if the switch forwards only packets of type *SSH* from port *6* to Host1, it suffices to show, using the NetKAT equational theory, that the following holds:

$$\left(\begin{array}{c} \neg(typ = SSH) \cdot (port = 6) \cdot \\ SW \cdot \\ (port = 1) \end{array} \right) \equiv 0 \tag{2.2}$$

i.e. any non-SSH packet at port *6* of the switch should never end up with port value *1* after the switch policy is applied on the packet. In other words, the corresponding NetKAT property should be equivalent to *0*, or false.

The strong mathematical foundation of NetKAT inspired several tools and extensions of the language to be developed, such as KATch [36], a symbolic verifier for network-wide properties, Stateful NetKAT [33], an extension of NetKAT adding support for mutable global state that may be used in the packet processing functions, or Concurrent NetKAT (CNetKAT) [52], another NetKAT extension that enables the modeling of global states and concurrent packet processing. One shortcoming of most of these extensions is the fact that they allow reasoning about the data plane only, without considering the communication with the control plane. To overcome this, Caltais et al. proposed DyNetKAT [9].

DyNetKAT [9] is an extension of NetKAT that introduces new syntax and semantics to support parallel composition and synchronous communication between programs, which enables modeling and reasoning over entire SDNs. Figure 2.4 shows an overview of the language. The DyNetKAT syntax was developed on top of the NetKAT syntax without packet duplication, and the most notable contributions are the parallel composition operator ($\|$) and the operators for sending (**!**) and receiving (**?**) NetKAT policies over communication channels. In addition, sequential composition (**;**), nondeterministic choice ($\oplus$), and recursive variables are also supported. Unlike NetKAT, DyNetKAT defines structural operational semantics, which means that the semantic rules of the language are described as labeled transitions over 3-tuples of the form (d, H, H'), where d is a DyNetKAT policy, H is a list of packets to be processed and H' is the list of processed packets. Transition labels can be of four types: packet pairs (σ, σ'), policy receiving, denoted $x?p$, policy sending, denoted $x!p$, and reconfigurations, denoted **rcfg(x, p)**, where x is a communication channel and p a NetKAT policy. For example, an important semantic rule is the sequential composition ($\mathbf{cpol}^{\checkmark}_{\_;}$), which separates the NetKAT denotations from the DyNetKAT operations. The rule produces a step labeled with a pair of packets (σ, σ') if the NetKAT policy p successfully processes σ into σ', moving the packet from H to H' and the configuration to the next DyNetKAT policy d [9].

Intuitively, parallel composition ($\mathbf{cpol}_{\_\|}$ and $\mathbf{cpol}_{\|\_}$) of DyNetKAT programs allows them to run concurrently, any of the programs being able to make an execution step at any point in time. Communication is enabled by the DyNetKAT semantics ($\mathbf{cpol}_{!?}$ and $\mathbf{cpol}_{?!}$) whenever two programs synchronize over a communication channel x and a policy p, i.e. the next step of the two programs is sending ($x!p$) and receiving ($x?p$) p over x, respectively. Such synchronization produces a **rcfg(x, p)**-labeled step. Consider figure 2.5, where the previous example of the two hosts exchanging packets over a switch is extended with a control plane consisting of a controller that changes the behavior of the switch when the latter encounters a non-SSH packet. This behavior can be modeled in DyNetKAT as follows:

$$
\begin{aligned}
SW &\triangleq ((typ = SSH) \cdot (((port = 5) \cdot (port \leftarrow 2)) + ((port = 6) \cdot (port \leftarrow 1)))\,;\, SW \oplus \\
&\quad \neg(typ = SSH)\,;\, Help!1\,;\, SW \oplus \\
&\quad Up?1\,;\, SW' \\
SW' &\triangleq 0\,;\, \bot \\
C &\triangleq Help?1\,;\, Up!1\,;\, C \\
SDN &\triangleq SW \,\|\, C
\end{aligned}
\tag{2.3}
$$

Syntax

$$N ::= \text{NetKAT}^{-\mathbf{dup}}$$

$$D ::= \bot \mid N\,;D \mid x?N\,;D \mid x!N\,;D \mid D \,||\, D \mid D \oplus D \mid X$$
$$X \triangleq D$$

Semantics

$$(\mathbf{cpol}^{\checkmark}_{;})\frac{\sigma' \in [\![p]\!](\sigma::\langle\rangle)}{(p;q,\sigma::H,H') \xrightarrow{(\sigma,\sigma')} (q,H,\sigma'::H')} \qquad (\mathbf{cpol_X})\frac{(p,H_0,H_1) \xrightarrow{\gamma} (p',H_0',H_1')}{(X,H_0,H_1) \xrightarrow{\gamma} (p',H_0',H_1')} X \triangleq p$$

$$(\mathbf{cpol}_{\_\oplus})\frac{(p,H_0,H_0') \xrightarrow{\gamma} (p',H_1,H_1')}{(p\oplus q,H_0,H_0') \xrightarrow{\gamma} (p',H_1,H_1')} \qquad (\mathbf{cpol}_{\oplus\_})\frac{(q,H_0,H_0') \xrightarrow{\gamma} (q',H_1,H_1')}{(p\oplus q,H_0,H_0') \xrightarrow{\gamma} (q',H_1,H_1')}$$

$$(\mathbf{cpol}_{\_||})\frac{(p,H_0,H_0') \xrightarrow{\gamma} (p',H_1,H_1')}{(p||q,H_0,H_0') \xrightarrow{\gamma} (p'||q,H_1,H_1')} \qquad (\mathbf{cpol}_{||\_})\frac{(q,H_0,H_0') \xrightarrow{\gamma} (q',H_1,H_1')}{(p||q,H_0,H_0') \xrightarrow{\gamma} (p||q',H_1,H_1')}$$

$$(\mathbf{cpol}_{?})\frac{}{(x?p;q,H,H') \xrightarrow{x?p} (q,H,H')} \qquad (\mathbf{cpol}_{!})\frac{}{(x!p;q,H,H') \xrightarrow{x!p} (q,H,H')}$$

$$(\mathbf{cpol}_{!?})\frac{(q,H,H') \xrightarrow{x!p} (q',H,H') \quad (s,H,H') \xrightarrow{x?p} (s',H,H')}{(q||s,H,H') \xrightarrow{\mathbf{rcfg(x,p)}} (q'||s',H,H')}$$

$$(\mathbf{cpol}_{?!})\frac{(q,H,H') \xrightarrow{x?p} (q',H,H') \quad (s,H,H') \xrightarrow{x!p} (s',H,H')}{(q||s,H,H') \xrightarrow{\mathbf{rcfg(x,p)}} (q'||s',H,H')}$$

$$\gamma ::= (\sigma,\sigma') \mid x!q \mid x?q \mid \mathbf{rcfg(x,q)}$$

FIGURE 2.4: Syntax and semantics of DyNetKAT [9]

The final network is defined by the term *SDN*, which is the parallel composition between the switch and the controller, allowing them to run concurrently. The controller behavior is defined by *C*, which simply listens on channel *Help* for any incoming message from the switch. Whenever that happens, it responds with the NetKAT policy *1* through channel *Up*, then goes back to its initial state. The behavior of the switch is defined by *SW*, which can non-deterministically decide between three operations:

1. Process a packet according to the NetKAT policy defined in the previous example and go back to the initial state
2. If a non-SSH packet is encountered, request a policy update by synchronizing with the controller and send a message through channel *Help*, then go back to the initial state
3. Receive messages from the controller on channel *Up*, then change the behavior to *SW'*

For simplicity, the new behavior of the switch is to drop all incoming packets. Note that in this simple example, the policies sent over the channels do not matter since the communication is used to signal the change, rather than to install new policies. However, more complicated NetKAT programs can be sent through these channels, which can then be used by the switch to change the packet processing behavior.

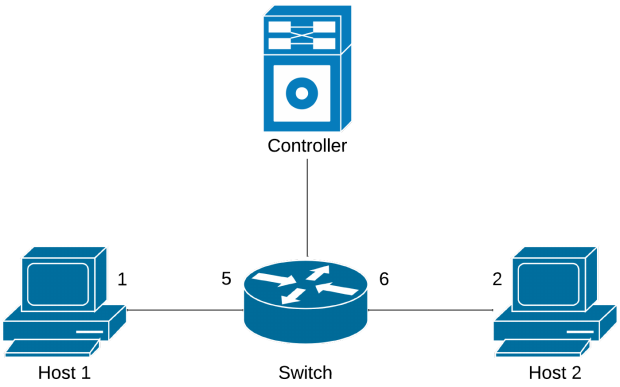


FIGURE 2.5: A simple SDN consisting of two hosts communicating over a switch managed by a controller

2.4 Detecting Race Hazards in SDNs

Various formal methods are used in race hazard detection, which are usually applied under two main verification methods: static and dynamic [30] [18]. Dynamic verification refers to reasoning about a particular system at runtime, typically using real inputs that can occur during usage [19]. Detection tools that do not actually run the target system, but do use information derived from the system's execution, such as execution traces, also fall under this kind of analysis. The main advantages of dynamic verification are that it can capture the true system behavior and it does not produce as many false positives, but not all execution paths can be explored as some of them may be extremely rare or simply not possible in the testing environment used. On the other hand, static analysis explores all possible execution paths of the system, often exhaustively, so it can capture rare events that may happen in practice, but it comes at a significantly higher performance overhead and is known to produce more false positives [19]. This section explores how these verification types are used for the detection of various race hazards in SDNs, which is mainly relevant for **RQ4**.

Dynamic Detection of Race Hazards Most of the tools described in the characterization of Lavado et al. [30] use dynamic verification; however, only two of them are concerned with race hazard detection, of which one offers only debugging capabilities, such as breakpoints and replay of packet traces, and does not perform any automatic detection of race hazards. Hence, from [30], the only representative in the dynamic verification category is SDNRacer [17].

SDNRacer [17] is a dynamic detection tool for SDN data races that occur between the control and the data plane. It is based on a happens-before model derived from the OpenFlow [34] specification and the implementations of two popular software programs

for network switches. Usually, a happens-before model defines an irreflexive transitive binary relation over a set of events or actions that specifies whether an event happens before another [29]. For example, if a controller sends a message that is received by a particular switch, then it is said that the sending happens before the receiving. When two events are not part of the happens-before relation, they occur concurrently [29]. In the context of SDNRacer, this relation is defined over a set of events that capture the behavior of controllers, switches, and hosts, as described in the specifications and the software programs analyzed. An important observation made by the authors of this tool, which also justifies the sub-question **a** of **RQ4**, is that not all concurrent behavior detected using the happens-before relation is necessarily harmful, leading to (possibly) many false positives that affect the output quality of the tool. To address this, a couple of filters were designed for SDNRacer, the most relevant being the one based on a commutativity model, which defines rules for when two events are allowed to commute or, in other words, behave concurrently. According to the authors, such filters eliminated approximately 99.97% of the false positives signaled.

A similar approach to detect SDN race hazards is also used in ConGuard [53]: a framework to detect and exploit harmful data races, particularly in SDN controller software. Although ConGuard does not specifically consider data races that occur due to network events, it does focus on how these events are processed by the control plane, which happens in a multithreaded fashion. Their dynamic analysis is aimed at the controller software, so they consider data races enabled by racing operations on shared memory, in particular, shared network state stored by the controller, such as host profiles or switches' state. To better focus their search and prevent false positives, they also base their detection on a happens-before model, unique to the SDN control plane, as the authors claim. In this case, the happens-before relation is defined over a representative set of actions related to network state access, such as event handling and dispatching, or reading and writing to variables. Unlike SDNRacer, ConGuard identifies commutativity races by treating the happens-before check as a graph reachability problem, instead of defining a custom commutativity model. Moreover, they further confirm that the detected data races are indeed harmful by instrumenting the SDN controller source code to enable the particular execution paths in which these data races occur. Finally, the authors conclude that their data race detection approach is complementary to the one in SDNRacer, because the latter focuses more on the memory operations performed in the data plane.

In [31], Lu et al. present SDN-predict: a sound predictive analysis framework to detect SDN race hazards that affect forwarding devices due to the interleaving of network events at the southbound interface. Similarly to the other two techniques, SDN-predict also focuses on defining ordering constraints between events exchanged by controllers and switches, such as happens-before rules; however, their approach to detect data races explores all possible event reorderings of a given trace using an SMT solver [4]. The reason behind this choice is the data races that may be missed by applying the ordering constraints only on the traces that are observed in the network, SDNRacer being susceptible to this, as the authors show. SDN-predict defines a data race as two consecutive events that access the same flow table of a switch, and at least one of these events attempts to modify the said flow table. The framework also defines three types of constraint for a given event trace: (i) *read consistency*, which specifies that the ordering of read events must be preserved with respect to the latest write event that precedes them; (ii) *must happen-before* constraints, which define happens-before rules between network events, just like SDNRacer; (iii) *asynchronous nondeterminism*, which simply refers to a particular subset

of events that can occur asynchronously and are allowed to be interchanged with read operations in an event trace, such as a switch requesting an update from the controller. To detect data races using these definitions, SDN-predict starts from a set of event traces that can be observed in the target network, e.g. by simulating the network. Then, given a particular trace, the tool encodes all the specified constraints for this particular trace, together with any pair of events that may cause a data race, into a formula that is fed to the SMT solver. If the formula is unsatisfiable, then no data races occur for the given trace; otherwise, the SMT solver returns the possible reorderings in which data races can be observed. To further improve the output quality of their tool, the authors also employ the two false positive filters defined in [17]. Using this approach, SDN-predict manages to perform, overall, better at data race detection than SDNRacer.

Static Detection of Race Hazards Many static detectors of SDN race hazards rely on model checking as their main verification method. Kuai [32] and CMurphi [45] are two such tools described in [30]. Despite not being directly concerned with race hazards but wider classes of properties, they are still able to detect some harmful race conditions when the specified properties are violated. Kuai was designed to verify safety properties in controllers execution using a distributed model checker. It requires the safety properties, the network topology and a model of the target controller as input, all described in a specification language called Murphi [15]. The main contributions of Kuai are the unique SDN-specific optimizations used to simplify the behavior of the network and drastically reduce the state space that the model checker has to explore. Sethi et al. [45] take a similar approach to verify SDN controllers using a general-purpose model checker called CMurphi, but instead focus on per-packet properties, such as invalid packet drops. In the same spirit as Kuai, the authors' main contributions are the two SDN-specific abstractions for the large number of packets flowing through the network and the switches' flow tables, which reduce the state space that has to be explored.

In [51], Vinarskii et al. present a more recent and direct approach to race hazard detection in SDNs using model checking. They derive an automata model for each SDN component, namely an application, a controller, and a switch, from the OpenFlow specification [34] and a popular controller software program. Based on these abstractions, as described in section 2.2, they formulate two types of race hazards: *input-output* and *intra-channel* races, which are individually verified on the derived models using the Promela specification language and the SPIN model checker [23]. For each race hazard type, corresponding Linear Temporal Logic (LTL) properties are formulated, which are verified with SPIN on the derived automatons, confirming through counter-examples the existence of the identified race hazards. The main disadvantage of this approach is the thorough analysis required for both, constructing the models and recognizing the possible race hazards that may occur, as the LTL properties cannot be properly specified otherwise. For example, to confirm that the installation and deletion of the same flow rule on a switch can cause a harmful data race, the corresponding LTL properties describing that the flow rule is successfully installed and deleted had to be specified. In this case, the harmful data race is triggered by the interleaving of the two operations, which prevents the flow rule to be deleted, causing a misconfiguration of the switch.

The authors of [3] explore a different static analysis approach in VeriCon: an SDN verification tool based on invariant checking using SMT solvers [4] that can reason about the correctness of controllers and switches over all possible network topologies and event sequences. The tool can indirectly detect harmful data races, among other undesired be-

haviors of the network, through user-defined invariants of three types: topology invariants, safety invariants, which have to hold for all event sequences, and transition invariants, which are checked after the execution of every event. The authors developed a simple imperative language, called CSDN, that is used to write SDN programs, incorporating semantics for controller and switch events, the latter being dictated by the OpenFlow standard [34]. Intuitively, the tool takes as input a CSDN program and a set of topology constraints and user-defined invariants specified in first-order logic, which are transformed into a so-called Verification Condition (VC) using the standard weakest preconditions [14]. The VC is then fed into an SMT solver [4] that checks the specified invariants in the given SDN program for all possible network topologies and event sequences that satisfy the constraints. If the VC is not satisfiable, the SMT solver returns a network topology and an event sequence as a counterexample. It is possible that the satisfiability check does not terminate; however, the authors claim that the generated formulas can be easily checked by SMT solvers in the majority of cases. Despite its thorough analysis, race hazards are not the main concern of VeriCon. Moreover, the tool requires the user to specify the corresponding invariants that enables race hazard detection, which, similar to model checking, requires the said user to have knowledge about the race hazards it is looking for.

Tracer Part of the main focus of this research is Tracer [54], a static analysis tool that aims to detect race hazards in SDNs using Vector Clocks [29] and the symbolic semantics of DyNetKAT [8].

Intuitively, vector clocks are a simple way of detecting when two parts of a system behave concurrently by tracking the order of events/actions of the entire system locally, at each of the said parts, and then compare all versions with each other to check for potential desynchronizations. Thus, for a distributed system with n subparts, a vector clock is defined as an array of length n, where the value at each position in the array represents the number of events/actions performed by a corresponding system component [29]. Each such component receives a vector clock, and the event/action tracking is performed according to two rules:

1. if component i performs an operation that does not involve communicating with another component, then it increases the value at position i in its associated vector clock by one
2. if component C_i wants to send a message to component C_j then:
   (a) C_i increases the value at position i in its own vector clock by 1
   (b) C_i sends the message and the updated vector clock to C_j
   (c) C_j modifies its vector clock by storing, at the corresponding positions, any of the newly received values that are higher than the current ones
   (d) Finally, C_j increases the value at position j in its own vector clock by 1

Two vector clocks are *incomparable* when neither all values of one vector clock are greater than or equal to the corresponding values of the other vector clock, nor are they all less than or equal. At any point in time, when two components have such vector clocks, then it is said that they behaved concurrently [29]. In other words, the incomparable vector clocks signal that one component witnessed a different execution trace of the system than the other component, so they are out of sync.

$$(\mathbf{Symb}_{||})\ \frac{h.n.f(q_i) \triangleq x!q; d_i \oplus r_i \quad h.n.f(q_k) \triangleq x?q; d_k \oplus r_k}{(q_1)_{\vec{c_1}} \,||\, \ldots \,||\, (q_i)_{\vec{c_i}} \,||\, \ldots \,||\, (q_k)_{\vec{c_k}} \,||\, \ldots \,||\, (q_n)_{\vec{c_n}} \xrightarrow{\mathbf{rcfg(x,p)}} (q_1)_{\vec{c_1}} \,||\, \ldots \,||\, (d_i)_{\vec{c_i}[i]++} \,||\, \ldots \,||\, (d_k)_{(max(\vec{c_i}[i]++,\vec{c_k}))[k]++} \,||\, \ldots \,||\, (q_n)_{\vec{c_n}}}$$

$$(\mathbf{Symb}_{\checkmark})\ \frac{p_i \in \text{NetKAT}^{-\mathbf{dup}} \qquad n.f.(p_i) = \Sigma_{\alpha_i \cdot \pi_i \in \mathcal{A}} \alpha_i \cdot \pi_i}{(p_i; q_i)_{\vec{c_i}} \,||\, \mathit{\Pi}_{\substack{1 \leq j \leq n \\ j \neq i}} d_{j\,\vec{c_j}} \xrightarrow{(\sigma_{\alpha_i}, \sigma_{\pi_i})} (q_i)_{\vec{c_i}[i]++} \,||\, \mathit{\Pi}_{\substack{1 \leq j \leq n \\ j \neq i}} d_{j\,\vec{c_j}}}$$

$$(\mathbf{Symb}_{\mathbf{X}})\ \frac{(p_i)_{\vec{c_i}} \,||\, \mathit{\Pi}_{\substack{1 \leq j \leq n \\ j \neq i}} d_{j\,\vec{c_j}} \xrightarrow{(\sigma,\sigma')} (p_i')_{\vec{c_i'}} \,||\, \mathit{\Pi}_{\substack{1 \leq j \leq n \\ j \neq i}} d_{j\,\vec{c_j}}}{(X_i)_{\vec{c_i}} \,||\, \mathit{\Pi}_{\substack{1 \leq j \leq n \\ j \neq i}} d_{j\,\vec{c_j}} \xrightarrow{(\sigma,\sigma')} (p_i')_{\vec{c_i'}} \,||\, \mathit{\Pi}_{\substack{1 \leq j \leq n \\ j \neq i}} d_{j\,\vec{c_j}}} X_i \triangleq p_i$$

$$(\mathbf{Symb}_{\oplus})\ \frac{(p_i)_{\vec{c_i}} \,||\, \mathit{\Pi}_{\substack{1 \leq j \leq n \\ j \neq i}} d_{j\,\vec{c_j}} \xrightarrow{(\sigma,\sigma')} (p_i')_{\vec{c_i'}} \,||\, \mathit{\Pi}_{\substack{1 \leq j \leq n \\ j \neq i}} d_{j\,\vec{c_j}}}{(p_i \oplus q_i)_{\vec{c_i}} \,||\, \mathit{\Pi}_{\substack{1 \leq j \leq n \\ j \neq i}} d_{j\,\vec{c_j}} \xrightarrow{(\sigma,\sigma')} (p_i')_{\vec{c_i'}} \,||\, \mathit{\Pi}_{\substack{1 \leq j \leq n \\ j \neq i}} d_{j\,\vec{c_j}}}$$

FIGURE 2.6: Symbolic semantics of DyNetKAT (relevant excerpt) [8]

This exact mechanism is also defined in the symbolic semantics of DyNetKAT [8], of which a relevant excerpt can be seen in figure 2.6. These are special semantic rules that operate on *symbolic configurations*, which are DyNetKAT policies of shape:

$$S_1\,\vec{c}_1 \parallel \ldots \parallel S_n\,\vec{c}_n \parallel C_1\,\vec{c}_{n+1} \parallel \ldots \parallel C_m\,\vec{c}_{n+m} \tag{2.4}$$

that encode SDNs with n switches and m controllers. Every device in the network has a corresponding encoding in this symbolic configuration, which is free of $\parallel$ operators and has a vector clock associated with it, denoted by $\vec{c_i}$. From this, the symbolic semantic rules simply specify the same behavior of vector clocks as explained above. For example, $\mathbf{Symb}_{\parallel}$ defines a transition rule that updates the vector clock of two components according to Rule 2 above whenever two components of the parallel composition communicate synchronously. One notable aspect is the fact that these rules operate only on network devices encoded with *guarded* DyNetKAT policies in *head normal form (h.n.f)*. These are simply rewritten DyNetKAT policies, using the language's set of axioms, in a form specifying all possible transitions that can be taken from the current step:

$$\Sigma^{\oplus}_{i \in I}\,(\alpha_i \cdot \pi_i)\,;\, d_i \oplus \Sigma^{\oplus}_{j \in J}\,c_j\,;\, d_j \tag{2.5}$$

where d_i and d_j are DyNetKAT policies, and c_j denotes communication actions (sending, receiving, or reconfiguration). α_i and π_i are *complete tests* and *complete assignments*, respectively, that *guard* these policies. Intuitively, a complete test is defined as a sequence of NetKAT tests on packet fields (e.g. $(f1 = v1) \cdot (f2 = v2) \cdot \ldots$) while a complete assignment is defined as a sequence of NetKAT assignments of packet fields (e.g. $(f1 \leftarrow v1) \cdot (f2 \leftarrow v2) \cdot \ldots$). Guards composed of complete tests and complete assignments are what enable the symbolic execution in Tracer because they essentially represent equivalence classes of packets, which allows the tool to reason about race hazards without feeding actual packets into the network.

For a better understanding of the symbolic semantics, consider again the DyNetKAT encoding given in equation (2.3). An excerpt of the corresponding execution trace that leads to two incomparable vector clocks according to the symbolic semantic rules is shown in figure 2.7. From the configuration defined by the term SDN, the initial switch (SW) encounters the non-SSH packet σ_b and takes one transition corresponding to the processing of the non-SSH packet guard ($\neg(typ = SSH)$). Then, it synchronizes with the controller on channel $Help$ and policy 1, triggering a reconfiguration step that updates the vector clocks of both components. At this point, the switch should wait for a new policy from the controller before continuing packet processing (step ⑤); however, in this example, the switch can still process SSH packets while the controller performs the policy change, which is a harmful race hazard signaled by the incomparable vector clocks (step ④).

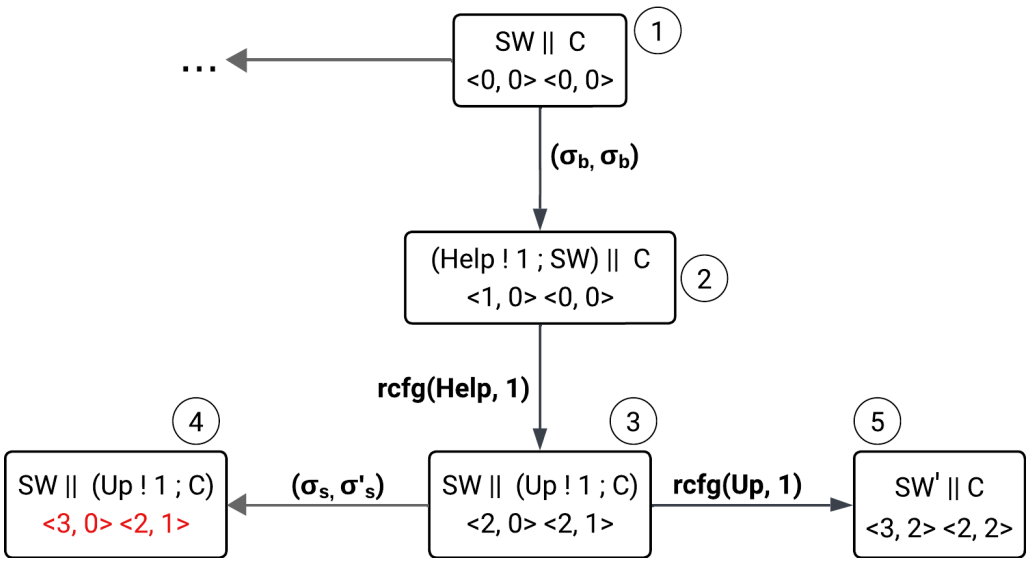


FIGURE 2.7: Symbolic execution snippet of equation (2.3)

With this background in mind, the prototype algorithm implemented for Tracer in [54] is straightforward. Based on an interaction between Python and Maude [11], the algorithm takes an SDN encoded in DyNetKAT and a search depth, and, in one recursive step, invokes Maude to rewrite the encoding in head normal form, then takes all next possible transitions enabled by the rewriting. With every transition taken by the algorithm, the vector clocks of the SDN components are updated accordingly. Whenever the prototype encounters two incomparable vector clocks, it assumes that a race hazard was witnessed, so it saves the symbolic execution trace that led to this result (similar to figure 2.7) and stops advancing on that path. When the given search depth is reached, all such traces are reported. It must be remarked that this strategy is complete in the sense that all possible

race hazards up to the given depth can be identified [8], and, as a consequence of the symbolic execution, the reported traces are minimal.

As mentioned earlier, the current implementation of the Tracer prototype was not properly evaluated in [54], the tool being used only on a few examples of small DyNetKAT networks. This is also what led to **RQ2**. Furthermore, the prototype's source code comes with a few issues. The most notable ones are the long processing time required by the tool even on small examples and reduced search depth, and some unsupported NetKAT operators, such as the recursive operator ($*$), which limits the amount of DyNetKAT networks that can be analyzed. The interplay between Python and Maude is suspected to contribute to the former drawback, because the rewriting that Maude performs is very costly. However, this technology was successfully employed before in other network verification tools based on DyNetKAT [9]. Thus, it still needs to be investigated in **RQ3 a** whether Maude is appropriate for the methodology used in Tracer. Finally, despite the fact that the Tracer detection strategy is complete, it poses the risk of reporting many false positives, which could affect the quality of the tool output and contribute to the bottlenecks mentioned in **RQ4 a**.

To see how easy it is for a harmless race hazard to be reported by Tracer, consider the execution trace snippet from figure 2.8, corresponding to an SDN with one controller and two switches. If the controller decides to install new policies for both switches through the corresponding channels $Ch1$ and $Ch2$, these two reconfiguration transitions are enough to make the vector clock of $Sw2$ incomparable to the vector clocks of the other two devices. Since this is a valid operation of the network that does not present any harmful behavior, the incomparable vector clocks would be reported as two false positives.

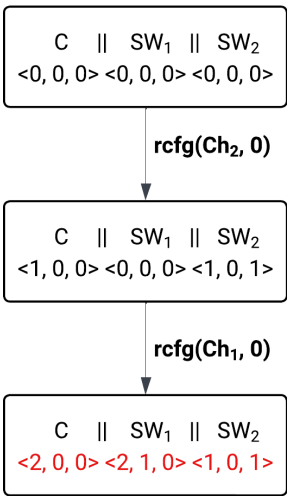


FIGURE 2.8: Harmless symbolic execution snippet of an SDN with 3 components leading to incomparable vector clocks

Having reviewed the foundational concepts behind SDNs, concurrency, and DyNetKAT, as well as existing approaches to SDN race hazard detection including the one used by Tracer and its limitations, we are now prepared to introduce a more robust symbolic execution framework. In the next chapter, we develop this framework in detail, focusing on formalizing three types of harmful SDN race hazards and optimizing the existing DyNetKAT symbolic semantics used for their detection.

Chapter 3

A Formal Framework for SDN Race Hazards

As shown in figure 2.8, the current approach for detecting race hazards using DyNetKAT does not provide a clear understanding of when such race hazards are actually harmful for a network. Recall that Software-Defined Networks heavily rely on concurrency, among other properties, to efficiently handle large volumes of network packets. Catching the majority of events that run concurrently in an SDN does not provide any benefit if their interleaving does not harm the SDN in some way. Therefore, in this chapter, we aim to provide an explanation of harmful race hazards in the context of Software-Defined Networking and formalize them using DyNetKAT and its symbolic semantics.

Section 3.1 presents our approach to modeling SDNs in DyNetKAT using the "one big switch" abstraction [24] and a communication mechanism inspired from the OpenFlow protocol [34], aimed at improving both expressiveness and scalability. Section 3.2 identifies three general categories of race hazards between the control and data planes, as observed in prior work. Section 3.3 defines an optimized version of the DyNetKAT symbolic semantics used by Tracer [54], forming the foundation of our symbolic execution framework. Finally, section 3.4 completes the framework by providing formal definitions of SDN race hazards and the three harmful race types introduced earlier.

3.1 DyNetKAT Big Switches and Open Flow

First, let us introduce the general DyNetKAT expressions that we are using to model the control and data plane of SDNs. These expressions are based on the OpenFlow protocol [34], one of the most popular communication protocols for switches and controllers.

For the data plane, we follow an approach similar to the one used in [2] and model the whole data plane as "*one big switch*" abstraction [24]. In our case, this is a DyNetKAT expression that models the aggregated behavior of all switches in the data plane. Intuitively, instead of considering the switches in a network and their interactions separately, we consider the data plane as a whole, abstracting away the inner interactions and looking only at the traffic that can enter and leave the network. The forwarding behavior of such *big switch* is equivalent to concatenating a set of switch flow tables $X_1, \ldots, X_n \subseteq \text{NetKAT}^{-dup}$ along a network topology $t \in \text{NetKAT}^{-dup}$. This can be encoded into a final NetKAT^{-dup} expression $N \triangleq ((X_1 + \ldots X_n) \cdot t)^*$.

For the communication between big switches and controllers, we model the following OpenFlow events:

- *Flow Mode*, denoted by $FM_i\,!\,N_i$, where $i \in I$; these events are sent by controllers to update the flow table of switches, modifying the way they handle the network traffic. In our approach, the controller will install the new forwarding policy N_i to a switch "listening" on channel FM_i.
- *Packet-In*, denoted by $PI\,!\,M_1$; these events are used by the switch to communicate to the controller any packets for which no flow rule is defined in its current flow table.
- *Packet-Out*, denoted by $PO\,!\,M_2$; these events are used by the controller to install the flow rule corresponding to the packet sent by the switch in a *Packet-In* event.

Figure 3.1 show the overall shape of DyNetKAT big switch and controller expressions that we will assume for the rest of the paper. Note that the big switch expression is parameterized, which enables us to model the changes made by the controller to the flow table of the switch.

$$SW_N \triangleq N\,;\, SW_N \oplus \Sigma^{\oplus}_{i\in I}\, FM_i\,?\,N_i\,;\, SW_{N_i} \oplus \Sigma^{\oplus}_{j\in J}\, PI_j\,!\,M_{1j}\,;\, SWR_{N,j}$$

$$SWR_{N,j} \triangleq N\,;\, SWR_{N,j} \oplus \Sigma^{\oplus}_{i\in I}\, FM_i\,?\,N_i\,;\, SWR_{N_i,j} \oplus PO_j\,?\,M_{2j}\,;\, SW_{M_{2j}}$$

$$C \triangleq \Sigma^{\oplus}_{i\in I}\, FM_i\,!\,N_i\,;\, C \oplus \Sigma^{\oplus}_{j\in J}\, PI_j\,?\,M_{1j}\,;\, CR_j$$

$$CR_j \triangleq \Sigma^{\oplus}_{i\in I}\, FM_i\,!\,N_i\,;\, CR_j \oplus PO_j\,!\,M_{2j}\,;\, C$$

$$SDN \triangleq SW_N \,\|\, C$$

FIGURE 3.1: $I, J \subseteq \mathbb{N}, \forall k \in I \cup J : M_{1k}, M_{2k},\ N, N_k \in \text{NetKAT}^{-dup}$, $PI_k, PO_k,\ FM_k -$ DyNetKAT channels

Intuitively, a DyNetKAT big switch can (i) forward packets according to its policy $N \in \text{NetKAT}^{-dup}$, (ii) "*listen*" for incoming flow mode messages from the controller, or (iii) communicate with the controller via packet-in and packet-out messages. Complementary, the controller can (i) send flow mode messages to a switch, and (ii) receive packet-in messages from the switch, and respond with corresponding packet-out messages.

3.2 SDN Race Hazards by Example

Based on the switch and controller expressions introduced in the previous section, we now introduce three general examples of SDN race hazards inspired from related work, such as [54], [9], or [22]. We use DyNetKAT to model the behavior of the network and showcase how vector clocks can signal the race hazards.

Example 1. The first example is inspired from [54] and showcase how a race hazard occurs in a simple SDN consisting of one switch and one controller. A possible set of DyNetKAT expressions modeling such SDN can be seen in equation (3.1). The model consists of a switch SW and a controller CT that run in parallel. SW can non-deterministically process packets according to a forwarding policy N or receive two policy updates from the controller via channel Ch, while CT can non-deterministically choose to install the two

policy updates on the switch.

$$\begin{aligned} SW_N &\triangleq (N\,;\,SW_N) \oplus (Ch\,?\,N_1\,;\,SW_{N_1}) \oplus (Ch\,?\,N_2\,;\,SW_{N_2}) \\ CT &\triangleq (Ch\,!\,N_1\,;\,CT) \oplus (Ch\,!\,N_2\,;\,CT) \\ SDN &\triangleq SW_N \parallel CT \end{aligned} \tag{3.1}$$

Figure 3.2 shows a possible execution trace of the SDN from equation (3.1) where a race hazard occurs between SW attempting to process a packet after receiving policy N_1 from CT (event 2) at the same time as CT reconfigures the forwarding behavior of the switch to N_2 (event 3a). Notice how the vector clocks associated with these events become incomparable.

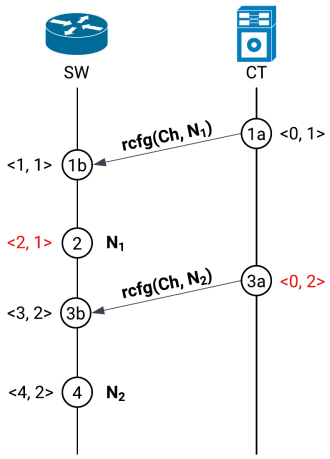


FIGURE 3.2: Possible execution of the SDN modeled in equation (3.1) where a race hazard is signaled between the switch SW and the controller CT by their incomparable vector clocks. Events 2 and 3a correspond to the switch and controller actions that happened concurrently.

Example 2. The second example is inspired from [9] and showcase how a race hazard occurs in an SDN consisting of one switch and two controllers that attempt to modify the switch at the same time. A possible set of DyNetKAT expressions modeling such SDN is presented in equation (3.2). The model consists of a switch SW and two controllers CT_1 and CT_2 that run in parallel. SW can non-deterministically process packets according to a forwarding policy N or receive a policy update from CT_1 and CT_2 through channels Ch_1 and Ch_2, respectively. The only action that the two controllers can make is to install their corresponding update on the switch.

$$\begin{aligned} SW_N &\triangleq (N\,;\,SW_N) \oplus (Ch_1\,?\,N_1\,;\,SW_{N_1}) \oplus (Ch_2\,?\,N_2\,;\,SW_{N_2}) \\ CT_1 &\triangleq Ch_1\,!\,N_1\,;\,CT_1 \\ CT_2 &\triangleq Ch_2\,!\,N_2\,;\,CT_2 \\ SDN &\triangleq CT_1 \parallel SW_N \parallel CT_2 \end{aligned} \tag{3.2}$$

Figure 3.3 shows a possible execution trace of the SDN from equation (3.2). After the switch forwards a packet according to policy N, CT_1 and CT_2 attempt to update the forwarding policy of SW at the same time (event 2a and 3a, respectively), leading to a race hazard. Notice how the vector clocks associated with these events become incomparable.

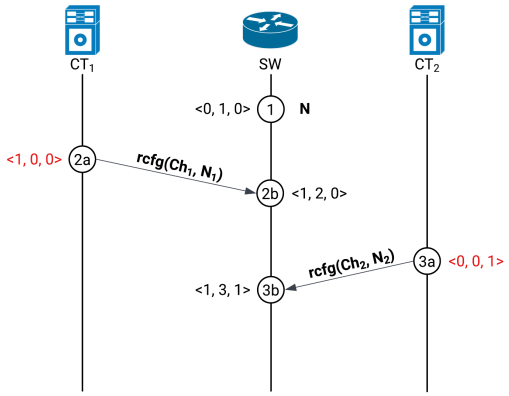


FIGURE 3.3: Possible execution of the SDN modeled in equation (3.2) where a race hazard is signaled between controller CT_1 and controller CT_2 by their incomparable vector clocks. Events 2a and 3a correspond to the actions of the two controllers that happened concurrently.

Example 3. The final example is inspired by [22] and illustrates how a race hazard can occur in an SDN that employs a hierarchical controller architecture. In this scenario, the SDN is composed of a data plane and multiple layers of controllers: the lowest layer directly updates the data plane, while the higher layers update the controllers in the layers below them. A possible set of DyNetKAT expressions modeling such SDN is presented in equation (3.3). The model consists of a switch SW and two controllers CT_1 and CT_2 that run in parallel. SW can non-deterministically process packets according to a forwarding policy N or receive a policy update from CT_1 through channel Ch_1. CT_1 can update the forwarding policy of SW or receive an update from CT_2 via channel Ch_2 that modifies the policy that it can install on SW. CT_2 is only concerned with updating CT_1.

$$\begin{aligned} SW_N &\triangleq (N\,;\, SW_N) \oplus (Ch_1\,?\,N_1\,;\, SW_{N_1}) \\ CT_1^{N_1} &\triangleq (Ch_1\,!\,N_1\,;\, CT_1) \oplus (Ch_2\,?\,N_2; CT_1^{N_2}) \\ CT_2 &\triangleq Ch_2\,!\,N_2\,;\, CT_2 \\ SDN &\triangleq CT_2 \parallel CT_1 \parallel SW_N \end{aligned} \tag{3.3}$$

Figure 3.4 shows a possible execution trace of the SDN from equation (3.3). After the switch forwards a packet according to policy N, CT_1 attempts to update the forwarding policy of SW at the same time as CT_2 updates the policy that CT_1 installs on SW (event 2a and 3a, respectively), leading to a race hazard. Once again, notice how the vector clocks associated with these events become incomparable.

3.3 DyNetKAT Symbolic Semantics - Updated

Recall from section 2.4 that the detection of SDN race hazards using DyNetKAT is done by applying the symbolic semantics of DyNetKAT (see figure 2.6) on *symbolic configurations* in head-normal form as in equation (2.4). Note that, currently, the transitions defined by the DyNetKAT symbolic semantics are concerned with processing symbolic packets. For example, according to the the symbolic rule $\mathbf{Symb}_{\checkmark}$, a transition is produced for all symbolic packets σ_{α_i} that can be processed into σ_{π_i}, where σ_{α_i} is a packet satisfying the

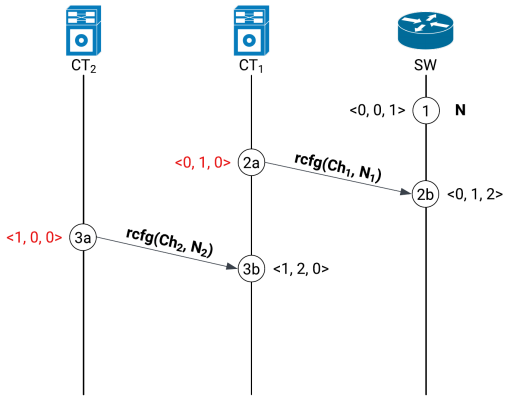


FIGURE 3.4: Possible execution of the SDN modeled in equation (3.3) where a race hazard is signaled between controller CT_1 and controller CT_2 by their incomparable vector clocks. Events 2a and 3a correspond to the actions of the two controllers that happened concurrently.

complete test α_i and σ_{π_i} is the packet obtained by applying the complete assignment π_i on σ_{α_i}.

As we will see in section 3.4, an important observation that we make about these symbolic semantics with respect to the detection of SDN race hazards is that we do not actually need to consider per symbolic packet transitions. Hence, we can optimize the DyNetKAT symbolic rules to produce transitions labeled with entire NetKAT^{-dup} expressions instead of symbolic packet tuples. In SDN terms, this corresponds to labels encoding an entire flow table.

To accommodate these changes, we define the optimized semantics over symbolic configurations in *symbolic normal form*. Similarly to expressions in head-normal form, DyNetKAT expressions in *symbolic normal form* specify all possible actions that can be performed in one step by an SDN from some given point. Such actions can be: processing packets according to a NetKAT^{-dup} policy encoding the SDN data plane as in figure 3.1, sending or receiving messages between control and data planes, or synchronous communication between the control and data planes.

for $p, q, r \in$ DyNetKAT and $z, y \in$ NetKAT$^{-\mathbf{dup}}$
for $a ::= z \mid x?z \mid x!z \mid \mathbf{rcfg}_{x,z}$

$$\begin{aligned}
\mathbf{0}\,;p &\equiv \bot && (A0)\\
(z+y)\,;p &\equiv z\,;p \oplus y\,;p && (A1)\\
p \oplus q &\equiv q \oplus p && (A2)\\
(p \oplus q) \oplus r &\equiv p \oplus (q \oplus r) && (A3)\\
p \oplus p &\equiv p && (A4)\\
p \oplus \bot &\equiv p && (A5)\\
p \,||\, q &\equiv q \,||\, p && (A6)\\
p \,||\, \bot &\equiv p && (A7)\\
p \,||\, q &\equiv p \lfloor\!\lfloor q \oplus q \lfloor\!\lfloor p \oplus p \mid q && (A8)\\
\bot \lfloor\!\lfloor p &\equiv \bot && (A9)\\
(a\,;p) \lfloor\!\lfloor q &\equiv a\,;(p \,||\, q) && (A10)\\
(p \oplus q) \lfloor\!\lfloor r &\equiv (p \lfloor\!\lfloor r) \oplus (q \lfloor\!\lfloor r) && (A11)\\
(x?z\,;p) \mid (x!z\,;q) &\equiv \mathbf{rcfg}_{x,z}\,;(p \,||\, q) && (A12)\\
(p \oplus q) \mid r &\equiv (p \mid r) \oplus (q \mid r) && (A13)\\
p \mid q &\equiv q \mid p && (A14)\\
p \mid q &\equiv \bot\ [owise] && (A15)
\end{aligned}$$

for $at ::= \alpha \cdot \pi \mid x?z \mid x!z \mid \mathbf{rcfg}_{x,z}$:

$$\begin{aligned}
\delta_{\mathcal{L}}(\bot) &\equiv \bot && (\delta_{\bot})\\
\delta_{\mathcal{L}}(at\,;p) &\equiv at\,;\delta_{\mathcal{L}}(p) \text{ if } at \notin \mathcal{L} && (\delta_{;})\\
\delta_{\mathcal{L}}(at\,;p) &\equiv \bot \text{ if } at \in \mathcal{L} && (\delta_{;}^{\bot})\\
\delta_{\mathcal{L}}(p \oplus q) &\equiv \delta_{\mathcal{L}}(p) \oplus \delta_{\mathcal{L}}(q) && (\delta_{\oplus})
\end{aligned}$$

for $n \in \mathbb{N}$:

$$\begin{aligned}
\pi_0(p) &\equiv \bot && (\Pi_0)\\
\pi_n(\bot) &\equiv \bot && (\Pi_{\bot})\\
\pi_{n+1}(at\,;p) &\equiv at\,;\pi_n(p) && (\Pi_{;})\\
\pi_n(p \oplus q) &\equiv \pi_n(p) \oplus \pi_n(q) && (\Pi_{\oplus})
\end{aligned}$$

$$p \equiv q \text{ if } \forall n \in \mathbb{N} : \pi_n(p) \equiv \pi_n(q) \quad (AIP)$$

E_{NK}

FIGURE 3.5: The axiom system E_{DNK} of DyNetKAT

Definition 1 (DyNetKAT symbolic normal form)**.** A DyNetKAT expression p is in *symbolic normal form* (s.n.f.) if it is of shape:

$$\sum_{1\leq i\leq n}^{\oplus} N_i \,;\, p_i \;\oplus\; \sum_{1\leq j\leq n}^{\oplus} c_j \,;\, p_j \;(\oplus\, \bot) \tag{3.4}$$

where N_i ranges over NetKAT^{-dup} policies, d_i, d_j range over DyNetKAT policies and $c_j ::= x?q \mid x!q \mid \mathrm{rcfg}(x, q)$ with q denoting terms in NetKAT^{-dup}.

Just as in [8], we will only consider *guarded* DyNetKAT expressions in symbolic normal form for the switch and controller expressions of the symbolic configurations.

Lemma 1 (DyNetKAT symbolic normalization)**.** Any guarded [9] DyNetKAT expression p can be reduced, according to the axiomatisation of DyNetKAT, to a *symbolic normal form.*

Proof. The lemma is proven by structural induction using E_{DNK} 3.5:

Base cases:

- $p \triangleq \bot$ trivially holds
- $p \triangleq N \,;\, d$ with $N \in$ NetKAT^{-dup} trivially holds
- $p \triangleq c \,;\, d$ with $c ::= x\,?\,q \,|\, x\,!\,q \,|\, \mathrm{rcfg}(x, q)$ where $q \in$ NetKAT^{-dup} trivially holds

Assume that $E_{DNK} \vdash q \equiv \sum^{\oplus}_{1\leq i\leq n} N_i \,;\, p_i \oplus \sum^{\oplus}_{1\leq j\leq n} c_j \,;\, p_j \;(\oplus\, \bot)$ holds for all DyNetKAT expressions q with a syntactic tree depth smaller than p. (**I.H.**)

Induction steps:

1. $p \triangleq p_1 \oplus p_2$ holds by I.H.
2. $p \triangleq p_1 \mid p_2$ holds by I.H. and axioms A_{12}, A_{13}, A_{14}, and A_{15}
3. $p \triangleq p_1 \parallel p_2$ holds by I.H and axiom A_8
4. $p \triangleq X$ is discarded because p is not guarded
5. $p \triangleq \pi_k(p')$ for $k \in \mathbb{N}$, holds by I.H., Π_0, $\Pi_\oplus$, $\Pi_;$, and $\Pi_\bot$.
6. $p \triangleq \delta_{\mathcal{L}}(p')$ for $\mathcal{L} \subset \{at \mid at ::= z \mid x\,?\,z \,|\, x\,!\,z \,|\, \mathrm{rcfg}(x, z),\ z \in \text{NetKAT}^{-dup}\}$, holds by I.H. and axioms $\delta_\oplus$, $\delta_;$, $\delta_;^\bot$, and $\delta_\bot$

Hence, by the principle of structural induction, lemma 1 holds. □

Now, consider the optimized symbolic semantics of DyNetKAT from figure 3.6. The meaning of the original symbolic rules presented in [8] is preserved almost entirely, our modifications only affecting the packet processing transition labels, which are replaced with their corresponding NetKAT^{-dup} expressions. Intuitively, this means that instead of considering each processed symbolic packet as a separate transition, we group them together into one transition that is associated with the corresponding NetKAT expression of these packets. In SDN terms, this corresponds to labels encoding an entire flow table.

For the remaining of the paper, we sometimes use $\mathrm{rcfg}(x, N, i, k)$ as the transition label of $\mathbf{Symb}_\parallel$ to explicitly state that the reconfiguration is made by q_i to q_k over channel x, i.e.

$$(\mathbf{Symb}_{\checkmark})\ \frac{N_i \in \text{NetKAT}^{-dup}}{(N_i\,;\,q_i)_{\vec{c_i}} \parallel \prod_{\substack{1\leq j\leq n\\ j\neq i}} d_{j\vec{c_j}} \xrightarrow{N_i} (q_i)_{\vec{c_i}[i]++} \parallel \prod_{\substack{1\leq j\leq n\\ j\neq i}} d_{j\vec{c_j}}}$$

$$(\mathbf{Symb}_{\mathbf{X}})\ \frac{(p_i)_{\vec{c_i}} \parallel \prod_{\substack{1\leq j\leq n\\ j\neq i}} d_{j\vec{c_j}} \xrightarrow{N_i} (p_i')_{\vec{c_i'}} \parallel \prod_{\substack{1\leq j\leq n\\ j\neq i}} d_{j\vec{c_j}}}{(X_i)_{\vec{c_i}} \parallel \prod_{\substack{1\leq j\leq n\\ j\neq i}} d_{j\vec{c_j}} \xrightarrow{N_i} (p_i')_{\vec{c_i'}} \parallel \prod_{\substack{1\leq j\leq n\\ j\neq i}} d_{j\vec{c_j}}}\ X_i \triangleq p_i$$

$$(\mathbf{Symb}_{\oplus})\ \frac{(p_i)_{\vec{c_i}} \parallel \prod_{\substack{1\leq j\leq n\\ j\neq i}} d_{j\vec{c_j}} \xrightarrow{N_i} (p_i')_{\vec{c_i'}} \parallel \prod_{\substack{1\leq j\leq n\\ j\neq i}} d_{j\vec{c_j}}}{(p_i \oplus q_i)_{\vec{c_i}} \parallel \prod_{\substack{1\leq j\leq n\\ j\neq i}} d_{j\vec{c_j}} \xrightarrow{N_i} (p_i')_{\vec{c_i'}} \parallel \prod_{\substack{1\leq j\leq n\\ j\neq i}} d_{j\vec{c_j}}}$$

$$(\mathbf{Symb}_{\parallel})\ \frac{s.n.f(q_i) \triangleq x\,!\,N\,;\,d_i \oplus r_i \qquad s.n.f(q_k) \triangleq x\,?\,N\,;\,d_k \oplus r_k}{(q_1)_{\vec{c_1}} \parallel \ldots \parallel (q_i)_{\vec{c_i}} \parallel \ldots \parallel (q_k)_{\vec{c_k}} \parallel \ldots \parallel (q_n)_{\vec{c_n}} \xrightarrow{\mathbf{rcfg(x,N)}} (q_1)_{\vec{c_1}} \parallel \ldots \parallel (d_i)_{\vec{c_i}[i]++} \parallel \ldots \parallel (d_k)_{max(\vec{c_i}[i]++,\vec{c_k})[k]++} \parallel \ldots \parallel (q_n)_{\vec{c_n}}}$$

with $N, N_i \in \text{NetKAT}^{-dup}$ and x a variable denoting a communication channel

FIGURE 3.6: (Optimized) symbolic semantics of DyNetKAT (relevant excerpt). For simplicity, the symmetric rules associated with $\mathbf{Symb}_{\oplus}$ and $\mathbf{Symb}_{\parallel}$ were ommited.

q_i is sending policy N on channel x and q_k is synchronously receiving N on x. In SDN terms, this intuitively corresponds to updating the flow table of q_k to N.

Furthermore, as an extension to the binary relations over vector clocks from [8], we define:

$$c \underset{(i,j)}{\parallel} c' \text{ iff } ((\vec{c}_{[i]} < \vec{c'}_{[i]}) \wedge (\vec{c}_{[j]} > \vec{c'}_{[j]})) \vee ((\vec{c}_{[i]} > \vec{c'}_{[i]}) \wedge (\vec{c}_{[j]} < \vec{c'}_{[j]})), \text{ for some } i, j \in \{1 \ldots k\}$$

for any vector clocks c and c' of size k. Intuitively, this new relation states that c and c' are incomparable with respect to some positions i and j, i.e. the values of c and c' at position i are strictly smaller, while the values at position j are strictly greater, or vice versa.

We end this section by defining a labeled transition system based on DyNetKAT symbolic configurations according to the semantics shown in figure 3.6. This serves as the main foundation for constructing symbolic execution traces of DyNetKAT network models.

Definition 2 (Symbolic LTS)**.** Let $d = \prod_{i=1}^{n} d_i$ be a DyNetKAT expression where d_i is free of $\parallel$ for all $i \in \{1, \ldots, n\}$. We use the notations SW and CT to denote expressions of shape d_i that describe the behavior of switches and controllers, respectively.

We call $T = (S,\ s_0,\ \Sigma,\ \rightarrow)$ the *symbolic LTS* associated with d where:

- S is a set of states of shape $\prod_{i=1}^{n} d'_{i\vec{c_i}}$ with:

- d'_i as a DyNetKAT expression free of $\|$ for $i \in \{1, \ldots, n\}$
- $\vec{c'_i}$ as a vector clock of size n for $i \in \{1, \ldots, n\}$

- $s_0 = \prod_{i=1}^{n} d_{i\vec{c_i}}$ with $\vec{c_i} = \langle 0, \ldots, 0 \rangle$ for $i \in \{1, \ldots, n\}$
- $\Sigma \subseteq \{\text{rcfg}(X, N, l, t) \mid X \text{ a variable denoting a communication channel}, N \in \text{NetKAT}^{-dup}, \text{ and } l, t \in \{1, \ldots, n\}\} \cup \text{NetKAT}^{-dup}$
- $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation where $s \xrightarrow{l \in \Sigma} s'$ denotes a transition according to the symbolic rules of DyNetKAT when starting in s, for $s, s' \in S$

For a state $s \triangleq \prod_{i=1}^{n} d_{i\vec{c_i}} \in S$, we call $d_{i\vec{c_i}}$ an *element* of s and write $s \downarrow i$. By abuse of notation, we use $d_{i[i]}$ to denote the value at position i in the vector clock of element $d_{i\vec{c_i}}$. We sometimes refer to the vector clock of element $d_{i\vec{c_i}}$ as the vector clock i of state s.

We call a *trace of* T a sequence $tr \triangleq s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} \ldots$ in T, and we write $tr \in T$. We write $s \in tr$ whenever $s \xrightarrow{l} s'$ or $s' \xrightarrow{l} s$ is a transition in tr.

We call a *finite trace of* T a finite sequence $tr \triangleq s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} \ldots \xrightarrow{l_k} s_{k+1}$ in T, and we write $tr \in T$.

For two finite traces $tr_1 \triangleq v_0 \xrightarrow{l_0} v_1 \xrightarrow{l_1} \ldots \xrightarrow{l_k} v_{k+1}$, $tr_2 \in T$, we call tr_1 a *subtrace of* tr_2 and write $tr_1 \subseteq tr_2$ whenever $tr_2 \triangleq s_0 \xrightarrow{\cdots} \ldots \xrightarrow{\cdots} v_0 \xrightarrow{l_0} v_1 \xrightarrow{l_1} \ldots \xrightarrow{l_k} v_{k+1} \xrightarrow{\cdots} \ldots$.

Figure 3.7 depicts a trace in the symbolic LTS associated with the variable SDN from equation (3.1). The transitions of the symbolic trace are equivalent to the events from figure 3.2.

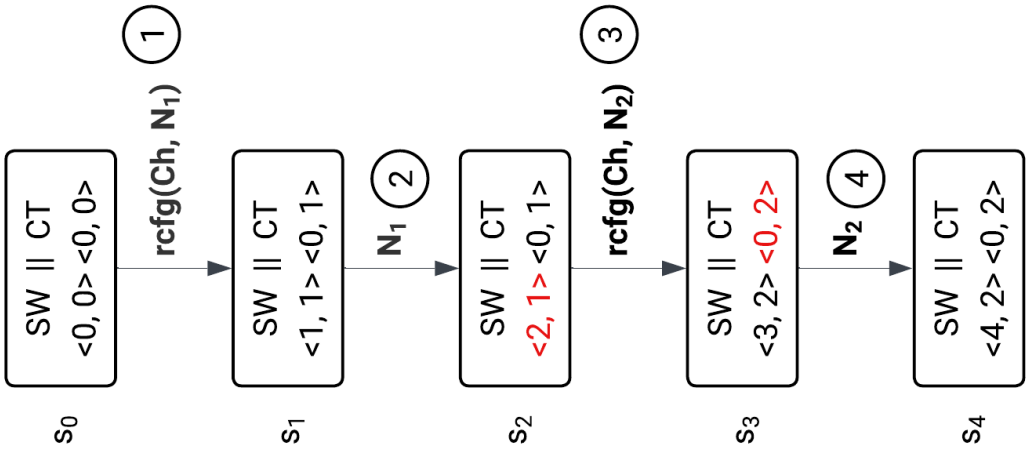


FIGURE 3.7: Trace in the symbolic LTS associated with expression SDN from equation (3.1). The trace transitions are equivalent to the events from figure 3.2.

3.4 Harmful DyNetKAT Races

In this section we leverage DyNetKAT and its optimized symbolic semantics introduced in the previous section to formalize the notion of SDN race hazards and define three cases when such hazards become harmful for an SDN.

Intuitively, a race occurs in a trace tr if there is a pair of states in tr where a vector clock from one state is incomparable with a vector clock from the other state (Definition 3). Such a race is *harmful* with respect to a *forwarding* property ϕ if it alters the packet processing behavior of the network in a way that leads to a different evaluation of ϕ. In our setting, a forwarding property specifies that packets can never traverse the network from a designated *ingress* point to a designated *egress* point. Hence, to verify if a race is harmful, we have to check whether ϕ evaluates differently on the forwarding policy of the network before and after the race manifested. Concretely, we can achieve this by searching in tr the two transitions that trigger the incomparable vector clocks between two states, then compare how the forwarding property evaluates on the labels of these transitions. Given Σ and the three examples from section 3.2, we differentiate between three types of harmful races:

Controller→Switch (CT→SW): a race occurring between a switch SW_t processing packets according to policy N_1 and a controller CT_l reconfiguring SW_t to N_2 is observed in tr through two labeled transitions: $\ldots s_i \xrightarrow{N_1} s_{i+1} \ldots s_j \xrightarrow{\text{rcfg}(X,N_2,l,t)} s_{j+1} \ldots$ that cause the vector clock t of s_{i+1} and the vector clock l of s_{j+1} to be incomparable at positions t and l. Such a race becomes harmful when applying ϕ on the current policy N_1 of SW_t leads to a different result than when applying it to the new policy N_2 from CT_l (Definition 5).

Controller→Switch←Controller (CT→SW←CT): a race occurring between two controllers CT_l^1 and CT_l^2 reconfiguring the same switch SW_t manifests in tr through two labeled transitions: $\ldots s_i \xrightarrow{\text{rcfg}(X_1,N_1,l_1,t)} s_{i+1} \ldots s_j \xrightarrow{\text{rcfg}(X_2,N_2,l_2,t)} s_{j+1} \ldots$ that cause the vector clock l_1 of s_{i+1} and the vector clock l_2 of s_{j+1} to be incomparable at positions l_1 and l_2. Such a race becomes harmful when one of the new policies N_1 and N_2 sent to SW_t satisfies ϕ, but the other does not (Definition 6). This signifies that the order in which CT_l^1 and CT_l^2 reconfigure SW_t matters for the satisfiability of the forwarding property.

Controller→Controller→Switch (CT→CT→SW): a race occurring between two controllers CT_l^1 and CT_l^2, where CT_l^2 is reconfiguring CT_l^1 and CT_l^1 reconfigures a switch SW_t, manifests in tr through two labeled transitions: $\ldots s_i \xrightarrow{\text{rcfg}(X_1,N_1,l_1,t)} s_{i+1} \ldots s_j \xrightarrow{\text{rcfg}(X_2,N_2,l_2,l_1)} s_{j+1} \ldots$. These transitions cause the vector clock l_1 of s_{i+1} and the vector clock l_2 of s_{j+1} to be incomparable at positions l_1 and l_2. Such a race becomes harmful when one of the policies N_1 and N_2 satisfies ϕ, but the other does not (Definition 7). This means that the update of CT_l^2 alters CT_l^1 such that the forwarding property evaluates to a different result than before, thus, the order in which the reconfigurations happen matters.

Remark: A race occurring between two different switches is characterized in tr by two labeled transitions: $\ldots s_i \xrightarrow{N_1} s_{i+1} \ldots s_j \xrightarrow{N_2} s_{j+1}$. Such a race becomes harmful when one of the policies N_1 and N_2 violates the forwarding property, but the other does not; however, this cannot happen in our context because switches do not communicate with

each other. Thus, this either means that one of the two policies did not satisfy ϕ from the beginning, or it is the result of one or more reconfigurations performed by controllers, implying that another harmful race caused it.

Definition 3 (Race)**.** Let $T = (S,\ s_0,\ \Sigma,\ \rightarrow)$ be the symbolic LTS associated with $d = \prod_{i=1}^{n} d_i$ and $tr \in T$. We say that tr exhibits a *race* whenever:

$$\exists \left(s = \prod_{i=1}^{n} d^{s}_{ic^{\vec{s}}_i},\, v = \prod_{i=1}^{n} d^{v}_{ic^{\vec{v}}_i} \right) \in tr.\ \exists c^{\vec{s}}_p \in \{c^{\vec{s}}_1, \ldots, c^{\vec{s}}_n\}.\ \exists c^{\vec{v}}_q \in \{c^{\vec{v}}_1, \ldots, c^{\vec{v}}_n\}.\ c^{\vec{s}}_p \underset{(p,q)}{\parallel} c^{\vec{v}}_q$$

Definition 4 (Forwarding property)**.** A *forwarding property* is a function

$$\phi^{\alpha_{out}}_{\alpha_{in}} : \text{NetKAT}^{-dup} \rightarrow \mathbb{B}$$

such that $\phi^{\alpha_{out}}_{\alpha_{in}}(N) = \texttt{true}$ whenever the axiom system of NetKAT entails $\alpha_{in} \cdot N \cdot \alpha_{out} \equiv \mathbf{0}$, and `false` otherwise. Here, α_{in} and α_{out} are NetKAT^{-dup} complete tests encoding ingress and egress conditions, and N is a NetKAT^{-dup} expression encoding data plane forwarding behavior.

We now specify assumptions and notation used in Definitions 5, 6 and 7. Let $T = (S,\ s_0,\ \Sigma,\ \rightarrow)$ be the symbolic LTS of $d \triangleq \prod_{i=1}^{n} d_i$ as in (2.4), $\phi^{\alpha_{out}}_{\alpha_{in}}$ be a forwarding property, and $tr \in T$. Suppose that $\exists\ v, v', u, u' \in tr$ such that $v = \prod_{i=1}^{n} d^{v}_{ic^{\vec{v}}_i}$, $v' = \prod_{i=1}^{n} d^{v'}_{ic^{\vec{v'}}_i}$, $u = \prod_{i=1}^{n} d^{u}_{ic^{\vec{u}}_i}$, $u' = \prod_{i=1}^{n} d^{u'}_{ic^{\vec{u'}}_i}$ and $\exists\ p, q \in \{1, \ldots, n\}$ such that $c^{\vec{v'}}_p \underset{(p,q)}{\parallel} c^{\vec{u'}}_q$. This formula identifies two elements with incomparable vector clocks, i.e. elements that run concurrently.

Definition 5 (Harmful CT→SW race)**.** We say that tr exhibits a *harmful CT→SW race* with respect to $\phi^{\alpha_{out}}_{\alpha_{in}}$ whenever:

$$(v \xrightarrow{N_1} v' \xrightarrow{\cdots} \ldots \xrightarrow{\cdots} u \xrightarrow{\text{rcfg}(X,N_2,q,p)} u') \subseteq tr \tag{3.5a}$$

$$\wedge\ (v \downarrow p)_{[p]} \neq (v' \downarrow p)_{[p]} \wedge (v' \downarrow p)_{[p]} = (u \downarrow p)_{[p]} \tag{3.5b}$$

$$\wedge\ (u \downarrow q)_{[q]} \neq (u' \downarrow q)_{[q]} \tag{3.5c}$$

$$\wedge\ (\phi^{\alpha_{out}}_{\alpha_{in}}(N_1) \neq \phi^{\alpha_{out}}_{\alpha_{in}}(N_2)) \tag{3.5d}$$

For this definition, the elements that run concurrently are a switch SW_p and a controller CT_q. The formulas in (3.5a)–(3.5c) capture the corresponding transitions that caused these incomparable clocks, specifically, the transitions that modified position p of $c^{\vec{v'}}_p$ and position q of $c^{\vec{u'}}_q$, respectively. Finally, (3.5d) checks whether the forwarding behavior changes: only one of N_1 or N_2 enables forwarding from *in* to *out*.

Remark: Definition 5 covers only one ordering of the two transitions, i.e. $\ldots v \xrightarrow{N_1} v' \xrightarrow{\cdots} \ldots \xrightarrow{\cdots} u \xrightarrow{\text{rcfg}(X,N_2,q,p)} u' \ldots$. The ordering $\ldots u \xrightarrow{\text{rcfg}(X,N_2,q,p)} u' \xrightarrow{\cdots} \ldots \xrightarrow{\cdots} v \xrightarrow{N_1} v' \ldots$, such that the assumptions made in the beginning of the section and formulas (3.5b) and (3.5c) hold, is not possible for the switch and controller expressions that we consider. If the reconfiguration transition happens first, it means that SW_p has already updated its policy to N_2, so it cannot produce packet processing transitions that are labeled N_1 anymore.

Definition 6 (Harmful CT→SW←CT race)**.** We say that tr exhibits a *harmful CT→SW←CT race* with respect to $\phi_{\alpha_{in}}^{\alpha_{out}}$ whenever:

$$(v \xrightarrow{\text{rcfg}(X_1,N_1,p,t)} v' \xrightarrow{\cdots} \ldots \xrightarrow{\cdots} u \xrightarrow{\text{rcfg}(X_2,N_2,q,t)} u') \subseteq tr \tag{3.6a}$$

$$\wedge\ (v \downarrow p)_{[p]} \neq (v' \downarrow p)_{[p]} \wedge (v' \downarrow p)_{[p]} = (u' \downarrow p)_{[p]} \tag{3.6b}$$

$$\wedge\ (u \downarrow q)_{[q]} \neq (u' \downarrow q)_{[q]} \tag{3.6c}$$

$$\wedge\ \ \forall u_1, u_2 \in (v' \xrightarrow{\cdots} \ldots \xrightarrow{\cdots} u),\ \forall i \in \{1 \ldots n\}.$$
$$(i \neq t) \Rightarrow (u_1 \downarrow t)_{[i]} = (u_2 \downarrow t)_{[i]} \tag{3.6d}$$

$$\wedge\ \phi_{\alpha_{in}}^{\alpha_{out}}(N_1) \neq \phi_{\alpha_{in}}^{\alpha_{out}}(N_2) \tag{3.6e}$$

For this definition, the elements that run concurrently are two controllers CT_p and CT_q updating a switch SW_t. Formulas (3.6b) and (3.6c) ensure that the reconfiguration transitions from (3.6a) correspond to these elements and trigger the incomparable vector clocks. In (3.6d), we ensure that SW_t is only processing packets in between the updates and no other reconfigurations target it. Finally, (3.6e) checks whether the forwarding behavior changes: only one of the controllers' updates, i.e., N_1 or N_2, enables forwarding from *in* to *out*.

Definition 7 (Harmful CT→CT→SW race)**.** We say that tr exhibits a *harmful CT→CT→SW race* with respect to $\phi_{\alpha_{in}}^{\alpha_{out}}$ whenever:

$$(v \xrightarrow{\text{rcfg}(X_1,N_1,p,t)} v' \xrightarrow{\cdots} \ldots \xrightarrow{\cdots} u \xrightarrow{\text{rcfg}(X_2,N_2,q,p)} u') \subseteq tr \tag{3.7a}$$

$$\wedge\ (v \downarrow p)_{[p]} \neq (v' \downarrow p)_{[p]} \wedge (v' \downarrow p)_{[p]} = (u \downarrow p)_{[p]} \tag{3.7b}$$

$$\wedge\ (u \downarrow q)_{[q]} \neq (u' \downarrow q)_{[q]} \tag{3.7c}$$

$$\wedge\ \ \forall u_1, u_2 \in (v' \xrightarrow{\cdots} \ldots \xrightarrow{\cdots} u),\ \forall i \in \{1 \ldots n\}.$$
$$(i \neq t) \Rightarrow (u_1 \downarrow t)_{[i]} = (u_2 \downarrow t)_{[i]} \tag{3.7d}$$

$$\wedge\ \phi_{\alpha_{in}}^{\alpha_{out}}(N_1) \neq \phi_{\alpha_{in}}^{\alpha_{out}}(N_2) \tag{3.7e}$$

For this definition, the elements that run concurrently are a controller CT_q reconfiguring another controller CT_p while it updates the forwarding policy of a switch SW_t. Formulas (3.7a)–(3.7c) ensure that the reconfiguration transitions from (3.7a) correspond to these elements and cause the incomparable vector clocks. In (3.7d), we ensure that SW_t is not the target of any updates in between the racing transitions and may only process packets. Finally, (3.7e) checks whether the forwarding behavior changes: only one of the controllers' updates, i.e., N_1 or N_2, enables forwarding from *in* to *out*.

Remark: Analogously to Definition 5, Definition 7 covers only one ordering of the two transitions, i.e. $v \xrightarrow{\text{rcfg}(X_1,N_1,p,t)} v' \xrightarrow{\cdots} \ldots \xrightarrow{\cdots} u \xrightarrow{\text{rcfg}(X_2,N_2,q,p)} u'$. The other ordering $v \xrightarrow{\text{rcfg}(X_2,N_2,q,p)} v' \xrightarrow{\cdots} \ldots \xrightarrow{\cdots} u \xrightarrow{\text{rcfg}(X_1,N_1,p,t)} u'$, such that the assumptions made in the beginning of the section and formulas (3.7b) and (3.7c) hold, is not possible for the switch and controller expressions that we consider. If CT_q reconfigures CT_p first, then CT_p would use the new policy N_2 when reconfiguring switches, including SW_t.

Building on the improved symbolic execution framework introduced in this chapter, we now turn to the development of a new tool for detecting harmful race hazards in SDNs. In the next chapter, we present RaceLoom, describe its architecture, and explore the main components of its implementation.

Chapter 4

RaceLoom: a Tool for SDN Race Hazards Detection

In this chapter we discuss the implementation details and design decisions behind RaceLoom [12], the new implementation of Tracer [54] based on the improved symbolic execution framework presented in chapter 3.

RaceLoom is a command-line tool that is primarily implemented using the Python programming language. Additionally, it leverages the following technologies:

- Maude 3.5 and Maude As A Library [43]: provides a package of code bindings that enables Maude 3.5 operations to be executed directly from Python. The Maude framework [11] is a high-performance system for specifying and analyzing systems based on rewriting logic. We leverage Maude's efficient rewriting logic capabilities to transform network expressions using the axiom system of DyNetKAT into symbolic normal form. This allows the tool to extract all possible steps that the network model can make from a given point, then link these steps together to construct symbolic traces.
- KATch [36]: state-of-the-art tool for checking equivalence of NetKAT programs. We employ KATch during the trace analysis step of RaceLoom to check the forwarding properties specified by the user on the packet processing policy of the network data plane.

As it can be observed, we still use Maude to perform the rewriting of DyNetKAT expressions, just like Tracer. This is mainly because Maude has been successfully used before in another DyNetKAT-based tool presented in [9], and the authors of Tracer did not explicitly point out any limitations with respect to this technology. However, implementing the entire tool using Maude did not seem feasible due to a lack of experience with the framework and the different programming paradigm that it uses, which makes certain features, such as invoking KATch, cumbersome to implement. Hence, Python was used to orchestrate the communication with the other technologies, particularly because it was the most tested programming language for the Maude code bindings [43], as the author claims. The choice behind KATch is straightforward: among the NetKAT verification tools available at the time of writing this paper, KATch is the fastest and most recent tool.

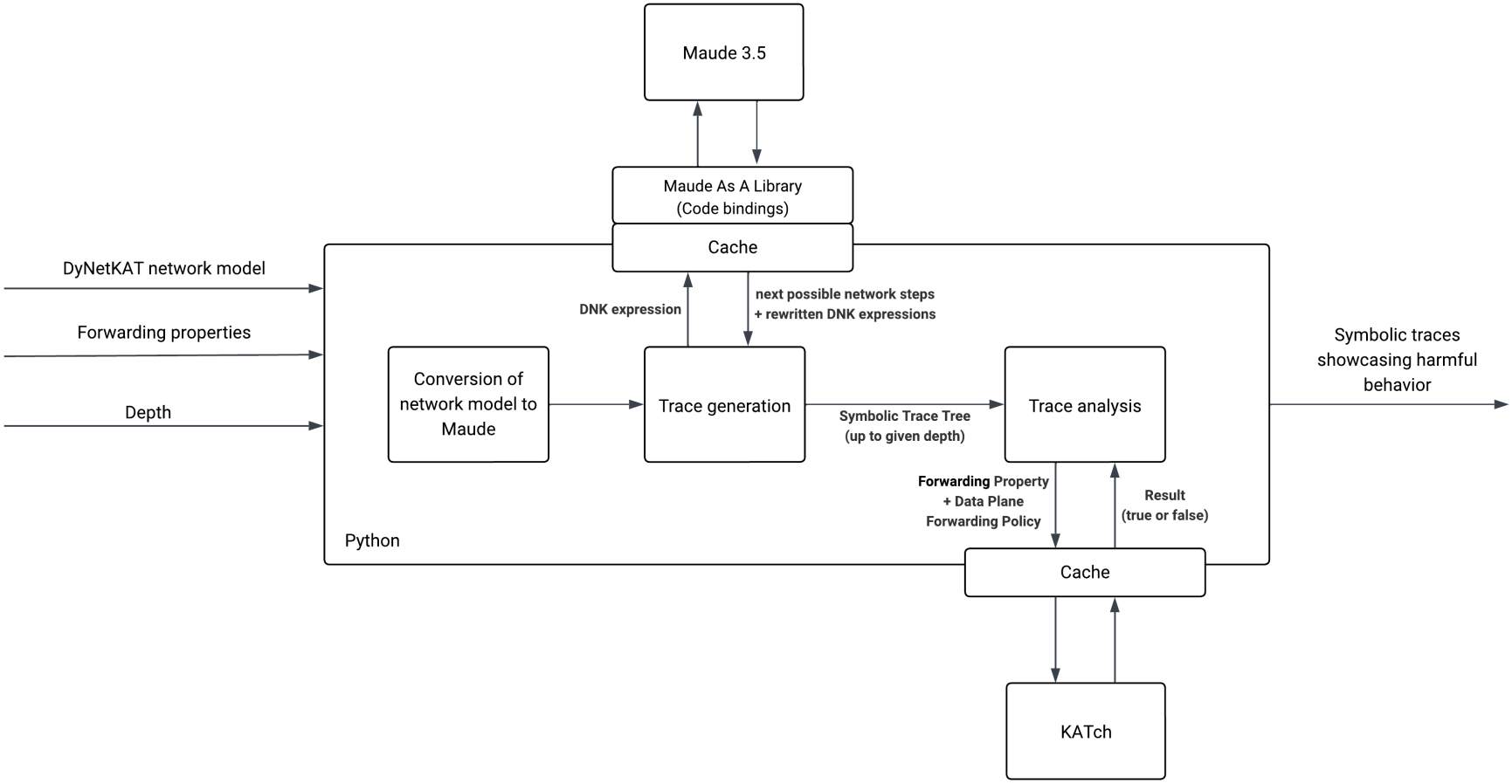


FIGURE 4.1: RaceLoom architecture

Figure 4.1 depicts the architecture of RaceLoom. The tool takes as input a trace depth and two JSON files: one containing the DyNetKAT network model to be analyzed, and one containing the forwarding properties to be checked when potential race hazards are found. RaceLoom converts the network input model into a Maude program, then generates all possible traces of the given network model up to the given depth. The trace generator constructs a symbolic trace tree by continuously rewriting the network expression into symbolic normal form using Maude and storing the resulting network steps into the tree. During this process, vector clocks are associated with each network step, resulting into symbolic traces similar to the one in figure 3.7. After the symbolic trace tree is generated, RaceLoom walks through every trace, identifies each pair of network events that ran concurrently, evaluates with KATch the forwarding property on the concurrent events, then marks the trace as harmful if the results of the two evaluations are different. Note that the tool only considers concurrent events that meet the conditions of the harmful race definitions formalized in chapter 3.

Output examples of RaceLoom can be found in chapter 5.

This chapter continues by detailing the most important parts of the tool's implementation. Section 4.1 provides a brief introduction to the Maude rewriting system and shows how DyNetKAT expressions are specified and manipulated using Maude. Section 4.2 describes the input format expected by RaceLoom and outlines how SDN models are translated into Maude programs. Section 4.3 details the process of generating symbolic traces from the network model, while section 4.4 explains how the tool analyzes these traces to detect harmful race hazards.

4.1 Maude Basics

Maude [11] is a high-performance system based on rewriting logic designed for specifying and executing formal systems, such as programming languages or protocols. The system models computation as rewrite rules applied to algebraic terms.

There are two types of specifications available in Maude:

1. Functional modules, which define equational theories through data types, operators, and equations.
2. System modules, which define rewrite theories through rewrite rules. These type of rules are applied non-deterministically whenever their conditions are satisfied and the left hand side of a rule matches part of the system state.

For the purpose of specifying DyNetKAT expressions, we use only functional modules. As a result, we are able to specify all DyNetKAT operators and axioms of the language as equational theories in Maude. Additionally, Maude supports operator properties such as associativity, commutativity, and idempotency, which improves its efficiency by performing equational reasoning modulo these properties.

Figure 4.2 shows a snippet of a Maude functional module specifying the DyNetKAT non-deterministic operator ($\oplus$), the dummy policy ($\bot$), and some of their associated axioms. It first declares a *sort* called `RDNK`, which is equivalent to a new data type, which we use to describe DyNetKAT expressions free of parallel composition ($\|$). Then it declares the non-deterministic choice operator and the dummy policy.

The $\oplus$ operator is applied on two arguments of type `RDNK` and results into a term of the

same type (`RDNK RDNK` → `RDNK`). The underscores on the left and right side of the operator are the placeholders for the arguments, which tells Maude where these arguments are placed with respect to the operator. Then, between the square brackets, we specify the properties of this operator, such as associativity (`assoc`) and commutativity (`comm`).

The dummy policy ($\perp$) is specified as an operator called `bot` without any arguments of type `RDNK`.

Finally, using a variable `P` of type `RDNK`, two equations are defined corresponding to the DyNetKAT axioms $A4$ and $A5$ shown in figure 3.5. The associativity and commutativity properties of the `o+` operator correspond to axioms $A2$ and $A3$.

This small snippet of equational theory now allows us to specify DyNetKAT expressions using non-deterministic choice and dummy policies in Maude, and use the system to rewrite such expressions based on four axioms of DyNetKAT. For example, the DyNetKAT expression `bot o+ bot` specified in Maude is rewritten by the system to `bot`.

Completing the functional module with the rest of the DyNetKAT operators and axioms enables us to specify SDN models in Maude. Then the system can rewrite them based on the DyNetKAT axioms and extract forwarding and reconfiguration transitions that the SDN can produce, just like in our symbolic framework.

Note that our Maude specifications of the DyNetKAT axioms operate on a slightly more complicated data type of shape `c(P, id)`, where P is an `RDNK` term and `id` is a natural number. Intuitively, this keeps track of an id with every big switch and controller expression, so we can differentiate between each element's actions and also identify between which components the reconfiguration transitions occur.

```
sort RDNK .

op _o+_ : RDNK RDNK -> RDNK [ctor comm assoc metadata "nondeterministic choice" prec 41] .
op bot : -> RDNK .

vars P : RDNK .

---A2: added comm to _o+_
---A3: added assoc to _o+_
eq [A4] : (P o+ P) = P .
eq [A5] : (P o+ bot) = P .
```

FIGURE 4.2: Snippet of a Maude functional module defining the DyNetKAT non-deterministic choice operator ($\oplus$) and the dummy policy ($\perp$)

4.2 Input Format and Conversion

We now described the input format expected by RaceLoom and how it is converted into the final DyNetKAT expressions used internally.

The SDN model is described to the tool in a JSON [39] file, whose structure is shown in figure 4.3. Due to time constraints, a proper parser was not possible to build, so we have to rely on this kind of structure to ensure the network is correctly converted into a big switch expression as in figure 3.1.

The meaning of each key in figure 4.3 is as follows:

- `Switches`: contains information about every switch in the SDN. The dictionary maps switch names to flow table and update information, which include:
  - `InitialFlowTable`: the NetKAT expression of the switch flow table before any updates. If the field is excluded, the default value is 0 (drop all packets).
  - `DirectUpdates`: correspond to *Flow Mode* updates as described in section 3.1. Each update consists of a NetKAT `Policy` received on a `Channel`.
  - `RequestedUpdates`: correspond to *Packet-In* and *Packet-Out* events as described in section 3.1. The switch produces a *Packet-In* event by sending a `RequestPolicy` on a `RequestChannel` to the controller, then waits for a corresponding *Packet-Out* event with policy `ResponsePolicy` on a `ResponseChannel`.
- `Links`: the NetKAT expression encoding all links in the SDN topology.
- `RecursiveVariables`: variables assigned to arbitrary DyNetKAT expressions. These are used to model controller behavior.
- `OtherChannels`: list of channels not specified in the `Switches` dictionary that are used between controllers/recursive variables.
- `Controllers`: a list of recursive variable names that define the entry points of the controllers in the SDN. E.g. the list ["`C1`","`C2`"] means that the behavior specified by variable `C1` should be used as the first controller and the behavior specified by variable `C2` should be used as the second controller.

To be able to distinguish between the different inner switches after we convert the network into a big switch, we enforce the channels to be unique for each one of them.

Note that the updates described in the `Switches` dictionary are expected to target the flow tables of the inner switches, and not the forwarding policy of the big switch.

Internally, this structure is converted into Maude operators and equations based on the DyNetKAT equational theory specified in Maude. The recursive variable expressions are transcribed as is. They are expected to "communicate" with the switches in the network based on the updates specified in the `DirectUpdates` and `RequestedUpdates` fields.

The `Switches` dictionary and the `Links` field are used to define in Maude a recursive variable with n arguments, each corresponding to a NetKAT expression of an inner switch flow table. Then, this variable is assigned to a DyNetKAT expression concatenating all `Direct` and `Requested` updates using non-deterministic choice as in figure 3.1. Since we use arguments for our recursive variable and we know which update affects which inner switch, we can instruct Maude to change these arguments accordingly when a reconfiguration transition is produced. Finally, an extra non-deterministic branch is added for packet forwarding events, which concatenates the flow tables of the inner switches and the `Links` field into a big switch forwarding policy as described in section 3.1.

The final entry point of the SDN model is a concatenation of the big switch variable and the controller variables specified in `Controllers` using parallel composition as in equation (2.4).

The forwarding policies are input to RaceLoom also as a JSON [39] file, which maps every type of harmful race to a NetKAT expression. These expressions support a special placeholder `@Network`, which marks the position where the forwarding policy of the big switch should be placed before evaluating the property.

```
{
  "Switches": {
    "SW1": {
      "InitialFlowTable": "<NetKAT expression>",  // Optional
      "DirectUpdates": [
        {
          "Channel": "<string>",
          "Policy": "<NetKAT expression>",
        }
        ...
      ],
      "RequestedUpdates": [
                    {
          "RequestChannel": "<string>",
          "RequestPolicy": "<NetKAT expression>",
          "ResponseChannel": "<string>",
          "ResponsePolicy": "<NetKAT expression>",
        },
        ...
      ]
    },
    ...
  },
  "Links": "<NetKAT expression>",  // Optional
  "RecursiveVariables": {
    "C1": "<DyNetKAT expression>",
    ...
  },
  "OtherChannels": ["<string>", ...],
  "Controllers": ["C1", ...]
}
```

FIGURE 4.3: JSON structure of SDN model file expected by RaceLoom

4.3 Trace Generation

Now that we have a way to write DyNetKAT SDN models into Maude programs and extract reconfiguration and forwarding transitions from these programs, we can generate symbolic traces.

Intuitively, whenever we rewrite a DyNetKAT expression using Maude, we extract a set of labels corresponding to the next possible transition steps that can be made by the network, but each individual step may result into a different expression. For example, the DyNetKAT expression $(N_1 \,;\, p_1) \parallel (N_2 \,;\, p_2)$ where $N_1, N_2 \in \text{NetKAT}^{-dup}$ and p_1 and p_2 are DyNetKAT policies free of $\parallel$, results in the label set $\{N_1, N_2\}$ of next possible transitions. However, if a transition labeled N_1 is produced, the new expression that has to be rewritten becomes $p_1 \parallel (N_2 \,;\, p_2)$, while a transition labeled N_2 would lead to the expression $(N_1 \,;\, p_1) \parallel p_2$.

Hence, at every depth level, our trace generation algorithm takes a list of DyNetKAT expressions and extracts from each one of them pairs of one transition step and its cor-

responding rewritten DyNetKAT expression. Then the transition steps are stored in a tree-like data structure, which we refer to as the symbolic trace tree, and the process repeats for the next depth level until the maximum depth is reached. As a result, the symbolic trace tree is populated in a breadth-first manner.

Parallel computation Since our algorithm has to perform a rewriting for every expression in the input list, we can divide the input list into multiple sublists and spread the workload between multiple threads/processes to achieve better performance. Unfortunately, we cannot achieve this directly from Python because the package that offers the code bindings for Maude is not thread-safe, so we have to achieve this using Maude.

In Maude, we can achieve multi-threading via meta-interpreters. Implementation-wise, Maude is a program interpreter. A meta-interpreter is simply another instance of a Maude interpreter, which is spawned from the main instance and runs as a separate process. The rewriting system provides a dedicated module to work with meta-interpreters, so most of this functionality is described in its manual. Additionally, we use [35] as a reference for our implementation.

Thus, we achieve parallel computation of our DyNetKAT expression list by implementing a Maude program which can initialize an arbitrary amount of meta-interpreters called workers, which receive a list of DyNetKAT expressions to rewrite and output a list of pairs of one transition step and its corresponding resulting DyNetKAT expression. Then, from the main Python program, we can split the initial list of expressions and prepare the input for Maude, then using the Maude program we spread the input over the available workers, merge the outputs of every worker, and send the result back to the Python program. Note that we also keep track of which transition steps are generated from which DyNetKAT expression via unique identifiers to be able to construct the symbolic trace tree correctly.

Pruning some of the transition steps Recall from chapter 3 that our big switch expression is modeled as a recursive variable that can produce transitions labeled with NetKAT policies, which correspond to packet forwarding events. Since this can happen at any time, it means that the big switch may produce such transitions multiple consecutive times along a trace, all labeled with the same NetKAT policy. However, to detect and reason about the harmfulness of a race condition, it suffices to witness a single packet forwarding transition, any other similar transitions becoming redundant if they appear one after the other.

Thus, we can use this as a heuristic to prune the state space. Whenever we ask Maude to rewrite a DyNetKAT expression, we also pass next to this expression the previous transition label and the identifier of the element that produced it. If the previous transition is a packet forwarding transition, we condition Maude not to extract another such transition if it is produced by the same element.

Adding vector clocks Recall from section 4.1 that each DyNetKAT big switch and controller receive an identifier that is used inside Maude programs to distinguish between the transition steps that each element can produce. The reason for this design choice is to avoid enriching the specifications with vector clocks and slow down Maude. Since these identifiers are stored for every transition in the symbolic trace tree, once the tree construction is finished, we can walk through it and assign vector clocks to each transition step, incrementing them accordingly based on the transition type and the elements that produced it.

4.4 Trace Analysis

The final part of our tool is concerned with analyzing the generated traces based on the three harmful race types defined in chapter 3.

From the symbolic trace tree we can extract an individual trace by choosing a leaf of the tree and walking backwards until we reach the root node, which corresponds to the start state of the SDN model. Once we have a trace, we essentially have a series of events performed by the individual elements of the SDN. To detect concurrency, we traverse the trace, keeping track of the last event of each element, and pairwise compare the vector clocks associated with these events. Whenever incomparable vector clocks are witnessed, we know that their corresponding events run concurrently, so we have to check them against the forwarding properties.

To determine which type of race we are dealing with, we check the conditions of each type of harmful race against the trace and the pair of concurrent transitions. Since the conditions for every race type are different, we either match the transitions to exactly one of them or to none at all. For example, two concurrent transitions match a CT→SW race if the first one corresponds to a packet forwarding event produced by a big switch and the second one to a reconfiguration event from a controller to the same big switch (equation (3.5a)). Additionally, we also have to check if the big switch does not produce any other events in between these transitions (equation (3.5b)).

The harmful race types presented in our symbolic framework assume that the NetKAT expressions of the transition labels encode forwarding policies of big switches. However, remember from section 4.2 that controller expressions produce reconfiguration events labeled with the flow tables of inner switches, and not the forwarding policy of the big switch itself. Thus, since each type of race condition involves a single big switch element, to check the pair of concurrent events against the forwarding properties we have to first reconstruct the big switch policy up to each event in the pair. Because each inner switch reconfiguration replaces the entire flow table of that switch, this simply involves walking through the trace and applying each reconfiguration update that targets the big switch up until the racing events.

With the big switch forwarding policy reconstructed for each concurrent event, we can now evaluate the forwarding property corresponding to the type of race we want to check. We replace the placeholder used in the property expression with each reconstructed policy and decide the equivalence to 0 using KATch. If the results of the two evaluations are not the same, it means the concurrent events define a harmful race condition, so we mark them and store the trace.

Filtering the final traces During the tool's implementation, we observed that the same pair of harmful concurrent events may appear across multiple traces, resulting in duplicate entries in the output. Although these traces differ in structure, the recurrence of the same harmful event pairs can lead to a large number of similar results, making manual inspection difficult. To address this, we filter out duplicates and retain only the trace in which the harmful concurrent events occur earliest, i.e. closest to the beginning of the trace.

Having completed the implementation of RaceLoom and established its core functionality, we now focus on evaluating the tool's practical effectiveness. In the next chapter, we test it on several SDN examples and benchmark its performance using SDNs derived from real-world network topologies.

Chapter 5

Experimental Evaluation

This chapter presents the experimental evaluation of our prototype tool. The goal of this evaluation is to answer the following research questions:

> **RQ1** Is RaceLoom capable of capturing harmful CT→SW, CT→SW←CT, and CT→CT→SW races?
>
> **RQ2** What is the performance of RaceLoom on examples based on real world network topologies of various sizes?

We begin by introducing a set of representative SDN examples designed to highlight the three types of harmful races presented in this paper, namely CT→SW, CT→SW←CT, and CT→CT→SW. These examples serve to validate the correctness of the tool by showcasing its ability to detect harmful races in controlled environments. Following this, we present in section 5.4 a performance evaluation inspired from NetKAT verification tools, such as [36], that measures the scalability and efficiency of RaceLoom on realistic network topologies.

The race detection for the three examples presented in this chapter is performed on a laptop running Linux Mint 21.3 with an Intel Core i7-10750H and 16GB of RAM.

5.1 Example 1: Stateful Firewall

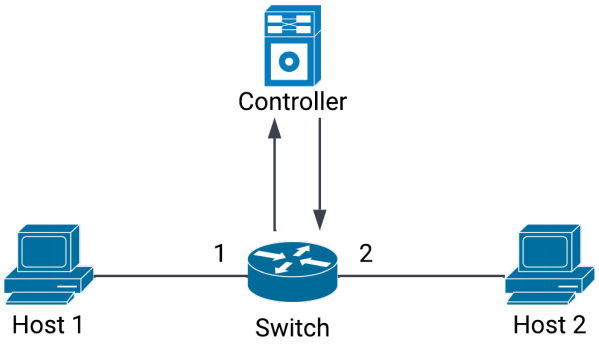


FIGURE 5.1: A network consisting of one switch and one controller connecting two hosts

Our first example is an adaptation of the stateful firewall example from [8], where Host

1 is sending packets to Host 2 over a network consisting of one switch and one controller running a firewall application (figure 5.1).

Initially, the switch forwards all packets flagged *regular*. Packets flagged differently are always sent to the controller, which decides how to update the flow table of the switch depending on the packet flag. For example, when the controller receives packets flagged *blocking-[T]*, it installs a new flow table on the switch that blocks all packets of type T and forwards any other *regular* packets. We distinguish between two types of packets that the firewall should block: SSH and UDP. The DyNetKAT expressions modeling this SDN behavior are as follows:

$$\begin{aligned} N &\triangleq (flag = regular) \cdot (pt = 1) \cdot (pt \leftarrow 2) \\ N_S &\triangleq (flag = regular) \cdot (type \neq SSH) \cdot (pt = 1) \cdot (pt \leftarrow 2) \\ N_U &\triangleq (flag = regular) \cdot (type \neq UDP) \cdot (pt = 1) \cdot (pt \leftarrow 2) \\ N_{SU} &\triangleq (flag = regular) \cdot (type \neq SSH) \cdot (type \neq UDP) \cdot (pt = 1) \cdot (pt \leftarrow 2) \end{aligned} \tag{5.1}$$

$$\begin{aligned} SW_N \triangleq\ & N\,;\, SW_N \\ & \oplus SSH\,!\,(flag = blocking\text{-}SSH)\,;\, SW \\ & \oplus UDP\,!\,(flag = blocking\text{-}UDP)\,;\, SW \\ & \oplus Up\,?\,N_S\,;\, SW_{N_S} \\ & \oplus Up\,?\,N_U\,;\, SW_{N_U} \\ & \oplus Up\,?\,N_{SU}\,;\, SW_{N_{SU}} \end{aligned} \tag{5.2}$$

$$\begin{aligned} C &\triangleq SSH\,?\,(flag = blocking\text{-}SSH)\,;\, Up\,!\,N_S\,;\, CS \\ &\quad \oplus UDP\,?\,(flag = blocking\text{-}UDP)\,;\, Up\,!\,N_U\,;\, CU \\ CS &\triangleq UDP\,?\,(flag = blocking\text{-}UDP)\,;\, Up\,!\,N_{SU}\,;\, \bot \\ CU &\triangleq SSH\,?\,(flag = blocking\text{-}SSH)\,;\, Up\,!\,N_{SU}\,;\, \bot \end{aligned} \tag{5.3}$$

$$SDN \triangleq SW_N \parallel C \tag{5.4}$$

Equation (5.1) encodes the flow tables used by the switch and the controller, where N is the initial forwarding policy of the switch, N_S and N_U are the policies blocking SSH and UDP packets, respectively, and N_{SU} is the policy blocking both types of packets. Equation (5.2) models the behavior of the switch, which can process packets according to a given policy N, send packets flagged *blocking-SSH* or *blocking-UDP* to the controller through channels SSH and UDP, respectively, and receive forwarding policy updates from the controller through channel Up. Finally, equation (5.3) models the behavior of the controller, which can receive packets from the switch through the SSH and UDP channels, and send to the switch new forwarding policies corresponding to the flag of the packets received. Note that the controller is encoded using three terms to account for the different orderings in which the new forwarding policies can be installed. For example, after the initial controller term C installed the policy blocking SSH packets, it changes its behavior to term CS. From then on, the controller waits for packets flagged *blocking-UDP*, then installs policy N_{SU} on the switch, blocking both SSH and UDP packets.

In this example, we consider the network "safe" when no packets of type T are forwarded by the switch after sending a packet flagged *blocking-[T]* to the controller. A forwarding

property encapsulating this condition with respect to SSH packets is:

$$\begin{aligned}\phi^{\alpha_{out}}_{\alpha_{in}}(N) \triangleq\ & (flag = regular) \cdot (type = SSH) \cdot (pt = 1) \\ & \cdot N \\ & \cdot (flag = regular) \cdot (type = SSH)\end{aligned} \tag{5.5}$$

In words, applying the forwarding policy of the switch on a *regular* packet of type SSH that is received on port 1 must not result into an SSH packet, i.e. the received packet is dropped. It may seem that this forwarding policy could raise false positives when the firewall is supposed to allow SSH packets to be forwarded. However, since $\phi^{\alpha_{out}}_{\alpha_{in}}(N)$ is only applied on racing transitions and the controller only installs updates when requested by the switch, the tool cannot report false positives in this case.

Using a trace depth of 10, we feed the DyNetKAT model and the forwarding property to RaceLoom, which finishes the analysis in approx. 2 seconds. The tool identifies 2 traces exhibiting unique CT→SW races violating the forwarding property in equation (5.5). One of these traces is shown in figure 5.2. We observe that our tool identified two transitions that happen concurrently after the switch sent a packet flagged *blocking-SSH* to the controller. The first transition corresponds to a forwarding action performed by the switch, while the second transition corresponds to a policy installation made by the controller. Notice how these transitions cause the vector clocks of the two elements to be incomparable. Because the policies sent by the controller concern only a flow table inside the big switch, RaceLoom has to reconstruct the whole forwarding policy of the network before checking the forwarding property. In this case, a harmful CT→SW race (5) occurs because the network concurrently forwards an SSH packet ($\phi^{\alpha_{out}}_{\alpha_{in}}(N)$ is true) while it receives an update from the controller that prohibits SSH packets to be forwarded ($\phi^{\alpha_{out}}_{\alpha_{in}}(N)$ is false).

Remark: The reconstructed network policies from figure 5.2 have a slightly different shape than the forwarding policy: $((X_1 + \ldots + X_n) \cdot t)^*$ of the big switch presented in section 3.1. This is because the NetKAT operator for repetition $(*)$ can also be taken zero times, dropping all packets and causing the forwarding property to always be true. To fix this, we always force the packet processing policy of the network to be applied at least one time using the NetKAT expression: $((X_1 + \ldots + X_n) \cdot t) \cdot ((X_1 + \ldots + X_n) \cdot t)^*$.

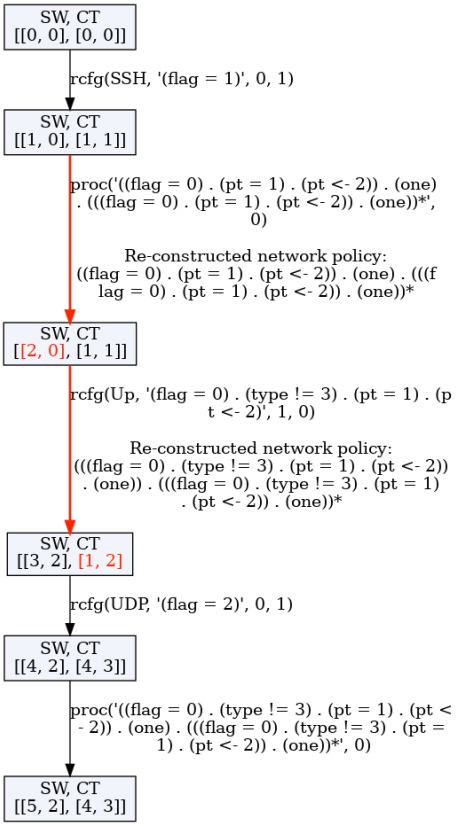


FIGURE 5.2: Trace generated by RaceLoom exhibiting a harmful CT→SW race on the Stateful Firewall example (5.1). Packet flags encoded as 0, 1, and 2 correspond to flags *regular*, *blocking-SSH*, and *blocking-UDP*, respectively. Packet types encoded as 3 and 4 correspond to types SSH and UDP, respectively.

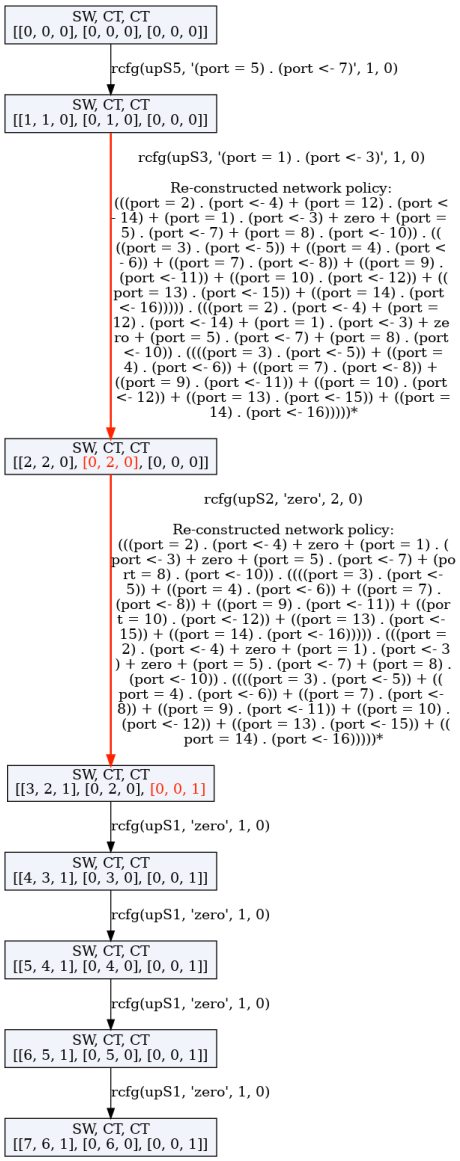


FIGURE 5.3: Trace generated by RaceLoom exhibiting a harmful CT→SW←CT race on the Independent Controllers example (5.2). The top row of each state lists the type of elements in the parallel composition from equation (5.8).

5.2 Example 2: Independent Controllers

The second case we examine is the independent controllers example presented in [9], where two controllers attempt to non-deterministically update a network, leading to a misconfiguration that cause the network traffic to be forwarded to the wrong host.

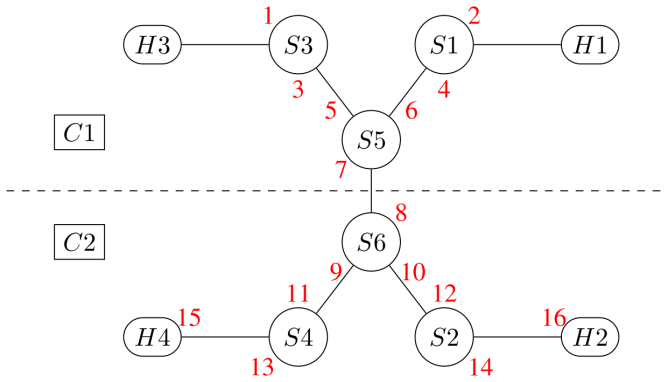


FIGURE 5.4: Network with 6 switches controlled by 2 independent controllers forwarding traffic between 4 hosts [9]

Figure 5.4 depicts the topology of the network. Controller $C1$ is responsible of updating the switches in the top part of the topology, while controller $C2$ updates the switches in the bottom part of the topology. Initially, the network is configured to forward packets from host $H1$ to host $H2$. The controllers are responsible for updating the switches such that the network forwards the packets from host $H3$ to $H4$. The DyNetKAT expressions from [9] modeling this example are as follows:

$$
\begin{aligned}
L \triangleq\ & (((port = 3) \cdot (port \leftarrow 5)) \\
& + ((port = 4) \cdot (port \leftarrow 6)) \\
& + ((port = 7) \cdot (port \leftarrow 8)) \\
& + ((port = 9) \cdot (port \leftarrow 11)) \\
& + ((port = 10) \cdot (port \leftarrow 12)) \\
& + ((port = 13) \cdot (port \leftarrow 15)) \\
& + ((port = 14) \cdot (port \leftarrow 16)))
\end{aligned}
$$

$$
\begin{aligned}
S_1 &\triangleq (port = 2) \cdot (port \leftarrow 4) \\
S_2 &\triangleq (port = 12) \cdot (port \leftarrow 14) \\
S_3 &\triangleq 0 \\
S_4 &\triangleq 0 \\
S_5 &\triangleq (port = 6) \cdot (port \leftarrow 7) \\
S_6 &\triangleq (port = 8) \cdot (port \leftarrow 10)
\end{aligned}
\tag{5.6}
$$

$$
\begin{aligned}
SDN_{X_1,\ldots,X_6} \triangleq\ & (X_1 + \ldots + X_6) \cdot L \cdot ((X_1 + \ldots + X_6) \cdot L)^* \,;\, SDN_{X_1,\ldots,X_6} \\
& \oplus \Sigma_{X_i' \in FT}\ upSi \,?\, X_i' \,;\, SDN_{X_1,\ldots,X_i',\ldots,X_6}
\end{aligned}
$$

$$
\begin{array}{lcl}
ft3 & \triangleq & (port = 1) \cdot (port \leftarrow 3) \\
ft4 & \triangleq & (port = 11) \cdot (port \leftarrow 13) \\
ft5 & \triangleq & (port = 5) \cdot (port \leftarrow 7) \\
ft6 & \triangleq & (port = 8) \cdot (port \leftarrow 9) \\
FT & = & 0, ft3, ft4, ft5, ft6
\end{array}
\tag{5.7}
$$

$$
\begin{array}{lcl}
C_1 & \triangleq & upS1\,!\,0\,;\, C_1 \oplus upS3\,!\,ft3\,;\, C_1 \oplus upS5\,!\,ft5\,;\, C_1 \\
C_2 & \triangleq & upS2\,!\,0\,;\, C_2 \oplus upS4\,!\,ft4\,;\, C_2 \oplus upS6\,!\,ft6\,;\, C_2
\end{array}
$$

$$
SDN \quad \triangleq \quad SDN_{S_1,\ldots,S_6} \parallel C_1 \parallel C_2 \tag{5.8}
$$

Notice that, as remarked in section 5.1, the forwarding policy of the big switch from equation (5.6) forces a packet to make at least one step through the network.

In this example, the network is considered "safe" as long as there are no packets forwarded from $H3$ to $H2$ or from $H1$ to $H4$. Given this constraint, a safe way the controllers can update the network is as follows: (i) S_1 and S_2 are disabled to prevent packets from being forwarded from $H1$ and to prevent $H2$ from receiving packets, (ii) S_5 is updated to forward packets received on port 5 instead of port 6, (iii) S_6 is updated to forward packets to port 9 instead of port 10, and, finally, (iv) S_3 and S_4 are changed to forward packets received on ports 1 and 11, respectively. However, since the controllers are installing new flow tables on the switches non-deterministically, a safe order of the updates cannot be guaranteed, leading to a violation of the forwarding property.

We check whether RaceLoom detects harmful races between the two controllers, which, depending on the update order, violate the forwarding constraint by allowing packets from H3 to reach H2. To achieve this, we use the following forwarding property:

$$
\phi^{\alpha_{out}}_{\alpha_{in}}(N) \quad \triangleq \quad (port = 1) \cdot N \cdot (port = 16) \tag{5.9}
$$

In words, applying the forwarding policy N of the network on a packet received on port 1 must not forward that packet to port 16.

With a trace depth of 7, we run RaceLoom on the DyNetKAT model and the forwarding property from above, which finishes the analysis in approx. 128 seconds. Note that the tool takes longer to finish the analysis because the SDN consists of 3 elements now, and the controllers are modeled using recursive variables, so they perform the same updates multiple times.

The tool identifies 16 traces showcasing harmful CT→SW races, and 8 traces showcasing harmful CT→SW←CT races. One of the latter traces is shown in figure 5.3. We observe that, after C_1 reconfigures S_5 to forward packets to port 7, RaceLoom identifies two transitions that run concurrently. The first transition corresponds to C_1 updating S_3, and the second one corresponds to C_2 updating S_2. In this case, a harmful CT→SW←CT race (6) occurs between the two controllers because the update to S_3 happens first, allowing packets to be forwarded to $H2$ ($\phi^{\alpha_{out}}_{\alpha_{in}}(N)$ is false), before the reconfiguration to S_2 is done, which prevents packets to reach $H2$ ($\phi^{\alpha_{out}}_{\alpha_{in}}(N)$ is true).

5.3 Example 3: 2-Layer Controller Hierarchy

Our final example is inspired from a body of work that focused on improving the efficiency of the SDN control plane by leveraging a hierarchy of controller instances, such as [22], [26], [16]. For example, in [22], the authors propose a 2-layer controller hierarchy for SDNs, where the bottom layer consists of local controllers, which are responsible for installing frequent, local updates on a subset of the network switches, while the top layer consists of a root controller that manages the bottom layer and orchestrates global updates of the network.

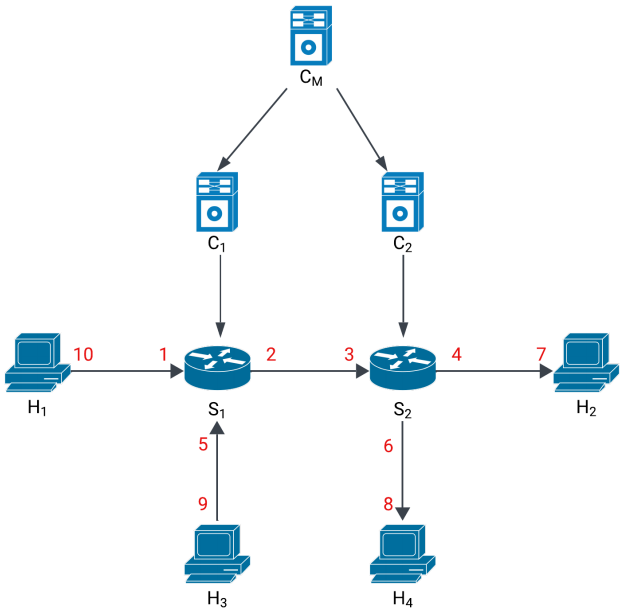


FIGURE 5.5: 4 hosts connected over a network with 2 switches managed by 3 controllers in a master-worker configuration. Controller C_M manages controllers C_1 and C_2, which install updates on switches S_1 and S_2, respectively. The arrows between switches and hosts depicts the traffic flow.

Figure 5.5 depicts the topology of the example network. In this case, we have 4 hosts connected to 2 switches, S_1 and S_2, which are managed by 2 independent controllers, C_1 and C_2, respectively. A master controller C_M is responsible to update C_1 and C_2.

Initially, the flow tables of the switches are empty, causing the network to drop all incoming packets. C_1 and C_2 update the switches to forward packets from host H_1 to host H_2. While the two controllers are busy, the master controller C_M is instructed to allow packets from host H_3 to be forwarded to host H_4, thus, C_M updates the behavior of C_1 and C_2 to install the new routing policy. The DyNetKAT expressions associated with this behavior are as follow:

$$
\begin{aligned}
N_1 &\triangleq (port = 1) \cdot (port \leftarrow 2) \\
N_2 &\triangleq (port = 3) \cdot (port \leftarrow 4) \\
N_3 &\triangleq (port = 5) \cdot (port \leftarrow 2) \\
N_4 &\triangleq (port = 3) \cdot (port \leftarrow 6)
\end{aligned}
$$

$$
\begin{aligned}
L \triangleq\ & (port = 2) \cdot (port \leftarrow 3) \\
& + (port = 6) \cdot (port \leftarrow 8) \\
& + (port = 4) \cdot (port \leftarrow 7)
\end{aligned}
\tag{5.10}
$$

$$
\begin{aligned}
SDN_{X_1,X_2} \triangleq\ & (X_1 + X_2) \cdot L \cdot ((X_1 + X_2) \cdot L)^* \,;\, SDN_{X_1,X_2} \\
& \oplus upS1\,?\,N_1 \,;\, SDN_{N_1,X_2} \\
& \oplus upS2\,?\,N_2 \,;\, SDN_{X_1,N_2} \\
& \oplus upS1\,?\,N_3 \,;\, SDN_{N_3,X_2} \\
& \oplus upS2\,?\,N_4 \,;\, SDN_{X_1,N_4}
\end{aligned}
$$

$$
\begin{aligned}
C_1 &\triangleq upS1\,!\,N_1 \,;\, C_1 \oplus upC1\,?\,N_3 \,;\, upS1\,!\,N_3 \,;\, \bot \\
C_2 &\triangleq upS2\,!\,N_2 \,;\, C_2 \oplus upC2\,?\,N_4 \,;\, upS2\,!\,N_4 \,;\, \bot \\
C_M &\triangleq upC1\,!\,N_3 \,;\, C_M \oplus upC2\,!\,N_4 \,;\, C_M
\end{aligned}
\tag{5.11}
$$

$$
SDN \triangleq SDN_{0,0} \,\|\, C_1 \,\|\, C_2 \,\|\, C_M \tag{5.12}
$$

Similarly to the independent controllers (5.2), we consider the SDN to be "safe" as long as there are no packets forwarded from $H3$ to $H2$ or from $H1$ to $H4$. Since the master controller C_M updates the two local controllers non-deterministically, there are orderings of the updates which violate the forwarding constraint. For example, if C_2 enables packets to be forwarded to H_2, but C_1 is updated by C_M to enable packets to be forwarded from H_3, the network becomes "unsafe".

With this example, we check if RaceLoom is capable of detecting races between the master and the working controllers. To do this, we use the following forwarding property, which ensures that no packets are forwarded from $H3$ to $H2$:

$$
\phi^{\alpha_{out}}_{\alpha_{in}}(N) \triangleq (port = 5) \cdot N \cdot (port = 7) \tag{5.13}
$$

In words, applying the network forwarding policy N to a packet received on port 5 of S_1 is not forwarded to port 7 of H_2.

With a depth setting of 10, RaceLoom analyzes the network model using the forwarding property from above in approximately 35 seconds. The tool identifies a total of 8 traces exhibiting unique harmful races, out of which: 4 traces showcase CT→SW races, 3 showcase CT→SW←CT races, and 1 showcase a CT→CT→SW race. The latter trace is shown in figure 5.6. It can be observed that after CT_2 updates S_2 to forward packets from port 3 to port 4, RaceLoom identifies 2 transitions that run concurrently. The first one corresponds to C_1 updating S_1 to forward packets received on port 1, while the second transition corresponds to C_M modifying CT_1 to instruct the switch to forward packets received on port 5. In this case, a harmful CT→CT→SW race occurs because applying the update of the first transition does not allow packets from H_3 to reach H_2 ($\phi^{\alpha_{out}}_{\alpha_{in}}(N)$ is true), but applying the new update from C_M violates the forwarding constraint ($\phi^{\alpha_{out}}_{\alpha_{in}}(N)$ is false).

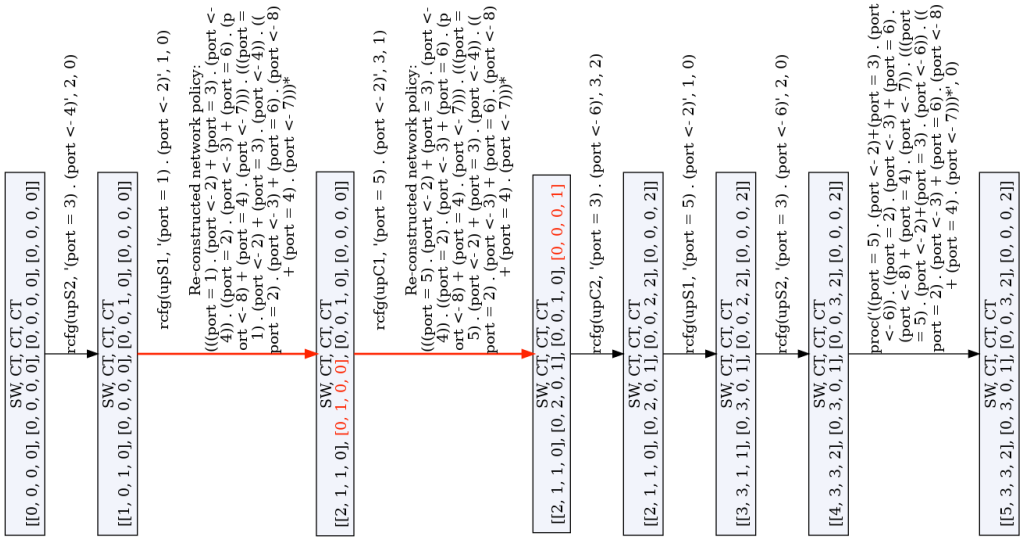


FIGURE 5.6: Trace generated by RaceLoom exhibiting a harmful CT→CT→SW race on the 2-Layer Controller Hierarchy example (5.3). The top row of each state lists the type of elements in the parallel composition from equation (5.12).

5.4 Performance Evaluation

The performance evaluation of RaceLoom follows a methodology inspired by the evaluation strategies employed in NetKAT verification tools, such as KATch [36]. This approach utilizes the Topology Zoo [25] dataset, which comprises 261 real-world network topologies, varying in size from 4 to 754 switches. For each topology, a destination-based routing scenario is constructed by randomly distributing multiple hosts throughout the network. NetKAT policies are then generated to emulate packet forwarding behavior along the shortest paths between all pairs of hosts.

If we consider a connection between 2 hosts as a single problem, we observe that the performance evaluation of [36] solves multiple such problems in one execution. For our evaluation approach, we not only have to emulate the data plane of the network, but also the control plane. Thus we decide to only focus on one problem of host connectivity at a time. Due to this choice, we cannot place the hosts at random through the network anymore because the shortest path between these hosts may not scale appropriately with the size of the network. Hence, we explicitly connect these hosts to the switches found at the ends of the diameter of the network, which represents the longest shortest-path possible for the given network topology.

Concretely, for each connected topology in the Topology Zoo, we build our test case as follows:

1. Identify the diameter of the topology and connect one host to each end
2. Assign ports to every switch and link in the topology
3. Generate NetKAT policies for every switch to forward packets in one direction between the 2 hosts
4. Construct a DyNetKAT big switch expression (as in 3.1) combining the behavior of all switches in the network. Initially, this expression either drops all packets or

accepts updates from controllers, thereby representing the DyNetKAT encoding of the network's data plane.

5. Generate a DyNetKAT expression representing a controller, which *sequentially* issues updates to the data plane. Each update corresponds to the NetKAT policy generated in step 3 for an individual switch of the network.

6. Concatenate the DyNetKAT expressions into one final model of the network using parallel composition (as in 2.4).

With this approach, we evaluate our tool on 2 scenarios:

1. One controller that establishes the connection of the two hosts in only one direction.

2. Two controllers independently updating the network data plane, where one controller establishes the connection of the two hosts in one direction, and the other in the opposite direction.

For both of the experimental scenarios we instruct the tool to generate traces up to a depth of 10 using a total of 75 threads. The forwarding property used to check if the races are harmful is: $\phi_{\alpha_{in}}^{\alpha_{out}}(N) \triangleq N \cdot (dst = 999)$, which always evaluates to `true` according to 4 because the host destination 999 does not exist. If $\phi_{\alpha_{in}}^{\alpha_{out}}(N)$ always evaluates to `true`, it means that, according to the definitions 5, 6, and 7, none of the races will be marked as harmful by RaceLoom. This forces the tool to inspect all possible races within a trace, which better reflects its performance. Note that we still use policy N in our forwarding property, which corresponds to the encoding of the data plane forwarding behavior. This ensures that when $\phi_{\alpha_{in}}^{\alpha_{out}}(N)$ is evaluated, the result is not trivial to compute (unlike other possible forwarding properties such as $\phi_{\alpha_{in}}^{\alpha_{out}}(N) \triangleq 0$).

Results

The performance evaluation was conducted in a cloud environment running Ubuntu 22.04.3 LTS. The hardware used for the experiments involved a Dell PowerEdge R7615 processor with 64 cores at 3.675 GHz and 1TB of RAM. In figure 5.8 and figure 5.9 we report the execution time of the three main technologies employed in our tool, namely Python, Maude [11], and KATch [36].

The results for scenario 1 (figure 5.8) show that our prototype tool can handle all generated DyNetKAT network models in less than 16 seconds. Despite caching the results, the main bottleneck in this scenario is the evaluation of the forwarding property using KATch. While this NetKAT evaluation tool is the fastest one available [36] at the time of writing this paper, the multitude of external calls and the overhead added to initiate every call and retrieve the result still impacts significantly the overall performance of our tool. Apart from this bottleneck, all other operations performed by RaceLoom, such as building the input network model, generating traces, or identifying possible races, are consistently performed in approximately 2 seconds for all network topologies except the last one. The last topology is also the one with the longest diameter (59), thus, our tool also takes longer to finish (approx. 3 seconds).

The amount of KATch calls made for every network diameter is also shown in figure 5.7. It can be observed that since we set the depth of the traces to 10, the networks with a diameter smaller than 9 require less KATch calls because the controller finishes installing the policies in less than 10 steps. For diameters higher than 9, the amount of calls does not increase anymore because all traces end after fewer steps than required by the controller

to finish installing all policies. Whenever the diameter length of the network increases by one, the amount of calls made to KATch also increases, and this is reflected in the execution time shown in figure 5.8. Eventually, the amount of calls reaches a maximum of 18, but even when RaceLoom has to call KATch only 5 times for the smallest network model, KATch still takes around 4 seconds to complete the calls, which is double the time that it takes for all other operations to finish (approx. 2 seconds).

For scenario 2, the results presented in figure 5.9 are drastically different than the ones for scenario 1. In this case, our tool could not finish executing all the generated network models in a reasonable amount of time. In figure 5.9 we present the execution time of our tool on one network model for each diameter length available up to 10 switches. It can be observed that for networks with a diameter length less than 9, our tool can still finish the analysis in less than 2 minutes. For networks with diameter lengths of 9 and 10 switches, the execution time of RaceLoom grows substantially to around 8 and 42 minutes, respectively. If for networks with diameters up to 8 the main bottleneck of our tool is still the NetKAT evaluation tool, for networks with a diameter of 8 and longer the main bottleneck becomes the trace generation step that leverages Maude [11]. Although we only employ Maude to re-write the symbolic configuration of the network (i.e. $S_{\vec{c_1}} \parallel C_{1\vec{c_1}} \parallel C_{2\vec{c_2}}$) into head-normal form, the tool starts to significantly struggle with the re-writing as the diameter increases. Whenever the diameter increases by one, the number of non-deterministic branches of the big switch increases by two. Since the symbolic configurations now contain three DyNetKAT elements in parallel, the more non-deterministic branches $S_{\vec{c_1}}$ has, the more re-writes Maude has to perform to extract all possible actions that the big switch and the two controllers can execute at a given point.

Despite the slow execution time of RaceLoom for networks with diameters larger than 9, it has to be noted that 50% of the network topologies in the Topology Zoo [25] have a diameter length ≤ 8, while 75% of the dataset consists of networks with diameter length ≤ 10. Based on these statistics, we conclude that RaceLoom is able to analyze a good majority of real world network topologies in less than 45 minutes.

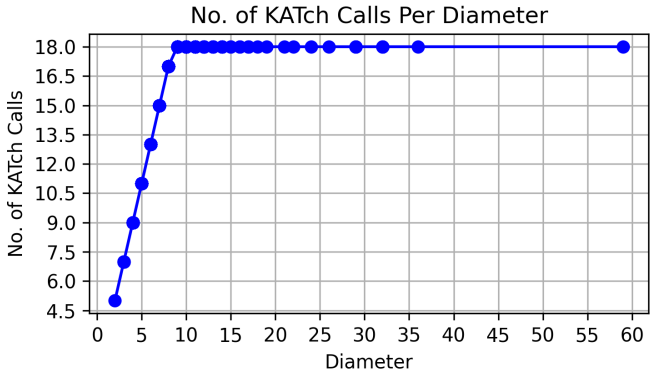


FIGURE 5.7: Scenario 1: Number of KATch calls made per network diameter.

After demonstrating RaceLoom's ability to detect harmful race hazards and benchmarking its performance on real-world network topologies, we now conclude the paper by summarizing our contributions, discussing key limitations, and highlighting possible directions for future research.

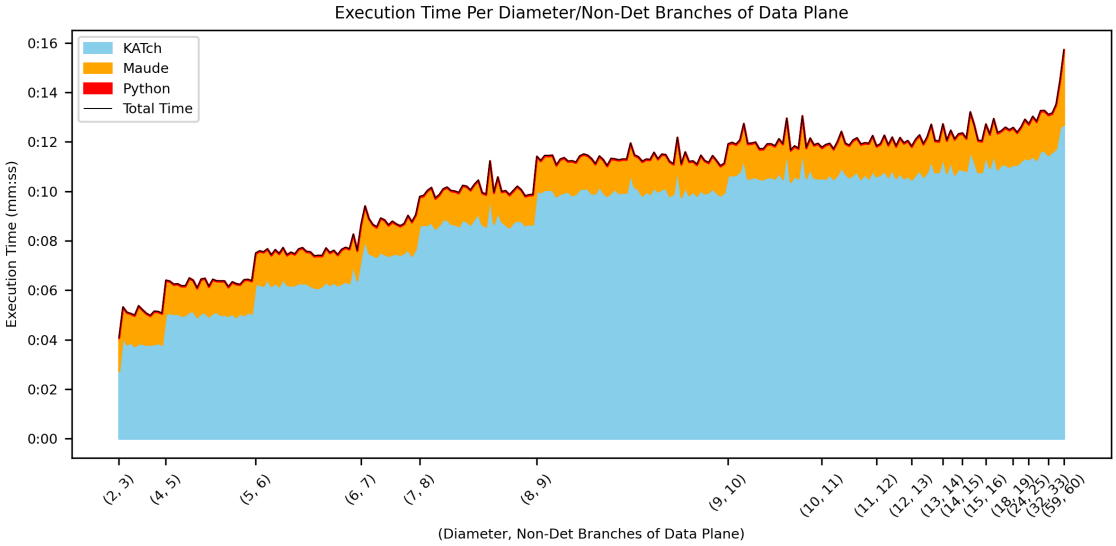
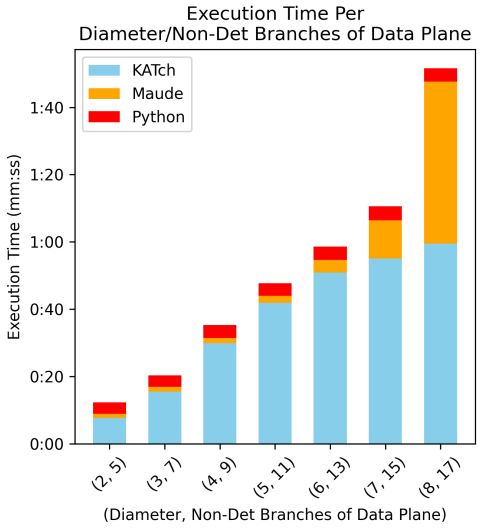
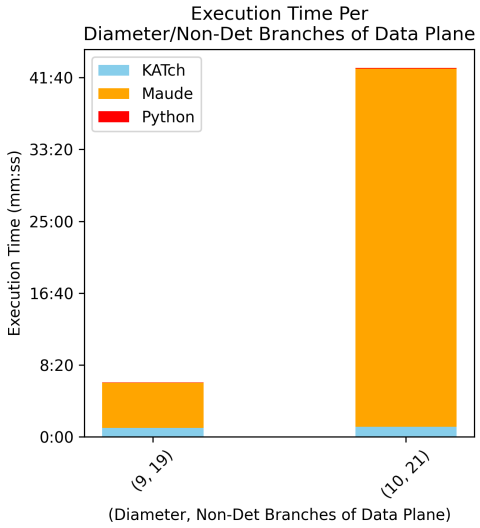


FIGURE 5.8: Scenario 1: Execution time of RaceLoom per network diameter length.



(A) Execution time for networks with a diameter up to 8



(B) Execution time for networks with diameter 9 and 10

FIGURE 5.9: Scenario 2: Execution time of RaceLoom per network diameter length.

Chapter 6

Conclusion and Remarks

In this chapter, we reflect on the results of our implementation evaluation and address the research questions posed in chapter 1. We summarize key findings, evaluate the strengths and weaknesses of our approach, and highlight technical challenges uncovered in the process.

6.1 Harmful Race Condition Detection in SDNs (RQ1, RQ4)

Software-defined networking is a new approach to computer network design that brings many advantages over traditional methods, such as flexibility and improved network programmability [28]. Despite this, they also bring many challenges and bugs, some unique to the networking paradigm and others inherited from existing computer science fields, most notably distributed systems [6].

One major challenge faced by SDNs is concurrency, in particular race conditions that occur either locally, at a specific network component, or between events and messages exchanged between these components. They can have serious consequences on the network itself and on the hosts connected to the network [53] [17], so an important step to prevent them is to catch and understand when and how these race conditions occur. Although many SDN verification tools exist, not all consider the control and the data plane of the SDN together, and only a few are directly targeted at race hazards. Among the ones that specifically perform race condition detection, various formal methods are employed, including happens-before models, satisfiability checking using SMT solvers, or model checking [30]. However, very few of these tools exploit symbolic execution, which allows the analysis of SDNs statically, without feeding actual packets into the network.

Tracer [54] is one such tool that leverages the advantage of symbolic execution to detect race conditions in SDNs using DyNetKAT, a domain-specific language for encoding SDNs with a complete and sound equational theory. Tracer makes use of the symbolic semantics of DyNetKAT based on vector clocks [29] to statically detect concurrent events that occur between SDN elements. Despite the novelty of the approach, we identify and address a series of shortcomings that hinder the practical performance of this methodology in response to **RQ4**. Particularly, we remark that:

1. the detection focuses on identifying concurrent events without addressing whether they are harmful for the network, which leads to a great amount of false positive, as shown by other tools such as SDNRacer [17]

2. the DyNetKAT symbolic semantics focus on symbolic packets derived from NetKAT expressions, significantly increasing the amount of traces that have to be generated and analyzed
3. there is a lack of abstractions for the SDN data plane, forcing all switches of the network to be encoded separately, which not only increases the amount of false positives output by the tool, but also hinders its scalability

In chapter 3, we enhance the symbolic execution framework from [8] to address these shortcomings by:

1. formulating three general classes of race conditions that can occur in SDNs as observed in related literature
2. optimizing the symbolic semantics of DyNetKAT to use NetKAT expressions encoding entire flow tables instead of symbolic packets
3. adapting the "one big switch" abstraction introduced in [24] for DyNetKAT to improve the performance of the methodology in practice
4. improving the DyNetKAT modeling of SDNs through a switch-controller communication mechanism inspired from the OpenFlow protocol [34]

Our work confirms that the symbolic execution approach via DyNetKAT is a promising, but underutilized approach for modeling and reasoning about harmful concurrent behaviors between the control and data plane of SDNs. Based on our improved DyNetKAT symbolic execution framework, we are able to construct symbolic traces, identify concurrent events through incomparable vector clocks, and reason about their negative impact on SDNs with respect to arbitrary forwarding properties, effectively addressing **RQ1**.

6.2 Reimplementation of Tracer (RQ3 a)

While Tracer provides a sound theoretical foundation, its implementation proved difficult to extend and benchmark due to the lack of documentation and slow execution times. To address this, we reimplemented the tool as *RaceLoom*, maintaining the DyNetKAT foundation but significantly improving the architecture.

RaceLoom uses Python and Maude [11] as the main technologies to extract network events from SDNs modeled with DyNetKAT, especially because they were successfully used before in another DyNetKAT based verification tool [9]. However, we remove some of the friction created by the inter-process communication between Python and Maude through a library of code bindings presented in [43], which allows us to execute Maude operations directly from our Python code base. For harmfulness checking, we integrate KATch [36], a state-of-the-art NetKAT verification tool. Together, these design choices lead to a more maintainable and testable framework that enables experimentation with reasonably sized SDN models. In chapter 5, we showcase RaceLoom's capability of detecting harmful SDN race conditions through three illustrative examples.

6.3 Performance Benchmarking (RQ2, RQ3 b)

To evaluate the practical limits of RaceLoom, we adapt an existing benchmarking methodology from NetKAT verification tools to the DyNetKAT setting. Using SDN models gen-

erated from the Topology Zoo [25], a data set of real world network topologies, we analyze the time required to verify race conditions across networks of varying size and complexity.

Our results show that RaceLoom can handle single-controller networks of arbitrary size within 16 seconds. However, verification time increases significantly for multi-controller networks, with two-controller models taking up to 45 minutes. We observe that the main performance bottlenecks vary by scenario: forwarding property checking dominates the runtime in single-controller models, while symbolic trace generation (via Maude) becomes the limiting factor in multi-controller ones.

Despite applying optimizations such as data plane abstraction, parallel trace generation, and heuristic trace pruning, scalability remains limited for SDN models that use multiple controllers or a large number of non-deterministic branches. This highlights ongoing challenges in symbolic analysis for distributed, concurrent networks.

Nonetheless, our work contributes with one of the first benchmarking methodology designed for DyNetKAT and with a performance baseline for future extensions and race condition detection tools that leverage this domain-specific language.

6.4 Limitations

While our approach advances the detection of harmful race hazards in SDNs, several limitations remain that affect the generality, scalability, and maturity of our proposed solution.

Scalability. Despite applying optimizations such as the "one big switch" abstraction [24] and parallel trace generation, our tool performance degrades rapidly on multi-controller networks, according to our benchmark. This is primarily due to the slow rewriting of the DyNetKAT SDN models using Maude, which is significantly influenced by the number of elements used by the model and their amount of non-deterministic branches. The discrepancy of the results between the two experimental scenarios indicate that Maude eventually reaches very large DyNetKAT expressions which are hard to handle. Additionally, the trace depth also plays an important factor because the state space and, implicitly, the number of traces grow exponentially as the depth increases. Hence, employing a more strategic approach to apply the DyNetKAT axioms on the SDN models or finding more heuristic to prune the state space could significantly improve RaceLoom's performance.

Tooling. The initial Tracer implementation lacked robustness and extensibility, and while RaceLoom improves on this, many improvements can still be made. In particular, better error handling, input format, and testing have the highest priority. Currently, no parser exists for DyNetKAT and the time constraints did not allow the implementation of one, so the user is forced to input their SDN model to RaceLoom as a JSON file with a rigid structure. More intricate testing examples and edge cases could also be added to the test suite of the tool to improve its reliability. Finally, the tool could handle KATch and Maude errors much better, as these can currently affect the final result or crash the tool during the analysis.

Modeling Assumptions. The SDN models used by our DyNetKAT symbolic framework rely on a data plane abstraction [24] and a communication mechanism inspired from OpenFlow [34]. This makes the framework less generalizable to SDNs that use different protocols or switch behaviors, and may oversimplify certain scenarios.

Expressiveness of Forwarding Properties. The definition of harmfulness depends on forwarding properties expressed in NetKAT that specify what packets can be forwarded by the network from an *ingress* to an *egress*. While this enables us to reason about what harmful concurrent events mean in a given SDN context, it greatly limits the type of properties that we can check and the amount of harmful race conditions that can be detected by our framework.

These limitations highlight areas for future research and implementation improvements, including more efficient handling of large DyNetKAT expressions in Maude, or supporting other types of properties when checking the harmfulness of SDN race conditions.

6.5 Conclusion

This work has addressed the problem of detecting harmful race conditions in Software-Defined Networks using symbolic execution in DyNetKAT. Motivated by the limitations of Tracer [54], which lacked precision, scalability, and practical benchmarking, we proposed a refined verification approach that improves both the symbolic semantics of DyNetKAT and the implementation of harmful race hazards detection.

We began by optimizing the symbolic semantics of DyNetKAT, replacing the symbolic packets with NetKAT-based flow-table abstractions and adapting the "one big switch" model [24] of the SDN data plane for DyNetKAT. Then, we formally characterized three general classes of harmful race hazards in SDNs, inspired by control and data plane interactions observed in the SDN literature. These improvements allow the race condition detection not only to identify concurrent events through incomparable vector clocks, but also to evaluate their harmfulness with respect to forwarding properties.

To support experimental validation, we introduced RaceLoom, a new implementation of Tracer built with Python, Maude [11], and KATch [36]. This new architecture reduces overhead and enables symbolic trace generation, concurrent events detection, and forwarding property checks of these concurrent events in a unified pipeline. Our benchmarks, derived from real-world topologies in the Topology Zoo dataset [25], show that RaceLoom can effectively detect and explain harmful race hazards in SDNs of moderate size. We observed strong performance in single-controller networks, while scalability remains limited for highly non-deterministic or multi-controller DyNetKAT networks.

Nonetheless, our results show that static analysis based on DyNetKAT symbolic execution is a viable approach for harmful race condition detection in SDNs, especially when enhanced with data plane abstractions and property-aware reasoning. While scalability remains the biggest challenge, our methods significantly reduce the gap between theoretical precision and practical applicability.

6.6 Future Work

While this work advances the detection of race hazards in SDNs, several directions remain open for future exploration.

First, scalability remains a key challenge. While we introduced optimizations, abstractions, caching, and parallel trace generation, performance degrades with increasing model complexity. Future work could explore a more refined approach to rewrite SDN models encoded in DyNetKAT using the axiom system of the language and new pruning strategies. The

former is especially important as it was observed during the evaluation of RaceLoom that Maude spends a long time extracting the possible transitions that the SDN can produce from its initial state. A promising solution is to use Maude's system modules to implement the axiom system of DyNetKAT. This approach would allow specifying rewriting strategies in Maude [11], thereby controlling the order in which the DyNetKAT axioms are applied during rewriting. In this way, the axioms that quickly increase the size of the SDN expression, such as axiom A8 from figure 3.5, can be applied last, reducing the rewriting time.

A pruning strategy similar to the one described in section 4.3 can also be applied to reconfiguration transitions. Specifically, consecutive reconfigurations with the same label, source element, and destination element can be excluded from symbolic traces, thereby pruning additional branches of the trace tree.

However, this approach has the drawback of potentially omitting valid execution paths, depending on the behavior of the controller expressions. In contrast to the "big switch" expressions, which have a fixed structure, controller expressions offer greater flexibility and may intentionally include such repeated reconfigurations. While the SDN encoding can be slightly adjusted to mitigate this limitation, it is important to recognize and account for its impact on trace completeness.

An additional optimization for pruning the symbolic trace tree involves integrating the harmful race condition detection directly into the trace generation process. RaceLoom reports only the first harmful race condition encountered along a trace, as the network execution is considered compromised beyond that point, rendering further analysis along the same path unnecessary.

By performing detection and analysis during trace generation, harmful race conditions that occur early, i.e. near the root of the tree, can be identified sooner. This allows for early termination of the corresponding branch, effectively reducing the number of traces that must be explored.

Second, RaceLoom is designed to detect harmful races with respect to forwarding properties specified in NetKAT. This involves verifying whether race hazards in SDN programs violate given reachability queries. A promising direction for future work is to extend the expressiveness of the supported properties beyond simple reachability. For instance, even without extending NetKAT, RaceLoom could be adapted to support additional classes of properties such as *waypointing*, which ensures that network traffic passes through a designated device (a waypoint); *isolation*, which guarantees that traffic between distinct parts of the network remains separated; and *equivalence checking*, which verifies that the forwarding behavior of the data plane matches a reference NetKAT expression.

More expressive properties, such as those expressed in Linear Temporal Logic, could be supported using Temporal NetKAT [5]. However, reasoning about such temporal properties requires extending the NetKAT language itself. As a result, enabling this level of expressiveness may necessitate corresponding extensions to DyNetKAT, which could be a non-trivial and technically challenging task.

Finally, from a practical standpoint, the current input and output formats of RaceLoom introduce significant friction, which may hinder its adoption in real-world settings. One way to reduce this barrier is to extend RaceLoom with support for automatic extraction of SDN behavior from controller log files. This would involve scanning the logs and generating DyNetKAT expressions based on recorded network events, thereby reconstructing partial

models of the SDN.

FPSDN [21] is an existing tool that performs such automated behavior extraction from OpenFlow log files using DyNetKAT. It captures the network behavior based on *Packet-In*, *Packet-Out*, and *Flow Mod* events, and subsequently verifies reachability properties to detect potential faults in the inferred models. Our symbolic framework relies on the same categories of events for constructing the "big switch" abstraction, which suggests that a similar approach could be employed to enhance RaceLoom. Enabling RaceLoom to accept OpenFlow log files, which are much more common in deployed environments, could significantly enhance its practical applicability.

The primary limitation of this approach, however, is that the SDN must be either executed or simulated in order to produce the necessary log files. As a result, the verification process would become dynamic in nature, similar to SDN-Predict [31]. Nonetheless, this enhancement would allow RaceLoom to detect harmful race conditions in real, running SDNs and facilitate evaluation using benchmarks that more closely reflect operational network deployments.

Overall, this work opens the door to building scalable, semantically grounded tools for reasoning about concurrency in SDNs, an area with many open challenges and practical importance.

Bibliography

[1] Ehab Al-Shaer and Saeed Al-Haj. FlowChecker: configuration analysis and verification of federated OpenFlow infrastructures. In *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration*, SafeConfig '10, page 37–44, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1866898.1866905.

[2] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 113–126, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2535838.2535862.

[3] 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, June 2014. doi:10.1145/2666356.2594317.

[4] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018. doi:10.1007/978-3-319-10575-8_11.

[5] Ryan Beckett, Michael Greenberg, and David Walker. Temporal netkat. *SIGPLAN Not.*, 51(6):386–401, June 2016. doi:10.1145/2980983.2908108.

[6] Ayush Bhardwaj, Zhenyu Zhou, and Theophilus A. Benson. A Comprehensive Study of Bugs in Software Defined Networks. In *Proceedings - 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021*, pages 101–115. Institute of Electrical and Electronics Engineers Inc., 6 2021. doi:10.1109/DSN48987.2021.00026.

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014. doi:10.1145/2656877.2656890.

[8] Georgiana Caltais, Hossein Hojjat, and Nate Foster. Symbolic Race Identification in DyNetKAT, 2024. URL: https://drive.google.com/file/d/1UdpsgqhoRTzNTKKi6RJeLCZTLuobVwst/view.

[9] Georgiana Caltais, Hossein Hojjat, Mohammad Mousavi, and Hunkar Can Tunc. DyNetKAT: An Algebra of Dynamic Networks, 2021. URL: https://arxiv.org/abs/2102.10035, arXiv:2102.10035.

[10] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE way to test OpenFlow applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 10, USA, 2012. USENIX Association.

[11] Manuel Clavel, Francisco Durán, Carolyn Talcott, José Meseguer, Narciso Martí-Oliet, Patrick Lincoln, and Steven Eker. *All About Maude– A High-Performance Logical Framework*. Springer-Verlag Berlin Heidelberg, 2007.

[12] Andrei Covaci. RaceLoom: A tool for automated symbolic detection of race conditions in software-defined networks. https://github.com/andreioff/RaceLoom, 2025. Accessed: 2025-06-20.

[13] Godson Dawuni. *Comparision of three OpenFlow SDN controllers - OpenDaylight, Floodlight, and HPE VAN*. PhD thesis, University of Ghana, October 2020. doi:10.13140/RG.2.2.25124.35202.

[14] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975. doi:10.1145/360933.360975.

[15] David L. Dill. *A Retrospective on Murφ*, pages 77–88. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-69850-0_5.

[16] Advait Dixi, Fang Hao, Sarit Mukherjee, T. V. Lakshman, and Ramana Rao Kompella. ElastiCon: An elastic distributed SDN controller. In *ANCS 2014 - 10th 2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 17–27. Association for Computing Machinery, 10 2014. doi:10.1145/2658260.2658261.

[17] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin Vechev. SDNRacer: Concurrency analysis for Software-Defined Networks. *ACM SIGPLAN Notices*, 51:402–415, 6 2016. doi:10.1145/2908080.2908124.

[18] Danial Entezari. Comparative Analysis of Dynamic Data Race Detection Techniques, 6 2022. doi:10.48550/arXiv.2206.10338.

[19] Michael D. Ernst. Invited talk static and dynamic analysis: synergy and duality. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '04, page 35, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/996821.996823.

[20] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: a network programming language. *SIGPLAN Not.*, 46(9):279–291, September 2011. doi:10.1145/2034574.2034812.

[21] Mohammadreza Ghobakhlou. FPSDN: Fault Prediction in Software-Defined Networks(SDNs). https://github.com/mghobakhlou/FPSDN/tree/9283be8ba3b06a14fd0e04b3b2a59ce043a70e8c, 2025. Accessed: 2025-07-18.

[22] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, page 19–24, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2342441.2342446.

[23] Gerard Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 01 2004.

[24] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the "One Big Switch" Abstraction in Software-Defined Networks. In Kevin C. Almeroth, Laurent Mathy, Konstantina Papagiannaki, and Vishal Misra, editors, *Conference on emerging Networking Experiments and Technologies, CoNEXT '13, Santa Barbara, CA, USA, December 9-12, 2013*, pages 13–24. ACM, 2013. doi:10.1145/2535372.2535373.

[25] Simon Knight, Hung X. Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications*, 29:1765–1775, 10 2011. doi:10.1109/JSAC.2011.111002.

[26] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: a distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 351–364, USA, 2010. USENIX Association.

[27] Dexter Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, May 1997. doi:10.1145/256167.256195.

[28] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, 103(1):14–76, 2015. doi:10.1109/JPROC.2014.2371999.

[29] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. doi:10.1145/359545.359563.

[30] Leticia Lavado, Laura Panizo, María del Mar Gallardo, and Pedro Merino. A characterisation of verification tools for software defined networks. *Journal of Reliable Intelligent Environments*, 3:189–207, 9 2017. doi:10.1007/s40860-017-0045-y.

[31] Gongzheng Lu, Lei Xu, Yibiao Yang, and Baowen Xu. Predictive analysis for race detection in software-defined networks. *Science China Information Sciences*, 62, 6 2019. doi:10.1007/s11432-018-9826-x.

[32] Rupak Majumdar, Sai Deep Tetali, and Zilong Wang. Kuai: A model checker for Software-Defined Networks. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*, pages 163–170, 2014. doi:10.1109/FMCAD.2014.6987609.

[33] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Černý. Event-driven network programming. *ACM SIGPLAN Notices*, 51:369–385, 6 2016. doi:10.1145/2908080.2908097.

[34] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008. doi:10.1145/1355734.1355746.

[35] Canh Minh Do, Adrián Riesco, Santiago Escobar, and Kazuhiro Ogata. Parallel Maude-NPA for Cryptographic Protocol Analysis. In Kyungmin Bae, editor, *Rewrit-*

ing Logic and Its Applications, pages 253–273, Cham, 2022. Springer International Publishing.

[36] Mark Moeller, Jules Jacobs, Olivier Savary Belanger, David Darais, Cole Schlesinger, Steffen Smolka, Nate Foster, and Alexandra Silva. KATch: A Fast Symbolic Verifier for NetKAT. *Proceedings of the ACM on Programming Languages*, 8, 6 2024. doi:10.1145/3656454.

[37] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 217–230, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2103656.2103685.

[38] João Carlos Pereira. Testing for Race Conditions in Distributed Systems via SMT Solving, 6 2020. doi:10.1007/978-3-030-50995-8_7.

[39] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of JSON schema. In *Proceedings of the 25th International Conference on World Wide Web*, pages 263–273. International World Wide Web Conferences Steering Committee, 2016.

[40] Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, pages 151–166, 2013. doi:10.1145/2509136.2509538.

[41] John Regehr. Race condition vs. data race, Mar 2011. URL: https://blog.regehr.org/archives/490.

[42] Joshua Reich, C. Monsanto, Nate Foster, Jennifer Rexford, and D. Walker. Modular SDN programming with Pyretic. *USENIX Login*, 38:128–134, 01 2013.

[43] Rubén Rubio. Maude as a Library: An Efficient All-Purpose Programming Interface. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 13252 LNCS, pages 274–294. Springer Science and Business Media Deutschland GmbH, 2022. doi:10.1007/978-3-031-12441-9_14.

[44] C.L. Schuba, I.V. Krsul, M.G. Kuhn, E.H. Spafford, A. Sundaram, and D. Zamboni. Analysis of a denial of service attack on TCP. In *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No.97CB36097)*, pages 208–223, 1997. doi:10.1109/SECPRI.1997.601338.

[45] Divjyot Sethi, Srinivas Narayana, and Sharad Malik. Abstractions for model checking SDN controllers. In *2013 Formal Methods in Computer-Aided Design*, pages 145–148, 2013. doi:10.1109/FMCAD.2013.6679403.

[46] Steffen Juilf Smolka. *A (Co)algebraic Approach to Programming and Verifying Computer Networks*. PhD thesis, Cornell University, 2019.

[47] Jungmin Son and Rajkumar Buyya. A Taxonomy of Software-Defined Networking (SDN)-Enabled Cloud Computing. *ACM Comput. Surv.*, 51(3), May 2018. doi:10.1145/3190617.

[48] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gün Sirer, and Nate Foster. Merlin: A Language for Managing Network Resources. *IEEE/ACM Transactions on Networking*, 26(5):2188–2201, 2018. doi:10.1109/TNET.2018.2867239.

[49] Xiaoye Steven Sun, Apoorv Agarwal, and T. S.Eugene Ng. Controlling Race Conditions in OpenFlow to Accelerate Application Verification and Packet Forwarding. *IEEE Transactions on Network and Service Management*, 12:263–277, 6 2015. doi:10.1109/TNSM.2015.2419975.

[50] Martinus Richardus van Steen and Andrew S. Tanenbaum. *Distributed Systems*. Self-published, open publication, 3rd edition, February 2017.

[51] Evgenii Vinarskii, Jorge López, Natalia Kushik, Nina Yevtushenko, and Djamal Zeghlache. A Model Checking Based Approach for Detecting SDN Races. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11812 LNCS, pages 194–211. Springer, 2019. doi:10.1007/978-3-030-31280-0_12.

[52] Jana Wagemaker, Nate Foster, Tobias Kappé, Dexter Kozen, Jurriaan Rot, and Alexandra Silva. *Concurrent NetKAT: Modeling and analyzing stateful, concurrent networks*, page 575–602. Springer International Publishing, 2022. URL: http://dx.doi.org/10.1007/978-3-030-99336-8_21, doi:10.1007/978-3-030-99336-8_21.

[53] Lei Xu, Jeff Huang, Sungmin Hong, Jialong Zhang, and Guofei Gu. Attacking the brain: races in the SDN control plane. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 451–468, USA, 2017. USENIX Association.

[54] Ervin Zvirbulis. Automated detection of race conditions in DyNetKAT. Bachelor's thesis, University of Twente, 2024.