# TOWARD PREDICTABLE CONTROL OF SOFTWARE-DEFINED NETWORKS

BY

JASON CROFT

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Associate Professor Matthew Caesar, Chair and Director of Research
Associate Professor P. Brighten Godfrey
Associate Professor Madhusudan Parthasarathy
Dr. Ratul Mahajan, Intentionet

# Abstract

Traditional computer networks require manual configuration of potentially hundreds of forwarding devices. To ease this configuration burden, software-defined networking (SDN) provides centralized, programmatic control of switch data planes in the form of applications, or *control modules*. This modular design simplifies development, and allows network operators to leverage a potentially vast library of open source SDN applications.

However, configuring an SDN is still prone to misconfigurations, software bugs, and unexpected behavior. An operator must reason about the configuration on at least three planes in the network: the management plane, the control plane, and the data plane. At the control plane, an operator must ensure the applications driving the network are *correct*, or free of software bugs. At the management plane, an operator must determine how to compose together multiple applications to create a coherent forwarding behavior. This requires reasoning about inter-application interactions and how to resolve conflicts between them. Finally, she must understand the execution of the applications on top of the controller, and how any bugs in the controller's implementation may introduce unexpected behavior.

This thesis explores techniques in verification and synthesis to ease the burden of configuring of an SDN and provide techniques to automatically search for unpredictable behavior. We present three primitives to search program behavior at the management plane, control plane, and data plane, without the need to understand the implementation details of the controller or applications. Specifically, we design primitives for orchestration, fast-forwarding, and autocorrect.

First, to tackle the problem of SDN application correctness, we introduce a *fast-forwarding* primitive to verify and explore future behaviors of a control plane application. Unlike traditional approaches to software verification, this primitive faithfully models time — an important source of complexity in control programs. We implement this idea in a model checker called DeLorean, and develop techniques to speed up the exploration of control programs. As a result, DeLorean can explore the future behavior of programs faster than they occur.

Next, to reduce the complexity of composing together multiple, independent applications, we provide an *orchestration* primitive. This technique automatically discovers dependencies between applications and can

suggest conflict-free composition plans to operators. With this primitive, we adopt a relational representation of the network and use a standard PostgreSQL database to develop a controller called Ravel. In addition to logic-based reasoning of inter-application dependencies, Ravel enables ad-hoc creation of new programming abstractions. Furthermore, Ravel can act as a runtime on which different abstractions or controllers can execute using translations to Ravel's schema.

Finally, to reduce the operator's burden of understanding the low-level implementation details of an application or controller, we introduce an *autocorrection* primitive with NEAt (Network Error Autocorrection). NEAt builds on synthesis techniques to prevent applications or a controller from installing policy-violating updates, ensuring the behavior of the network is correct and predictable at the data plane. NEAt allows backward-compatibility with existing SDN deployments and acts as a transparent layer between the network and controller, enforcing correctness by repairing updates that violate the network policy.

We test our primitives on real-world datasets and applications. With DeLorean and NEAt, we find bugs in real configurations running on deployed systems.

*To my parents and sister, for their love and support.*

# Table of Contents

# List of Tables

# List of Figures

# List of Abbreviations

| | |
|---|---|
| FSA | Finite State Automata |
| GCD | Greatest Common Divisor |
| HA | Home Automation |
| ILP | Integer Linear Programming |
| NEAt | Network Error Autocorrection |
| SDN | Software-Defined Networking |
| SQL | Structured Query Language |
| TA | Timed Automata |
| VC | Virtual Clock |

# Chapter 1

# Introduction

Ensuring a computer network reliably transfers data from a source to a destination is a complex process. Networks may consist of hundreds or thousands of forwarding devices, which a network operator must individually configure. A modern network can be divided into three planes of functionality: the management plane, the control plane, and the data plane. To monitor the state and performance of the routers and switches on a network, as well as define high-level policy, an operator can use *management plane* services and protocols, such as SNMP (Simple Network Management Protocol). To configure the protocols that discover paths connecting endpoints on the network, an operator configures the *control plane* of the forwarding devices that run routing protocols such as OSPF (Open Shortest Path First). These protocols generate forwarding rules in the *data plane*, which provides fast packet forwarding through the network. Although routing protocols provide reliable routing even in the face of link failures, a network operator must configure hundreds or thousands of parameters for these protocols through command-line interfaces (CLIs) or application programmer interfaces (APIs).

In a traditional network, the control plane and data plane are tightly coupled. Both functional components reside on a forwarding device, and therefore an operator must individually configure each device. In doing so, the operator must translate high-level policies into low-level commands and parameters. As the size of enterprise networks increase, so does the complexity of configuring and maintaining a network. Worse yet, networks require constant human intervention and maintenance to remain operational. The network may grow, requiring the addition of new router or switches, and devices may fail or need upgrading. One recent study found that operators make changes to their network configuration at least once per day [63]. These updates, even those that fix a bug, can be problematic from an operator's perspective. In the same study, 89% of operators were never certain a change would be bug-free, and 82% were concerned the change would affect existing functionality unrelated to the change.

In recent years, a new paradigm called software-defined networking (SDN) has emerged, separating the control plane and data plane. With SDN, a centralized controller acts as the control plane, while switches simply forward packets. The controller sends updates to the switches to install or remove forwarding rules from their data plane, and switches send events to the controller notifying it of a new connection, the arrival

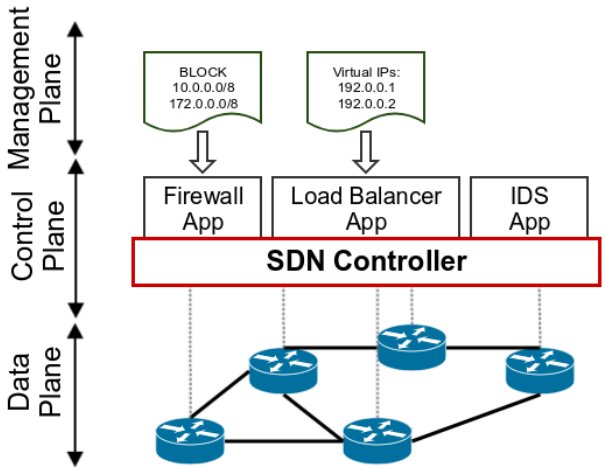



Figure 1.1: SDN architecture

of certain packets, or the state of flow counters. As a result of this decoupling of the control and data planes, the data plane adopts a unified interface (e.g., OpenFlow) and the forwarding behavior of switches is abstracted into a simple match-action model. Furthermore, the introduction of a centralized controller offers a global view and control of the network. Controllers [14] can expose APIs, as well as high-level languages and novel abstractions [45, 78, 87, 63, 81], which developers can leverage for programmatic control of the network. Thousands of lines of configurations can be replaced with a few hundreds of lines of Python or Java running on an SDN controller. This modular design simplifies development, and allows network operators to leverage a potentially vast library of open source applications.

Nevertheless, configuring and managing the applications — or *control modules* — driving an SDN is inherently difficult. Like traditional networks, SDNs are prone to misconfigurations, software bugs, and unexpected behaviors. At the control plane, an operator must ensure the applications are *correct*, or free of software bugs. These applications can be complex — they may depend on time, requiring an operator to reason about how the application will respond not only to events, but also the time at which the events occur. At the management plane, with multiple applications collectively controlling the network, the operator must reason about their interactions and how to resolve conflicts between them. In many cases, this requires the operator to understand each application at the implementation level. Finally, to prevent unexpected behavior at the data plane, an operator must understand the execution model of the controller — the system on which the applications run — and how each application's output is processed and delivered to the switches. Failure to correctly reason about any of these three dimensions can lead to unpredictable behavior, compromising the safety, security, and efficiency of the network. This configuration management nightmare, however, is not limited to enterprise networks. With the proliferation of Internet-enabled devices for homes

(e.g., thermostats, lights, cameras), even home networks and home automation systems can be difficult to manage. Unintended behaviors in this context can compromise privacy and increase cost (e.g., in the case of an improperly configured thermostat).

Recent work toward solving this challenge has focused on three approaches: domain-specific languages, verification, and synthesis. High-level, domain-specific languages [45, 78, 63] simplify the development of new SDN applications, but require porting existing functionality into the new language. Further, because of the level of abstraction of these languages, the resulting low-level behavior of the program may not be fully understood by the programmer. Composing or integrating together multiple applications in these languages is also not a simple task. Pyretic [78] and other frameworks [56, 76, 93] targeting this integration problem can require an operator to coordinate the composition at a low level. As a result, a network operator must take on the role of a developer and understand the details of the implementation to determine how the applications will interact and how to resolve conflicts between them.

Orthogonal to these language-based approaches is verification of existing or new configurations. For SDNs, this can target the correctness of the control plane [27, 30] or the data plane [61, 59, 58, 34]. Traditional model checkers [46, 79, 62] and those that verify control plane applications [27], however, abstract away time in their exploration of the application's behavior. As a result, for SDN and other control programs that depend intimately on time, these approaches can miss potential behavior or explore behavior that may never occur in practice. Prior work [19, 24] that addresses this problem requires modeling the applications as automata. The modeling process may introduce bugs if the process is manual and the model does not faithfully capture the behavior of the program. The model may also become inconsistent with the implementation during development [79], and the tool will fail to catch any bugs introduced after the model is constructed.

In place of verifying old or new configurations, another technique is synthesis. Work in this space [90, 104, 91, 89] automatically configures the network from examples or partial specifications. The resulting configuration obeys the correctness properties derived from the specification. However, the performance of these tools can be too slow for real-time applications, for example, to act as a transparent layer repairing buggy updates from applications. Additionally, specifying program behavior using examples may be tedious and more time consuming than implementing the application in a lower-level language.

Although there is no silver bullet to this challenge, a common trend in SDN research is to move network management into software, thus becoming a software development task. Software testing tools, such as verification and synthesis, can then be more easily applied to network configurations. However, the abstractions provided by these SDN software frameworks and domain-specific languages have an inherent trade-off

between the level of control and level of abstraction. Too high-level of an abstraction and an operator or developer maybe not be able to fully express the desired functionality and the system may hide important technical details. With too low-level of an abstraction, the operator may be burned with too many technical details, resulting in buggy or undesirable network behavior. Building off the insight that network management is moving toward a software engineering approach, this thesis aims to leverage principles from formal methods to simplify SDN management for network operators by introducing primitives to make the task of managing this software more predictable.

## 1.1 Terminology

In this thesis, we define the task of *configuring* an SDN as the process of implementing a specific behavior in the network. This behavior is often the composition of multiple, high-level demands (e.g., industry standards or regulatory requirements). For example, this might include how paths are rerouted when links fail, how traffic is balanced across a set of links, or how clients connecting to a web server are balanced across a set of web server replicas. This behavior is collectively realized over a set of forwarding devices that are centrally controlled by a *controller*. The controller exposes an API to programmatically modify the state of these devices.

*Applications*, or *control modules*, build on this API to implement a specific feature (e.g., load balancing). An application may have inputs, such as an IP whitelist or blacklist in the case of a firewall application. Applications are event-driven and typically structured as a collection of event handlers. These event handlers execute in response to events occurring in the system, such as the arrival of a packet at the controller. An event handler can set a timer to fire at a point in the future, which may trigger the execution of another event handler. Additionally, a specific time of day can trigger an event handler to execute.

To run multiple applications simultaneously, one must define a *composition plan* that specifies the ordering of application execution or how to resolve conflicts between applications. One such composition plan is the assignment of priorities to applications, such that the highest-priority application executes last in response to an event. For example, suppose a load balancer and firewall are composed together, and the firewall is assigned the highest priority. If the load balancer installs a path that violates the firewall's blacklist, the firewall will override this update before it is installed in the switch.

To modify the state of the forwarding devices, applications generate *updates* that the controller sends to the devices. An update may add, remove, or modify a forwarding rule in a device. Although devices also send updates to the controller to inform it of the state of links or flow counters, only updates from the

controller can modify forwarding rules.

Therefore, the configuration process for an operator consists of at least three tasks: *i*) implementing or collecting a set of applications that will control the network; *ii*) determining the inputs for these applications based on the specific demands, policies, and topology of the network; and *iii*) defining a composition plan for the applications.

## 1.2 Research Questions

The main question that drives this thesis is *how can we make control of software-defined networks more predictable without requiring an operator to understand the implementation details of different layers in the SDN stack?* Although understanding the implementation details of an application may burden a network operator, she must have some knowledge of an application's behavior and how it interacts with the other applications simultaneously controlling the network. However, understanding how these applications behave — independently and collectively — in response to all events is a difficult task. This task becomes even more challenging as additional applications are added to the network configuration.

In answering this question, this work develops three new primitives to automatically explore and search program behavior for unexpected behavior: fast-forwarding, orchestration, autocorrection. These primitives build on exploration of program behavior, in the form of verification and synthesis. The aim of these primitives is to aid the operator in configuring the network and provide confidence that the configuration produces the intended behavior at each layer: the management plane, the control plane, and the data plane. With orchestration [99, 98], an operator can compose together multiple applications — written at different levels of abstraction — without the need to specify a low-level composition policy. Orchestration searches for and synthesizes a policy that avoids conflicts between the operations within each application, ensuring applications are configured correctly in the management plane. With fast-forwarding [30, 29], the operator can verify and interactively inspect the future behaviors of the applications, ensuring the application is correct in the control plane. Should bugs exist in the controller, or intricacies of its implementation require non-intuitive changes to an application, autocorrection [105] ensures the applications and controller produce the correct data plane rules. An operator can define correctness properties and run unmodified SDN application, with the confidence that updates violating these properties will be repaired automatically.

### 1.2.1 Fast-Forwarding

Predictable and bug-free behavior in an SDN needs a correctly implemented SDN application. As such, in the first part of this thesis, we develop a fast-forwarding primitive for home automation (HA) and software-defined networking (SDN) programs that builds on software verification. HA and SDN programs depends intimately on time, both on absolute time (i.e., time of day) and relative time (i.e., the elapsed time between two or more events). In enterprise networks, behavior can depend on absolute time as operators define different actions for peak load time, data usage policies over a fixed period (e.g., monthly usage quotas), or for billing and accounting. Behavior can also depend on relative time, for example, for DHCP leases or cache expiry. Similarly, automated homes may have different policies for day vs. night, or timers to turn on lights for certain periods after detecting motion.

To realize this primitive, we design a tool called DeLorean, which uses timed automata (TA) [16] to explore program behavior of real HA and SDN programs. Unlike previous TA-based model checkers [19, 24], DeLorean does not require a model of the code. Fast-forwarding entails a hard performance constraint as such a tool must explore all possible future behaviors before they can occur in real time. To this end, we introduce new techniques to speed up exploration and find DeLorean can fast-forward HA programs up to 36K times faster than real-time.

This work is inspired by the following questions:

- How can we scalably examine the behavior of a control plane application by checking all possible events occurring at all possible times?
- How can we perform this exploration without first deriving an automaton from the program?
- What types of control programs benefit most from checking temporal correctness properties?
- What types of bugs are not easily uncovered by traditional verification tools that do not maintain temporal consistency?

### 1.2.2 Orchestration

Although SDN simplifies the management of networks by providing a large library of existing applications, composing together and configuring these applications is not a simple task. To correctly predict the interaction of and conflicts between the applications, an operator may need to understand each application at an implementation level. Orchestration automates this search for dependencies and conflicts, and builds on the idea that network control fundamentally revolves around data representation. We design a controller,

Ravel, that adopts a relational representation of the network, where the network topology, forwarding, and control applications are represented as SQL tables, views, and triggers.

While this is not the first work to leverage a SQL-like syntax for SDN applications, it is the first SDN controller built entirely on top of a standard (PostgreSQL [12]) database. Furthermore, the design of Ravel enables building on principles from the database community to solve challenging SDN problems, such as the composition and configuration of multiple applications. For example, this problem can be formulated as a data integration problem [68, 35] of merging data sources into an integrated whole, where network management tasks represent the individual sources, and the data plane as the integrated whole.

The research questions associated with this approach are presented below.

- With constantly changing network demands, what are the "right" programming abstractions for network applications? Can a relational representation of the network enable easier design of new abstractions and composition across them?
- Can this representation allow for easier and automated composition of multiple SDN applications?

### 1.2.3 Autocorrection

To aid in the management of existing network configurations and SDN applications, the final part of this thesis proposes an autocorrection primitive. To design such a primitive, we explore a real-time application of synthesis through the design of a tool called NEAt (Network Error Autocorrection). NEAt prevents unmodified SDN applications from installing updates that might violate an operator's intended policy. Rather than blocking these updates, as in the case of real-time verification tools, NEAt attempts to repair violations introduced by the update. NEAt automatically searches for changes to the update, such that the resulting configuration obeys the correctness properties defined by the network operator.

The main research questions for this primitive are:

- How can we enforce and repair correctness on-the-fly without modifications to control programs?
- How can we constrain divergence between application state and network state in the face of repairs?

## 1.3 Thesis Statement

Our thesis is the following:

Configuring an SDN requires reasoning about a large space of program behaviors. An operator must consider the implementations of the applications controlling the network, their interactions with other ap-

plications, and the controller on which it runs, among other complexities. Automatic search techniques can help an operator efficiently reason about this large space of behaviors.

## 1.4 Key Contributions

The key contributions of this thesis are three new primitives enable automatically searching SDN configurations for unexpected behavior. Using fast-forwarding, orchestration, and autocorrection, network operators can examine how the configuration executes at the management plane, control plane, and data plane. In developing these primitives, this thesis makes the following concrete contributions:

- **The first TA-based model checker that operates on real code.** Building on real-time systems verification and the theory of timed automata, we build DeLorean, which explores future behaviors of real HA and SDN programs. Unlike previous TA-based techniques [19, 24], this is the first to verify real programs, rather than a model of the code. Furthermore, DeLorean is the first application of TAs as a technique to explore future program behaviors. DeLorean also achieves better coverage of source code and program states than existing model checkers that maintain temporal consistency [103].

- **Novel optimizations for TA-based exploration of program behavior.** Similar to traditional model checkers [79], DeLorean faces the state space explosion problem, in which programs can generate an intractable number of programs states to explore. Worse yet, TAs use *virtual clocks* (VCs) defined by the constraints on variables storing time. These VCs define a finite number of *regions*, or equivalence classes, and an exploration requires examining every reachable program state in every reachable time region. To scalably explore this larger state space, this work uses three optimizations: predicting successor states, reducing the number of VCs, and identifying and exploring independent control loops in the program.

- **An SDN controller built on a standard database.** To automatically configure and compose together multiple applications, we propose a design for a new SDN controller called Ravel that is built on top of a standard PostgreSQL [12] database. In addition to the ability to leverage research from the database community, Ravel presents the network to the operator as a collection of SQL tables. Not only is SQL-broadly familiar and a time-tested language for manipulating data, it additionally allows for easy creation of new abstractions in the form of SQL views and triggers.

- **A formulation of the application composition problem as a data integration problem.** An SDN controller built on top of a standard database enables an investigation into principles from

database research and provides a foundation for an orchestration primitive. Orchestration aims to automate the configuration and composition of multiple SDN applications. In particular, we formulate this problem as a data integration problem [68, 35]. We develop an algorithm to translate static Pyretic [78] applications to Ravel (SQL) applications. We then demonstrate the feasibility of automatically combining together and avoiding conflicts across multiple applications simultaneously controlling a network by building on irrelevance reasoning [22, 70].

- **A technique to repair policy-violating updates for unmodified SDN applications in real-time.** To enable an autocorrection primitive for network operators, we develop a tool called NEAt that formulates the problem of repairing policy-violating updates as an integer linear programming (ILP) problem. With NEAt, operators define correctness policies in the form of graphs. To meet real-time performance constraints, we propose an algorithm to compress the network topology and equivalence classes of the forwarding behavior, and solves the ILP problem on these compressed graphs.

| Primitive | Tool | Chapter | Layer of Operation |
|---|---|---|---|
| Fast-Forwarding [30, 29] | DeLorean | 2 | Control Plane |
| Orchestration [99, 98] | Ravel | 3 | Management Plane |
| Autocorrection [105] | NEAt | 4 | Data Plane |

Table 1.1: Primitives introduced in this thesis

### 1.4.1 Thesis Outline

Table 1.1 outlines each primitive discussed in this thesis and the layer at which it operators in an SDN. In the following chapters, we describe in further detail each of these primitives. In Chapter 2, we present our work on a fast-forwarding primitive through the design of DeLorean. In Chapter 3, we describe Ravel, a SQL-based SDN controller that provides a foundation for an orchestration primitive for composing together multiple SDN applications. In Chapter 4, we detail NEAt, an autocorrection primitive that repairs updates on-the-fly. We highlight related work in Chapter 5. In Chapter 6, we discuss future work and the application of this thesis to other domains driven by control programs. We conclude in Chapter 7.

# Chapter 2

# Systematically Exploring Control Program Behavior

One step in effectively controlling a software-defined network (SDN) is correct and predictable application behavior in the control plane. Composing together or repairing updates from applications ridden with bugs is a futile exercise. As such, this thesis first introduces a *fast-forwarding* primitive to verify and explore the behavior of control programs, such as SDN and home automation (HA) applications, before examining an orchestration primitive to automatically compose applications.

In particular, this chapter tackles the problem of systematically exploring the behavior of control applications that depend intimately on time. SDN and HA applications are written to depend on time, for example, to take different actions for day vs. night, or expire DHCP leases. Traditional approaches toward verification, i.e., model checkers, either require models of the code or do not accurately model time in favor of scalability. Instead, this work aims to scalably verify temporal properties of actual code for HA and SDN programs and, for HA programs, do so with a real-time performance constraint.

The work in this chapter was co-authored with Ratul Mahajan, Matthew Caesar, and Madan Musuvathi. It was published in the 2015 USENIX Annual Technical Conference [30].

The main research questions associated with this primitive are:

- How can we scalably examine the behavior of a control plane application by checking all possible events occurring at all possible times?
- How can we perform this exploration without first deriving an automaton from the program?
- What types of control programs benefit most from checking temporal correctness properties?
- What types of bugs are not easily uncovered by traditional verification tools that do not maintain temporal consistency?

## 2.1 Introduction

Control programs that orchestrate the actions of "dumb" devices are becoming increasingly popular. The number of such devices, including locks, thermostats, motion sensors, and packet forwarding switches, is projected to grow beyond 50 billion by 2020 [39]. In a similar vein, SDN has become a multi-billion dollar market.

While control programs for these devices may be specified using simple languages (e.g., ISY [54], in the case of home automation), reasoning about their correctness is an incredibly complex task. The programs can have complex interactions across rules due to shared variables and device states. Further, time plays an important role in program behavior, as the behavior can change with the time of day or the time between occurrences of certain events. System behavior is often programmed directly to depend on time, in terms of policies (e.g., different actions during day vs. night) and protocol behavior (e.g., DHCP leases). Therefore, the behavior of these control programs is hard to verify by running the program a few times (e.g., during development) and, as a result, many bugs are discovered in production. These bugs can compromise the safety, security, and efficiency of the system.

One method of uncovering bugs is to systematically explore program behavior using model checking. However, prior work [27, 46, 62, 79, 102] does not address an important aspect of program behavior, specifically time. Instead, these tools abstract away time and, as a result, assume timers of different periods can fire at any time and in any order. Similarly, comparisons involving time can nondeterministically return true or false. Such an imprecise analysis of time is unacceptable for control programs because, as we show later, it generates many states that are not reachable in practice. This can force developers to sort through many false positive bugs reported by these tools. Furthermore, by abstracting time, these tools preclude developers from verifying correctness properties involving time (e.g., that timers fire at the correct time and under the correct conditions). Tools that use coarse heuristics to model time [103] eliminate false positives at the expense of incomplete exploration.

Accurately modeling time when exploring program behavior is a non-trivial problem. The challenge arises because events can occur at any time. To explore all possible behaviors, in theory, we must study all possible events occurring at all possible times. However, this is an ill-defined concept since time is continuous. We describe in § 2.2.1 why circumventing this issue by naïvely discretizing time is unsatisfactory.

We investigate the use of timed automata (TA) [16] to systematically explore the behavior of control programs. TAs have been previously used to verify models of real-time systems. A TA is a finite state machine extended with real-valued (not discrete) virtual clocks. TA transitions can specify constraints on clock variables. For instance, a timeout transition should happen only when a particular clock variable is

greater than a constant. The analyzability of TAs arises from the fact that, under certain conditions on clock constraints, one can define a finite number of *regions* [16]. All program states within a region are equivalent with respect to the untimed behavior of a system. Thus, "all possible times" can be safely translated to "all possible regions."

Prior work [19, 24] on exploring temporal behavior with TAs analyzes only an abstract model of a program or system. However, errors can be introduced in the model if it does not faithfully capture the behavior of the program, and the model can "drift" as the program evolves during development [79]. In this chapter, we focus on using TAs to verify temporal properties of actual code. In particular, we ask: *can TAs be used to analyze executable programs? If so, what are the limitations of applying this theory to practice?*

Exploring the temporal behavior of a program without the need to first derive a TA introduces several new challenges. A TA-based exploration requires the set of temporal constraints that appear in the program. We develop a method to extract this information using program analysis [65]. As with many verification techniques, TA-based exploration inherits the state space explosion problem. As prior work is limited to exploring only abstract models, most heuristics to reduce the state space assume full knowledge of the model and cannot be used in our exploration. We develop three new techniques to boost exploration efficiency: *i*) reducing the number of clock variables in the program, to cut down the number of regions we must explore; *ii*) exploring the program as multiple, independent control loops; and *iii*) predicting the response of the program to certain events, reducing the number of times we must execute the program.

We implement our approach for TA-based exploration in a tool called DeLorean and evaluate it in two diverse domains: home automation and software-defined networks. We explore 10 real HA programs and three SDN applications. Though DeLorean does not completely bridge the gap between the theory and practice of exploring real code with TAs, we see measurable benefit. DeLorean finds bugs uncovered by existing verification tools and new bugs that cannot be uncovered with existing techniques. We find we can achieve higher fidelity in exploring behavior, resulting in improved state and code coverage. We achieve up to 94% code coverage, compared to 76% in existing techniques that explore temporal behavior [103].

## 2.2 Background and Motivation

Many networked systems today are logically centralized. An HA system is composed of a controller and devices such as light switches, motion sensors, and locks. The controller receives notifications from the devices (e.g., when motion is sensed), can poll them for their current state (e.g., current temperature), and can send them commands (e.g., turn on the light switch). It uses these capabilities to coordinate the devices.

Similarly, in SDNs, a controller manages the operation of switches by configuring them to forward packets as desired. The switches inform the controller when they receive packets for which forwarding actions have not been configured.

At the core of logically centralized control systems is a control program that determines its behavior. While the implementation languages for different systems and domains are different, control programs have a common structure. Their operation can be understood in terms of a set of rules. Each rule has a trigger and associated actions. A trigger is either an event in the environment (e.g., sensed motion, arrival of a packet) or a firing timer. Actions include setting a device state (e.g., turn on the light, installing a new rule in a switch) or a variable and setting timers. Actions can be conditioned on device state, variable and timer values, and time of the day. Programs are single-threaded and each rule runs until completion before another is processed.

Figure 2.1 shows an example program with three rules. Assume that the user wants to turn on the front porch light when motion is detected and it is dark out, and to automatically turn off this light after 5 minutes if it is daytime. Rule 1 is triggered when motion is detected by the front porch motion sensor. It turns on the light if motion is detected twice within 1 second and the light level sensed by a light meter is less than 20. The first condition is a heuristic to filter out false positives in motion sensing, and the second ensures that light is turned on only when it is dark. Rule 1 also updates the time when motion was last detected. Rule 2 is triggered when the front porch light goes from off to on (either programmatically or through human action) and sets a timer for 5 minutes. Rule 3 is triggered when this timer fires, and turns off the light if the current time is between 6 AM and 6 PM.

### 2.2.1 Reasoning about Program Correctness

The correctness of control programs can be hard to reason about. Even if individual rules are simple, reasoning about the program as a whole can be difficult because of complex interactions across rules. These interactions arise from shared state across rules due to the state of variables and devices. Thus, the program's current behavior depends not only on the current trigger but also on the current state, which in turn is a function of the sequence and timings of rules triggered in the past. This dependency and the number of possible sequences makes predicting program behavior difficult.

As an example, even the simple program in Figure 2.1 has a behavior that may not be expected by the user. Suppose the light is turned from off to on at 9:00 PM either due to sensed motion or by the user, triggering Rule 2. Then, the user walks on to the front porch at 9:04:50 PM, triggering Rule 1. This user might expect the light to stay on for at least 5 minutes, but it goes off unexpectedly 10 seconds later (at

```
1   /* Rule 1 */
2   motionFrontPorch.Detected:
3       if (Now - timeLastMotion < 1 secs
4           && lightMeter.LightLevel < 20)
5           FrontPorchLight.Set(On);
6       timeLastMotion = Now;
7
8   /* Rule 2 */
9   frontPorchLight.StateChange:
10      if (frontPorchLightState == On)
11          timerFrontPorchLight.Reset(5 mins);
12
13  /* Rule 3 */
14  timerFrontPorchLight.Fired:
15      if (Now.Hour > 6 AM && Now.Hour < 6 PM)
16          FrontPorchLight.Set(Off);
```

Figure 2.1: An example home automation program

9:05 PM). The fix here is of course to reset the timer in Rule 1, but that may not be apparent to the user until this behavior is encountered in practice.

Control programs are not the only ones whose correctness is difficult to reason about; the same holds true for almost all real-world programs such as network protocols and distributed systems. As a result, in a range of settings, researchers have developed a variety of techniques and corresponding tools, called model checkers, to automatically explore program behavior [27, 79].

#### Complex Dependence on Time

The behavior of control programs can depend intimately on time, both absolute time and the relative timing of triggers. For instance, the behavior of the program in Figure 2.1 depends on the time of day and on how close in time two motion events fire.

Existing model checkers do not systematically model time. Most [27, 62, 79] perform *untimed model checking*. They completely abstract time (in the interest of scalability) and do not maintain temporal consistency. During exploration, calls to `gettimeofday()` return random values and timers can fire in any order, regardless of their values. This can lead to many false positives (§ 2.5.4), i.e., bad states that will not arise in practice. False positives can be highly problematic, and often worse than missing errors [103], because they can send developers on a wild goose chase. Equally important for our context, since time is abstracted, untimed model checkers cannot verify time-related properties of a system, which are of prime interest for control programs.

One model checker that maintains temporal consistency is MoDist [103]. It has a global virtual clock,

```
1  Trigger0:
2    timeTrigger0 = Now;
3    timeTrigger1 = Now;
4    trigger1Seen = false;
5  Trigger1:
6    if (Now - timeTrigger0 < 5 secs)
7      trigger1Seen = true;
8  Trigger2:
9    if (trigger1Seen)
10     if (Now - timeTrigger1 < 2 secs)
11        DoOneThing();
12     else
13        DoAnotherThing();
```

Figure 2.2: An example home automation program

which is used to return values for `gettimeofday()`. Timers are fired in order and, when they do, the virtual clock is advanced accordingly. It uses static analysis of program source to infer all timers, including implicit timers. (Line 3 of Figure 2.1 represents an implicit timer that is set in Line 6 to expire in 1 second. Line 15 checks if the timer has fired, and the program behaves differently for the two cases.) During exploration, MoDist explores two cases, one in which the timer has expired and one in which it has not. In each, the clock value is set appropriately to a value that is consistent with the explored case.

While MoDist's approach does not produce false positives, it does not comprehensively explore all possible behaviors because exploring both cases for timers is not enough. Consider the simple example in Figure 2.2. This program has three triggers: Trigger0 resets the control loops; Trigger1 is considered as seen if it occurs within 5 seconds of Trigger0; and Trigger2 does different things depending on whether it occurs within 2 seconds of Trigger0. Assume that when Trigger0 fires, the virtual clock time of MoDist is $T$ (seconds). While exploring Trigger1, to cover both cases MoDist will select one virtual clock time in the range $[T, T+5)$ and one greater than $T+5$. But now it has a problem: while exploring Trigger2, it can only explore one of the two branches (Line 10 or 12) and not both. If it had picked $T+1$ in the first case, it cannot explore the path on Line 13; if it had picked $T+3$, it cannot explore the path on Line 11.

Note that at the point of exploring Trigger1, MoDist has no reason to believe the specific selection in the range $[T, T+5)$ matters. All choices lead to the same program state and paths, and only later the choice has an impact. This is just one simple example; in reality, temporal constraints in the program can be highly complex (e.g., the same timer may drive behavior in multiple places).

Without systematic modeling of the temporal behavior of the program, the only way MoDist can explore all possible program behaviors is to explore all possible times of all possible triggers. But "all possible

times" is ill-defined because time is continuous. We could discretize time and assume events happen only at discrete moments. But picking a granularity of discretization is tricky — if it is too fine, the exploration will have too much overhead as we would explore too many event occurrences; if it is too coarse, the exploration will miss event sequences that occur at finer granularity in practice and lead to different behaviors. Simply picking the smallest time-related constant in the program is also not enough [16]. Thus, there appears no satisfactory way to pick a granularity that works for all events and programs.

We thus systematically reason about time by exploring the control program as a timed automaton (TA) [16]. This lets us carve time into equivalence regions such that the exact timing of events within a region is immaterial. Thus, instead of exploring all possible times, it suffices to explore all possible regions.

#### Time-Bound Correctness Properties

Untimed model checkers find violations of properties such as liveness (i.e., the system will *eventually* enter a good state) and safety (i.e., the system *never* enters a bad state). Since these tools abstract time, they cannot verify properties involving concrete time. For example, consider an SDN program caching mappings of ports to MAC addresses. Entries should expire a certain period after their last access. An untimed model checker can prove the entry expires after a certain sequence of events, but not that it expires after a certain period of time. To prove such a property, we must prove *time-bound liveness*. Furthermore, untimed model checkers cannot verify correctness properties based on absolute time, such as a light turning on at the right time or never being on at a certain time. In our evaluation of HA programs, we find three bugs (P9-1, P9-2, and P10-1 in § 2.5.5) with this type of invariant.

### 2.2.2 Timed Automata

To reason systematically about time, we use timed automata [16] to guide our exploration. TAs are finite state machines extended with real-valued virtual clocks (VC), where a VC represents time elapsed since an event. (Wall clock time is simply one possible VC, which measures elapsed time since Jan. 1, 1970.) The state of a TA is the combination of the state of the underlying finite state machine together with the values of all VCs. A TA transition changes the machine state and may reset one or more VCs. Each transition specifies a set of clock constraints and is enabled from states that satisfy the constraint.

Figure 2.3 shows a TA that captures the behavior of the program in Figure 2.1. There are four states, corresponding to the Cartesian product of whether the front porch light (`FPL`) is on or off and the current light level (`CLL`). The TA uses three VCs to capture the time since *i*) the last motion (`tlm`), *ii*) the light was turned on (`tfpl`), and *iii*) midnight (`td`). Transitions are labeled with their triggers (underlined), the clock

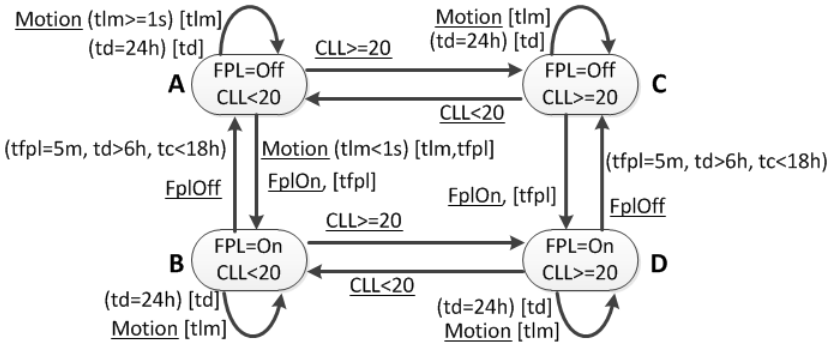



Figure 2.3: A TA for the program in Figure 2.1

constraints (in parenthesis), and the clocks that are reset (in brackets). `Motion` denotes motion, and `FplOn` and `FplOff` denote the physical acts of manipulating the light. Some transitions have multiple labels, one for each situation where the TA can go from the source to the sink state. Transitions that have no triggers are taken as soon as the clock constraints are met.

In general, systematic exploration of TAs is infeasible, even in theory, as VCs hold non-discrete real values. However, the seminal work on TAs [16] shows that an exhaustive exploration is feasible provided the VC constraints obey certain conditions. The conditions are that arithmetic operations cannot be performed between two VCs and a VC cannot be involved in a multiplication or division operation. But adding or subtracting constants to VCs is allowed, and so is comparing two VCs (potentially after adding or subtracting constants).

Under these conditions the possible behaviors of the TA can be discretized into regions, such that the (uncountably many) states in a region behave the same with respect to the correctness properties of the TA. How such regions emerge can be intuitively understood if one observes that for the TA in Figure 2.3, after a motion event, the future behaviors are determined by whether the succeeding motion event occurs before or after 1 second. The exact timing of the second motion event is not critical. Regions exist in multi-dimensional space where each dimension corresponds to one VC, and a point in the space represents concrete values of all the VCs. Regions encompass a set of points such that the exact point is immaterial for the purposes of comprehensive exploration.

The size of the region is proportional to the greatest common divisor (GCD) of constants in clock constraints, and hence the exploration will be faster if the GCD is larger. Regions get exponentially smaller as more VCs are included in the TA, because the plane for each pair of VCs divides open spaces into two parts. Once the regions are known, fully exploring the TA's behavior requires *i*) exploring all possible transitions, in response to all possible triggers, from the current state; and *ii*) exploring exactly one *delay transition* in which there is no state transition but all VCs advance by the same amount. This amount is

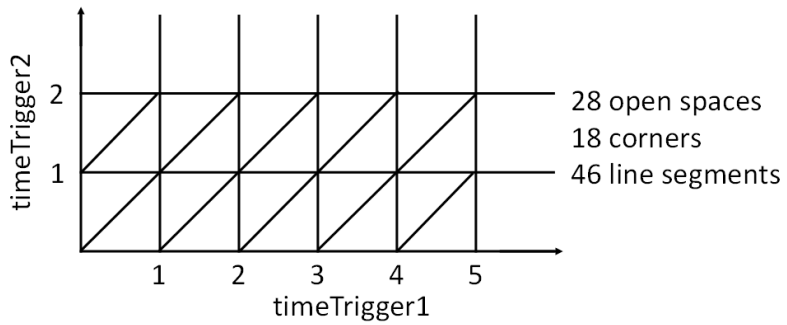



Figure 2.4: Time regions for the example in Figure 2.2

such that the time progresses to the immediately succeeding region. Figure 2.4 shows the regions for the example in Figure 2.2, which has two VCs. The constants in the clock constraints are 5 and 2, and thus the GCD is 1. We get 92 regions in this example.

## 2.3 Our Approach

Our goal is to systematically explore control program behavior. From a starting time and state, we want to predict all possible program behaviors. The exploration should be virtual so the actual state of the devices is not impacted. The output should be the set of unique states the system can be in, along with the sequence of events (i.e., triggers, actions) leading to that state.

### 2.3.1 Introducing Virtual Clocks

Control programs do not contain explicit references to VCs, but as mentioned previously, all time-related activities in effect manipulate VCs.

There are three kinds of time-related activities in control programs. The first is measuring the relative time between two events of interest (e.g., consecutive motion events). Here, a variable (e.g., `timeLastMotion` in Figure 2.1) is used to store the time of the first event, which is then subtracted from the wall clock time of the second event. To express this as a VC, we set the VC to zero when the first event occurs; the value of the VC when the second event occurs yields the delay, since VCs progress at the same rate as the wall clock unless reset.

The second time-related activity is a timer (e.g., `timerFrontPorchLight` in Figure 2.1). To capture this activity using a VC, we reset the VC when the timer is set, and queue a timer trigger to fire after the desired delay, after removing any previously queued event.

The third time-related activity is a sleep call, where actions for a rule are taken after a delay (e.g., turn

on fan, sleep 30 seconds, turn it off). We express this by introducing a new timer and new rule. The actions of the new rule corresponds to post-sleep actions of the original rule. The sleep and post-sleep actions in the original rule are replaced by a timer that fires after the desired delay. In our treatment of sleep calls, if the trigger for the original rule occurs again before the timer set by an earlier occurrence fires, the post-sleep actions that correspond to the earlier trigger will not be carried out (because the earlier timer event will be dequeued). This behavior is consistent with the semantics of the systems we study.

### 2.3.2 Systematically Exploring Behavior

Given a control program and its starting state as input, our goal is to explore a given duration of wall clock time. A duration must be specified since wall clock time is unbounded and has an infinite number of regions. For programs with no dependency on the wall clock (e.g., some SDN applications), no duration is needed.

We assume that the program can be modeled as a TA. That is, all timers and variables that store time in program are, in effect, VCs. To leverage time regions, these VCs must satisfy the conditions mentioned above. We believe that these conditions are met in many contexts. They are certainly met in the different systems that we study in § 2.5 and § 2.6. The scripting languages of some of these systems cannot even express complex clock operations.

However, our exploration does not assume a TA has been derived from the actual code. Existing methods [19, 24] can explore the behavior of a TA, but deriving the entire TA corresponding to the program may not be feasible. Even the smallest of control programs can have extremely large TAs, as it needs to capture the program logic and its response to possible events.

Thus, we explore the TA dynamically, akin to how FSA-based model checkers dynamically explore the FSA instead of deriving the complete FSA of the program. From a starting program state, we repeatedly derive successor states resulting from triggers or delay transitions. For delay transitions, we must know the timed regions in advance to compute the delay amount. Fortunately, constructing regions does not require the complete TA, but only the constraints on the values of VCs [16]. We extract these constraints using analysis of program source.

Figure 2.5 shows how we comprehensively explore program behavior. Assume we want to explore `FFDuration` of behavior, starting from the program state $S_0$. Program state includes the values of (non-time) variables, VCs, and enabled timers (i.e., ready to fire). We do a breadth-first exploration using a queue of unexplored states. Obtaining all successors of a state entails firing all possible events and all enabled timers. If a successor state is not similar to any previously seen state, we add it to the queue. Two states are similar

```
1: EndWC=Time.Now + FFduration;          ▷ How long to explore
2: S_0.WC = Time.Now;                    ▷ Set the wall clock
3: ES = {};                              ▷ explored states
4: US={S_0};                             ▷ unexplored states
5: while US ≠ φ do
6:     S_i = US.pop();
7:     ES.push(S_i);
8:     for all e in Events, Si.EnTimers do
9:         S_o = Compute(S_i, e);
10:        if !Similar(S_o, (US ∪ ES)) then
11:            US.push(S_o);
12:        end if
13:    end for
14:    if S_i.EnTimers = φ then
15:        delay = DelayForNextRegion(S_i.Region);
16:        if S_i.WC + delay > EndWC then
17:            continue;
18:        end if
19:        S_o = S_i.AdvanceAllVCs(delay);
20:        for all timer in S_o.Timers do
21:            if timer.dueTime >= S_o.WC then
22:                S_o.EnTimers.Push(timer);
23:            end if
24:        end for
25:        if !Similar(S_o, (US ∪ ES)) then
26:            US.push((S_o, t));
27:        end if
28:    end if
29: end while
```

Figure 2.5: Pseudocode for basic TA exploration

if their variable values and set of enabled timers are identical and if their VC values map to the same region; VC values need not be identical since the exact time within a region does not matter.

If the state being explored has no enabled timer, it is eligible for a delay transition. This represents a period of time where nothing happens and time advances to the succeeding region. States with enabled timers need to fire all enabled timers before time can progress. We ignore the successor if this delay takes us past the end time (i.e., starting wall clock time + specified duration). Otherwise, the successor state is computed by advancing all VCs. We treat wall clock time, which is virtualized during exploration, as any other VC except that it never resets; it tracks the progress of absolute time. We then check if any of the timers have been enabled because of this delay and mark them as such. The construction of time regions guarantees that no timers are skipped during the delay transition.

### 2.3.3 Achieving Scalable Exploration

The basic TA-based exploration above correctly handles time but is too slow to be practical. We use three general techniques to make it practical:

#### Predicting Successor States

Our first technique reduces the time to obtain successor states of a state being explored. We must first define the notion of *clock personality*. Two program states have the same clock personality if their values of all the VCs are equivalent with respect to all the clock constrains of the TA. Two program states can have the same clock personalities even if they are not in the same region.

If two states $S1$ and $S2$ with the same clock personalities have identical variable values and enabled timers, then any stimulus (i.e., combination of trigger and environmental conditions) will have exactly the same effect on both states. Thus, it is necessary to compute the successor of only one such state, say $S1$. The successor of $S2$ can be obtained from the successor of $S1$ while retaining the clock values of $S2$ for all VCs except those that are reset by the current stimulus.

Computing a successor requires deserializing the parent's state, running the program, subjecting it to the stimulus, and serializing the successor's state. These are costly operations. In contrast, prediction requires only copying the state and modifying VC values.

#### Independent Control Loops

Our second optimization is based on the observation that large control programs may often be composed of multiple, independent control loops manipulating different parts of the program state. For instance,

thermostats and furnaces may be controlled by a climate control loop, and locks and alarms by a security loop. These two may manipulate different variables and clocks, but otherwise share no state. In such cases, we can explore the loops independently, instead of exploring them jointly. Separate exploration is faster since joint exploration considers the Cartesian product of the values of independent variables and clocks. We use taint tracking to identify independent loops.

**Reducing the Number of Clocks**

The number of VCs in the program has a significant impact on exploration efficiency because the size of regions shrinks exponentially with it. When transforming a control program, we should introduce the minimum number of VCs. We exploit two opportunities. First, consider cases where the actions in a rule have multiple sleeps, e.g., `action1; sleep(5); action2; sleep(10); action3`. Instead of using two timers (one per sleep), we can use only one because the two sleeps can never be active at the same time [31]. To retain the original dynamic behavior, we introduce a new program variable to track which actions should be taken when the timer fires. In the example above, when the rule is triggered, after `action1` is taken, this variable is reset to 0 and the timer is set to fire after 5 seconds. When the timer fires: *i*) if the variable value is 0, `action2` is taken, the variable is set to 1, and the timer is set to fire after 10 seconds; *ii*) if the variable value is 1, `action3` is taken.

Second, control programs often have daily action for different times of day (e.g., sunrise, one hour after sunrise). The straightforward translation is to introduce a new timer per unique activity. A more efficient method is to use one timer to conduct all such activities, using a method similar to the above — introduce an additional program variable to cycle through the different actions and reset it after the last action is conducted.

### 2.3.4 Theory-Practice Gap

Existing TA-based model checkers work with abstract models and assume the model is provided as input. In building the model incrementally and dynamically, we uncover several gaps in using TAs on real code.

A transition in a TA must occur instantaneously since time only progresses explicitly through a delay transition. In practice, however, the processing of an event (e.g., in response to motion occurring) may involve a non-trivial amount of time. In our implementation, we assume processing time is instantaneous, but propose a technique to handle events with non-trivial processing times. For each of these event handlers in the program, we can introduce a timer to expire after the expected processing time. When the timer is active and has not expired, the system is processing the event. If the timer is inactive, either because the

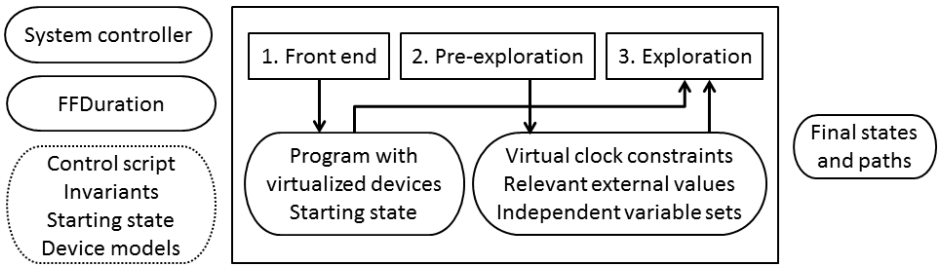



Figure 2.6: Overview of DeLorean

timer has expired or has not been activated, the system is not processing the event.

In some systems, clocks may be created in response to events. In SDN programs, for example, flows installed in switches in response to a packet arrival at the controller introduce two new VCs — one each for the soft and hard timeouts. We can use symbolic execution to extract the clock constraints for all possible values of timeouts, but we cannot determine the number of occurrences of the event triggering this behavior. Rather, this is dependent on the number of times the event occurs along a path generating a specific state. We cannot add a new clock with a new constraint to the region construction, as we cannot change regions during exploration. Regions are constructed using the GCD of the clock constraints and adding a new clock mid-exploration could change the delay if the GCD changes. Instead, we pre-allocate a number of VCs, each with a specific constraint. During exploration, when a new VC needs to be created, it is allocated an available VC from a queue of pre-allocated VCs.

## 2.4 Design

We now describe the design of DeLorean, our TA-based model checker. As shown in Figure 2.6, the primary inputs to DeLorean is the control program — including a model of the controller (on which the program runs) and devices — and the duration of wall clock time to explore (`FFDuration`). If the program has no dependence on wall clock time, i.e., timers are relative, `FFDuration` is not needed. The user can also specify three optional inputs. The first is a list of invariants on device states and system behavior, which should be satisfied at all times. These are specified in a manner similar to the $if$ conditions in the rules and can include time. The second is the wall clock time at the start of the exploration. The third is the starting state of (a subset of) devices and variables from which exploration should begin.

The output of DeLorean is all the unique states of the devices. If invariants are specified and violated by any state during the exploration, we also output the state. In addition, DeLorean outputs the path that leads to each state — a timestamped sequence of triggers, along with the values of environmental factors

during those firings.

DeLorean has three stages. First, the front end converts the program to one where clocks have been virtualized, using the method in § 2.3.1. Second, pre-exploration analyzes this program to recover information required for the optimizations in § 2.3.3. The final stage is the exploration itself.

### 2.4.1 Pre-Exploration

This stage analyzes the program produced by the front end to recover the information needed for constructing timed regions and implementing the optimizations in § 2.3.3. Here, we use symbolic execution [65] of program source. Symbolic execution simulates the execution of code using a symbolic value $\sigma_x$ to represent the value of each variable $x$. As the symbolic executor runs, it updates the symbolic store that maintains information about program variables. For example, after the assignment $y{=}2x$ the symbolic executor does not know the exact value of $y$ but has learned that $\sigma_y{=}2\sigma_x$. At branches, symbolic execution uses a constraint solver to determine the value of the guard expression given the information in the store. The executor only explores the branch corresponding to the guard's value as returned by the constraint solver, ensuring infeasible paths are ignored. If there is insufficient information to determine the guard's value, both branches are explored. This produces a tree of all possible program execution paths. Each path is summarized by a path condition that is the conjunction of branch choices made to go down that path.

We symbolically execute the program's main control loop, which is the starting point for all processing activity. We configure the symbolic executor to treat the following entities as symbolic: program state (variables and clocks) and the parameters of the control loop. The output of the symbolic executor is the set of possible paths for each possible trigger. For each path, we obtain the *i*) constraints that must hold for the program to traverse that path, and *ii*) the program state that results after its traversal. The constraints and the resulting program state are in terms of input symbols, the entities we made symbolic in the configuration.

We can now recover the following information.

**Virtual Clock Constraints** These are required for constructing time regions and for predicting successor states. We obtain them from the output of symbolic execution by taking the union of constraints on VCs along each path. Additionally, program statements that reset a timer $x$ to $k$ seconds are essentially clock constraints of the form $x \geq k$. We extract such statements from the program source and add corresponding constraints to the set.

**Independent Control Loops** We also use the output of symbolic execution for taint tracking. We analyze the program state that results from each path. If the final value of a variable along any path is different from its (symbolic) input value, that variable is impacted along the path. This impact depends on the input

symbols that appear in the output value (data dependency) and path constraints (control dependency). The variables corresponding to those input symbols are tainting the variable.

We use this information to identify independent sets of variables and VCs. Two variables or VCs are deemed dependent if they either taint each other in the program, or they occur together in a user-supplied invariant (as we must do a joint exploration in this case as well). After determining pairwise dependence, we compute the independent sets that cover all variables and VCs.

### 2.4.2 Exploration

This stage implements the method outlined in § 2.3. To start, it runs the program and initializes the starting state. We then checkpoint the program by serializing its internal state. The checkpoint captures the values of all variables, including time related variables, and the times when various timers will fire.

We maintain a table that contains the values of the VCs of a state. Many states differ only in VC values — the successor state after a delay transition differs from the parent only in VC values, so does the successor that is predicted from another state. Maintaining this table separately lets us quickly obtain these successor states. It also helps reduce the memory footprint, since two states that differ only in VC values can share the same checkpoint. However, this implies that the VC values in a table can be out of sync with those embedded in the checkpoint. Thus, when restoring a state, we update its VC values from the table before any other processing.

### 2.4.3 Implementation

DeLorean is implemented with 10k+ lines of C# code. The bulk of this code implements the pre-exploration and exploration stage, which we developed from scratch. We could not use existing tools for exploring TAs [19, 24] because we do not have the complete TA for the program. Our implementation includes models of controllers and devices in the two domains we study: home automation (§ 2.5) and software-defined networks (§ 2.6).

For HA applications, we implemented front end modules for two systems: ISY [54] and ELK [37]. We chose these two because of their popularity. The front end parses ISY or ELK programs using ANTLR [17] and produces a C# program that captures the behavior of the program and contains additional variables, rules, and actions needed for modeling devices. As the state of these devices is typically simple and can be represented using boolean or integer variables, we can model the devices automatically from the ISY or ELK program.

The pre-exploration stage uses Pex [94] to symbolically execute the main event loop of this C# program.

| | type | #rules | #devs | SLoC | #VCs | GCD (s) |
|---|---|---|---|---|---|---|
| P1 | OmniPro | 6 | 3 | 59 | 2 | 7200 |
| P2 | Elk | 3 | 3 | 75 | 2 | 1800 |
| P3 | MiCasaVerde | 6 | 29 | 143 | 2 | 300 |
| P4 | Elk | 13 | 20 | 193 | 5 | 5 |
| P5 | ActiveHome | 35 | 6 | 216 | 14 | 5 |
| P6 | mControl | 10 | 19 | 221 | 4 | 5 |
| P7 | OmniIIe | 15 | 27 | 277 | 6 | 60 |
| P8 | HomeSeer | 21 | 28 | 393 | 10 | 2 |
| P9 | ISY | 25 | 51 | 462 | 6 | 60 |
| P10 | ISY | 90 | 39 | 867 | 6 | 10 |

Table 2.1: The HA programs we studied

Pex is a modern symbolic execution engine that mixes concrete and symbolic execution ("concolic" execution) to boost path coverage and efficiency.

## 2.5 Case Study: HA Networks

To evaluate a TA-based exploration against existing verification techniques, we examined DeLorean in two environments: home automation networks and SDNs.

### 2.5.1 Domain-Specific Optimizations

A common behavior in HA is a dependence on environmental factors (e.g., temperature, light level) sensed by devices in the system. For a comprehensive evaluation, we must explore all combinations of values of external factors. To address this challenge, we build on prior work and combine symbolic execution with model checking [27]. We use symbolic execution of program source to infer equivalence classes of combinations of values of environmental factors. In Figure 2.1, for example, there are two equivalence classes, corresponding to light level values below or higher than 20. During exploration, we then use one set of values per class, instead of exploring all possible combinations of values.

### 2.5.2 Dataset

We evaluated DeLorean using real HA programs. We solicited these programs on a mailing list for HA enthusiasts. We picked the 10 programs shown in Table 2.1. We selected them for the diversity of HA systems and the number of rules and devices. We see that most installations have tens of rules and devices, with the maximums being 90 and 51. This points to the challenge users face in predictably controlling their homes. Collectively, these installations had 19 different types of devices, including motion sensors,

| | # Transitions | CPU Time (sec) | Reduction w/ Prediction |
|---|---|---|---|
| P1 | 72 | 0.10 | -11.11% |
| P2 | 123 | 0.12 | -20% |
| P3 | 178 | 0.15 | -7.14% |
| P4 | 19.7M | 176.10 | 75.16% |
| P5 | 78.7K | 1.03 | 61.28% |
| P6 | 51K | 1.04 | 48% |
| P7 | 36.M | 17.87 | 89.53% |
| P8 | 8.1M | 89.50 | 84.36% |
| P9 | 121M | 793.90 | 95.24% |
| P10 | 256M | 998.0 | 83.5% |

Table 2.2: Performance for exploring one hour of wall clock time and the reduction in CPU time from predicting states

temperature sensors, sprinklers, and thermostats.

The table shows the source lines of code (SLoC) and the number of VCs in the program after transformation in the first stage. Systems which we have not implemented a front end yet were transformed manually. We see that most installations have five or more VCs, indicating a heavy reliance on time. The table also shows the GCD across all constants in VC constraints in the program. The GCD can be coarsely thought of as the detail with which the program observes the passage of time. Since the size of the regions depends on the GCD, it also heavily influences the exploration time.

### 2.5.3 Exploration Performance

We ran DeLorean over all 10 programs and conduct 20 trials, each with a randomly selected starting state and time (since program behavior depends on both). All experiments used an 8 core 2.5Ghz Intel Xeon PC with 16GB RAM. Table 2.2 shows the number of transitions and average CPU time needed to explore one hour of wall clock time for each program. We estimate DeLorean makes 200k transitions per second. Since HA programs depend on wall clock time, we can also measure the CPU time with respect to wall clock time. We also see that DeLorean can explore real programs 3.6 times to 36K times faster than wall clock time.

An important element in obtaining quick explorations for these programs is predicting successor states. While this is a general optimization for dynamically exploring TAs, its effectiveness depends on how often we encounter non-similar states with identical clock personalities, variables, and enabled timers. To evaluate it, Table 2.2 shows the percentage reduction in CPU time when prediction is used compared to when it is not used. For the smallest programs, prediction leads to slower exploration. This is because in such cases the overhead associated with checking for past states that can be used for prediction is greater than any

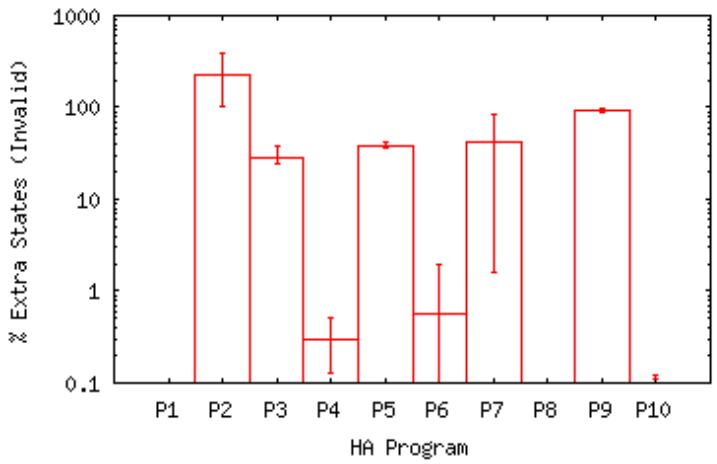



Figure 2.7: Invalid states generated by untimed exploration

benefit it brings. However, for larger programs, prediction brings substantial benefit. For P9, prediction cuts the exploration time by 95%, i.e., exploring without prediction is slower by a factor of 20.

### 2.5.4 Comparison with Alternatives

#### Untimed Exploration

As mentioned earlier, current model checkers ignore time and can thus generate invalid program states that will not be generated in real executions. If there were just a few, it is conceivable that users would be willing to put up with occasional incorrectness. However, we found that untimed exploration results in many incorrect states. Figure 2.7 shows the percentage of additional, invalid states produced by untimed model checking,[1] when beginning from the same starting state as DeLorean and running until it cannot find any new states. Untimed exploration differs from DeLorean in three aspects: *i*) in addition to successors based on device notifications, each state has successors based on each queued timer, independent of the target time of the timer; *ii*) if a comparison to time is encountered during exploration both true and false possibilities are considered; *iii*) there are no delay transitions. The graph averages results over 10 paired trials with different starting inputs, and the error bars shows maximum and minimum percentage of invalid states.

We see that untimed exploration produces a significant number of invalid states. For most programs, the number of invalid states is of the same order as the number of valid states produced by DeLorean. Closer inspection of results from untimed exploration provides insight into how some invalid states are produced. One common case is where devices such as lights are programmed to turn on in the evening, using a timer.

[1]This comparison based on invalid states alone hides one additional limitation of untimed model checking. Untimed exploration is incapable of verifying program behaviors that depend on time (e.g., light turned off a second after turning on).

Because timers can fire anytime, untimed model checking incorrectly predicts that the light can be off in the evening, which will not happen in practice. Another case is where certain actions are meant to occur in a sequence, e.g., open the garage door after key press and then close it 5 minutes later. With DeLorean, these actions are carried out in the right sequence, correctly predicting that the door is left in the closed state. But both possible sequences are explored by untimed exploration, one which incorrectly predicts that the garage door is left open.

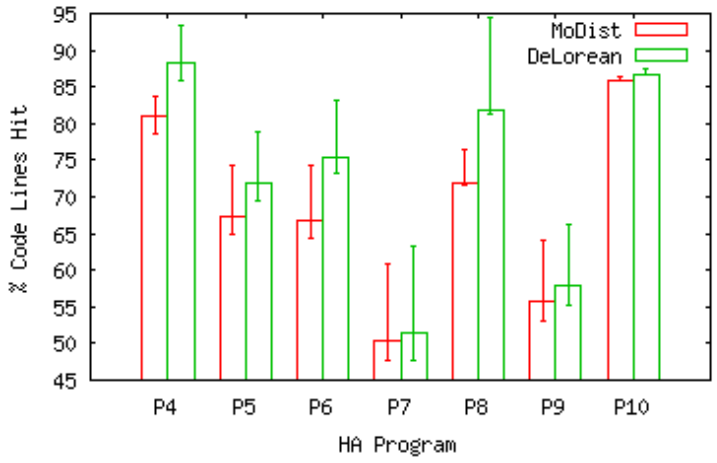



Figure 2.8: Code coverage

**MoDist**

Unlike untimed exploration, MoDist maintains temporal consistency during exploration, but at the expense of incomplete exploration (§ 2.2.1). To illustrate this, we implemented MoDist's algorithm for exploring timers in DeLorean and compared it with our exploration. We compared two metrics: state coverage and code coverage. State coverage measures the number of unique program states explored and code coverage measures the number of lines of code exercised during exploration. Figures 2.8 and 2.9 show code and state coverage, respectively, for MoDist and DeLorean averaged over 24 trials, each exploring one hour. Programs omitted in Figure 2.8 have equivalent coverage in MoDist and DeLorean.

### 2.5.5 Unintended Behaviors

To informally gauge DeLorean's ability to find such behaviors, we inspected comments in two of the programs (P9, P10) and translated them into invariants for which DeLorean should report violations. We found four violations.

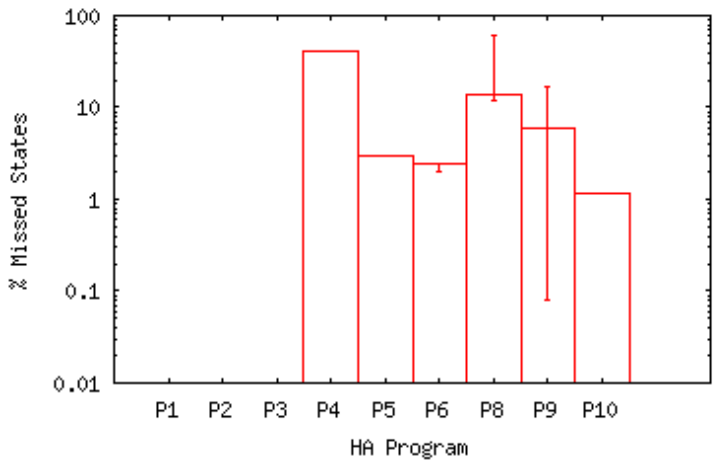



Figure 2.9: States missed by MoDist

**P9-1** A comment indicated the lights in the back of the house should turn on if motion is detected in the evening (i.e., sunset to 11:35PM). But DeLorean found that the lights could be on even if there was no motion. A rule appeared misprogrammed — instead of using conjunction as the condition to turn on the light (`sunset < Now < 11:35PM && MotionDetected`), it was using disjunction (`sunset < Now < 11:35PM || MotionDetected`)

**P9-2** A comment indicated the front porch light should stay on from a half hour after sunset until 2AM. There were two rules to implement this invariant: one turning the light on at a half hour after sunset and one turning it off at 2AM. But DeLorean found cases where the light was off in that time window. Inspection revealed another rule to turn off the light at 7:45PM. Thus, the invariant is violated if sunset occurs after 7:15PM, which can happen where the user of P9 resides. Exploring sunset values higher than 7:15PM uncovered the violation.

**P10-1** A comment indicated the user wanted to turn on a dimmer switch in the master bath room when motion is detected. But we found instances where the motion occurred but the dimmer was not on. Inspection revealed that the user's detailed intent, implemented using two rules, was to turn on the dimmer half-way when motion occurs during the day, and to turn it on fully when it's detected during night. But the way day and night time periods were defined left a 2 minute gap where nothing would happen in response to motion.

**P10-2** A comment indicated the user wanted to treat three devices identically (i.e., all on or all off). Inspection of a violation of this invariant showed that while three of the four rules that involved these devices correctly manipulated them as a group, one rule had left out a device.

| | SLoC | #VCs | GCD (s) | #trans 1 VC | #trans 2 VCs | #trans 4 VCs |
|---|---|---|---|---|---|---|
| PySwitch | 234 | 13 | 1 | 6210 | 49k | 8.8M |
| LoadBalancer | 2063 | 14 | 2 | 351k | 512k | 3.8M |
| EnergyTE | 434 | 10 | 5 | 442k | 1.7M | 21M |

Table 2.3: The OpenFlow programs we studied

## 2.6 Case Study: SDN

To further demonstrate the value of TA-based exploration, we modeled and tested SDN programs in DeLorean. Similar to NICE [27], we created a model of the NOX platform in C#, including the controller, switches, and hosts. We discover relevant packet headers during pre-exploration, using Pex to symbolically execute the event handlers that make up the OpenFlow program. Since OpenFlow switches have complex internal behavior (e.g., flow tables) that we cannot observe externally, we manually defined models of OpenFlow switches and hosts. OpenFlow programs have no dependency on absolute time, therefore we used no wall clock time.

### 2.6.1 Dataset

We evaluated DeLorean using three real programs: a MAC-learning switch (`PySwitch`), a web server load balancer[100], and energy-efficient traffic engineering (`EnergyTE`) [96]. We manually translated the programs from Python into C# for testing in DeLorean. Table 2.3 shows the source lines of code (SLoC), the total number of VCs that can be dynamically created during an exploration, and the GCD of the clock constraints. As in NICE, we bound the state space by limiting certain events, such as the number of times a host sends a packet.

Each program has dependencies on relative time. `PySwitch`, for example, uses a timer to periodically check entries in a cache of MAC address-port mappings and expire entries older than a specific time. In this case, a VC is needed to express the timer scheduling the periodic check, and another VC for each entry in the cache.

### 2.6.2 Comparison with Alternatives

We compared DeLorean to NICE, a model checker for OpenFlow programs. Similar to untimed model checking, NICE does not systematically model time. Instead, application-specific heuristics are used to trigger timers in each of the SDN applications tested. We constructed a model of the NOX controller [14] for DeLorean and simulated NICE's exploration by running DeLorean with no VCs. We also implemented

NICE's heuristics for exploring timer behavior. To informally gauge DeLorean's ability to find unintended behaviors, we created invariants from the 11 bugs discovered by NICE. We find DeLorean can reproduce violations for all 11 bugs.

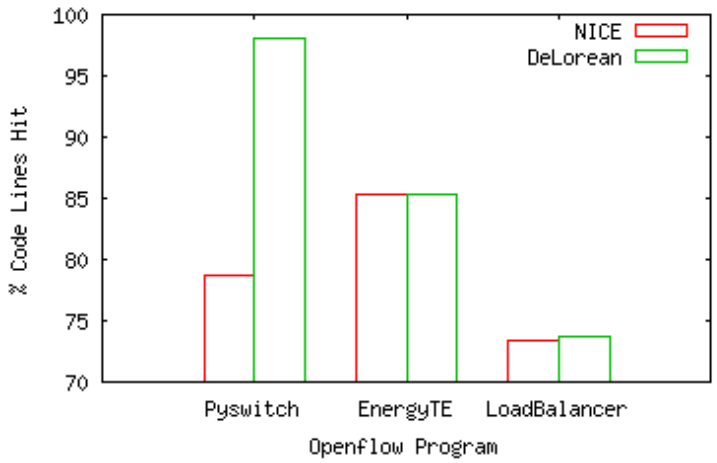



Figure 2.10: Code coverage in DeLorean and NICE

Next, we compared DeLorean's coverage of a program's state space to that of NICE. In Figure 2.10, we show the code coverage of DeLorean and NICE for the three programs. In `PySwitch`, NICE does not explore the timer for periodically checking cache entries, leaving a function unexplored.

In Table 2.4, we show the number of missed and incorrect states generated by NICE compared to explorations in DeLorean using 1, 2, and 4 VCs. Because NICE does not trigger any timers in `PySwitch`, it misses potential behavior, but does not introduce any invalid states that would be generated from timers firing at incorrect times from incorrect states The heuristic for triggering timers in the `EnergyTE` application, however, fires timers from every possible state, resulting in invalid states. Similarly, in `LoadBalancer`, timers can also fire from invalid states.

Further, NICE's heuristics do not test the expiration of flows. Correctly exploring this behavior, as well as verifying correctness properties related to flow expirations, requires more systematic treatment of time. This results in missed states in both the `EnergyTE` and `LoadBalancer` applications.

We see that with non-systematic treatment of time, program exploration can introduce false behaviors or miss potential behaviors. In programs dependent on absolute and relative time, such as HA programs, we found untimed exploration can produce too many invalid states to be useful. Even in programs with dependencies only on relative time, such as SDN programs, we see non-systematic treatment of time can also produce as many invalid states as valid states.

| | % Missing | | | % Incorrect | | |
|---|---|---|---|---|---|---|
| | 1 VC | 2 VCs | 4 VCs | 1 VC | 2 VCs | 4 VCs |
| PySwitch | 0% | 34% | 84% | 0% | 0% | 0% |
| LoadBalancer | 95% | 95% | 95% | 117% | 123% | 123% |
| EnergyTE | 69% | 87% | 97% | 26% | 12% | 46% |

Table 2.4: Missing and invalid states generated by NICE, compared to explorations in DeLorean using 1, 2, and 4 VCs

## 2.7 Observations and Future Work

A tool such as DeLorean is most useful to programs with a deep dependency on time. In particular, we found that programs that depend on both absolute (i.e., time of day) and relative time exhibit the most interesting behavior to analyze with TAs. Moreover, these types of programs tend to be harder to reason about, since one must consider the interaction of relative timers with absolute time (e.g., the bug in Figure 2.1 described in § 2.2.1). All HA programs we found had such complex interactions, while all SDN programs we examined use only relative time.

A confluence of the two domains we study is Lithium [64], which extends the control domain for SDNs to include time (both relative and absolute), history, and users, rather than only flows.[2] Policies can be defined to allow guest access to a network during certain times of day with a limit on data usage. We believe that as more "dumb" devices become network-capable, the demand for this type of control of a network will only increase, as will the importance of tools to comprehensively explore the behavior of these types of policies.

Additionally, we believe timed model checking could be a useful tool in checking performance properties in these domains. We plan to investigate this application of DeLorean in future work, as well as other contexts where time is important [28].

[2]We considered studying Lithium policies in our evaluation, but the authors' prototype did not include the implementation for any policies involving time.

# Chapter 3

# Orchestrating Applications with Database-Defined Networking

Given a set of correct control plane applications, a network operator can then focus on ensuring the applications are correct at the management plane. To do so, an operator must determine a composition plan for the applications that results in a coherent and consistent data plane. However, operations within applications may conflict with operations in other applications, requiring a prioritization across applications (or operations). Therefore, an operator may need to understand the implementation details of each application driving the network, how each will respond to every possible event, and how those responses will conflict with or trigger behaviors in the other applications controlling the network.

To help an operator compose together applications, this chapter introduces an *orchestration* [98, 99] primitive through a new approach for thinking about SDN applications. In particular, we investigate the use of a database as the building block for an SDN controller, and the power it brings to solve challenging application composition and management problems. Although prior domain-specific languages [81, 78] for SDN controllers have built on a SQL-like syntax, this is the first work to build an SDN controller entirely on top of a standard database. This work is motivated by the belief that SDN fundamentally revolves around data representation, and by adopting a relational representation of the network, we can build on a large body of database research that has applications to current challenges in SDN.

The work in this chapter was co-authored with Anduo Wang, Matthew Caesar, and P. Brighten Godfrey. It was published in the 2016 ACM Symposium on SDN Research (SOSR) [99].

This work seeks to answer the following research questions:

- With constantly changing network demands, what are the "right" programming abstractions for network applications? Can a relational representation of the network enable easier design of new abstractions and composition across them?
- Can this representation allow for easier and automated composition of multiple SDN applications?

## 3.1 Introduction

SDNs employ APIs to control switch data planes via a logically centralized controller. This provides a platform so that it is possible to write software that can control network-wide behavior. To make that programming *easy*, it is generally agreed that higher-level abstractions are needed, but there is little consensus on what are the "right" abstractions. Frenetic [44], Pyretic [78], and NetCore [77] introduce functional constructs that enable modular creation of SDN control. Flowlog [81] extends this functional abstraction to the data plane by adding SQL-like rule constructs. FatTire [89] and Merlin [91] add additional language-level support for fault-tolerance and resource provisioning. On the other hand, to address stateful middlebox and service chaining, Kinetic [63] and PGA [87] raise the abstraction level using state automata and graph expressions.

Indeed, as the network evolves over time, the need for abstractions will likely outgrow the abstractions of the present, thus requiring a continual "upgrade" that extends or raises the level of existing abstraction. Each "upgrade" incurs the tremendous effort of careful design of a new abstraction and engineering of the supporting runtime. With the deployment of drastically different abstractions, the network will be jointly driven by multiple controls created with disparate abstractions. Orchestration of these controls across many abstractions is therefore needed. Current approaches, however, only offer a fragmented solution: higher-level coordination often depends on and is restricted to the use of certain abstractions [78, 87, 63], while abstraction-agnostic support depends on common structures (e.g., OpenFlow rules, network state variables) that are usually low-level [76, 93, 56].

To this end, we champion a perspective that SDN control fundamentally revolves around data representation. We discard any application-specific structure that might be outgrown by new demands. Instead, we adopt a plain data representation of the entire network — network topology, forwarding, and control applications — and seek a universal data language that allows application programmers to transform the primitive representation into any high-level representations to present to applications or network operators. Driven by this insight, we present a system, Ravel, that implements an entire SDN network control infrastructure within a standard SQL database. We take the entire SDN network control system under the hood of a standard SQL database, and rely on SQL for data manipulation, the database runtime for data mediation, and propose a novel protocol that refines the database runtime to enforce only orchestrated execution.

Furthermore, by adopting a relational representation of the network, we can build on database research to solve challenging SDN problems. In particular, we formulate the application composition problem as a data integration problem. In doing so, we can build on database research in irrelevant updates [22, 70, 38] to discover dependencies between applications.

We present a realization of our approach, Ravel, which offers attractive advantages:

**Ad-hoc programmable abstractions via database views.** The database *view* construct enables new abstractions to be constructed ad-hoc and enables them to build on each other. For example, from the "base tables" defining the network topology and OpenFlow forwarding rules, one can construct a view representing a virtual network that spans a subset of the topology; and one can further derive a load balancer's view of host availability within the virtual network. These SQL views form the structure of the abstractions. Furthermore, integrity constraints over the views express high-level policy invariants and objectives. These views and constraints can be expressed via SQL statements and can even be constructed ad-hoc, i.e., dynamically in the running controller.

**Orchestration across abstractions via view mechanisms.** Once we have a mechanism to construct ad-hoc abstractions, we need to coordinate across these multiple views of a single network. View maintenance [47] and update [18, 23, 40, 32, 60] mechanisms are the "runtime" for view abstractions, constructed through normal SQL operations (i.e., queries and updates). First, *view maintenance* continuously refreshes the view abstractions of a dynamic network. Second, to translate updates in a derived view back down to a lower-level view, Ravel allows users to define a view update policy that governs how updates on the higher-level abstraction are realized on the underlying data plane via *triggers*, which can incorporate custom heuristics at runtime to optimize applications.

**Orchestration across applications via a data mediation protocol.** In Ravel, an orchestration protocol mediates multiple applications whose database modifications affect each other. The protocol assumes a simple conflict resolution strategy — an ordering of view constraints where lower-ranked constraints yield to the higher-ranked. Given a view update request as input, the protocol produces as output an orchestrated update set that respects all applications constraint (subject to conflict resolution). The orchestrated set may append to the request additional updates for completion (e.g., invoking a routing application when an access control application attempts to remove an unsafe path) or reject the update if a cohesive update is not possible.

**Automated orchestration with irrelevance reasoning.** Building on our relational representation of the network, we extend our data mediation protocol and introduce *dependency reasoning* to automatically determine dependencies between applications. Using these dependencies, Ravel can automate orchestration by searching for conflict-free compositions plans.

**Network control via SQL.** The entire Ravel system above is exposed to application programmers and network engineers via standard SQL interfaces and database view, constraint, and trigger mechanisms. We

believe this is likely to be valuable both technically and as a way to encourage more rapid uptake of SDN. SQL databases have proved over decades to be an effective platform for high-level manipulation of data, and are broadly familiar (e.g., compared to domain-specific languages). Moreover, network architects today need to combine heterogeneous data sources — network forwarding rules, data flow and QoS metrics, host intrusion alerts, and so on — to produce a cohesive view of the network and investigate problems. This could be eased by the interoperability of SQL databases.

We note that certain past SDN controllers have employed databases as specific limited modules, including state distribution, distributed processing, and concurrency and replication control. For example, Onix [66] delegates to distributed databases for concurrency and replication of low-level network state in a common pre-defined data schema. The database silently accepts and executes transactions submitted by external control applications. Declarative networking [72, 71, 74], on the other hand, uses a distributed query engine for fast processing of customized routing protocols, where the database executes routing queries submitted by end-hosts. In contrast, Ravel makes the database an active participant that uses views to incorporate multiple high-level and low-level abstractions of the network, which are orchestrated online. In other words, in Ravel, the database *is* the controller.

While this is an ambitious long-term goal, our Ravel prototype built on the PostgreSQL database [12] exhibits promising performance even for large scale networks. On fat-tree (up to 5,120 switches / 196,608 links) and ISP (up to 5,939 nodes / 16,520 links) topologies, we microbenchmark Ravel delay, uncovering the sources of database overhead, showing that the database operations and orchestration of the various applications all scale to large network and policy sizes. On a physical SDN network, we find Ravel can translate changes in the database to flow modification commands within 1ms and install the change in switch flow tables within 50ms. On real-world firewall rules, Ravel control applications can maintain service chain sizes up to one million within 11 seconds, and handle queries in <1ms. Our Ravel prototype also integrates into the database runtime a classic view maintenance algorithm that automatically translates an arbitrary user view into an equivalent high-performance materialized table. This improves access on views by one to two orders of magnitude with a small maintenance overhead on the materialized table. To the best of our knowledge, Ravel showcases the first high-performance database controller over a hardware OpenFlow network, and our evaluation presents the first comprehensive feasibility study of such a controller.

## 3.2 Motivation

### 3.2.1 A Motivating Example

Consider the task of managing an enterprise network with SDN-enabled abstractions. In the beginning, network control is simple end-to-end communication policies in the form of routing and security (e.g., a firewall). As the business grows and the organization splits into several independent compartments including human resources (HR) and research and development (R&D), the network desires additional support to enable the compartments to jointly drive the network. Specifically, each compartment could independently create, modify, and control a (potentially overlapping) sub-net of the enterprise network. The compartments also desire a more natural and concise expression of end-to-end communication. Rather than explicitly enumerating all pairs of individual endpoints, the compartments need an aggregated expression for policies between endpoint groups, the member endpoints of which are under the same policy due to a shared property.

### 3.2.2 Enlarging and Fragmented Abstractions and Orchestration

#### Abstractions

Initially, the requirement for network abstraction is only concerned with high-level control for basic forwarding, including routing and access control through a firewall. It is thus natural to use *Pyretic* [78], which uses a programming framework that features a forwarding policy as function abstraction and a runtime that compiles this abstraction into lower-level OpenFlow representations. However, Pyretic lacks a language-level construct to directly control a firewall, or middleboxes in general. Instead, we add an additional layer for this purpose, namely *Kinetic* [63]. Kinetic adds state automata abstractions to Pyretic that allow control of a firewall (or any stateful middlebox) through the state automata construct, and extends the runtime to compile the state automata policy into lower-level Pyretic code.

Next, as the need for multiple compartments to independently control endpoint groups outgrows the capacity of Pyretic and Kinetic, a new abstraction found in PGA [87] can be added. Like Kinetic, PGA raises the abstraction level of Pyretic with constructs for service chain graphs over groups, as well as a runtime that analyzes and merges the service chains. At this point, our repository of abstraction grows into a list of disparate constructs — functions, state automata, and chain graphs — as shown in Figure 3.1 (left).

While enjoying the higher-level control from each construct, we are faced with the problem of making them coexist. Integration of Pyretic and Kinetic and likewise Pyretic and PGA is straightforward. As Kinetic and PGA are Pyretic-extensions that add higher-layer constructs, they are "user-space" tasks that do not require understanding of the system internals. However, making Kinetic and PGA coexist requires more

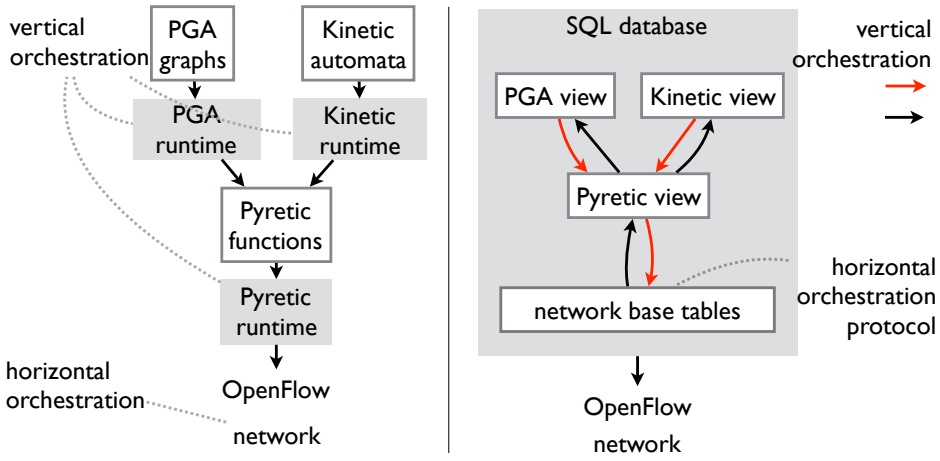



Figure 3.1: Abstraction and orchestration: current approach vs. Ravel

work — a "system-space" job that requires re-writing the internals of the two systems. In the case of Kinetic and PGA, one has to modify the systems to feed Kinetic outputs (i.e., intermediate Pyretic representation of the middleboxes) to PGA for service chain analysis and merging. Additional syntax-sugar wrapper that eases the usage of two drastically different constructs might also be desirable.

As long as an abstraction depends on a set of pre-compiled, fixed constructs, it is likely that even the best design of the present will be outgrown by the requirements of the future. Researchers will have to develop an ever-growing body of abstractions (and supporting runtimes), and network operators will have to continually upgrade their system to the latest one. Depending on the nature of the newly added abstraction, the upgrade can incur additional integration overhead for re-wiring system internals.

### Orchestration

Because the enterprise network can be collectively driven by multiple controls expressed with a diverse set of abstractions, we need proper orchestration across these abstractions.

First, a control policy can correspond to multiple layers of abstractions. For example, a security policy controlling access through a firewall could be specified by an operator in the form of a PGA graph expression. Eventually, it will be compiled by the PGA runtime into intermediate Pyretic functions. Finally, it will be implemented by the Pyretic runtime as OpenFlow rules. We term this translation of the same policy across multiple abstraction layers as *vertical orchestration*. Vertical orchestration with existing abstractions depends on the underlying runtime of each abstraction involved.

Second, multiple policies can also simultaneously direct overlapping flow space on shared network resources. Consider scenario *(A)* of a flow bypassing a firewall. This behavior can be jointly controlled by

a Pyretic routing application and a PGA service chain. In a more interesting case *(B)*, suppose servers within a network accept in-bound flows from external hosts only if the server initiates the connection. Such a policy can be achieved using a Kinetic automata and a PGA graph, where the Kinetic automata specifies the stateful firewall behavior while the PGA graph directs all in-bound flows through the Kinetic firewall. Besides these collaborative scenarios, controls can also compete for resources. For example, in scenario *(C)*, consider two compartments simultaneously migrating networks that may overload overlapping switches and links. This undesirable race condition is not addressed by any participant's control logic, and must be detected and resolved. Overall, we term the coordination of multiple abstractions to form a consistent packet processing behavior as *horizontal orchestration*.

While a generic system exists, it can operate at a common level that is often too low-level for a network operator. For example, CoVisor [56] merges control on OpenFlow rules and Statesman [93] depends on hard-wired network state variables. Therefore, existing abstractions often provide some degree of orchestration within their system scope, such as modular composition of abstraction constructs. However, depending on the nature of the orchestration scenario, it is likely that operators will have to rely on an ensemble of fragmented facilities, ranging from language-level composition at compile time to specialized state management frameworks at runtime. For example, in *(A)*, we can leverage Pyretic's modular composition facility to combine routing and access control, given that both component modules are developed in Pyretic. In *(B)*, which crosses two system boundaries and involves two distinctly different abstractions, additional integration is needed to wire control together. For *(C)*, one must use a low-level state management system like Statesman.

### 3.2.3 Our Approach: An Open and Consolidated Solution using SQL

To address the limitations found in today's network abstractions, Ravel discards any fixed abstraction constructs that are bound to application features. Instead, Ravel leverages a unifying system — a standard SQL database — to provide ad-hoc extensible abstractions that are sufficient for the diverse needs of the present and the future, and offer an out-of-box high-performance runtime that creates an insertion point for orchestration protocols.

#### Abstractions

To support a variety of abstractions without dependencies on specialized constructs, Ravel uses a plain data representations and a universal data language, namely database relations and SQL statements.

As shown in Figure 3.1 (right), the various forms of abstraction (e.g., Pyretic functions, Kinetic state automata) all take the form of plain database relations. The relations are defined using simple, familiar SQL

statements. Ravel pre-defines a network base (i.e., stored base tables) that capture the data plane, including topology, forwarding behavior, and traffic (§ 3.4). Using SQL queries over this base, Pyretic functions and PGA graphs become relations with schemas that capture the function domain, codomain, and graph entities, respectively. To capture Kinetic state automata, we first create a relation whose schema fits the automata states, and then augment the relation with triggers that simulate state transitions. Note that while the Pyretic view is built using SQL queries directly from the base tables, PGA and Kinetic views are queries over the Pyretic view. In general, application views are used as normal base tables, meaning each application enriches the set of data that can be used to create new abstractions.

#### Orchestration

While vertical orchestration with existing techniques requires a variety of disparate runtimes, in Ravel it is handled by the unifying view mechanism. As shown in Figure 3.1 (right), the two separated compilation processes from the PGA graph and Kinetic automata to Pyretic are reduced in Ravel to view updates. Here, changes on a higher-level PGA/Kinetic view are pushed down to the lower-level Pyretic view. Updates to the Pyretic view are translated to those on the base tables. A key challenge in Ravel is horizontal orchestration that involves multiple interacting controls, a scenario unusual in traditional database. For example, to coordinate PGA and Kinetic policy changes, Ravel needs to coordinate the corresponding view updates. To this end, Ravel enhances the database with a mediation protocol that refines the runtime execution in a manner that the underlying network only transitions between consistent states (§ 3.5).

## 3.3 System Architecture

Figure 3.2 shows the main components of Ravel: the users, the Ravel runtime, and the network. The network includes the services and resources being controlled. The users include network operators and a Ravel administrator that interact with the network via the view interface. The Ravel administrator can modify the behavior of Ravel, such as enriching the control abstractions by adding new views, or changing the orchestration behavior (i.e., the database execution model) by adding new protocols. These interactions all take the form of normal SQL statements: queries, updates, and triggers.

The main body of Ravel is the runtime, which sits in the middle between the users and the network. It is the engine that enables multiple controls to jointly drive the network. The runtime consists of two components. The database component interfaces with the users and is the brain that controls the variety of abstractions. The network component interfaces with the network, to bridge database events (i.e., table

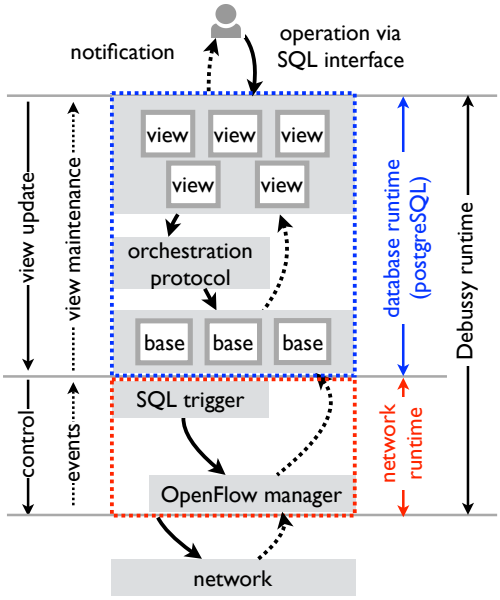



Figure 3.2: Ravel overview

updates) and external network events (e.g., control messages and network changes).

The database runtime operates on a set of tables — the pre-defined base tables (§ 3.4) and a hierarchy of user-defined control application views. The base tables interact with the network runtime via SQL triggers and with users via SQL queries and updates. The database runtime offer two services: vertical orchestration that *executes* individual application control, and horizontal orchestration that *coordinates* these executions.

Vertical orchestration is implemented using a view mechanism. View maintenance automatically refreshes application views, while view updates translate application updates to network controls. Modern databases are capable of automatic view update for simple cases. For complicated scenarios that involve ambiguity (e.g., a reachability view update corresponds to multiple viable path assignments in a lower base table), the operator can direct the database via a trigger before deployment, or alter the trigger behavior at runtime.

Horizontal orchestration is implemented using a mediation protocol. The protocol assumes a priority ordering among the control applications. Upon an initial control request that attempts to modify the base data, the protocol consults all higher ranked applications that are affected, allowing them to check the proposed update and make modifications according to its local logic. The resulting updates are combined and populated back to the lower ranked applications. Finally, the orchestrated updates are grouped into a transaction committed to the base tables.

The network runtime is controlled via database triggers. Database updates that result in changes to switch flow tables invoke an OpenFlow manager (via a SQL trigger) that pushes the change to the network. Changes to network state, such as the removal of a link, are passed on to the database through this manager.

## 3.4 Abstractions

**Shared Network State: The Base Tables**

Ravel takes the entire network under the hood of a standard relational database, where the network states and control configurations are organized in two levels. At the lower level is the tables storing the shared network state and at the higher-level is the virtual views pertaining to each application. We consider the network tables as part of the base. Ravel pre-defines the network base — a set of stored database tables — based on our extensive study of state-of-the-art control applications. The base table schema design is as follows:

```
tp(sid,nid)          # topology
rm(fid,sid,nid,vol) # end-to-end reachability (matrix)
cf(fid,sid,nid)      # configuration (forwarding table)
```

`tp` is the topology table that stores link pairs `(sid,nid)`. `rm` is the end-to-end reachability matrix that specifies for each flow (`fid`) the reachability (from `sid` to `nid`) and bandwidth (`vol`) information. `cf` is the flow table that stores for each flow (`fid`) the flow entries (i.e., the next hop `nid` for the switch `sid`). All attributes are symbolic identifiers with integer type for performance. For each attribute, Ravel keeps an auxiliary table that maps to real-world identifiers (e.g., an IP prefix for `sid` and `nid` in `cf`).

Ravel's base tables provide fast network access and updates while hiding the packet processing details. They interact directly with the underlying network via OpenFlow. Initially, the base is loaded with the network configurations (e.g., topology, flow tables). As the network changes, the Ravel runtime updates the base (e.g., switches, links) accordingly. Similarly, as the base entries for flow rules change due to network control produced by the application, the runtime sends the corresponding control messages to the network.

**Application-Specific Policy: SQL Views**

While the shared base talks continuously to the network, a separate higher-level permits abstraction pertaining to individual applications. By running SQL queries on the base, SQL views derive the data relevant to individual applications from the base, and restructure that data in a form that best suits the application's logic. SQL views allow a non-expert to add abstractions on demand, and the resulting views are immediately usable — referenced and updated via the database system runtime (§ 3.5). This frees users from committing to a fixed "right model that fits all", thus making Ravel abstractions ad-hoc extensible as requirements change or evolve.

For example, a load balancer abstraction is a view `lb` defined by the following SQL query.

```
CREATE VIEW lb AS (
  SELECT nid AS sid,
         count(*) AS load
  FROM rm
  WHERE nid IN server_list);
```

SQL views are also *composable*, like function composition via procedure call. A composite view is built by a query that references other views. In a SQL query, views are indistinguishable from normal tables. Imagine a tenant network managed via view `tenant_policy` as follows:

```
CREATE VIEW tenant_policy AS (
  SELECT * FROM rm
  WHERE host1 IN (SELECT * FROM tenant_hosts) AND
        host2 IN (SELECT * FROM tenant_hosts));
```

`tenant_policy` monitors all traffic pertaining to the tenant slice. To manage load in this slice, we can build a composite view `tlb` from `tenant_policy`, just like `lb` is built from the base `rm`:

```
CREATE VIEW tlb AS (
  SELECT sid,
         (SELECT count(*)
          FROM tenant_policy
          WHERE host2=sid) AS load
  FROM tlb_tb);
```

**Control Loop: Monitoring and Repairing Policy Violations with Rules/Triggers**

SDN application dynamics typically follow a control loop: the application monitors (reads) the network state against its constraints, performs some computation, and modifies (writes) the network state according to its policy. For example, a load balancer checks traffic on servers of concern and re-balances traffic when an overload occurs (i.e., the load on a server exceeds a threshold `t`).

Ravel allows a natural translation of application policy as integrity constraints over view data and the control loop that maintains the policy as SQL rule processing, in the form:

```
ON event DO action WHERE condition
```

For the load balancer, its invariant is a simple constraint `load < threshold (t)` over `lb`. Thus, we monitor the violations with a query on `lb` against this constraint:

```
CREATE VIEW lb_violation AS (
  SELECT * FROM lb WHERE load >= t);
```

To repair violations, we simply empty the `lb_violation` view with a `DELETE` operation on `lb_violation`. We introduce the rule `lb_repair` to translate the repair onto the actual operations on `lb`, which simply sets the load to the default threshold:

```
CREATE RULE lb_repair AS
  ON DELETE TO lb_violation
  DO ALSO (
    UPDATE lb SET load=t WHERE sid=OLD.sid);
```

Another example action is to move two units of flow away from a particular server with id `k`:

```
UPDATE lb
  SET load = (SELECT load FROM lb WHERE sid=k) - 2
  WHERE sid=server_id;
```

## 3.5 Orchestration

### 3.5.1 Vertical Orchestration via View Mechanisms

Vertical orchestration is concerned with synchronizing the views and base by leveraging two view mechanisms. View mechanisms power view-based network control on individual applications: view maintenance populates base dynamics to the views, and view update translates view updates to the base. View maintenance is well-established, with a family of fast incremental algorithms developed over a decade [107, 48, 47]. Ravel implements classic view maintenance algorithms [48, 47] that automatically optimize application views for faster access (described in § 3.8 and § 3.9.1). An interesting use of view maintenance is that an application can ask interesting questions (e.g., a summary or aggregate) by submitting a SQL query on its view. For example, to find overloaded servers exceeding some threshold `t`, an application can use the query:

```
SELECT sid FROM lb WHERE load > t;
```

Conversely, view update pushes the changes on the view back to the base. That is, changes requested by the application are propagated to the network via this mechanism. [View update is a difficult and open research problem [60, 18, 23] because an update may correspond to multiple translations on the underlying base. To disambiguate, Ravel relies on user input for an update policy. For example, consider an update on `lb` that re-balances traffic to randomly-chosen, lightly-selected servers. Specified by the following actions, it reduces the load on a server from `OLD.load` to `NEW.load` by picking a server with lowest load and redirecting the `OLD.load-NEW.load` oldest flows in `rm` to that server.

```
UPDATE rm
  SET nid =
    (SELECT sid FROM lb
     WHERE load = (SELECT min (load)
                   FROM lb LIMIT (OLD.load - NEW.load))
     LIMIT 1)
  WHERE fid IN
     (SELECT fid FROM rm WHERE nid = NEW.sid
      LIMIT (OLD.load - NEW.load));
```

In general, users can program an update policy with rules of the form:

```
CREATE RULE view2table AS
  ON UPDATE TO view
  DO INSTEAD UPDATE TABLE --- actions ---
```

Here, `action` can be an arbitrary sequence of SQL statements, or a procedure call to external heuristics. This has the benefit of making Ravel abstractions open. Users can dynamically control the implementation of its high-level policy — how the view update maps to the underlying network — by simply creating or modifying the associated rules.

Similar to views derived directly from the base tables, nested views can be updated to manage the underlying network states. The update policy only needs to specify how changes on the view are mapped to its immediate source. For example, `tlb` only needs an update policy on `tenant_policy`, from which it is derived, and `tenant_policy` will handle the mapping down to the network base tables.

```
CREATE RULE tlb2tenant_policy AS
  ON UPDATE TO tlb
    DO INSTEAD
    UPDATE tenant_policy ...;
```

### 3.5.2 Horizontal Orchestration via Mediation Protocol

Eventually, all control application operations are translated to updates on the network base data. To integrate the individual controls into a consistent base, it is sufficient to control their updates on the shared base. To this end, Ravel enhances the database runtime with a mediation protocol that instructs the shared data access.

The mediation protocol offers participating applications three primitive operations. *i*) *Propose*: an application attempts a tuple insertion, deletion, or update to its view. *ii*) *Receive*: an application observes updates (i.e., side-effects) reflected on its application view due to network updates initiated by another participant. *iii*) *Reconcile*: an application checks the received view updates against its policy and performs either accept (no violation), overwrite (to resolve a conflict), or reject (no conflict resolution exists). Conflict occurs when updates made by one application violate the constraints of another. Ravel adopts a simple conflict resolution policy based on priorities. Applications are required to provide a global ranking among all the constraints, and one application can have multiple constraints. Higher-ranked constraints can then overwrite updates from lower-ranked constraints if a conflict occurs.

Starting from a consistent network state where all application policies (i.e., invariants) are satisfied, the protocol takes an proposed update and a globally agreed upon priority list, producing a set of orchestrated updates as follows. First, Ravel computes the effects of the proposal on other applications. A view up-

date affects another view if the corresponding update on the shared data item causes modification to that view. Next, the affected applications sequentially reconcile their received update against their constraints in increased ranks. Finally, all reconciled updates are merged and form the orchestrated output set. The orchestrated output is applied atomically as one transaction in the database, which transforms the network from its current state to the next.

**Correctness**

Ravel orchestration handles two scenarios. In one, applications are called to collaborate for a common task (e.g., when a firewall drops an unsafe path, the routing component shall be invoked to remove the corresponding path). In the other, independent applications are prevented from inadvertently affecting each other (e.g., traffic engineering and device maintenance). Combined, the goal is to avoid partial and conflicting updates that lead a network into an inconsistent state. Formally, a network state is consistent if it is compliant with all the application policies. That is, consistent network states correspond to (derive) applications with invariant-preserving views. A set of network updates potentially contributed by multiple applications is correctly orchestrated if the set transforms a consistent (i.e., correct) network state into only another consistent states.

**Proposition 3.1.** *Ravel orchestration preserves network consistency.*

*Proof.* (Sketch) The data mediating protocol translates each application update into a set of updates (contributed by all relevant applications) which, combined, satisfy all application constraints (subject to conflict resolution), thus preserving network consistency. □

## 3.6 Guiding Orchestration with Dependency Reasoning

Next, we extend our notion of orchestration and build on logic reasoning to automatically synthesize part of the control logic when composing applications. The modular design of SDN — where multiple control modules collectively realize all management objectives — still requires an operator to understand a member module's internals to determine the event-level interactions in all execution runs. To address this challenge, we characterize the composition problem as a *data integration* problem of merging data sources into an integrated whole. We use this formulation to introduce *dependency reasoning*, which provides a logical characterization of the finer-grained interactions between control modules. These interactions, or *dependency relations*, include: *independence*; *dependence*, where one control module requires further action from another; and *codependence*, where divergence, conflicts, and refinement to dependent cases can occur.

By building on database irrelevance reasoning, Ravel can automate dependency reasoning by reducing it to a satisfiability problem. Ravel also extends prior satisfiability formulations to include aggregates, which provides useful networking semantics (e.g., `count` for load balancer). The extension is inspired by a novel rewriting technique, inspired by incremental view maintenance, that transforms an aggregate into an incrementally maintained "normal" function [48].

In addition to supporting SDN management at design phase — when a preset control logic based on domain knowledge is assumed [42, 78] — we extend Ravel to existing controller implementations with a pure syntactic translation. Without requiring additional user annotations, Ravel automatically extracts the formal model for automatic dependency reasoning, and maps the output logic back to the original platform. We use the Pyretic platform as an example: we formally identify a core sub-language of Pyretic, and recursively build the mapping between Pyretic objects, the Ravel model, and the output logic.

### 3.6.1 Motivating Examples

#### Automating Pyretic Composition

To keep up with the complexity of SDN management, it is generally agreed that modular programming with high-level network objects plays a key role. While [78, 44] provide excellent high-level modular platforms, they do not enforce a clean module-level composition interface. For example, the simple case [78] of a firewall (`afw`) and load balancer (`alb`) requires the operator to foresee the subtle difference in composition (`afw>>alb` versus `alb>>afw`):

```
fwlb = if_(from_client, afw >> alb, alb >> afw)
```

An operator will need to correctly identify the composition patterns that correspond to all possible executions, which often depend on the traffic being processed. Thus, the correct composition, or *master program*, still relies on the operator's understanding of the internals of each individual module and the finer-grained, "beneath-module" interactions. [87] calls this the "decompose and re-compose" problem, and shows that additional requirements can quickly add complexity [1].

To this end, Ravel aims to synthesize a correct composition by automatic, yet finer-grained dependency reasoning. The challenge is to find a syntactic translation that automatically extracts a formal model from a Pyretic program and converts the reasoning result back to Pyretic composition.

[1] In fact, this motivates PGA [87] to automate composition of static service chains

**Preventing Security Violations**

While SDN enables flexible programmable control, it also presents a security challenge. Malicious or innocent-but-careless users can inject controller programs that breach security. Consider a firewall that blocks communication between $(h_1, s_1)$, dropping packet with header $(src : h_1, dst : s_1)$. A rewriting program that combines two seemingly legitimate rules $(h_1, s_2) \rightarrow (h_2, s_2)$ and $(h_2, s_2) \rightarrow (h_2, s_1)$ will breach the firewall, allowing traffic flow from $h_1$ (with header $(h_1, s_2)$) to $s_1$. FortNOX [86] formulates such security breaches as OpenFlow rule conflicts, and introduces an *alias set* to detect the conflicts through runtime checking of OpenFlow rule updates. Like other runtime efforts, FortNOX is postmortem and requires exhaustively checking every update. More importantly, FortNOX, which operates on low-level OpenFlow rules, does not understand higher-level control logic. Thus, for a particular detected breach, FortNOX cannot shed light on the problematic control module.

To this end, Ravel aims to prevent security breaches via logic reasoning on the interactions between controller programs. The strength of Ravel is the performance gained by non-exhaustive static analysis and the ability to reveal the conflicting control logic prior to deployment. The challenge is to fold the alias set into automatic dependency reasoning.

**Synthesizing Content-based Dynamic Policies**

To extend the SDN paradigm to full-featured middleboxes, the FlowTags system [42] introduces a tag-extension that realizes complex middleboxes policies. However, the correct interaction among middleboxes can be tricky. For example, between a group of internal hosts and some external server, one may want to place a proxy and a firewall. The proxy improves host experience by caching server contents for faster responses, while the firewall controls communication between the host and the server. The subtlety is that the proxy has to take into account the decisions of the firewall, otherwise a to-be-blocked host might still be able access the data in the proxy which is previously cached by a legal host. FlowTags assumes a preset specification — a dynamic policy graph (DPG) — that correctly recognizes such subtle context-dependent dynamic interactions[2].

To this end, Ravel aims to synthesize a DPG that correctly knits together the middleboxes. The key challenge in doing so is synthesizing the context-based dynamic dependency.

[2]Automatic generation of DPG specification is described as future work of FlowTags.

### 3.6.2 Composition as a Data Integration Problem

To tackle this application composition problem in SDN, we argue we can formulate many network management tasks as data integration problems of merging data sources into an integrated whole. In our formulation, the entire network is stored as relational data. We assign distinct roles, where the control applications serve as the individual data sources that produce network data, and the data plane becomes the integrated whole.

*Data integration* [68, 95, 69] examines the problem of combining data from a variety of sources to form a new, unified whole. More formally, a data integration system is defined as $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, with a global schema $\mathcal{G}$, semantic mappings $\mathcal{M}$, and data sources $\mathcal{S}$. The system exposes to users a single and coherent interface, through a mediated global schema $\mathcal{G}$. Users can query and access data from local sources while the system hides the details and inconsistencies of the constituting sources. Semantic mappings $\mathcal{M}$ specify the relationship between the schemas of the data sources $\mathcal{S}$ and the global schema $\mathcal{G}$. The semantic mappings allow answering queries over $\mathcal{G}$ using queries over $\mathcal{S}$. We consider data sources that are *relational*.

There are two approaches to (virtual) data integration, which differ in the way $\mathcal{M}$ is defined: *global-as-view* (GAV) and *local-as-view* (LAV). In GAV, the global tables are defined in terms of the local tables. Each global table is thus a *view* of the local tables. In LAV, $\mathcal{M}$ is defined in a somewhat dual manner: the local tables are defined in terms of the global tables. Hence, each local table is a *view* of the global tables.

An important problem in a data integration system is the reconciliation of inconsistencies between the data sources. Inconsistencies are captured by integrity constraints that specify what data instances are acceptable. Integrity constraints can be specified directly over the global schema, or expressed on the individual sources and eventually expanded and unfolded on the global schema. The constraints can accelerate data access (i.e., reads) in a read-dominant $\mathcal{I}$, but they deteriorate $\mathcal{I}$'s performance when faced with frequent updates (i.e., writes) due to the overhead of checking the integrity constraints.

In SDN, we define *network integration* as the problem of enabling multiple independently created, potentially overlapping and/or conflicting control applications to collectively drive a network in a coherent manner. Using Ravel, where the entire network state (i.e., the control applications and data plane) is modeled as relational data, we can cast this problem to a data integration problem and define a network integration system as $\mathcal{I}^{\mathcal{N}} = \langle \mathcal{G}^{\mathcal{N}}, \mathcal{S}^{\mathcal{N}}, \mathcal{M}^{\mathcal{N}} \rangle$. The global schema $\mathcal{G}^{\mathcal{N}}$, augmented with integrity constraints, characterizes a consistent data plane. The source schemas $\mathcal{S}^{\mathcal{N}}$ describe network states contributed by independently operated control applications. The goal of network integration is to find a mapping $\mathcal{M}^{\mathcal{N}}$ that synchronizes between the control applications' states and the network data plane under the integrity constraints.

### 3.6.3 Behavioral Model

The intended behavior of a task is governed by the control logic of the corresponding module. Depending on the nature and development stage of the task, an operator may use different control abstractions to describe or program the task. For a basic forwarding task, Pyretic [78] offers a functional abstraction that enables modular programming. For a stateful task with correctness requirements, Kinetic [63] offers a verifiable language construct based on state machines. Though seemingly different, the behavior of most control modular follows a common pattern: a continuous control loop of monitoring network states, followed by performing a local computation, and then reconfiguring the network.

We make one further abstraction. Let the requirement (i.e., intention) of the task be denoted by some invariants, captured by a collection of violation views. Conceptually the local computation will serve two roles: measuring the network state to detect invariant violations; and reconfiguring the network to repair any violations. Thus, we model a control unit — an cycle of measuring and reconfiguring one invariant — by a pair of datalog-like rules $q, r$.

```
q v(X',Y',...) :- net1(X), net2(Y),..., vc(X,Y,...,Z)
r u_i(X') :- v(X,Y,...), uc(X,Y,...,X');
  u_d(Y') :- v(X,Y,...), uc(X,Y,...,Y'); ...
```

The $q$ rule tests the network states `net` against violation condition `vc`, outputting the violation view `v`. `X,Y,...` are vectors of network attributes, and `Z` is the module-specific attributes[3]. The $r$ rule computes network reconfigurations, i.e., updates (either an insertion '`u_i`' or a deletion '`u_d`') to *net* tables that will restore the violation view to empty.

For example, the behavioral model for the previously discussed firewall module is as follows. The $q$ rule says a flow in `rm` is violating the firewall policy if its source and destination (`src,dst`) match the local computation (`block`). The $r$ rule models the violation repair, which removes `rm` rows that are detected in the violation view `fw_v`.

```
q fw_v(fid) :- rm(fid,src,dst), block(src,dst)
r rm_d(fid,src,dst):- fw_v(fid)
```

#### Complete Model and Assumptions

**Definition 3.1** (Network Model)**.** *A network collectively driven by a set of management tasks is a 5-tuple* $(N, V, U, q, r)$*, where* $N$ *is a finite set of network tables,* $V$ *is a finite set of violation views,* $U$ *is a finite set of update tables to* $N$*,* $q = \{q_k | q_k : N \to v_k, v_k \in V\}$ *is the measuring rules, and* $r = \{r_k | r_k : v_K \times N \to u_k, v_k \in V \wedge u_k \in U\}$ *is the reconfiguring rules.*

[3]Technically, rule $q$ joins `net` and `vc`, and selects rows in `net` that fail the test (condition `vc`)

A snapshot of the *network state* is denoted by the tuple $(N, V, U)$. As the management module measures and reconfigures the network, the network state evolves. A sequence of network states is called an execution run. A network is *consistent* if it is compliant with all invariants, i.e., when $\forall v_k \in V, v_k = \varnothing$. As shown in Figure 3.3 (left), we call the individual set $(q_k, v_k, r_k, u_k)$ that continuously measures and reconfigures the network a control unit. In general, a control module can involve multiple distinct control units. For example, a NAT module may contain two units, one for traffic arriving on the public face and one for the private face.

We make two assumptions when a module involves multiple control units. First, the invariants that define each control unit's violation view are disjoint. Second, for any two control units $i, j$ in the same module, a reconfiguration by one $(u_i)$ will not introduce new violations to another $(v_j)$. Thus, we assume each module is capable of partitioning and organizing its concerns into *disjoint* and *coherent* control units. However, control units across modules can overlap and contradict, resulting in complex interactions.

### 3.6.4 Automated Reasoning

Consider the management tasks described in § 3.6.1. Despite their apparent differences — diverse abstraction constructs (functions, graphs, rules), different stages in development (design, implementation, runtime), and the varying nature (forwarding, middleboxes, security) — the recurring theme is the search for a correct control logic that maintains network consistency.

The challenge is that the control units across the modules are collectively driving the same shared network, and exhibit finer-grained complex interactions. Our goal is to formally characterize and statically determine the various types of control unit-level dependencies that are necessary in forming a consistent management plane.

### 3.6.5 Behavioral Dependency

#### Notations

As illustrated in Figure 3.3 (right), suppose we have two control units $x = (q_k, v_k, r_k, u_k)$ and $y = (q_y, v_y, r_y, u_y)$. We use the term *active* to describe a control unit when its violation view is nonempty. A control unit is active if the network is in an inconsistent state that violates its invariant. We say the control unit $x$ *activates* $y$, denoted by $x \to y$, if in some network states, the network updates contributed by $u_x$ can cause changes to $v_y$, potentially introducing new violations to $y$. We say $x$ does not activate $y$, denoted by $x \not\to y$, if in all network states, reconfigurations from $x$ leave $v_y$ unchanged. When $x$ activates $y$ $(x \to y)$, we draw a directional arrow from $x$ to $y$. We use *execute* $x$ to describe the process of network measurement and

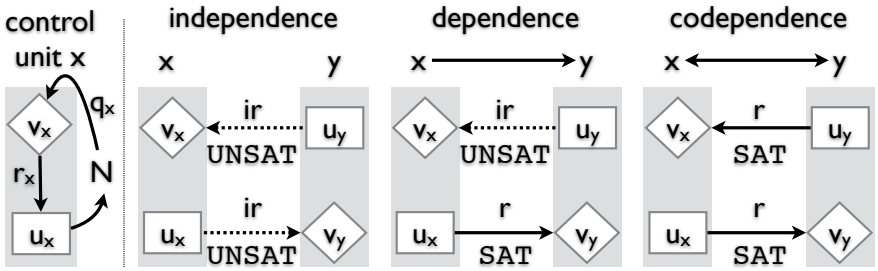



Figure 3.3: Formal model (left) of a control unit that measures and reconfigures the network. Logical characterization and detection (right) of dependency relations between the control units

reconfiguration (i.e., the application of $q_x$ and $r_x$ that produces $v_x$ and $u_x$). An execution of $x$ will repair any network inconsistency with regard to $x$, but may activate other control units. We use $[\![x \to y]\!]$ to denote the set of network states where $u_x$ always changes $v_y$.

### Independence and Dependence

**Definition 3.2** (Independence relation)**.** *$x$ is independent of $y$ if $x \not\to y$ and $y \not\to x$.*

The independence relation is symmetric: if $x$ is independent of $y$, $y$ is also independent of $x$. When two independent units $x, y$ are both active, we can execute $x$ and $y$ independently without additional coordination. Control units in the same module are always independent of each other based on our assumptions. If we have the additional knowledge that the controls units from different modules are also independent, the entire control logic can safely execute in parallel, while preserving overall network consistency.

**Definition 3.3** (Dependence relation)**.** *$x$ depends on $y$ if $x \to y$ and $y \not\to x$.*

The dependence relation is asymmetric. When $x$ depends on $y$, additional coordination is needed to maintain management consistency. On one hand, even if the inconsistency only concerns $x$ (i.e., only $x$ is active), executing $x$ may still trigger and require further actions from $y$. Thus, we also need to execute $y$ to take into account $y$'s reconfiguration. On the other hand, when the inconsistency concerns both $x, y$ (i.e., $x$ and $y$ are both active), if we execute $y$ prior to $x$, the latter reconfiguration by $x$ can trigger actions of $y$ the second time, potentially overwriting $y$'s earlier actions. Thus, if $x \to y$, it is better to execute in the order of $x, y$.

In general, for the set of control units from two modules, if the units form an independence relation and/or a set of acyclic dependence relations, we can deterministically schedule their execution, defining an execution plan that is guaranteed to restore overall network consistency by executing each control unit exactly once. Essentially, the execution plan keeps track of every pair of $\{x, y | x \to y\}$, $x$ is executed prior to $y$.

**Codependence**

**Definition 3.4** (Codependence relation)**.** *$x$ and $y$ are codependent if $x \to y$ and $y \to x$.*

The codependence relation is symmetric. The coordination of two codependent control units $x, y$ is more difficult. We identify and address three special cases. Case 1 assumes that we observe $x \to y$ and $y \to x$ on disjoint sets of network states. Formally, $[\![x \to y]\!] \cap [\![y \to x]\!] = \varnothing$. In this case, we can always reduce the codependent $x, y$ into two pairs of dependent control units. Without loss of generality, we can break $x$ into $x_a, x_b$, where $x_a$ is the restriction of $x$ on $[\![x \to y]\!]$) and $x_b$ is the restriction of $x$ on $[\![y \to x]\!]$). Obviously, $x_a \to y$ and $y \to x_b$. Thus, we call this the *false-codependence* case. Case 2 assumes that an inconsistency that activates both $x, y$ can be repaired by either $x$ or $y$. That is, $x, y$ each concern some aspect of an inconsistent network. Executing $x$ ($y$) can also repair the violation on $y$ ($x$). We call this the *divergence* case as the final network state depends on which of $x, y$ is executed. Finally, Case 3 assumes that $x, y$ can form some execution loop, where an execution of $x$ activates $y$, which further activates $x$, and so forth. That is, the execution loop can be infinite. We call this the *conflicting* case as the execution loop is caused by conflicting control logic.

In the false-codependence case, we can first reduce the control units to some dependence relations and coordinate them accordingly. In the divergence case, we can pick an execution from a preset prioritization of control units. Note that this is more finer-grained and flexible than a rigid module-level priority. In the conflicting case, however, we cannot find an execution plan that is guaranteed to restore overall network consistency without looking closer at and altering the control logic themselves.

### 3.6.6 Satisfiability Reasoning

We now describe how to automatically determine the dependency relations by only examining the logical rules that define the control units. The challenge is that a dependency relation, by definition, is a dynamic property over all execution runs. Our contribution is to reduce the checking of such runtime property to a static decision problem called *irrelevance update* [22, 70, 38].

By definition, the dependency relations (independence, dependency, codependency) can be decided by checking $x \overset{?}{\nrightarrow} y$ and $y \overset{?}{\nrightarrow} x$. Imagine that we store all the network tables and views in a database, the network reconfiguration $u_x$ is nothing but a database update, and the violation view $v_y$ a regular database derived view. As a result, $x \not\dashrightarrow y$ is reduced to checking, for all possible database states, if the update $u_x$ will ever change the derived view $v_y$. This is called the irrelevant update problem in database research: $u_x$ is said to be irrelevant to $v_y$ if the update will not affect $v_y$ for any database instances. Otherwise, update

$u_x$ is said to be relevant to $v_y$.

**Proposition 3.2** (dependency and (ir)relevance)**.** *$x \not\rightarrow y$ ($x \rightarrow y$) iff $u_x$ is irrelevant (relevant) to $v_y$ respectively.*

Figure 3.3 illustrates how to decide the dependency relation by automatic (ir)relevance reasoning. We leverage prior work [22, 70, 38] that formulates (ir)relevance reasoning as a satisfiability problem. Our goal is to decide (ir)relevance between $u_x$ and $v_y$ by static analysis of $r_x$ (i.e., the reconfiguring program that generates $u_x$) and $q_y$ (i.e., the measuring program that generates $v_y$).

**Proposition 3.3** ((ir)relevance reasoning as satisfiability)**.** *If $u_x = t$, $u_x$ is (ir)relevant to $v_y$ if and only if $vc_y(t)$ is (un)satisfiable. If $u_x = \{t|uc_x(t)\}$, $u_x$ is (ir)relevant to $v_y$ if and only if $uc_x \wedge vc_y$ is (un)satisfiable.*

Recall that, as formalized in § 3.6.4 in $r_x$, $u_x$ is either an insertion of a tuple $t$, or a deletion of tuples $\{t|uc_x(t)\}$ that violates $x$'s update condition $uc_x$. $q_y$ generates $vc_y$ by running a local violation check $uc_x$ on the network.

#### Extensions for Aggregates

Prior work focuses on irrelevant updates where the derived views are built from standard operations (*select, join, projects*, or the datalog equivalent), leaving out important aggregate operations, such as `MIN, MAX, AVG, COUNT`. However, these aggregates offer useful semantics for network management. For example, a load balancer needs `COUNT` in its logic. Formally, an aggregate is a function $f : G \rightarrow a$ that maps a group of values ($G$) to a new value ($a$), and by definition, depends on every elements in $G$.

The challenge is that if the invariant condition $vc_y$ involves an aggregate over a group, and the updates $t$ (insertion) / $uc_x$(deletion) belong to that group, the information in $t$ / $uc_x$ will not be sufficient to decide the new aggregate value of the group. Thus, we will not have sufficient information to deduce the truth value of $vc_y(t)$ / $uc_x \wedge vc_y$.

We overcome this by borrowing ideas from incremental view maintenance [48]. Incremental view maintenance computes a view's new value by only looking at the old value and the change to the database. When the view is defined by aggregates, the key idea is to rewrite the aggregate function into an incremental one that avoids referencing every element in the group: $f(G) = a$ in the above becomes $f'(G \cup \delta) = f'(a, \delta)$. Formally, $f(G \cup \delta) = f'(f(G), \delta)$. For example, `count` will be rewritten to $f'(G \cup \delta) = \texttt{count}(G) + |\delta|$. The occurrence of the aggregate $f$ in the predicate $vc_y$ is then replaced by the $f'$, referencing only the delta $t/uc_x$.

**Primitive Actions:**

```
A  ::=  drop | modify(h=v)
```

**Predicates:**

```
P  ::=  all_packets | no_packets |
        match(h=v) | P&P | (P|P) |  P
```

**Policies:**

```
C  ::=  A | P[C] | (C | C) | C >> C
```

Figure 3.4: Pyretic sub-language amenable to reasoning

| Pyretic Predicate | Violation View Condition |
|---|---|
| match(h=v) | fid FROM rm WHERE h=v |
| P & P | fid FROM rm WHERE P AND P |
| P \| P | fid FROM rm WHERE P OR P |
| all_packets | fid FROM rm |

| Pyretic Action | Repair Update |
|---|---|
| drop | DROP fid FROM rm |
| modify(h=v) | UPDATE rm SET h=v |

Table 3.1: Summary of Pyretic translation

### 3.6.7 Syntactic Translation

In our proof of concept, we demonstrate extracting a model amenable to reasoning from Pyretic [78] policies. We examine a subset of the Pyretic language for defining static policies, shown in Figure 3.4. We chose this subset as we can express a large range of static policies (e.g., for firewall and NAT applications), with the intuition that most policies can be expressed in the form `P >> A`. We leave extracting dynamic and query policies to future work.

For example, suppose we have a firewall policy that blocks flows between hosts 10.0.0.1 and 10.0.0.2, and a NAT policy that specifies flows with source IPs 10.0.0.3 and 10.0.0.4 should be rewritten to 10.0.0.11. The resulting policy can be expressed in Pyretic as:

```
# firewall
(match(srcip=10.0.0.1) & match(dstip=10.0.0.2)) |
(match(srcip=10.0.0.2) & match(dstip=10.0.0.1)) >> drop

# nat
(match(srcip=10.0.0.3) | match(srcip=10.0.0.4)) >>
modify(srcip=10.0.0.11)
```

The model we extract from a Pyretic policy operates on Ravel's base schema design. Thus, a Pyretic match predicate on a flow's source IP address translates to a query over flow table `rm` that matches the `sid` column. To map between `sid` and the source IP address, we can introduce a `nodes` table to translate between the two:

```
nodes(sid, ip, mac)
```

When analyzing a Pyretic application, we use the existence of the location for a rule to define local and global rules. Global rules impose constraints across all flows, while local rules impose constraints only on rules installed in a given switch. Therefore, a policy that does not specify a location for a rule becomes a global rule. Global policies query `rm` (the network-wide reachability matrix), while local rules query `cf` (the per-switch configuration matrix).

Following our formal model, the extracted Pyretic model must specify violations of the policy's constraints, as well as a set of repair actions. Pyretic predicates (and combinations of predicates) define violations, while actions define repairs. We model violations as a SQL view (equivalent to datalog rules in § 3.6.3) on `rm` and `cf`. Table 3.1 shows our translation process, where the violation view condition defines the query for violating flows, and the repair update defines the update to `rm` or `cf`. For example, a drop action for a given node or flow implies an entry in a blacklist, so a drop action on a given predicate entails the absence of flows in `rm` for that pair of nodes. Repairing the violation is a matter of removing the violating flow `fid` from `rm`. We repeat this process recursively for each match-action term.

Once the reasoning engine analyzes the Pyretic model, we can use the result to compose the Pyretic applications. For an application $x$ that is dependent on $y$, we compose the two sequentially as $y >> x$. Otherwise, applications that are not dependent on each other can be composed in parallel as $x|y$.

## 3.7 Example Ravel Applications

We present a full implementation of an example network consisting of a simplified PGA service chain application, a simplified Kinetic stateful firewall, and the orchestration facility that combines them. The disparately different PGA and Kinetic paradigms fit naturally into Ravel's SQL statements: tables, views, and rules. We also give an example of priority assignment, and the automatically generated orchestration implementation. In addition to providing a lightweight alternative to present systems like PGA and Kinetic, Ravel does not rely on application-specific constructs or optimization tuning.

### 3.7.1 (PGA) Service Chain Policies for Groups

#### Specifying Control Policy

In Ravel, the PGA graph expressions for service chain policies on sub-domains are two plain tables:

```
CREATE TABLE sub_domain (
       gid integer,
       eid integer);

CREATE TABLE service_chain (
       gid1 integer,
       gid2 integer,
       MB text);
```

`sub_domain(gid, eid)` reads as a membership statement that an endpoint `eid` (i.e., any individual address) belongs to group `gid`. `service_chain(gid1, gid2, MB)` specifies the middlebox to be traversed for the two groups `gid1, gid2`. To merge the group-level service chain policies into those for individual endpoints, PGA employs an integrated specialized algorithm that involves two key steps. The first, graph composition, computes the policy over the individual endpoints. Next, middlebox ordering analyzes middlebox dependencies and merges multiple ones into a single chain with heuristics. Ravel also splits merging into two steps: *i*) generating the composition endpoint-level policy with a three line query as follows, and *ii*) determining an ordering using a mediation protocol (§ 3.5.2). What is unique to Ravel is that the two steps are separated and each relies on a generic application-agnostic technique.

```
CREATE VIEW service_chain_composite AS(
  SELECT DISTINCT sd1.eid as eid1, sd2.eid as eid2, MB
  FROM sub_domain sd1, sub_domain sd2, service_chain
  WHERE sd1.gid=gid1 AND sd2.gid=gid2);
```

This query states that we have an endpoint-level service chain `service_chain_composite(sid1, sid2, MB)` if the two endpoints belong to groups and there is a corresponding group-level service chain. While PGA improves graph composition performance with specialized heuristics drawn on novel insights, Ravel relies on standard view maintenance algorithm to accelerate access.

### Control Loop: Monitoring and Repairing Violations

To detect violations of the service chain policy, we simply query `service_chain_composite(sid1, sid2, MB)` against the underlying network reachability in `rm`.

```
CREATE VIEW sc_violation AS (
  SELECT fid, MB
  FROM rm, service_chain_composite
  WHERE src=eid1 AND dst=eid2 AND (MB='FW' AND FW=0));
```

A violation occurs if for some endpoint pairs `(eid1, eid2)`, `service_chain_composite` requires a waypoint through a middlebox (`MB='FW'`) that is not respected by `rm` (`FW=0`). The query also extracts the flow (`fid`) in the violation. To repair a violation, we simply delete rows in `sc_violation`. The delete operation is translated onto the underlying `rm` by rule `sc_repair`, which simply sets `rm`'s `FW` value to `1`.

```
CREATE RULE sc_repair AS
  ON DELETE TO sc_violation
  DO INSTEAD
    UPDATE rm SET FW=1
    WHERE fid=OLD.fid AND OLD.MB='FW';
```

### 3.7.2 (Kinetic) Stateful Firewall

#### Specifying Control Policy

In Ravel, a stateful firewall policy (§ 3.2) comprises two tables: `FW_white(eid1, eid2)` denotes the endpoint pair `(eid1, eid2)` that is allowed to communicate (i.e., the whitelist), and the user table `FW_in(eid)` stores internal endpoints whose in-bound traffic is allowed only after they first initiate an out-bound request. To capture this dynamic, we add two rules: `FW1` modifies the whitelist to allow in-bound routes in response to a request, and `FW2` disallows an outdated in-bound routes.

```
CREATE TABLE FW_white (
       end1 integer,
       end2 integer);

CREATE TABLE FW_in (
       uid integer);

CREATE RULE FW1 AS
  ON INSERT TO rm
  WHERE ((NEW.src, NEW.dst) NOT IN
         (SELECT end2, end1 FROM FW_white)) AND
        (NEW.src IN (SELECT * FROM FW_in))
  DO ALSO (INSERT INTO FW_white VALUES (NEW.dst, NEW.src););

CREATE OR REPLACE RULE FW2 AS
  ON DELETE TO rm
  WHERE (SELECT count(*)
         FROM rm WHERE src=OLD.src AND
                       dst=OLD.dst) = 1;
  DO ALSO
     DELETE FROM FW_white
     WHERE end2=OLD.src AND
           end1=OLD.dst;
```

#### Control Loop: Monitoring and Repairing Violations

Violations are detected by a query on `rm` that finds pairs that are not in the firewall whitelist, as stated in `FW_violation`. To repair the violation, we simply remove these pairs, as shown in rule `FW_repair`.

```
CREATE VIEW FW_violation AS (
   SELECT fid
   FROM rm
   WHERE FW=1 AND
         (src, dst) NOT IN (SELECT end1, end2 FROM FW_white));

CREATE RULE FW_repair AS
   ON DELETE TO FW_violation
   DO INSTEAD
         DELETE FROM rm WHERE fid=OLD.fid;
```

### 3.7.3 Putting It All Together: Orchestrating Service Chains and Firewall

Implementing the orchestration protocol using Ravel is straightforward with PostgreSQL triggers [83, 101, 33, 101]. Assuming a pre-assigned priority, Ravel automatically generates the orchestration implementation consisting of two parts: triggers that register an application, and triggers that enforce an execution order according to the priority.

#### Registering Applications in the Protocol

To register an application, Ravel binds it to an auxiliary table and a rule. As the protocol proceeds, the table is set on and runs the application in the rule. For example, we bind to the firewall a table `p_FW` and a rule `run_FW`. The table and rule generated for service chain is similar.

```
CREATE UNLOGGED TABLE p_FW (
  counts  integer,
  status  text);

CREATE RULE run_FW AS
  ON INSERT TO p_FW
  WHERE (NEW.status='on')
  DO ALSO (
     DELETE FROM FW_violation;
     UPDATE p_FW SET status='off'
     WHERE counts=NEW.counts);
```

#### Implementing the Protocol

To determine the priority among the service chain (`SC`), firewall (`FW`), and routing (`RT`) applications, we analyze the view dependency among them. First, observe that both `FW` and `SC` updates will invoke `RT` to install the corresponding change to network path. Hence, we assign `RT` the highest priority so it is always called in the end to append the path-level data write needed to complete the updates. Second, `SC` will affect `FW`. Consider a flow bypassing the firewall — if the flow is also forbidden by the firewall, an `SC` update that

instructs the flow to traverse the firewall will cause `fw_violation`. As such, we assign `FW` a higher rank so it can repair the violation (e.g., drop the flow). Together, we have a priority list of `[SC,FW,RT]` (low to high).

To enforce priority, we only need to ensure an execution ordering of `SC`, `FW`, and `RT`. Ravel generates rules `PGA2FW`, `FW2RT`, and `RT2Fin` to pass control along this ordering.

```
-- starts orchestration
INSERT INTO p_PGA VALUES (MAX_CLOCK_TICK, 'on');

-- enforces an order of execution sequence
CREATE OR REPLACE RULE PGA2FW AS
       ON UPDATE TO p_PGA
       WHERE (NEW.status='off')
       DO ALSO
           INSERT INTO p_FW values (NEW.counts, 'on');

CREATE OR REPLACE RULE FW2RT AS
       ON UPDATE TO p_FW
       WHERE (NEW.status='off')
       DO ALSO
           INSERT INTO p_RT values (NEW.counts, 'on');

CREATE OR REPLACE RULE RT2Fin AS
       ON UPDATE TO p_RT
       WHERE (NEW.status='off')
       DO ALSO
           INSERT INTO clock values (NEW.counts);
```

## 3.8 Implementation

Our Ravel prototype is open source [1, 2]. It consists of two components: a PostgreSQL database runtime and a network runtime.

### 3.8.1 Database Runtime

We chose PostgreSQL [12] as it is an advanced, open source database popular for both academic and commercial purpose. It is highly customizable with useful features for networking (e.g., pgRouting [84]). The database component is implemented in 1000+ lines of SQL, consisting of three subcomponents: the network base, the control applications (routing, load balancer, access control, and tenants), and the runtime. Table 3.2 summarizes the implementation size and the PostgreSQL features used.

During development of Ravel's prototype, we learned two optimization tactics that reflect the time-space trade-off in a database-centered system. First, Ravel optimizes the base table schemas to avoid recursive path computation. By adding an additional reachability requirement table `rm` and triggers that recursively

| | Ravel component | Lines (#) | Language/Feature |
|---|---|---|---|
| app | routing (rt) | 230 | SQL, rule, trigger |
| | load balancer (lb) | 30 | SQL, rule |
| | access control (acl) | 20 | SQL, rule |
| | tenant (rt, lb, acl) | 130 | SQL, rule, trigger |
| db | network base | 120 | SQL |
| | runtime | 200 | rule, trigger |
| network | SQL trigger | 100 | SQL, trigger |
| | Python trigger | 300 | Python, RPC, MQ |
| | OpenFlow manager | 1100 | Python, flowmod |

Table 3.2: Summary of Ravel components

reconfigure the corresponding paths on the per-switch configuration table, higher-level views can control routing paths via non-recursive queries and updates over `rm`. Second, Ravel optimizes the performance of application views by integrating a classic view maintenance algorithm [48], which avoids wasteful re-computation of the views from scratch as the network changes.

### 3.8.2 Network Runtime

Ravel interacts with a physical SDN network via a network runtime implemented in 1700 lines of Python and SQL code, summarized in Table 3.2. Figure 3.5 shows the high level architecture. The runtime consists of two subcomponents: database triggers and an OpenFlow manager. Database triggers are invoked on changes to flows (i.e., insertion, deletion, or modification). A trigger is invoked for each switch along the flow's path to update the switch's flow table. The trigger queries the database for the flow's match and action rules, and passes it to the OpenFlow manager.

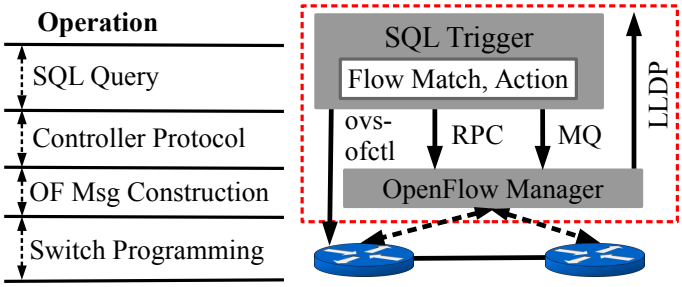



Figure 3.5: Ravel network runtime architecture

We implemented three mechanisms for triggers to communicate with the OpenFlow manager: remote procedure calls (RPC), message queues (MQ), and a wrapper around the `ovs-ofctl` [82] command. With RPC, the OpenFlow manager can reside on a separate machine than the Postgres database. The OpenFlow manager receives the flow entry from the database trigger and constructs a flow modification message. We

use Pox [14] to send the OpenFlow messages to the appropriate switch. We also leverage Pox's (LLDP-based) `discovery` module to monitor changes to link states and update the database appropriately using `psycopg` [88].

## 3.9 Evaluation

We evaluate Ravel under two scenarios. First, we present a comprehensive evaluation of the database component, profiling and micro-benchmarking database overhead on all key operations: routing decision, application orchestration, and control application optimizations. We demonstrate promising results, showing Ravel's database component scales gracefully to large network topologies and incurs small absolute delay on moderate network size. Second, we evaluate Ravel on a physical SDN testbed, showing Ravel's network runtime can transform changes in the database to changes in a switch's flow table within a few milliseconds.

### 3.9.1 Profiling Database Performance

As Ravel uses a database for network control and performance tuning, we first profiled the database component overhead. Specifically, we microbenchmarked database operations for route insertion and deletion on large fat-tree and ISP network topologies. Next, we examined scalability in orchestrating applications. Finally, we evaluated the performance improvement from view optimizations. The results are promising: on a fat-tree topology with 196608 links and 5120 switches, and the largest ISP topology (Rocketfuel dataset) of 5939 switches and 16520 links, the delay for rule/trigger and key-value query overhead remained in the single digit millisecond range. All experiments were run on a 64bit VM with a single-core CPU and 8GB RAM.

#### Network Setup

We used the three fat-tree topologies and four ISP topologies (taken from the Rocketfuel [92] dataset) shown in Figure 3.3. A fat-tree with `k` pods contains `k` core switches, `k/2` aggregation switches, `k/2` edge switches, and `k/2` hosts.

We used the smallest AS (`4755`), two moderate sized ASes (`3356`, `2914`), and the largest AS (`7018`). ASes `4755`, `3356`, and `7018` are used to study how Ravel operations scale as network size increases. AS `2914` is pre-populated with a policy configuration (i.e., forwarding rules) to study Ravel scalability against policy size. We used Route Views [13] BGP feeds, each of which specifies a route to an external prefix collected at a vantage point, to synthesize the policy as follows. Vantage points were mapped to a set of random AS

| fat-tree | | | ISP | | |
|---|---|---|---|---|---|
| k | switches | links | AS# | nodes | links |
| 16 | 320 | 3072 | 4755 | 142 | 258 |
| 32 | 1280 | 24576 | 3356 | 1772 | 13640 |
| 64 | 5120 | 196608 | 7018 | 25382 | 11292 |
| | | | 2914 | 5939 | 16520 |

Table 3.3: Topology setup: (left) fat-tree with k pods, (right) four Rocketfuel ISP topologies with varying size

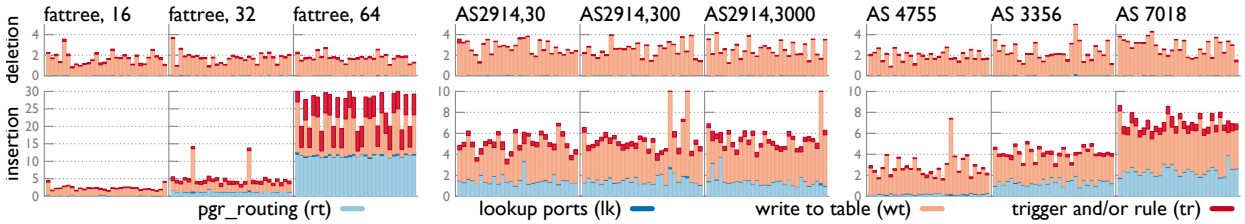



Figure 3.6: Sources of Ravel delay (ms) for route insertion and deletion

switches, which served as border routers for the AS. Next, the external prefixes were randomly assigned to one of the borders. With these two mappings, each BGP feed was translated to a route between two border routers, a route that carries traffic for the external prefix.

### Profiling Delay

We started by profiling the primitive database operations that constitute Ravel delay: `pgr_routing` (`rt`) that computes a shortest path between two nodes, `lookup ports` (`lk`) that fetches the outgoing port, `write to table (wt)` that installs the per-switch forwarding rules, and `trigger and rules` (`tr`) that optimize application operations and coordinate their interactions. We found the delay is dominated by the routing component (`rt`) and database writes (`wt`), which are dependent on the network topology. The additional database overhead of `lk` and `tr` remains low even for large network sizes, demonstrating good scalability. The details of the constituted delays are shown in Figure 3.6.

As the network size grows for fat-tree and ISP topologies, the delay caused by `tr` and `wt` operations increases linearly. Even for `fattree k=64`, both operations only double to 5ms. While the path-computation component `rt` grows and quickly dominates delay, it can easily be replaced by a faster implementation with better heuristics or precomputing k-shortest paths. In contrast to network size, the growth of the policy size has a smaller effect on the operations. This is because Ravel handles each flow `fid` independent of the rest of the policy space. Therefore, the scalability study below examines only the network size.

### Scaling Orchestration

We measured, on increasing network sizes, Ravel's delay in orchestrating applications, including a load balancer (`lb`), access control (`acl`), tenant network (`t`), and routing application (`rt`). `acl` removes (i.e., prohibits) traffic requirements from a blacklist. `lb` re-balances traffic requirements among a list of provisioned servers. `rt` is the component that actually realizes the traffic requirement with per-switch rules. Tenant network `t` is a 10-node full-mesh network that controls the traffic slice pertaining to its topology.

We examined vertical and horizontal orchestration scenarios. We use `x@t` to denote vertical orchestration of an application `x` controlling a tenant network `t`. For horizontal orchestration, we use `x+y+...` to denote applications `x,y,...` collectively controlling a network. For example, `lb+acl+rt` denotes the scenario that upon a newly inserted route, the load balancer is engaged first, followed by access control, and finally by the routing application to set up the actual path on the underlying network state. The CDF in Figure 3.7 shows the overall orchestration delay in milliseconds.

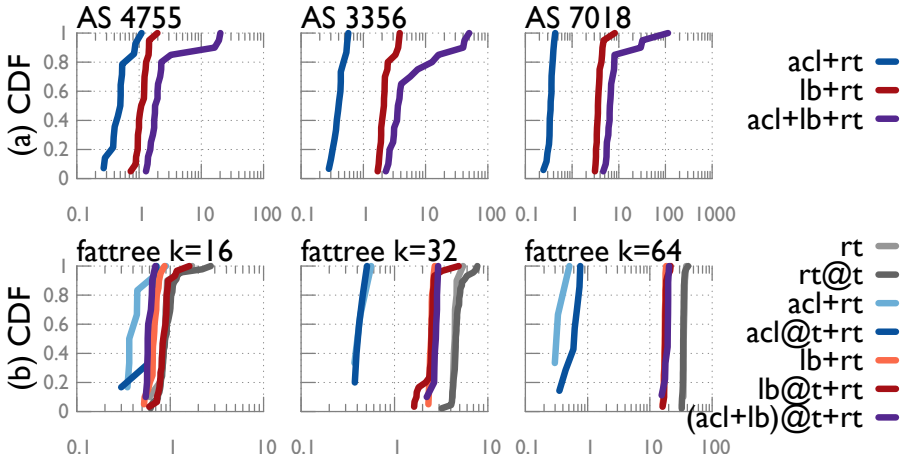



Figure 3.7: CDF of orchestration delay: normalized per-rule orchestration delay (ms) on various network sizes

We found that Ravel adds a small delay for orchestration, around 1ms for most scenarios. Delay is dominated by `rt` because of its semantics. `rt` must compute the path and reconfigure switches. In contrast, `acl` imposes a negligible delay (<1ms) since it must only read from its blacklist, i.e., a fast key-value lookup. `lb` sits between these two applications and handles the extra path computation to direct traffic to a less loaded server. In particular, `lb+acl+rt` is bound by `rt`, and `x@t` is almost identical to that of `x`.

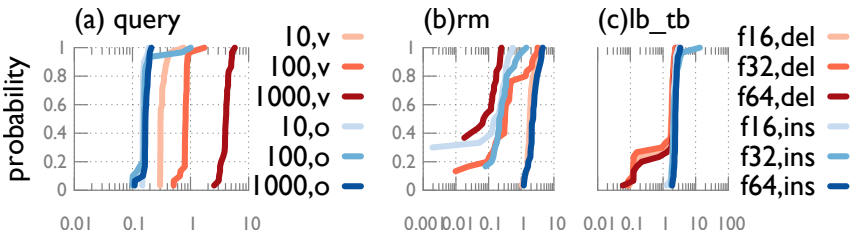



Figure 3.8: (a) CDF of query time (ms) on a view and its materialized equivalent. (b,c) CDF of maintenance delay (ms)

**Optimizing Application Views**

Ravel optimizes application views by translating them into equivalent materialized tables that offers faster access with small overhead. Figure 3.8 (a) compares the performance (i.e., query delay) on a load balancer view (`v`) and its materialized version (`o`) for three policy sizes (`10,100,1000`). Queries on optimized views (blue lines) are one magnitude faster (0.1ms vs. 1-2ms). As policy size grows (from 10 to 1000), the performance gain is more obvious. Figure 3.8 (b,c) shows the overhead of view maintenance, measured on three fat-tree topologies (`k=16,32,64`) and two scenarios: updates (deletion and insertion) to `lb_tb` and `rm`. In all cases, view maintenance incurs small delay (single digit millisecond) that scales well to large network size.

### 3.9.2 Physical Testbed Evaluation

We evaluated Ravel on a physical SDN testbed with Pica8 Pronto 3290 switches. Average round-trip time between the controller and switches is <0.5ms.

First, we examined sources of delay in the network runtime by profiling the database triggers and OpenFlow manager during flow insertion, deletion, and rerouting for a five-hop path. We used message queues (MQ) for communication between the triggers and OpenFlow manager. Flow insertion (deletion) consists of inserting (removing) a flow from Ravel's `rm` table. This change in the database invokes the triggers that look up the corresponding flow's match fields and action, and pass it to the OpenFlow manager. The triggers are invoked once for each of the five switches along the path. We installed two flows in each switch, for the forward and reverse path. For rerouting, we removed a link along the path between a random source and destination five hops apart. The update triggers Ravel to find a new path (also five hops) then invokes the network-specific triggers to remove the flow from the switches along the old path and insert the flow into the switches along the new path. We report the per-switch average in Figure 3.9 across 30 trials.

Trigger execution consists of several SQL queries and queuing the flow information for processing by the

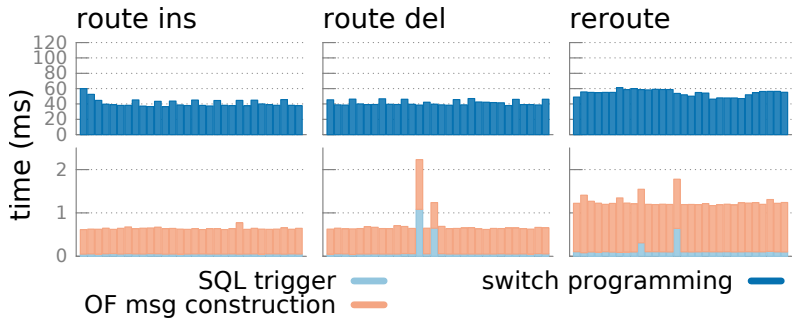



Figure 3.9: Sources of delay for route insertion, deletion, and rerouting in the network runtime using message queues. Reported times are per-switch average along a five-hop path

OpenFlow manager. Thus, the database trigger terminates after the message is queued. In Figure 3.9, on the bottom, we can these operations take less than 0.5ms. The message queue is read by the OpenFlow manager, which constructs a flow modification message and sends it to the appropriate switch using Pox. This operation takes less than 1ms for adding and removing a flow. Rerouting, which requires removing the old flow and inserts a new flow, takes around 1ms. The top of Figure 3.9 shows the switch programming delay for a flow entry. Switch delay is measured using a barrier message sent after the flow modification message. The end-to-end delay for inserting or removing a flow (i.e., the forward and reverse path) on a single switch is under 50ms.

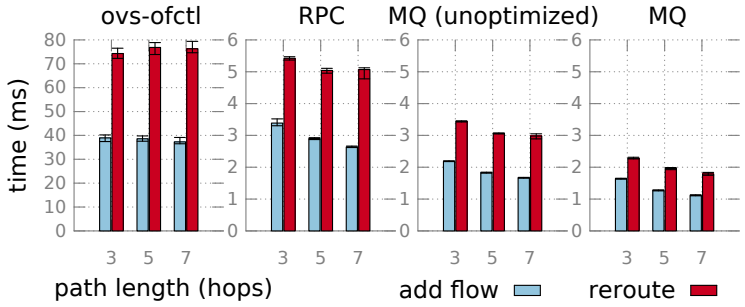



Figure 3.10: Comparison of average per-switch flow install and reroute time for a path length of five, using different protocols for triggers to communicate with the OpenFlow manager

Next, we examined the performance of the three protocols a trigger can use to communicate with the OpenFlow manager: RPC, message queues, and `ovs-ofctl`. We tested two variants of our message queue protocol. The first is an unoptimized version with only one trigger written in Python. Database queries for the flow's match and action rules use `psycopg`. The optimized version separates the trigger into two parts: a SQL trigger that queries the database and a Python trigger that adds the result of the query to the message queue. We measured end-to-end time for flow insertion and rerouting (i.e., starting by adding a new flow into `rm` or bringing down a link in `tp`) for paths of length 3, 5, and 7 over 30 trials and report time as the

per-switch average in Figure 3.10.

We can see `ovs-ofctl` takes around 100ms per switch to install a flow. RPC and both variants of message queues improve flow insertion time by an order of magnitude since these protocols do not block on the switch programming and instead install the rule asynchronously. Additionally, we see that separating the triggers into SQL and Python parts reduces the execution for the message queue protocol by 30%.

# Chapter 4

# Repairing Updates with Autocorrection

In Chapters 2 and 3, we propose techniques to provide an operator with techniques to automatically search for unpredictable behavior of SDN applications at the control and management planes. In this chapter, we examine the output of applications and the correctness of the data plane updates they produce. The execution of an application's event handler, and any subsequent handlers it may trigger, will produce data plane updates. Even given a set of correct applications and composition plan, bugs in the controller's implementation could introduce unexpected behavior through unexpected updates. For example, there are cases where bugs in the controller can cause flows to be removed at the wrong time [10], failure of flows to be removed [9, 11], and incorrect logic that could affect the expected output of an application [5]. As a result, an operator may need to understand the low-level implementation details of the controller to comprehensively reason about the output of the composed application set. Additionally, unexpected behavior may arise if an operator's specification of correctness when exploring application behavior at the management and control planes is incomplete.

Therefore, in this chapter, we introduce an *autocorrection* [105] primitive to automatically repair SDN updates that violate operator-specified policies. Unlike prior work on synthesis for SDN [91, 90, 104], this project approaches it with a real-time performance constraint. We introduce NEAt, a tool to modify updates at runtime from unmodified SDN applications to satisfy policies such as reachability and segmentation.

The work in this chapter was co-authored with Wenxuan Zhou, Bingzhe Liu, Matthew Caesar, and P. Brighten Godfrey. [1] It was published in 2017 ACM Symposium on SDN Research (SOSR) [105].

This main research questions motivating this work are presented below.

- How can we enforce and repair correctness on-the-fly without modifications to control programs?
- How can we constrain divergence between application state and network state in the face of repairs?

[1] §§ 4.5.1–4.5.2, 4.8.1 were developed by Wenxuan Zhou, and § 4.6.2 by Bingzhe Liu.

## 4.1 Introduction

Modern enterprise networks must comply with highly stringent security demands, including regulatory requirements, or industry standards, such as PCI, HIPAA, and SOX. As a result, network operators must carefully design and maintain their networks to follow those policies, by mapping out device contexts and access to sensitive resources, assessing risk, and installing access control policies that effectively mitigate that risk. However, mistakes and errors in implementing the policies can result in costly data breaches, segmentation violations, and infiltrations. Through 2020, Gartner predicts 99% of firewall breaches will be caused by misconfigurations [3, 4].

While discovering and troubleshooting these bugs is essential to maintaining network security, doing so is notoriously hard. Relying on humans to configure and maintain the network configuration is not only prone to mistakes, but slow. Given the sophistication and speed at which new attack vectors propagate, manually updating and testing new configurations leaves the network in a vulnerable state until the attack vector is fully secured. Furthermore, maintaining a security posture in the presence of SDN is even more challenging. While SDN enables new functionality, application designers may not be aware of the policy or security requirements of the networks on which their applications will be deployed. Worse yet, SDN applications written in general-purpose languages such as Java or Python can be arbitrarily complex. Requiring applications to implement and modify their behavior to support a broad spectrum of policies needed across a large range of networks presents an almost insurmountable challenge.

To this end, we present NEAt, a transparent layer to automatically repair policy-violating updates in real-time. NEAt secures the network with a mechanism similar to a smartphone's autocorrect feature, which enables on-the-fly repair to policy-violating updates and ensures the network is always in a state consistent with policy. Unlike prior work on update synthesis, NEAt maintains backward compatibility and flexibility to run general SDN application code. To do this, NEAt does not synthesize network state from scratch, but rather *influences* updates from an existing SDN application toward a correct specification. In particular, NEAt enforces a concrete definition of correctness by influencing and constraining dynamically arriving network instructions. To formulate correctness criteria, we construct a set of *policy graphs* to represent a human operator's correctness intent, which is based on the observation that important error conditions can be caught by a concise set of boundary conditions. NEAt sits between an SDN controller and the forwarding devices, and intercepts the updates proposed by the running SDN applications. If the update violates an operator's defined policy, such as reachability or segmentation, NEAt transforms the update into one that complies with the policy.

A key challenge we face in this approach is discovering update repairs in real-time. In NEAt, we build

on prior work on verification to efficiently model packet forwarding behavior as a set of Equivalence Classes (ECs) [61, 106]. Upon receiving an update from the controller, NEAt computes the set of affected ECs and checks for a violation in the same manner as [61]. To repair the violation, we cast the problem as an optimization problem, to find the minimum number of changes (added or deleted edges) to repair the violating EC's forwarding graph. To rapidly compute repairs on arbitrarily large networks, we exploit two optimization techniques, *topology limitation* which "slices" away irrelevant part of the network, and *graph compression*, to compress both an EC's forwarding graph and the topology. We then solve the optimization problem on the sliced and compressed graphs.

Furthermore, as NEAt repairs policy-violating updates, stateful applications — without knowledge of the violating or repaired updates — will diverge from the underlying network state. To address this problem, applications can interactively propose updates to NEAt and receive notifications of repairs with minor modifications to application code. Thus, applications can remain unmodified and leverage NEAt transparently in a *pass-through* mode, with a risk of state divergence, or propose updates in an *interactive* mode.

A preliminary evaluation of our prototype shows promising results. On topologies with up to 125 switches and 250 hosts, NEAt can discover repairs in under 1 second for applications with non-overlapping rules, and under 2 seconds for applications with more complex dependencies. Furthermore, we find NEAt can verify and repair updates on realistic data planes. On a large enterprise network with one million forwarding rules, NEAt discovers and repairs 28 loop violations. Simulations on this dataset show NEAt can verify and repair reachability and loop freedom policies in under a second.

## 4.2 Background and Motivation

Enterprise network policies must compose together requirements from a variety of demands, such as government or industry regulations, to mitigate risk for attack vectors, and limit access to sensitive resources. As a result, network operators must take into account complex, composed policies when configuring or updating a network.

Composing together such policies is a slow and often error-prone process for a human operator. The operator may introduce errors translating the demands into high-level policies, or translating the policies into low-level routing configurations. A recent study [63] found that operators make changes to their networks at least once per day, while more than 80% were concerned updates would break existing functionality unrelated to a given change. While tools [61, 58] exist to automatically discover misconfigurations in real-time, they offer the operator no guidance on how to repair the misconfiguration beyond the type of correctness property

that is violated.

Instead, a system to automatically repair updates, ensuring the network always remains consistent with the operator's policy, can relieve a slow and error-prone task from the configuration process. If an update violates a given property in the network, a *repair* should fix the cause of the violation while maintaining the original purpose of the update. We argue a minimal change is best, to repair the update with the least number of added or removed edges. Furthermore, such a system should improve upon a manual effort with transparency in both architecture and performance. A system that requires hours or days to verify and repair a network is not useful if the process can be completed manually in just a few minutes. It should also not require modifying or redesigning existing infrastructure.

However, accomplishing this task in real-time is challenging due to the size of the network and the data plane state. To efficiently reason about the data plane, we build on previous work in data plane verification [61, 106] that separates forwarding behavior into Equivalence Classes (ECs) of packets. All packets within an EC are forwarded in precisely the same manner. From each EC, we can extract a *configuration graph* that defines the forwarding behavior for packets within the EC. A repair for a given EC must then explore additions or deletions of links in the configuration graph. Finding a link addition requires examining the *topology graph* defined by the edges in the physical topology. To efficiently discover repairs, we propose two optimization techniques to compress the configuration and topology graphs, described in § 4.6. We refer to the outcome of these techniques as the *compressed configuration graph* and *compressed topology graph*.

Finally, a repair system should ideally support both unmodified and modified applications. Burdening developers or operators with the task of modifying their applications to support the latest extension is unreasonable, but those that want more specific control over the repair process should have the option to do so. A *pass-through* mode could allow an operator to transparently leverage automated repairs, without the need to modify existing applications, while an *interactive* mode could provide the application with finer-grained control over the choice of repairs. Furthermore, an interactive mode would allow the application to ensure its state is consistent with that of the network, as it can update its own state after choosing one of several potential repairs for the policy-violating update.

## 4.3 Design

At the core of NEAt is a verification and correction layer, which ensures only updates conforming to the network policies are sent onto the network. This layer receives updates from one of two integration modes with the SDN control infrastructure: *pass-through* and *interactive*.

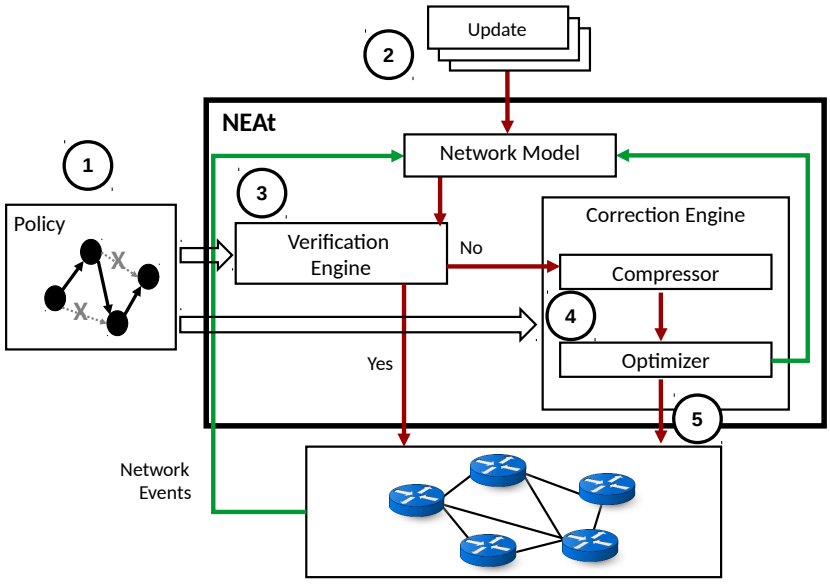



Figure 4.1: System architecture of NEAt

### 4.3.1 Verification and Repair

NEAt's verification and correction engines ensure the network is always consistent with the defined policy. To start, NEAt takes as input a *policy graph* (①), which defines the network policies (e.g., reachability, segmentation, waypointing) in the form of a directed graph. Next, NEAt receives updates (e.g., flow modification messages) from the SDN control infrastructure. For each update (i.e., each flow modification message) (②), NEAt applies the change to a network model, from which the ECs affected by the update are computed. Using the policy graph, NEAt checks each affected EC in the network model for policy violations using the verification engine (③). If the update does not introduce any violations, it is sent onto the network. However, if it does introduce a violation, the configuration graph and topology graph are compressed and passed to the correction engine (④). The optimizer returns a set of edges to be added or removed to the EC's configuration graph, which are then applied to the network model, converted to OpenFlow rules, and sent to the forwarding devices (⑤).

NEAt's correction engine models the process of discovering repairs as an optimization problem. Our exploration of alternative approaches guided us toward this optimization problem-based solution for performance considerations. For example, consider a brute force approach that discovers repairs for a given EC by testing all possible permutations of edge additions and removals to the EC's configuration graph. A repair that requires only adding edges, from 10 possible unused topology edges, would need to explore 10! (˜3.6M) permutations. If the violating property can be checked in just 1ms, each EC could take up to 10 minutes to find a repair. Therefore we use the formulation described in § 4.5 for our repair discovery process.

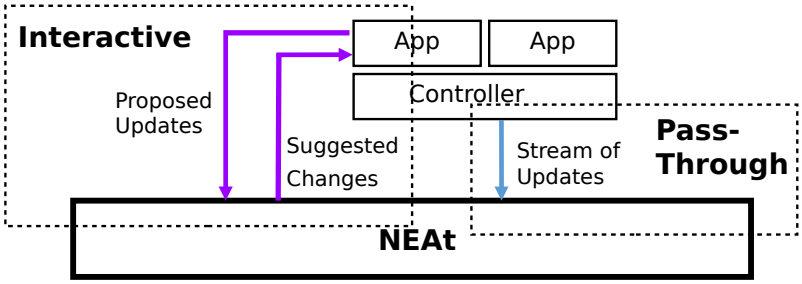



Figure 4.2: Integration modes of NEAt

### 4.3.2 Integration Modes

With each repair, inconsistencies between application state and network state will arise. To prevent applications from diverging from the underlying network state, NEAt exposes two integration modes: *pass-through* and *interactive*.

In pass-through mode, NEAt acts as a transparent layer that sits between the controller and forwarding devices. This mode enforces network policies without modifications to the controller applications. Both the controller and applications are unaware of NEAt in this mode. NEAt intercepts updates from the controller, as well as updates from the network about link and switch state, and passes it to the verification and repair engines.

Interactive mode enables applications to leverage NEAt's verification and repair process by checking proposed updates. An application passes to NEAt a set of updates, which are checked against the current network model. If the updates introduce a violation, NEAt returns a set of repaired updates, which the application can accept or reject. If the application accepts the changes, it can send them onto the network and update its state, ensuring the application and network state are consistent. If the application rejects the changes, it can propose another set of updates to NEAt. Interactive mode requires modifications to applications to update its state with the accepted change.

NEAt maintains consistency between the integration modes, allowing applications and the controller to both simultaneously benefit from NEAt's automated repair. For example, one application can use NEAt's API while another remains unmodified, allowing its updates to be checked by NEAt in pass-through mode.

## 4.4 Policies as Graphs

Many existing tools reason about individual network paths [61, 59]. While this approach has proven effective for network data plane verification, synthesizing network state changes requires viewing the entire network as a whole (i.e., a graph), as changes that repair one path may influence the correctness of other

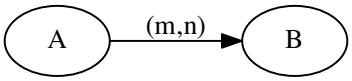



Figure 4.3: Policy edge

paths. In addition, expressing network correctness conditions as a graph instead of a collection of paths enables dealing with a richer set of policies, for instance, path consistency and load balancing. Based on this intuition, NEAt takes as input a set of intended policies, and formulates these policies as directed graphs called *policy graphs*.

Edges on a *policy graph* are marked with notations denoting different types of reachability constraints. For example, the graph in Figure 4.3 specifies a requirement that at least $m$ paths exist from node $A$ to $B$, each bounded by $n$ hops.

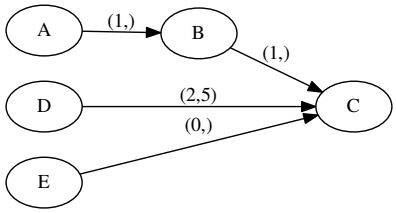



Figure 4.4: Policy graph

A basic *reachability* requirement can be expressed with $m = 1$ and $n$ unspecified, as shown in edges $A \rightarrow B$ and $B \rightarrow C$ in Figure 4.4. When $n$ is specified, we get a *bounded path length reachability* policy (e.g., $D \rightarrow C$ in Figure 4.4). *Shortest path* policies can be viewed as a special case of a *bounded path* policy, and can be encoded in a similar way. When $m > 1$, the edge expresses a *multipath* invariant (e.g., $D \rightarrow C$ in Figure 4.4), whereas $m = 0$ specifies that two ends of the edge are required to be *isolated* from each other (e.g., $E \rightarrow C$ in Figure 4.4). Furthermore, concatenating edges together can denote *service chaining* policies. As shown in Figure 4.4, traffic from node $A$ should traverse a waypoint $B$ before reaching $C$.

One special case is expressing a *load balancing* policy, as this typically requires distributing traffic in a certain way among a pool of servers. In our policy model, this can be expressed by assigning $m$ a fractional value, in contrast to the integer values it takes in the previous examples. Figure 4.5 denotes a policy that requires traffic from client $C$ to be distributed evenly among five servers. In this way, NEAt uses *policy*

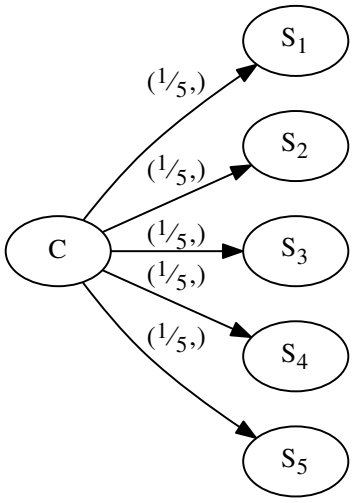



Figure 4.5: Load balancing policy

*graphs* to express both qualitative and quantitative reachability constraints on the network.

## 4.5 Repair Algorithm

In this section, we present NEAt's core algorithm for repairing violations in real-time constrained by a given policy graph. First, we introduce the network model and give an overview of the algorithm. Next, we describe our formulation of the repair problem for basic reachability policies as an integer linear programming (ILP) problem. We then generalize this approach to repair the wider range of policies discussed in § 4.4.

#### Network Model

As described in § 4.3, upon intercepting an update, NEAt constructs a network graph model for each affected EC that captures the forwarding behavior for all packets within the EC. This directed *configuration graph* $\ell_c$, along with a topology graph $T$ and policy graph $\wp$ serve as inputs to the repair algorithm.

Each node in these graphs represents a host or a networking device, and each edge between a pair of nodes defines reachability between them. The policy graph $\wp$ is a directed graph constructed from a set of conflict-free policies that represents the expected behavior of the whole network and hence should not be violated at runtime. Each node in the policy graph represents a device or a group of equivalent devices, and each edge $(u, v)$ represents an expected path from $u$ to $v$. Policies' conflict freedom can be guaranteed by

| *Symbol* | *Description* |
|---|---|
| $\ell_c$ | The configuration graph for equivalence class $c$. |
| $\wp$ | The policy graph. |
| $T$ | The topology graph. |
| $(i, j)$ | The edge from node $i$ to node $j$. |
| $\rho_{ij}$ | The paths between node $i$ and node $j$. |
| $C_i^c$ | The cluster of node $i$ for equivalence class $c$. |
| $c_i$ | The compressed node $i$ for $C_i^c$. |
| $E(a)$ | The set of all edges in graph $a$. |
| $N(E(a))$ | Number of all edges in graph $a$. |
| $NB_a(i)$ | The set of all neighbors of node $i$ in graph $a$. |

Table 4.1: Key notations in problem formulation

tools like PGA [87], which is out of the scope of this work. A topology graph $T$ is an undirected graph that represents the physical topology of the network.

**Algorithm Overview**

When the verification engine discovers a violated EC, the repair algorithm is executed. Its goal is to repair the detected violations optimally, i.e., with the minimum number of changes to the original configuration. Upon receiving the violated EC $c$ together with its configuration graph $\ell_c$, NEAt formulates the problem as an optimization problem: we aim to add or delete the minimum number of edges on $\ell_c$ so that the modified $\ell_c$ complies with $\wp_c$. $\wp_c$ is a subgraph of $\wp$ that is relevant to EC $c$. Note that the added or deleted edges are constrained within the topology graph $T$. We solve the optimization problem using ILP.

In § 4.5.1, we describe the repair algorithm for basic reachability policies, and § 4.5.2 enhances the basic algorithm to cope with the entire set of policies in § 4.4. We complete the section with our repair algorithm for forwarding loops (§ 4.5.3). Table 4.1 summarizes the key notations used in this section and § 4.6.

### 4.5.1 Repair Basic Reachability

After receiving a configuration graph $\ell_c$ that violates the desired policies from the verification engine, the optimizer determines the minimum number of edges that needs to be added or deleted to ensure $\ell_c$ is consistent with the policy graph $\wp_c$ using Integer Linear Programming (ILP). We start with the basic case where $\wp_c$ contains only reachability constraints.

Our integer program has a set of binary decision variables $x_{i,j,p,q}$ and $x_{i,j}$ where

$$x_{i,j,p,q}, (i,j) \in E_T, (p,q) \in E_{\wp_c} \tag{4.1}$$

$$x_{i,j}, (i,j) \in E_T \tag{4.2}$$

$E_T$ and $E_{\wp_c}$ denote the set of all edges in $T$ and $\wp_c$ respectively. Variable $x_{i,j,p,q}$ defines the mapping between a physical edge and a policy graph edge. It is 1 if a directed edge $(i,j)$ is mapped to policy edge $(p,q)$ for the current EC $c$, i.e., the flow from $p$ to $q$ will be forwarded through edge $(i,j)$ from $i$ to $j$. Variable $x_{i,j}$ defines whether or not edge $(i,j)$ is used for forwarding this EC's traffic regardless of which flow uses it. Edge $(i,j)$ in $T$ is selected if any flow $(p,q)$ is forwarded through $(i,j)$ (Equation 4.3). Similarly, for the other direction $(j,i)$, we have Equation 4.4. No physical link can be selected to forward traffic for the same EC on both directions (Equation 4.5) to avoid tight loops.

$$\forall (i,j) \qquad x_{i,j} \geq \sum_{(p,q) \in E_{\wp_c}} \frac{x_{i,j,p,q}}{N(E_\wp)} \tag{4.3}$$

$$\forall (j,i) \qquad x_{j,i} \geq \sum_{(p,q) \in E_{\wp_c}} \frac{x_{j,i,p,q}}{N(E_\wp)} \tag{4.4}$$

$$\forall (j,i) \qquad x_{i,j} + x_{j,i} \leq 1 \tag{4.5}$$

Equations 4.6-4.8 are the flow conservation equations for policy-level reachability $(p,q)$.

$\forall (p,q), \forall i \in T$:

$$\begin{cases} \sum_{j \in NB_T(i)} x_{i,j,p,q} = 1 \\ \sum_{j \in NB_T(i)} x_{j,i,p,q} = 0 \end{cases} \qquad \text{if } i = p \tag{4.6}$$

$$\begin{cases} \sum_{j \in NB_T(i)} x_{i,j,p,q} = 0 \\ \sum_{j \in NB_T(i)} x_{j,i,p,q} = 1 \end{cases} \qquad \text{if } i = q \tag{4.7}$$

$$\begin{cases} \sum_{j \in NB_T(i)} (x_{i,j,p,q} - x_{j,i,p,q}) = 0 \end{cases} \qquad \text{otherwise} \tag{4.8}$$

The optimization objective is to minimize the number of changes (additions and deletions) on the original configuration graph $\ell_c$.

$$\min \Big( \sum_{(i,j) \notin E_{\ell_c}} x_{i,j} - \sum_{(i,j) \in E_{\ell_c}} x_{i,j} \Big) \tag{4.9}$$

### 4.5.2 Generalizing the Algorithm

To support the general reachability policies as discussed in § 4.4, we encode several additional constraints into the ILP.

#### Isolation

We introduce a special $DROP$ node. If two nodes are required to be isolated, i.e., the nodes are connected with a $(0,)$ edge in the policy graph, we change the way the flow conservation equations are defined. More specifically, Equation 4.7 is changed to Equation 4.10. That is, a flow from $p$ to $q$ should sink at $DROP$ before reaching $q$.

$$\begin{cases} \sum_{j \in NB_T(i)} x_{i,j,p,q} = 0 \\ \sum_{j \in NB_T(i)} x_{j,i,p,q} = 1 \end{cases} \quad \text{if } i = DROP \tag{4.10}$$

#### Service Chaining

With service chaining, or waypointing, we enhance our flow conservation equations with Equation 4.11. This extends it beyond the individual reachability requirements in the policy, and takes into account dependencies between policy edges. The resulting mapping is guaranteed to satisfy chaining of reachability requirements. For instance, if a policy node $i$ is required to reach $q$ through $p$, because of this equation, $i$ cannot be mapped to the path segment $(p, q)$. Otherwise, $p$ might be skipped on the path from $i$ to $q$.

$$\begin{cases} \sum_{j \in NB_T(i)} x_{i,j,p,q} = 0 \\ \sum_{j \in NB_T(i)} x_{j,i,p,q} = 0 \end{cases} \quad \text{if } i \in \wp_c \text{ and } (\exists \rho_{i,p} \text{or } \exists \rho_{q,i}) \tag{4.11}$$

#### Bounded or Equal Path Length

A special case is a shortest path policy, where the bounded length is the length of the shortest physical path. If a path length bound $n$ is specified for a policy edge $(p, q)$, then a new constraint is added (Equation 4.12):

$$\sum_{(i,j) \in E_T} (x_{i,j,p,q} + x_{j,i,p,q}) \leq n \tag{4.12}$$

**Multipath**

If at least $m$ link-disjoint paths are required for flow $(p, q)$, the flow conservation equations (Equations 4.6 and 4.7) are updated with Equations 4.13 and 4.14, respectively.

$$\begin{cases} \sum_{j \in NB_T(i)} x_{i,j,p,q} \geq m \\ \sum_{j \in NB_T(i)} x_{j,i,p,q} = 0 \end{cases} \quad \text{if } i = p \tag{4.13}$$

$$\begin{cases} \sum_{j \in NB_T(i)} x_{i,j,p,q} \geq m \\ \sum_{j \in NB_T(i)} x_{j,i,p,q} = 1 \end{cases} \quad \text{if } i = q \tag{4.14}$$

**Load Balancing**

As discussed in § 4.4, policy edges within a load balancing policy are denoted with a decimal path count. Correspondingly, in our optimization problem, variables that map physical edges to policy edges are also decimal values between zero and one, instead of binary values. In addition to this change, we introduce a new equation (Equation 4.15) to capture how flow distribution propagates.

$$\prod_{x_{i,j,p,q} \neq 0} x_{i,j,p,q} = m \tag{4.15}$$

For example, consider a physical topology shown in Figure 4.6, with two layers of load balancing between client $C$ and servers $S1 - S5$. If the policy in Figure 4.5 is required, the solutions for variables $(x_{i,j})$ are shown in Figure 4.6.

### 4.5.3 Repairing Loops

The preceding repair algorithm operates on a *loop-free* configuration graph. As such, we first check for and remove loops from each configuration graph before compressing and repairing violations of any other property type. Our objective for repairing loops is to minimize change to the network, with a preference to affect as few equivalence classes as possible, as well as remove the minimal number of rules. Thus, our algorithm will remove a forwarding rule matching packets destined to 10.0.0.1/32 over one for 10.0.0.0/8. Since loops are repaired first, and NEAt will later check reachability properties on each equivalence class, our loop repair algorithm does not need to consider introducing permanent reachability violations by removing rules.

Algorithm 4.1 presents our loop repair algorithm. $\Theta(c)$ denotes the set of all loops appearing in a

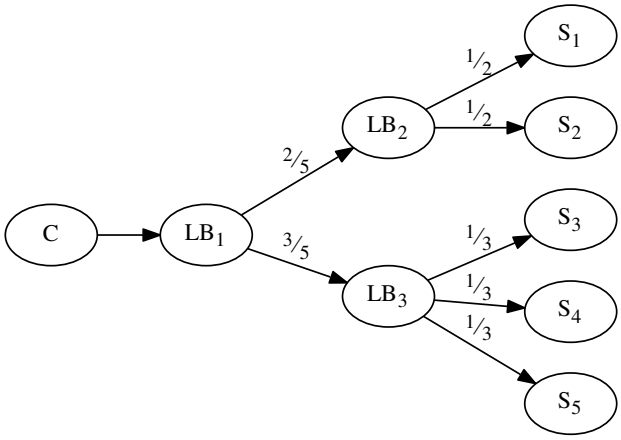



Figure 4.6: Load balancing configuration

configuration graph $\ell_c$ and $N(\Theta(c))$ the number of loops in $\ell_c$. $\theta_i$ is a subgraph of $\ell_c$, and $N(\theta_i) = 0$ when the subgraph contains no loops. The algorithm begins by finding and removing all intersecting edges across $\ell_c$'s loops. For each loop in $\ell_c$ that is not repaired by removing these edges, next remove an edge $(i, j)$ where $i$'s IP address is the destination, if such an edge exists. While $\theta_i$ still has loops, remove an edge in the loop that has the most specific match rule (e.g., longest prefix). Each edge is mapped to a specific forwarding rule at a particular switch when we compute the equivalence classes.

Removal of a forwarding rule is accomplished by replacing it with a drop rule, to prevent a coarser match in a separate equivalence class from introducing another loop. For example, if a rule matching destination IP 10.0.0.1/32 is simply deleted from a switch's forwarding table, another rule matching 10.0.0.1/31 on the same switch and forwarding to the same next hop could prevent the loop from being repaired. To conserve switch memory during repairs, NEAt checks all coarser drop rules to determine if multiple rules can be aggregated together.

## 4.6 Optimizations

While conceptually straightforward, the mapping algorithm in § 4.5 does not scale to a large network. In the optimization problem formulation, the number of variables for one EC is approximately the product of the number of edges in the physical topology and the number of edges of the policy graph, which can easily exceed 100k. In this section, we present two techniques that dramatically optimize the repair speed.

**Algorithm 4.1** Loop repair

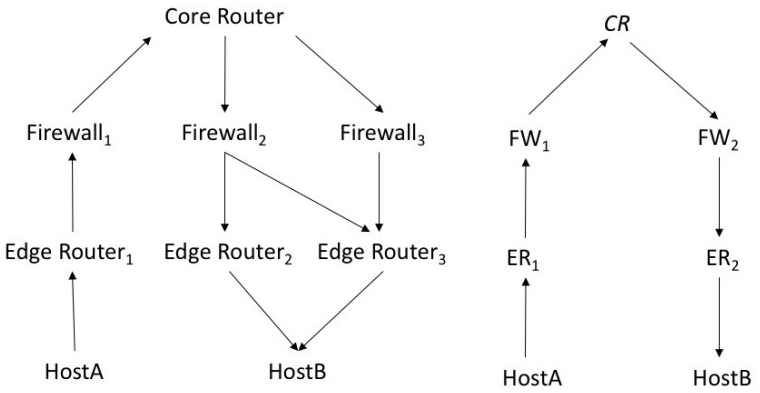
```
procedure RemoveLoop(ℓ_c, Θ(c))
    # remove edges appearing in multiple loops
    remove {(i, j) | (i, j) ∈ θ_k ∧ (i, j) ∈ θ_m ∀k, m ∈ Θ(c)}
    if N(Θ(c)) = 0 then
        return ℓ_c
    end if
    for all θ_i ∈ Θ(c) do
        while N(θ_i) > 0 do
            # remove edges forwarded out the destination
            remove (i, j) if i is destination
        end while
        while N(θ_i) > 0 do
            # remove most specific forwarding rule
            remove (i, j) ∈ θ_i with longest prefix
        end while
    end forreturn ℓ_c
end procedure
```

### 4.6.1 Topology Limitation

This technique aims to "slice" away irrelevant or redundant part of the network, and thus shrink the size of the optimization problem. After detecting a policy violation on a forward graph, before running our optimization mapping problem, we first remove disconnected components in the physical topology. Next, we localize the potential affected area in the topology. Fortunately, most modern networks are designed with a hierarchical structure. Examples include data centers arranged in a fat-tree topology, and enterprise networks divided into multiple sites joined by a backbone network. Such a structure implies certain communication patterns: communication within a subtree should stay local, for example, and communication between subtrees normally doesn't traverse other subtrees, i.e., traversing through a valley. Based on this, in our linear programming problem, typically only a subset of the topology is considered mappable to a policy edge. Results in § 4.8 shows the effectiveness of this technique.

### 4.6.2 Graph Compression

Other than hierarchical structures, most large networks are designed using patterns that enforce symmetry to some extent [85] for load balancing or resilience. For example, in a data center fat-tree topology, devices in the same layer (access, aggregate, core) are symmetrically connected to multiple devices on the neighboring layers. We exploit such regularities to compress the graphs that the repair algorithm operates on. The key to this idea is that the compressed graphs must be equivalent to the original graphs with respect to the policies of interest. To this end, we leverage a *graph pattern preserving* algorithm [41] as the




(a) Before compression (b) After compression

Figure 4.7: Example of compression

major building block of NEAt's compressor (Figure 4.1). The algorithm compresses a labeled directed graph according to a *bisimulation relation*.

**Bisimulation Relation [36]**

We denote $G = (V, E, L)$ as a labeled directed graph. $V$ represents a set of nodes and $(u, v) \in E$ represents a directed edge from node $u$ to node $v$. $L(u) \in \Gamma$ represents the label of node $u$, where $\Gamma$ is the set of labels applied to $V$. In the context of a networked system, the labels may represent a set of similarly functional nodes, e.g., hosts, firewalls, load balancers. For example, in Figure 4.7a, we label the network nodes as *Firewall*, *Edge Router* and *Core Router*, and we label the two hosts as $HostA$ and $HostB$.

A *bisimulation relation* on a graph $G = (V, E, L)$ is a binary relation $BR \subseteq V \times V$ such that for each $(u, v) \in BR$, *i*) $L(u) = L(v)$; *ii*) for each $(u, u') \in E$, there exists an edge $(v, v') \in E$ such that $(u', v') \in BR$; and *iii*) for each $(v, v') \in E$, there exists an edge $(u, u') \in E$ such that $(u', v') \in BR$. One can verify that in Figure 4.7a $Firewall_2$ and $Firewall_3$ are *bisimular* to each other, while $Firewall_1$ is not bisimular to any other firewall. Because $HostB$ is solely in a bisimular cluster, $EdgeRouter_1$ and $EdgeRouter_2$ are bisimular as they only have one child $HostB$. As $Firewall_2$ and $Firewall_3$ have children that are bisimular, they are bisimular to each other. While $Firewall_1$'s child is *Core Router*, which has a different label than *Edge Router*, $Firewall_1$ is not bisimular to anyone.

**Bisimulation Based Compression**

Algorithm 4.2 presents the compression algorithm on the given graphs $\ell_c$, $\wp_c$ and $T$. We compute the *bisimulation relation* on $\ell_c$ using the algorithms presented in [36] and then compress the graphs based on

their bisimularity. However, unlike $\ell_c$ and $\wp_c$, $T$ is not a directed graph, and thus the original algorithm is not applicable. To compute $T^{cp}$, we first compress the parts in $T$ that overlap with $\ell_c$ according to the undirected version of $\ell_c^{cp}$. We then add edges between the non-overlapping parts and the compressed parts with their original edges in $T$. The time complexity of the compression algorithm is $O(|E|log|V|)$. Figure 4.7b shows the compression result on graph $\ell_c$. $Firewall_2$ and $Firewall_3$ are bisimular and are compressed to a new cluster named $FW_2$. $Firewall_1$ remains by itself as $FW_1$. $\wp_c$ is compressed in a similar way.

**Algorithm 4.2** Graph pattern preserving compression

**procedure** CLUSTERING($\ell_c$, $\wp_c$, $T$)
  compute the maximum bisimulation relation $BR$ of $\ell_c$
  compute the clusters $clusters = V/BR$
  collapse the nodes in each $cluster \in clusters$
  compute compressed $\wp_c$, $\ell_c^{cp}$, $T^{cp}$
  **return** $\wp_c$, $\ell_c^{cp}$, $T^{cp}$
**end procedure**

We evaluate the compression algorithm on a simulated fat-tree topology and a large enterprise network. We denote the compression rate $r_c$ as the ratio of the number of remaining nodes in $\ell_c^{cp}$ to the number of nodes in the original graph $\ell_c$. Table 4.2 shows the compression results. From the results we can conclude that the compression algorithm could result in a much smaller amount of nodes for large-scale networks.

| Topology | $1 - r_c$ |
|---|---|
| Fat-tree (6750 hosts, 1125 switches) | 99.38% |
| Enterprise (236 routers) | 88.98% |

Table 4.2: Compression results

### Incremental Compression

Further leveraging the incremental compression algorithm from [41], we incrementally maintain the compressed configuration graphs. In response to changes on the original graphs, the incremental algorithm computes the new compressed graph using the changes and the compressed graph as input, independent of the original graph. That is, there is no need to decompress the graph to propagate the changes.

### Repairing Compressed Graphs

With the compression module in place, when a violation is detected, the graphs are compressed and then passed to the optimizer. Note that one compressed topology graph edge may represent a collection of original topology graph edges. This works with single-path reachability types of policies, such as reachability, isolation, or service chaining. However, it will break Equations 4.13 and 4.14 for link-disjoint multipath

policies. Our solution is to label predecessors of a multipath policy destination (e.g., $q$ for policy edge $(p, q)$) node differently, such that they are not compressed into a single cluster. In addition, the compressed topology graph is modeled as a weighted graph, where the weight on each edge is the number of original edges the compressed edge represents. Multipath policy constraint Equation 4.13 is modified accordingly as Equation 4.16, whereas Equation 4.14 remains the same because there are never multiple edges pointing to the destination node $q$.

$$\begin{cases} \sum_{j \in NB_{T^{cp}}(i)} (x_{i,j,p,q} * weight_{i,j}) >= m \\ \sum_{j \in NB_{T^{cp}}(i)} x_{j,i,p,q} = 0 \end{cases} \quad \text{if } i = p \tag{4.16}$$

**Map Back**

The last step is to map the results back to the original graphs. The optimization result is a set of changes (added or deleted edges) on the compressed graphs. To map back to the original configuration graph, a changed edge $(c_i, c_j)$ could become a set of changed edges between the cluster $C_i^c$ and cluster $C_j^c$. If an edge $(c_i, c_j)$ is supposed to be added to the compressed configuration graph, then on the original configuration graph, for every node $i$ in the source cluster $C_i^c$, there should be an edge added from $i$ to *one* of its neighbor node $j$ that is in the target cluster $C_j^c$. By the definition of bisimulation, such a node $j$ always exists. On the other hand, if an edge $(c_i, c_j)$ should be removed from the compressed configuration graph, *all* edges between the two clusters should be removed on the original graph. Afterward, the computed changes are translated into forwarding instructions and sent to the network devices.

**Policy Preservation**

We prove the compression algorithm preserves equivalence between the compressed graph $G_c$ and the original graph $G$ with respect to the scope of policies in § 4.4. Equivalence is proved in [41] for graph pattern queries. A graph pattern query is effectively asserting single-path type reachability and bounded path length. Therefore, we only need to prove the conclusion also holds for multipath policies.

**Theorem 4.1** (Multipath Equivalence)**.** *A multipath policy for a flow* $(p, q)$ *holds on* $G$ iff *the policy also holds for* $(p, q)$ *on* $G_c$.

*Proof.* Consider a multipath policy that requires at least $m$ paths for flow $(p, q)$. Trivially, if a flow $(p, q)$ satisfies the policy on $G$, the policy also holds for $(p, q)$ on $G_c$, and the flow conservation equations (Equations 4.16, 4.14 and 4.7) are satisfied. Therefore, we must prove the policy holds when Equation 4.16, 4.14, and 4.7 are satisfied.

Let $path_1^c, path_2^c, \ldots, path_n^c$ $(n \leq m)$ be the set of paths from $p$ to $q$ on $G_c$ that collectively satisfy Equations 4.16, 4.14, and 4.7. If $n$ equals $m$, then there are $m$ link-disjoint paths on $G_c$ between $p$ and $q$, and thus there are $m$ link-disjoint paths on $G$, i.e., the policy is satisfied.

If $n < m$, there must be at least a path on $G_c$, whose starting edge's weight is greater than one. Let those paths be $path_{m0}^c, \ldots, path_{mj}^c$, whose starting weights are $k_0, \ldots, k_j$ respectively. Consider path $path_{m0}^c$ first. This means its starting edge is pointing from $p$ to a cluster $C_{next}$ which contains at least $k_0$ nodes that are also $p$'s successors. Because the predecessors of $q$ are labeled differently, each is a separate cluster. By definition of the bisimulation relation, two nodes are bisimular (and thus can be clustered together) only if their children's label set is the same. Via back propagation, and constrained by Equation 4.7, there must be at least $k_0$ disjoint paths on $G$ from $p$'s successors in $C_{next}$ to $q$'s predecessors. Similarly, iterating through all paths from $p$ to $q$ on $G_c$, there are at least $m$ paths from $p$ to $q$ on $G$.

□

## 4.7 Implementation

We implemented a prototype of NEAt in Python. NEAt requires no modifications to the controller or switches. The verification engine is based on [61] and we use the Gurobi Optimizer [6] within our optimization engine to solve the ILP.

NEAt's pass-through mode is implemented as a proxy between the controller and switches, listening for flow modification messages. The interactive mode is implemented as an XML-RPC API, allowing it to be compatible with applications written in any language or for any controller. More specifically, NEAt exposes a `check()` function that accepts an OpenFlow flow modification message to check against the network policy. NEAt updates the network model with the proposed change, verifies the model, and searches for a set of repairs if any violations are found. The application can choose to receive the repairs as a set of OpenFlow flow modification messages or as a set of edge tuples. For example, a load balancer application may wish to receive the repairs as a set of tuples (e.g., *[(s2, h1)]*) to easily re-assign a client to a particular server replica, rather than parsing OpenFlow messages from NEAt.

## 4.8 Evaluation

In this section, we examine the performance of repairs in NEAt, as well as the end-to-end latency experienced by applications.

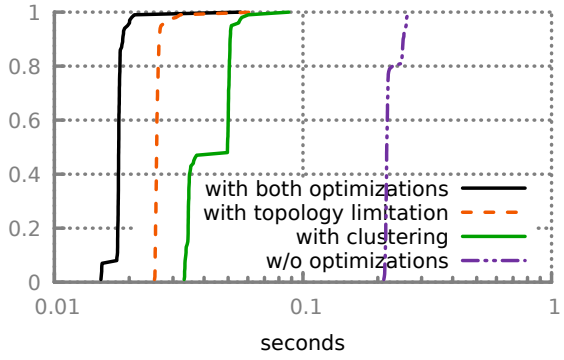
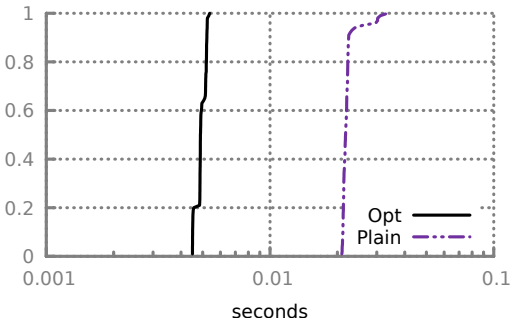
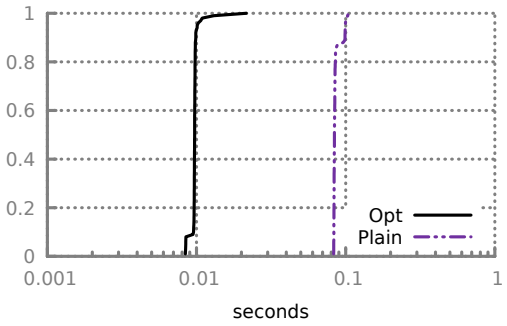


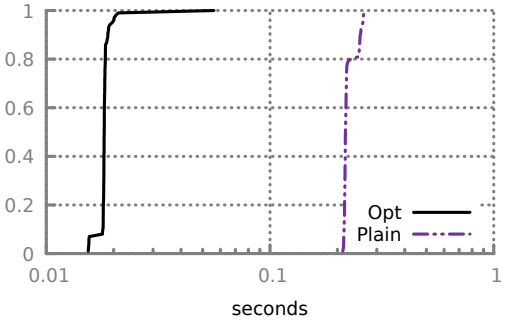

Figure 4.8: Effect of optimizations



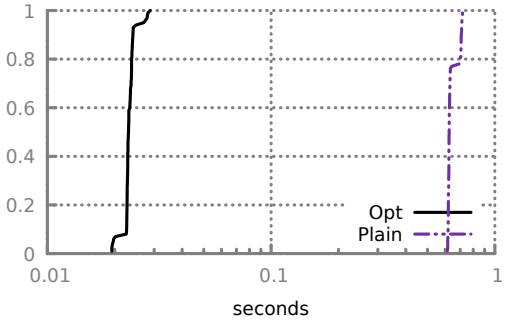

(a) 54 hosts, 45 switches



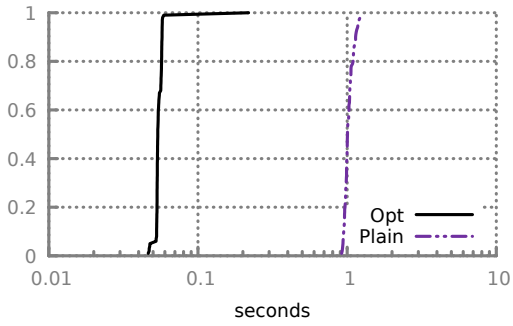

(b) 128 hosts, 80 switches




(c) 250 hosts, 125 switches




(d) 432 hosts, 180 switches




(e) 686 hosts, 245 switches

Figure 4.9: Repair time under random removals of exact matching rules

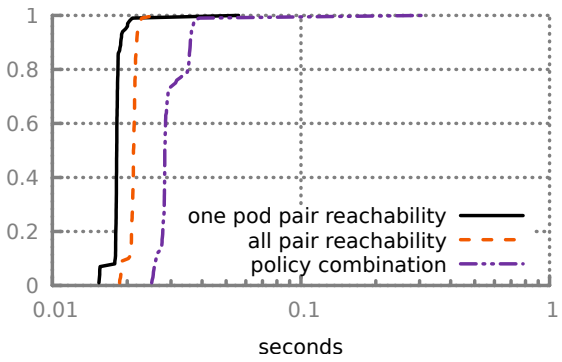



Figure 4.10: Repair time for different policies

### 4.8.1 Repair Performance

To evaluate the feasibility and scalability of NEAt's repair process, we synthesized a set of fat-tree topologies with various sizes, and used NEAt to maintain a variety of network-wide policies, including reachability, segmentation, bounded path length and multipath policies. More specifically, on each topology, under random removals of rules, we measured the repair time for each removal that caused a violation.

#### Exact Matching Rules

We first focus on flow-based traffic management applications, which are widely used in SDN [57, 49, 20, 55, 53]. Any forwarding rule produced by such applications at a switch matches at most one flow. In our terms, each rule only affects at most one EC.

For each fat-tree topology, we randomly selected a pair of pods. We used a *pod-pair reachability* policy, which specifies that any host in one of the pods should be able to reach every other host in both selected pods. We randomly removed a rule and triggered the optimizer to perform a repair on each EC violating this policy. We verified the resulting configuration graph after each repair and found that it satisfied the policy in every case.

On a fat-tree topology with 250 hosts and 125 switches, we measured the time taken to repair *pod-pair reachability* policy using four mechanisms: *i*) plain mapping, *ii*) mapping with topology limitation, *iii*) mapping with graph compression, and *iv*) mapping with both compression and topology limitation. Figure 4.8 compares the CDFs of the repair time for these four repair mechanisms. We can see the combination of graph compression and topology limitation (leftmost curve) brings approximately one order of magnitude speed-up over plain mapping (rightmost curve). Figure 4.9 (a-e) shows the speed-up increases as the network size scales. Even on a network with 686 hosts and 245 switches, the repair time is bounded under 0.1 second for the majority of cases, close to 1/20 of the repair time using plain mapping.

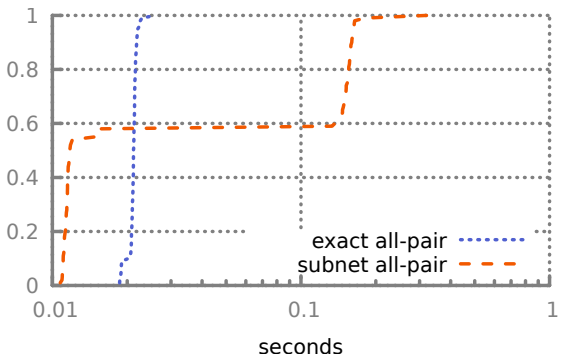



Figure 4.11: Exact matching rules vs. overlapping rules

We next explored how NEAt handles a larger set of policies and a combination of different types of policies. We use a policy specifies that every pair of hosts should be able to reach each other, which we will refer to as an *all-pair reachability* policy. On a fat-tree topology with 250 hosts and 125 switches, we measured the repair time under random rule removals against this *all-pair reachability* policy and report the results in Figure 4.10. The policy size is increased by approximately 10 times compared with *pod-pair reachability* policy, but the repair time only increases slightly.

To test a more complex setting, we randomly selected three pods in a fat-tree topology. Between the first two pods, we specify that hosts should be isolated from each other (*segmentation*), and between the first and third selected pods, hosts should be connected by at least two paths (*multipath*). For host pairs that do not fall into the previous two conditions, they should be able to reach each other (*all-pair reachability*). Both *multipath* and *all-pair reachability* are combined with a bounded path length policy, to avoid flows between pods traversing a valley. Note that unlike the previous pure single-path reachability policy, where repairs are all edge additions, in this case a repair can be a combination of edge additions and deletions. Furthermore, to satisfy the multipath requirement, more additions are necessary. Due to this complexity, the repair time increases, but is still on the same order of magnitude as the previous reachability policy cases, as shown in Figure 4.10. We verified each repair and found that the violation was repaired without affecting any of the other policies in every case.

### Overlapping Rules

For networks that use wild-carded rules or longest prefix matching, the assumption in the previous subsection does not hold. One rule may affect multiple ECs, and thus potentially trigger repairs on multiple graphs. Fortunately, there is a trend to move such overlapping rules to network edge or even hosts [75, 8, 25], leaving the core with exact matching rules. In order to study how NEAt performs under this less preferable

but less common scenario, we assign IP addresses within the same prefix subnet to hosts within the same pod on the fat-tree topologies. We then aggregated rules on the switches as much as possible. For example, each core switch has only $k$ forwarding rules, where $k$ is the number of pods, and each rule matches on one pod's prefix. Similar to the previous experiment, we used NEAt to guarantee an *all-pair reachability* policy, and our engine discovered repairs for all violations. Figure 4.11 compares the CDFs of the repair time for overlapping and exact matching rules on a 250-host-125-switch fat-tree topology. The repair took longer compared to applications with exact match rules due to the increased number of affected ECs. With our graph compression and topology limitation techniques, the repair finishes within 0.4 seconds in the worst case.

### 4.8.2 Repair Correctness

NEAt verifies and, if necessary, repairs each update from the controller separately (i.e., one flow modification message at a time). We ran the algorithm over a wide variety of real and synthetic networks and did not encounter a single case where the repair failed on the compressed configuration graph when run in this manner. Furthermore, we compared repairs computed on the compressed and uncompressed graphs, and the resulting changes were equivalent in all cases. However, as additional assurance, NEAt verifies the repair on the uncompressed graph and falls back to computing repairs on the uncompressed graph if the post-repair verification fails. To further reduce repair latency, we compute repairs on both the compressed and uncompressed graph in parallel, and terminate the latter process once the repair on the compressed graph is verified.

Nevertheless, we microbenchmarked this scenario by modifying our implementation to produce an incorrect repair on the compressed configuration graph. We evaluated the time for the entire process of computing repairs on both graphs in parallel and falling back to the uncompressed graph repair. We found that the resulting latency was still under 0.5 seconds on a 10-pod fat-tree network (i.e., 250 hosts, 125 switches) with a pod-pair reachability policy.

### 4.8.3 End-to-End Delay

Next, we examined the application-level delay introduced by NEAt when using its interactive mode. We tested NEAt on various-sized fat-tree topologies using Mininet [7] and the Pox controller [14]. A learning switch application and load balancer application run on top of Pox. The load balancer balances flows between the two replicas in a round-robin fashion, and we modified it to leverage NEAt's API to check the assignment of clients to replicas. If NEAt suggests a repair, the application updates its client-to-replica

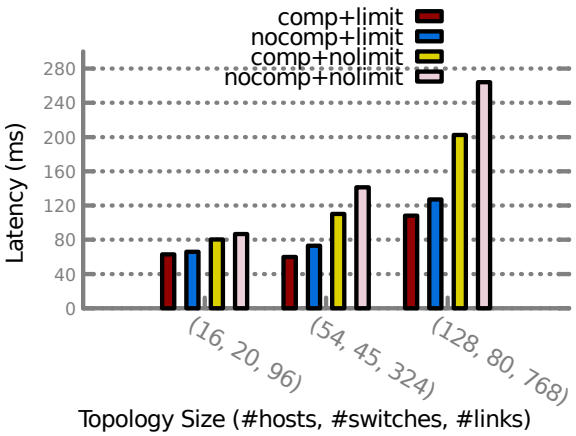



Figure 4.12: Application-perceived latency of NEAt, on various fat-tree topologies, showing performance for a reachability policy with/without graph compression and topology limitation

mapping with one suggested by NEAt. While the learning switch remains unmodified and unaware of NEAt, its updates are transparently checked by NEAt. This setup demonstrates the ability of NEAt to interact with the controller and applications simultaneously through its two integration modes.

The load balancer application runs on an edge switch in the fat-tree topology, with clients and server replicas placed in different pods. To trigger an update, a client pings the virtual IP of the load balancer. When the appropriate event handler in the load balancer is executed, it invokes NEAt's `check()` function. We measured the total latency introduced by NEAt as the time to invoke the `check()` function and apply any repairs to the application's state. This includes the time to verify an update (i.e, calculate equivalence classes affected by the update, compute their configuration graphs, and verify them) and repair violations in any of the affected equivalence classes.

For each topology size, we examined the total latency for a reachability policy, with and without our compression and topology limitation optimizations. Figure 4.12 shows the total delay experienced by the load balancer. Topology limitation has the largest speed-up of our optimizations, but when used in combination with compression of the topology and configuration graphs, NEAt can verify and repair an update in under 120ms.

### 4.8.4 Enterprise Network Trace Study

Finally, we examined traces from a large enterprise network, to evaluate NEAt's performance on real forwarding graphs. We examined two dumps of the data plane from 2014 and 2017. These datasets contained more than one million forwarding rules across more than 200 forwarding devices. The 2014 dataset contains 27k equivalence classes, while the 2017 trace contains 285k.

#### Bugs

For each dataset, we constructed loop and reachability policies and check for violations. In the 2014 dataset, NEAt found nine different loops. In the 2017 dataset, NEAt found 19. We examined the forwarding table and found several of these were caused by default routes with prefix 0.0.0.0/0. Only equivalence classes with more specific rules on the device were free of loops in these cases. Another cause we discovered is load balancing – a device can forward packets out one of two ports, one of which will result in a path containing a loop.

#### Simulation

We next used the 2017 dataset to evaluate NEAt's scale and performance on a data plane with a realistic number of equivalence classes. First, we verified and repaired any loops in the 285k equivalence classes contained in the dataset. We then constructed artificial updates, choosing a destination IP address and prefix length with the same probability as they appear in the dataset's forwarding rules. An update can either add a rule, delete a rule, or introduce a loop. Loops are chosen from the list of those that were discovered and repaired at the start of the simulation. An update has a 10% chance of introducing one of these loops, which may affect multiple ECs. We simulated 100 updates, affecting an average of eight ECs per update.

We applied the set of random updates to different combinations of policies, including loop freedom, reachability, and our compression and topology limitation optimizations. Since the compression and topology limitation optimizations only apply to reachability-type policies, we do not test loop freedom in combination with compression or topology limitation. Figure 4.13 shows a CDF of the total update time, including verification and repairs (when repairs are necessary). Of the 100 updates, 20 loops violations needed repair, as well as 24 reachability violations. Median and 98th percentile update times were 10ms and 1300ms, respectively, for a reachability policy with compression and topology limitation enabled. For a loop freedom property, median and 98th percentile update times were 35ms and 730ms, respectively. Combining these two policies, without compression or topology limitation optimizations, resulted in median and 98th percentile times of 36ms and 193 seconds. Adding our two optimizations reduced these times to 36ms and 6 seconds.

### 4.8.5 Repair as Configuration Synthesis

In this section, we explore the application of NEAt to synthesis — that is, generating an entire data plane state from scratch. Specifically, we examined the performance of NEAt on fat-tree topologies to "repair" an empty configuration graph. In Figure 4.14, we show the total time to repair and verify an all-pair

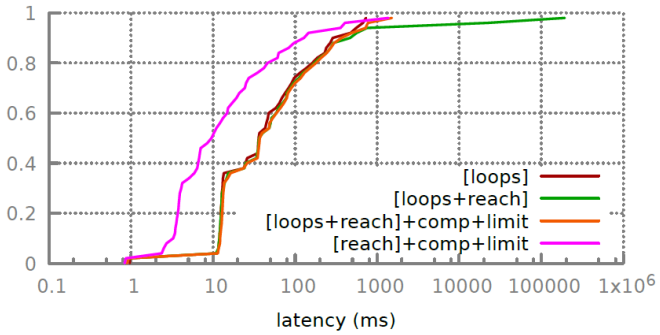



Figure 4.13: Total update time for different combinations of policies and optimizations, on a model of a real-world data plane trace

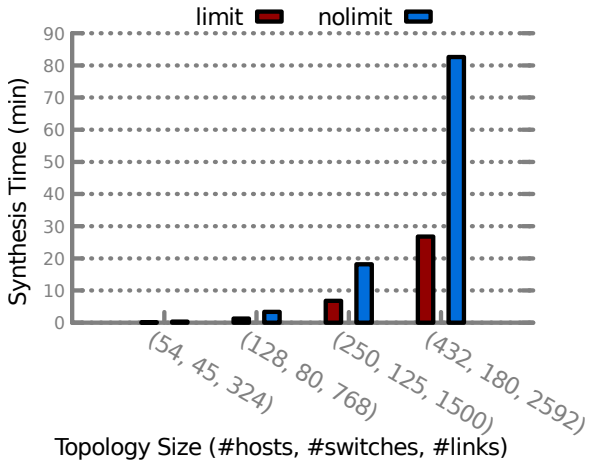



Figure 4.14: Repair time of an all-pair reachability property on an empty configuration graph

reachability policy. We report the repair time with and without our topology limitation optimization. For a 10-pod fat-tree topology, the median synthesis time was 6.8 seconds.

## 4.9 Synthesis and Repair

In the context of this work, we distinguish between synthesis and repair based on the scope and urgency of the change. We envision repair as a tool for temporary and immediate application, e.g., preventing security violations or downtime, given a partially correct implementation and a set of correctness policies to guide the repair. Afterward, the original cause of the policy violation might still need to be addressed, such as updating the configuration to reflect the new security policy or fixing the bug in the control program's implementation. A repair in NEAt, for example, is a change to a network after a single update. It expects the network to be in a "mostly correct" state – that is, prior to the policy-violating update, all correctness properties in the network were satisfied. Furthermore, except for aggregating drop rules to repair loops, NEAt does not

modify prefixes or match fields, for example, to reduce the number of flow table rules. We consider synthesis, on the other hand, to be useful for construction of longer-lived configurations and programs, without the need for a partial implementation. In keeping with our smartphone autocorrect analogy, a repair could correct a single misspelled word, while synthesis would correct the grammar of a complete sentence.

# Chapter 5

# Related Work

## 5.1 Fast-Forwarding

**Model Checking Programs** One class of techniques is model checking, where programs are modeled as FSAs and their behavior is comprehensively explored [46, 62, 79]. Recent work, like DeLorean, also combines model checking with symbolic execution [26, 34, 102]. However, most model checking work ignores time. This approach works well for programs that have a weak dependence on time, but the behavior of control programs that we study is intricately linked with time. Ignoring time in such programs can lead to exploring infeasible executions, and it cannot discover unexpected behaviors in which the mismatch is the time gap between events. One exception is NICE, which studies OpenFlow applications whose behavior can vary considerably based on packet timings [27]. However, its treatment of time is not systematic and instead relies on heuristics to explore timer behavior.

**Model Checking using TA** There has been much work on TA-based model checking in the real-time systems community. It includes developing efficient tools to explore the TA [19, 24] as well as transformations that speed up explorations [31, 50]. This body of work assumes that the entire TA is known in advance, and it does not target program analysis. While we draw heavily on the insights from it, to our knowledge, our work on DeLorean is the first to use TA to model check programs. We describe general methods to dynamically and comprehensively explore program executions and techniques to optimize exploration.

**Other Debugging Techniques** Explicit state model checking, which we use in DeLorean, is complementary to other program debugging approaches. Record and replay [67] can help diagnose faults after-the-fact and is especially useful for non-deterministic systems; in contrast, we want to determine if faults can arise *in the future*. There has also been work on "what-if" analysis in IP networks, e.g., with the use of shadow configurations [15] and route prediction [43]. These focus on computing the outcomes of configuration changes; in contract, we study the dynamic behavior of more general programs.

## 5.2 Orchestration

**Declarative Networking** In the pre-SDN era, declarative networking [72, 71, 74] — a combined effort of deductive database (recursive datalog) and distributed system (distributed query optimization) research — uses a distributed recursive query engine as an extensible and efficient routing infrastructure. This allows rapid implementation and deployment of new distributed protocols, making it an alternative design point that strikes a balance among its peers like overlay [73] and active networks [51]. In Ravel, we build on relational database research, making novel use of SQL views and contributing new data mediation techniques, with target usage — mediating applications with higher-level user support in a centralized setting — better described in network OS and SDN programming APIs.

**Database Usage in Network Controllers** The use of databases and the notion of network-wide views are not unfamiliar. Distributed controllers such as Onix [66] and ONOS [21] provide consistent network-wide views over distributed network elements and multiple controller instances. Unlike Ravel, these systems use the database as a transactional repository to "outsource" state management for distributed and replicated network states, and treat the database as a passive recipient that only executes queries and transactions. Furthermore, the network-wide views are often pre-defined by the system (e.g., Onix's NIB APIs with fixed schemas for all control applications), making little use of user-centered database views. In Ravel, the database *is* the reactive controller with user-centered database views. Control applications and the dynamic orchestrations are moved into the database itself, while SQL offers a native means to create and adjust application-specific ad-hoc views.

## 5.3 Autocorrect

**SDN Programming Languages** Many programming languages have been proposed abstractions to program SDNs, e.g., Frenetic [45], Pyretic [78] and Maple [97]. These allow programmers to compose complex rules without the need to manually resolve conflicts between rules. However, these languages face limitations implementing general policies that deliver higher-level intent, such as expressing middleware functionality or QoS constraints.

**SDN Synthesis Platforms** Network state can also be synthesized from a set of pre-specified correctness conditions. NetGen [90], for example, takes as input a specification using regular expressions to define path changes and a set of equivalence classes to modify. It uses an SMT solver to find the minimal number of changes. However, similar to Merlin [91] and FatTire [89], this tool is designed to be used as a compiler, with performance that is less applicable for real-time applications (i.e., minute-scale synthesis). While using

NetGen in place of our ILP is possible, doing so would additionally require translating each update into an equivalent NetGen specification. Similarly, Marham [52] proposes a framework for automated repair, but with slower performance — on the order of several seconds for topologies with tens of nodes and links. Margrave [80] analyzes changes to access control policy changes, highlighting to an operator the effect it has on the policy, without suggesting repairs to violations.

# Chapter 6

# Future Work

In this thesis, we discuss three primitives to help discover unpredictable behaviors in software-defined networks. However, these techniques are useful to other domains where a collection of devices are centrally controlled by applications, such as home automation (HA) or Internet-of-Things (IoT). In particular, searching for unpredictable behavior in a general control system can be applied to three similar challenges: composing the applications controlling the devices, verifying that each application is free of software bugs, and ensuring the output from the controller (i.e., the commands send to the devices) are correct.

In Chapter 2, we describe a model of control modules as a collection of event handlers that we believe applies to other domains. Furthermore, the focus on faithfully modeling time during program exploration is applicable to other domains whose applications depend on the relative and absolute time of events, such as IoT. Although extending DeLorean to another domain would require implementing a new model of the controller and devices, the exploration of application behavior would remain unchanged.

However, extending Ravel and NEAt to other domains presents additional challenges. Even though the underlying primitive implemented in these tools are applicable to other systems, they both build on the graph structure of networks. For example, Ravel abstracts a network into a set of tables that models application demands and data plane state as a graph. Similarly, correctness policies in NEAt are expressed as graphs. One direction for future work is to explore how to model other domains with this graph abstraction, as well as other abstractions that can be used with these tools.

In addition, these techniques may have applications to other properties beyond correctness and predictability. DeLorean, for example, could be used to check performance or security [28]. Consider scenarios where the firing or ordering of timers directly impacts performance (e.g., convergence of a distributed protocol). By systematically exploring future program states, DeLorean could report best case or worst case times.

Similarly, NEAt's transparent modification of updates could also address performance. NEAt could act as an optimization layer, transforming updates to meet switch memory constraints, balance traffic across links, or minimize path length. Effectively transforming updates to meet such diverse demands, however, introduces additional challenges. In particular, modifications to updates — whether to repair or optimize

a change to the data plane state — should preserve the intention of the update. In future work, we could explore this in two directions: analysis of applications and analysis of network changes. For the former approach, we could build on program analysis to extract invariants from programs. For the latter, we could mine properties and invariants about the network as it changes. Using these invariants, we could inform our search for repairs or modifications to updates. In the context of orchestration, we could additionally leverage these mined properties to resolve conflicts between applications that are codependent.

Additionally, we believe NEAt could have additional applications beyond repair, such as a security substrate to quickly and temporarily enforce updated security policies. For example, an operator could use NEAt to block traffic when hosts become infected with malware.

# Chapter 7

# Conclusion

This thesis explores techniques in verification and synthesis to ease the burden of configuring and controlling a software-defined network. In configuring an SDN, an operator must consider the behavior at the management plane, control plane, and data plane. At the management plane, an operator must reason about the interaction and conflicts between the applications controlling the network. At the control plane, she must reason about each individual application's execution to ensure it is free of software bugs. Finally, she must understand the implementation of the controller, to prevent unexpected behavior arising from the intricacies of the controller's implementation and processing of updates.

To this end, this thesis builds on verification and synthesis to propose three new primitives to help operators automatically search for unpredictable behavior in the applications controlling an SDN. Specifically, we design primitives for fast-forwarding, orchestration, and autocorrection. First, we build on verification to detect bugs in applications dependent on time. We develop a model checker, DeLorean, that systematically explores the behavior of home automation and SDN applications. With new techniques for speeding up this exploration, DeLorean can explore the future of behavior of these programs faster than they occur and, in essence, fast-forward a program.

Next, we build on synthesis techniques to reduce the complexity of composing together multiple applications. In automating the search for dependencies between applications, we design Ravel. Ravel adopts a relational representation of the network and uses a standard PostgreSQL database at its core. This enables ad-hoc creation of new programming abstractions and logic-based reasoning of inter-application dependencies.

Finally, to prevent multiple applications and their interaction from installing policy-violating updates, we introduce an autocorrection primitive with NEAt. NEAt allows backward-compatibility with existing SDN deployments and acts as a transparent layer between the network and controller, enforcing correctness by verifying and repairing updates that violate the network policy.

# References


[1] http://www.ravel-net.org/.

[2] http://github.com/ravel-net/ravel.

[3] http://www.infosecurity-magazine.com/opinions/to-err-is-human-to-automate-divine/.

[4] http://www.verizonenterprise.com/verizon-insights-lab/dbir/2016/.

[5] flow_table: check_for_overlapping_entries does not do overlap checking. https://github.com/noxrepo/pox/issues/142.

[6] Gurobi optimization. http://www.gurobi.com/.

[7] Mininet. http://mininet.org/.

[8] Network virtualization for cloud data centers. http://tinyurl.com/c9jbkuu.

[9] No FlowMod or FlowRemoved sent after receiving PortStatus down - Issue #645 - floodlight/floodlight. https://github.com/floodlight/floodlight/issues/645.

[10] [ONOS-6250] Change FlowRuleManager to use non-wall clock time to remove old rules. https://jira.onosproject.org/browse/ONOS-6250?jql=issuetype%20%3D%20Bug%20AND%20priority%20in%20(Blocker%2C%20Critical%2C%20Major).

[11] [ONOS-7228] Packet Service "cancel Packets" is not removing flows from devices. https://jira.onosproject.org/browse/ONOS-7228?jql=issuetype%20%3D%20Bug%20AND%20priority%20%3D%20Major.

[12] PostgreSQL. http://www.postgresql.org.

[13] Route views project. http://www.routeviews.org.

[14] The pox controller. https://github.com/noxrepo/pox.

[15] R. Alimi, Y. Wang, and Y. Yang. Shadow Configuration as a Network Management Primitive. In *SIGCOMM*, 2008.

[16] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 1994.

[17] ANTLR parser generator. http://antlr.org/.

[18] F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. *ACM Transactions on Database Systems*, 6(4):557–575, December 1981.

[19] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL: A Tool Suite for Automatic Verification of Real-time Systems. In *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III*, 1996.

[20] T. Benson, A. Anand, A. Akella, and M. Zhang. Microte: Fine Grained Traffic Engineering for Data Centers. In *CoNEXT*, 2011.

[21] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar. ONOS: Towards an Open, Distributed SDN OS. In *HotSDN*, 2014.

[22] J. A. Blakeley, N. Coburn, and P.-V. Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. In *ACM Transactions on Database Systems*, 1989.

[23] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational Lenses: A Language for Updatable Views. In *PODS*, 2006.

[24] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A Model-Checking Tool for Real-Time Systems. In *CAV*, 1998.

[25] B.Raghavan, M.Casado, T.Koponen, S.Ratnasamy, and a. S. S. A.Ghodsi. Software-Defined Internet Architecture: Decoupling Architecture from Infrastructure. In *HotNets*, 2012.

[26] M. Canini, V. Jovanovic, D. Venzano, B. Spasojevic, O. Crameri, and D. Kostic. Toward Online Testing of Federated and Heterogeneous Distributed Systems. In *ATC*, 2011.

[27] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE Way to Test Openflow Applications. In *NSDI*, 2012.

[28] R. Corin, S. Etalle, P. H. Hartel, and A. Mader. Timed Analysis of Security Protocols. *Computer Security*, 15(6), 2007.

[29] J. Croft, R. Mahajan, M. Caesar, and M. Musuvathi. Back to the future: Forecasting Program Behavior in Automated Homes. *Microsoft Research, Technical Report No. MSR-TR-2012-131*, 2012.

[30] J. Croft, R. Mahajan, M. Caesar, and M. Musuvathi. Systematically Exploring the Behavior of Control Programs. In *ATC*, 2015.

[31] C. Daws and S. Yovine. Reducing the Number of Clock Variables of Timed Automata. In *RTSS*, 1996.

[32] U. Dayal and P. A. Bernstein. On the Updatability of Relational Views. In *VLDB*, 1978.

[33] U. Dayal, E. N. Hanson, and J. Widom. Active Database Systems. In *Modern Database Systems*, pages 434–456. ACM Press, 1994.

[34] M. Dobrescu and K. Argyraki. Software Dataplane Verification. In *NSDI*, 2014.

[35] X. L. Dong and D. Srivastava. Big Data Integration. In *ICDE*, 2013.

[36] A. Dovier, C. Piazza, and A. Policriti. A Fast Bisimulation Algorithm. In *CAV*.

[37] ELK products, inc. `http://www.elkproducts.com/`.

[38] C. Elkan. Independence of Logic Database Queries and Update. In *PODS*, 1990.

[39] D. Evans. The Internet of things. `http://blogs.cisco.com/news/the-internet-of-things-infographic/`, 2011.

[40] R. Fagin, J. D. Ullman, and M. Y. Vardi. On the Semantics of Updates in Databases. In *PODS*, 1983.

[41] W. Fan, J. Li, X. Wang, and Y. Wu. Query Preserving Graph Compression. In *SIGMOD*, 2012.

[42] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul. FlowTags: Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions. In *HotSDN*, 2013.

[43] N. Feamster and J. Rexford. Network-wide Prediction of BGP Routes. *IEEE/ACM Trans. Networking*, April 2007.

[44] N. Foster, A. Guha, M. Reitblatt, A. Story, M. J. Freedman, N. P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, and R. Harrison. Languages for Software-Defined Networks. *IEEE Communications Magazine*, 51(2):128–134, 2013.

[45] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ICFP*, 2011.

[46] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *POPL*, 1997.

[47] A. Gupta and I. S. Mumick. Materialized Views. chapter Maintenance of Materialized Views: Problems, Techniques, and Applications, pages 145–157. MIT Press, Cambridge, MA, USA, 1999.

[48] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *SIGMOD*, 1993.

[49] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: Saving Energy in Data Center Networks. In *NSDI*, 2010.

[50] M. Hendriks, K. G. Larsen, M. Hendriks, and K. G. Larsen. Exact Acceleration of Real-Time Model Checking. In *Electronic Notes in Theoretical Computer Science*, 2001.

[51] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Programming Language for Active Networks. In *PLDI*, 1998.

[52] H. Hojjat, P. Reummer, J. McClurgh, P. Cerny, and N. Foster. Optimizing Horn Solvers for Network Repair. In *FMCAD*, 2016.

[53] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving High Utilization with Software-Driven WAN. In *SIGCOMM*, 2013.

[54] Universal devices products/insteon/isy-99i series. `http://www.universal-devices.com/99i.htm`.

[55] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Holzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-Deployed Software Defined WAN. In *SIGCOMM*, 2013.

[56] X. Jin, J. Gossels, J. Rexford, and D. Walker. CoVisor: A Compositional Hypervisor for Software-Defined Networks. In *NSDI*, 2015.

[57] X. Jin, R. Mahajan, H. H. Liu, R. Gandhi, S. Kandula, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic Scheduling of Network Updates. In *SIGCOMM*, 2014.

[58] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *NSDI*, 2013.

[59] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *NSDI*, 2012.

[60] A. M. Keller. Updating Relational Databases Through Views, 1995.

[61] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-wide Invariants in Real Time. In *NSDI*, 2013.

[62] C. E. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI*, 2007.

[63] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable Dynamic Network Control. In *NSDI*, 2015.

[64] H. Kim, A. Voellmy, S. Burnett, N. Feamster, and R. Clark. Lithium: Event-Driven Network Control. *Georgia Institute of Technology SCS Technical Report GT-CS-12-03*, 2012.

[65] J. C. King. Symbolic Execution and Program Testing. *CACM*, 19(7), 1976.

[66] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-Scale Production Networks. In *OSDI*, 2010.

[67] T. Leblanc and J. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, 36, 1987.

[68] M. Lenzerini. Data Integration: A Theoretical Perspective. In *PODS*, 2002.

[69] A. Y. Levy. Logic-based Artificial Intelligence. chapter Logic-based Techniques in Data Integration. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[70] A. Y. Levy and Y. Sagiv. Queries Independent of Updates. In *VLDB*, 1993.

[71] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking: Language, Execution and Optimization. In *SIGMOD*, 2006.

[72] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking. In *CACM*, 2009.

[73] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *SOSP*, 2005.

[74] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. *SIGCOMM Comput. Commun. Rev.*, 35(4):289–300, August 2005.

[75] M.Casado, T.Koponen, S.Shenker, and A.Tootoonchian. Fabric: A Retrospective on Evolving SDN. In *HotSDN*, 2012.

[76] J. C. Mogul, A. AuYoung, S. Banerjee, L. Popa, J. Lee, J. Mudigonda, P. Sharma, and Y. Turner. Corybantic: Towards the Modular Composition of SDN Control Programs. In *HotNets*, 2013.

[77] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A Compiler and Run-time System for Network Programming Languages. In *POPL*, 2012.

[78] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software-Defined Networks. In *NSDI*, 2013.

[79] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *OSDI*, 2002.

[80] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The Margrave Tool for Firewall Analysis. In *LISA*, 2010.

[81] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi. Tierless Programming and Reasoning for Software-Defined Networks. In *NSDI*, 2014.

[82] Open vSwitch Manual. `http://openvswitch.org/support/dist-docs/ovs-ofctl.8.txt`.

[83] N. W. Paton and O. Díaz. Active Database Systems. *ACM Comput. Surv.*, 31(1):63–103, March 1999.

[84] pgRouting Project. `http://pgrouting.org/`.

[85] G. D. Plotkin, N. Bjrner, N. P. Lopes, A. Rybalchenko, and G. Varghese. Scaling Network Verification using Symmetry and Surgery. In *POPL*, 2016.

[86] P. Porras, S. Shin, V. Yegneswaran, M. Fong, M. Tyson, and G. Gu. A Security Enforcement Kernel for OpenFlow Networks. In *HotSDN*, 2012.

[87] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *SIGCOMM*, 2015.

[88] Psycopg: the PostgreSql adaptor for Python. `http://initd.org/psycopg/`.

[89] M. Reitblatt, M. Canini, A. Guha, and N. Foster. FatTire: Declarative Fault Tolerance for Software-defined Networks. In *HotSDN*, 2013.

[90] S. Saha, S. Prabhu, and P. Madhusudan. NetGen: Synthesizing Data-plane Configurations for Network Policies. In *SOSR*, 2015.

[91] R. Soulé, , S. Basu, P. Marandi, F. Pedone, R. Kleinberg, E. Sirer, and N. Foster. Merlin: A Language for Provisioning Network Resources. In *CoNEXT*, 2014.

[92] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *SIGCOMM*, 2002.

[93] P. Sun, R. Mahajan, J. Rexford, L. Yuan, M. Zhang, and A. Arefin. A Network-State Management Service. In *SIGCOMM*, 2014.

[94] N. Tillmann and J. de Halleux. Pex: White Box Test Generation for .NET. In *TAP*, 2008.

[95] J. D. Ullman. Information Integration Using Logical Views. In *ICDT '97*, 1997.

[96] N. Vasić, P. Bhurat, D. Novaković, M. Canini, S. Shekhar, and D. Kostić. Identifying and Using Energy-Critical Paths. In *CoNEXT*, 2011.

[97] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *SIGCOMM*, 2013.

[98] A. Wang, J. Croft, and E. Dragut. Reflections on Data Integration for SDN. In *SDN-NFV Security*, 2017.

[99] A. Wang, X. Mei, J. Croft, M. Caesar, and P. Godfrey. Ravel: A Database-Defined Network. In *SOSR*, 2016.

[100] R. Wang, D. Butnariu, and J. Rexford. OpenFlow-based Server Load Balancing Gone Wild. In *Hot-ICE*, 2011.

[101] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.

[102] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *NSDI*, 2009.

[103] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MoDist: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI*, 2009.

[104] Y. Yuan, R. Alur, and B. T. Loo. NetEgg: Programming Network Policies by Examples. In *HotNets*, 2014.

[105] W. Zhou, J. Croft, B. Liu, and M. Caesar. NEAt: Network Error Auto-Correct. In *SOSR*, 2017.

[106] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. Godfrey. Enforcing Customizable Consistency Properties in Software-Defined Networks. In *NSDI*, 2015.

[107] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *SIGMOD*, 1995.