Program Synthesis for Software-Defined Networking

by

Jedidiah McClurg

M.S., Computer Science, Northwestern University, 2013

B.S., Electrical Engineering, University of Iowa, 2009

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2018

This thesis entitled:
Program Synthesis for Software-Defined Networking
written by Jedidiah McClurg
has been approved for the Department of Computer Science

Prof. Pavol Černý

Prof. Bor-Yuh Evan Chang

Prof. Nate Foster

Prof. Dirk Grunwald

Prof. Mooly Sagiv

Prof. Sriram Sankaranarayanan

Date ____________________

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

McClurg, Jedidiah (Ph.D., Computer Science)

Program Synthesis for Software-Defined Networking

Thesis directed by Prof. Pavol Černý

Software-defined networking (SDN) is revolutionizing the networking industry, but even the most advanced SDN programming platforms lack mechanisms for changing the **global configuration** (the set of all forwarding rules on the switches) correctly and automatically. This seemingly-simple notion of global configuration change (known as a **network update**) can be quite challenging for SDN programmers to implement by hand, because networks are distributed systems with hundreds or thousands of interacting nodes—even if the initial and final configurations are correct, naïvely updating individual nodes can lead to bugs in the intermediate configurations. Additionally, SDN programs must simultaneously describe both static forwarding behavior, and dynamic updates in response to events. These **event-driven updates** are critical to get right, but even more difficult to implement correctly due to interleavings of data packets and control messages. Existing SDN platforms offer only weak guarantees in this regard, also opening the door for incorrect behavior. As an added wrinkle, event-driven network programs are often physically distributed, running on several nodes of the network, and this distributed setting makes programming and debugging even more difficult. Bugs arising from any of these issues can cause serious incorrect transient behaviors, including loops, black holes, and access-control violations.

This thesis presents a synthesis-based approach for solving these issues. First, I show how to automatically **synthesize network updates** that are guaranteed to preserve specified properties. I formalize the network updates problem and develop a synthesis algorithm based on counterexample-guided search and incremental model checking. Second, I add the ability to reason about transitions between configurations in response to events, by introducing **event-driven consistent updates** that are guaranteed to preserve well-defined behaviors in this context. I propose **network event structures** (NESs) to model constraints on updates, such as which events can be enabled simultaneously and causal dependencies between events. I define an extension of the NetKAT language with mutable state, give semantics to stateful programs using NESs, and discuss provably-correct strategies for implementing NESs in SDNs. Third, I propose a

synchronization synthesis approach that allows correct "parallel composition" of several event-driven programs (**processes**)—the programmer can specify each sequential process, and add a declarative specification of paths that packets are allowed to take. The synthesizer then inserts synchronization among the distributed controller processes such that the declarative specification will be satisfied by all packets traversing the network. The key technical contribution here is a counterexample-guided synthesis algorithm that furnishes network processes with the synchronization required to prevent any races causing specification violations. An important component of this is an extension of network event structures to a more general programming model called **event nets** based on Petri nets. Finally, I describe an approach for **implementing event nets** in an efficient distributed way on modern SDN hardware. For each of the core components, I describe a prototype implementation, and present results from experiments on realistic topologies and properties, demonstrating that the tools handle real network programs, and scale to networks of 1000+ nodes.

Dedication

To my wife Anna and my trusty little sidekick, Ernest the ragdoll cat.

Acknowledgements

I am indebted to many people who have helped and encouraged me in this undertaking. Thanks to my wife and family for being supportive every step of the way. Thanks to fellow climbers at the Boulder Rock Club for introducing me to Colorado and pushing me to climb harder. Special thanks to the students and faculty of the CUPLV group at CU Boulder, who welcomed me and made me feel like a valued member of the community. Many thanks to Nate Foster and Hossein Hojjat—I profoundly enjoyed our research collaborations, and hope we will continue to have many more. Last but not least, thanks to my advisor Pavol Černý. I owe a massive debt of gratitude to him for his guidance through the years, and I do not think I could have made it to this point without such an exceptional mentor.

Chapter 2 was originally published as "Efficient Synthesis of Network Updates" (McClurg, Hojjat, Cerny, and Foster [85]) in The 36th ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI 2015), and the Artifact Evaluation Committee (AEC) found that the prototype implementation "met or exceeded expectations." In regards to that paper, I would like to thank my co-authors for invaluable assistance and ideas, the anonymous PLDI reviewers and AEC members for insightful feedback on the paper and artifact, as well as Xin Jin, Dexter Kozen, Mark Reitblatt, and Jennifer Rexford for helpful comments. Andrew Noyes and Todd Warszawski contributed a number of early ideas through an undergraduate research project. The work was supported by the National Science Foundation (NSF) under awards CCF-1421752, CCF-1422046, CCF-1253165, CNS-1413972, CCF-1444781, and CNS-1111698; the Office of Naval Research (ONR) under Award N00014-12-1-0757; and gifts from Fujitsu Labs and Intel.

Chapter 3 was originally published as "Event-driven Network Programming" (McClurg, Hojjat, Foster, and Cerny [86]) in The 37th ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI 2016), and the AEC found that the prototype implementation “met or exceeded expectations.” In regards to that conference paper, I would like to thank my co-authors for their help, the anonymous PLDI reviewers for offering helpful and constructive comments, as well as Zach Tatlock for shepherding the paper and providing useful feedback. The work was supported by the NSF under grants CNS-1111698, CNS-1413972, CCF-1421752, CCF-1422046, CCF-1253165, and CCF-1535952; the ONR under grant N00014-15-1-2177; and gifts from Cisco, Facebook, Fujitsu, Google, and Intel.

Chapter 4 was originally published as “Synchronization Synthesis for Network Programs” (McClurg, Hojjat, and Cerny [84]) in The 29th Intl. Conference on Computer-Aided Verification (CAV 2017), and the AEC found that the prototype implementation “met or exceeded expectations.” In regards to that conference paper, I would like to thank my co-authors for their invaluable help, as well as Nate Foster and P. Madhusudan for fruitful discussions. The research was supported in part by the NSF under award CCF 1421752, and by DARPA under agreement FA8750-14-2-0263.

Contents

Tables

Figures

Chapter 1

Introduction

Networked computer systems are everywhere. Billions of people use web-based services every day for communication, social networking, navigation, etc., and these services rely heavily on the correct and efficient functioning of complex networked systems, such as datacenters. When things go wrong in these systems, the consequences can make headlines—for example, Google cloud services experienced a failure in April 2016 due to a combination of human error and software bugs [47], and Twitter went down in January 2016 due to problems caused by faulty software upgrades [116]. Beyond hurting users, downtime can cost companies tens of thousands of dollars per minute [28]. The stakes are only becoming higher, as networks are relying on programmable devices and more complex software, and this added complexity means that the human effort required in (re)configuring these systems could quickly become unmanageable. What is needed is **better programming languages/tools for networked systems**, and in this thesis, I outline a promising new approach based on program synthesis.

1.1 Difficulties of Programming Networked Systems

At the heart of many web-based applications is a datacenter containing a complex network, possibly with tens of thousands of interconnected servers. Traffic flows between these servers through devices known as **switches**, which process and forward packets efficiently using specialized hardware. The **network operator** can change the **configuration** of the network (the global set of forwarding tables on all switches) by communicating with individual switches. **Software-Defined Networking (SDN)** makes this process easier, by providing a uniform API for "programming" the switches (I will subsequently refer to the network operator as the "programmer").

SDN is a relatively new paradigm in which a logically-centralized controller machine manages a collection of programmable switches. The controller responds to **events** such as explicit commands from the programmer, topology changes, shifts in traffic load, or new connections from hosts, by pushing forwarding rules to the switches. Because the controller has global visibility and full control over the entire network, SDN programs can implement a wide range of advanced network functionality including fine-grained access control [21], network virtualization [70], traffic engineering [56, 54], and many others. SDN has been used in production enterprise, datacenter, and wide-area networks, and new deployments are rapidly emerging.

Much of SDN's power stems from the controller's ability to change the **global** state of the network. Controllers can set up end-to-end forwarding paths, provision bandwidth to optimize utilization, or distribute access control rules to defend against attacks. However, implementing these global changes in a running network is not easy. Networks are complex systems with many distributed switches, but for all practical purposes, the controller can only modify the configuration of one switch at a time. Hence, to implement even a simple global change between two static network configurations (referred to as a **network update**), an SDN programmer must explicitly transition the network through a sequence of intermediate configurations to reach the intended final configuration. The code needed to implement such a transition is tedious to write and prone to error—in general, the intermediate configurations may exhibit new behaviors that would not arise in the initial and final configurations.

Beyond just simple operator-initiated transitions between static initial and final network configurations, in real network applications, these transitions are often triggered **dynamically** by events such as packets arriving at certain switches. This complicates things further, since it then becomes necessary to reason about interleavings of control messages and regular data packets being processed by the switches.

Finally, even though SDN provides the abstraction of a centralized controller, in practice, such a setup would not provide adequate performance, so it is desirable to have **distributed** control operating at multiple nodes of the network. This adds an additional layer of complexity, since even correct control applications can work together improperly to produce incorrect behavior.

The goal of this thesis is to address these problems. More concretely, I examine the following four core problems.

(1) **High-level programmability**: SDN allows switch forwarding tables to be programmatically set, but how can programmers write a single program that describes behavior of the network as a whole?

(2) **Correctness of network programs**: while the network is "running" a program, how can programmers ensure that consistency/security invariants are not violated due to unexpected switch-level delays/errors, or concurrency bugs within the program itself?

(3) **Event-driven behavior**: beyond simply specifying static configurations, how can programmers write programs which efficiently and automatically change the configuration due to network events?

(4) **Correct distributed behavior**: high-level network behavior often takes the form of several applications operating concurrently—how can programmers specify such behavior, utilize errors/traces to rule out unwanted interleavings (concurrency bugs), and produce efficient distributed implementations for such programs?

1.2 Core Contribution: Synthesis of SDN Programs

Program synthesis is a recent approach for solving a hard problem: given a high-level specification of a program's **behavior**, automatically determine the program's code. Much success has been achieved using a **synthesize-verify loop**, where a **verifier** repeatedly checks whether a **candidate program** satisfies the specification, and **counterexamples** are used to guide the search for new candidates. This is often performed using off-the-shelf tools such as SMT solvers or model checkers, and typically cannot be directly applied to large (distributed) SDN programs.

In this thesis, I present a set of techniques for effectively extending program synthesis techniques to the context of network programs. By properly extending program synthesis techniques to the SDN context, I am able to directly address the previously-mentioned network-programming difficulties. In particular, I focus on two main goals in this thesis, (I) **designing new programming abstractions for SDN** which enable everyday programmers to write correct and efficient network functionality, and (II) **building scalable development tools**, e.g., **synthesizers** to assist programmers with writing difficult networking code, which are often powered by **verifiers** to check the correctness of existing networking code.

Thesis Statement: Program synthesis enables an accessible and efficient approach to producing real-world network programs which are correct and efficiently implementable in an SDN.

In particular, **accessible** means the tools and systems should be usable by network programmers, and **efficient** means the algorithms should readily scale to at least thousands of nodes, making them usable in the context of small real-world datacenters. **Real-world** means the languages and formalisms should allow actual network programs to be modeled, and **correct** means the resulting program should satisfy (customizable) correctness properties that are of interest in the networking literature. Finally, **efficiently implementable** means the running network programs should be distributed, i.e., avoid relying on the controller, and operate at line-rate, i.e., avoid the use of expensive mechanisms such as blocking.

In this document, I defend this thesis statement by providing strong evidence to support each of the following four hypotheses.

Hypothesis 1. The problem of performing a network update which is correct with respect to customizable single-packet properties can be solved efficiently using counterexample-guided program synthesis.

I use the term **single-packet property** to mean a formula that each packet being processed in the network must satisfy, e.g., "if a packet enters the network from Host 1, it must eventually arrive at Host 2," and **correctness** with respect to such properties means that packets in any state of the network (even intermediate transient states) must satisfy the property.

Hypothesis 2. In addition to languages/guarantees that work well in the context of network updates, it is also possible to develop intuitive languages/guarantees that enable easy reasoning in the event-driven context.

In other words, it is possible to produce a language with built-in support for event-driven behavior, as well as reasonable guarantees about which global configuration should process each packet. By **intuitive**, I mean that the language and guarantees should be similar to existing tools used by network programmers, and should be able to capture real networking applications.

Hypothesis 3. The problem of repairing concurrent event-driven programs, that is, synthesizing program constructs which cause the programs to satisfy a specification over single-packet traces, can be solved cleanly and efficiently using a counterexample-guided program synthesis approach.

By **concurrent event-driven programs**, I mean a set of event-driven applications which are intended to be operating in the network at the same time. By **cleanly and efficiently**, I mean that the solution should enable straightforward reasoning about concurrency in this context, and should scale to network programs and topologies of realistic size.

Hypothesis 4. It is possible to use a modern stateful switch architecture to build mechanisms which efficiently implement the event-driven programs.

By **modern stateful switch architecture**, I mean a platform which endows switches with stateful memory, i.e., local registers which can be read and written by incoming packets. By **efficient implementation**, I mean one that is distributed (i.e., does not reside solely on a single controller), and does not use expensive operations such as blocking of packets.

1.3 Overview of Contributions

In this section, I outline the individual components of my synthesis-based solution.

1.3.1 Synthesis of Network Updates

I begin by considering the most basic type of network program, **network updates**. Even in this context, writing network programs correctly is challenging due to the concurrency inherent in SDNs—switches may interleave packet and control message processing arbitrarily. Hence, programmers must carefully consider all possible message orderings, inserting synchronization primitives as needed. The algorithm presented here works by searching through the space of possible sequences of individual switch updates, learning from counterexamples and employing an incremental model checker to re-use previously computed results. The model checker is **incremental** in the sense that it exploits the loop-freedom of correct network configurations to enable efficient re-checking of properties when the model changes. Because the synthesis algorithm poses a series of closely-related model checking questions, the incrementality yields large performance gains on the update scenarios examined in the experiments.

Example: Simple Network Update. To illustrate key challenges related to network updates, consider the network in Figure 1.1. It represents a simplified datacenter topology [40] with core switches (C1

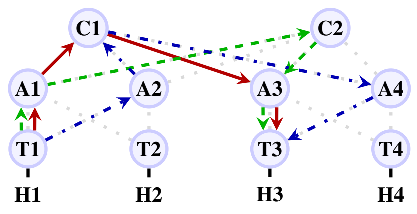


Figure 1.1: Example topology.

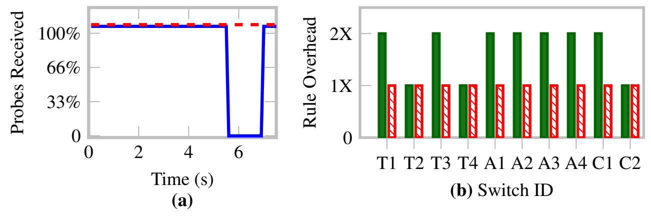


Figure 1.2: Example naïve (blue/solid-line), two-phase (green/solid-bar), and ordering (red/dashed) updates: (a) probes received; (b) per-switch rule overhead.

and C2), aggregation switches (A1 to A4), top-of-rack switches (T1 to T4), and hosts (H1 to H4). Initially, I configured switches to forward traffic from H1 to H3 along the solid/red path: T1-A1-C1-A3-T3. Later, I decided to shift traffic from the red path to the dashed/green path, T1-A1-C2-A3-T3 (perhaps to take C1 down for maintenance). To implement this update, the operator must modify forwarding rules on switches A1 and C2, but note that certain update sequences break connectivity—e.g., updating A1 followed by C2 causes packets to be forwarded to C2 before it is ready to handle them. Figure 1.2(a) demonstrates this with a simple experiment performed using my system. Using the Mininet network simulator and OpenFlow switches, I continuously sent ICMP (`ping`) probes during a "naïve" update (blue/solid line) and the ordering update synthesized by my tool (red/dashed line). With the naïve update, 100% of the probes are lost during an interval, while the ordering update maintains connectivity.

Consistency. Previous work [105] introduced the notion of a consistent update and also developed general mechanisms for ensuring consistency. An update is said to be **per-packet consistent** if every packet is processed entirely using the initial configuration or entirely using the final configuration, but never a mixture of the two. For example, updating A1 followed by C2 is not consistent because packets from H1 to H3 might be dropped instead of following the red path or the green path. One might wonder whether preserving consistency during updates is important, as long as the network eventually reaches the intended configuration, since most networks only provide best-effort packet delivery. While it is true that errors can be masked by protocols such as TCP when packets are lost, there is growing interest in strong guarantees about network behavior. For example, consider a business using a firewall to protect internal servers, and suppose that they decide to migrate their infrastructure to a virtualized environment like Amazon EC2. To ensure that this new deployment is secure, the business would want to maintain the same isolation properties enforced in their home office. However, a best-effort migration strategy that only eventually reaches the target configuration could step through arbitrary intermediate states, some of which may violate this property.

Two-Phase Updates. Previous work introduced a general consistency-preserving technique called **two-phase update** [105], which involves explicitly tagging packets upon ingress and using these tags to determine which forwarding rules to use at each hop. Unfortunately, this has a significant cost. During the transition, switches must maintain forwarding rules for **both** configurations, effectively doubling the memory requirements needed to complete the update. This is not always practical in networks where the switches store forwarding rules using ternary content-addressable memories (TCAM), which are expensive and power-hungry. Figure 1.2(b) shows the results of another experiment where I measured the total number of rules on each switch: with two-phase updates, several switches have twice the number of rules compared to the synthesized ordering update. Even worse, it takes a non-trivial amount of time to modify forwarding rules—sometimes on the order of 10ms per rule [59]! Hence, because two-phase updates modify a large number of rules, they can increase update latency. This can make two-phase updates a non-starter.

Ordering Updates. My approach is based on the observation that consistent (two-phase) updates are overkill in many settings. Sometimes consistency can be achieved by simply choosing a correct order of switch updates. I call this type of update an **ordering update**. For example, to update from the red path to the green path, I can update C2 followed by A1. Moreover, even when achieving full consistency is impossible, one can often still obtain sufficiently strong guarantees for a specific application by carefully updating the switches in a particular order. To illustrate, suppose that instead of shifting traffic to the green path, I wish to use the blue (dashed-and-dotted) path: T1-A2-C1-A4-T3. It is impossible to transition from the red path to the blue path by ordering switch updates without breaking consistency: I can update A2 and A4 first, as they are unreachable in the initial configuration, but if I update T1 followed by C1, then packets can traverse the path T1-A2-C1-A3-T3, while if I update C1 followed by T1, then packets can traverse the path T1-A1-C1-A4-T3. Neither of these alternatives is allowed in a consistent update. This failure to find a consistent update hints at a solution: if I only care about preserving connectivity between H1 and H3, then either path is actually acceptable. Thus, either updating C1 before T1, or T1 before C1 would work. Hence, if I relax strict consistency and instead provide programmers with a way to specify properties that must be preserved across an update, then ordering updates will exist in many situations. Recent work [80, 59] has explored ordering updates, but only for specific properties like loop-freedom, blackhole-freedom, drop-freedom, etc. Rather than handling a fixed set of "canned" properties, I use a specification language that is expressive enough to encode these properties and others, as well as conjunctions/disjunctions of properties—e.g. enforcing loop-freedom **and** service-chaining during an update.

In-flight Packets and Waits. Sometimes an additional synchronization primitive is needed to generate correct ordering updates (or correct two-phase updates, for that matter). Suppose I want to again transition from the red path to blue one, but in addition to preserving connectivity, I want every packet to traverse either A2 or A3 (this scenario might arise if those switches are actually middleboxes which scrub malicious packets before forwarding). Now consider an update that modifies the configurations on A2, A4, T1, C1, in that order. Between the time that I update T1 and C1, there might be some packets that are forwarded by T1 before it is updated, and are forwarded by C1 after it is updated. These packets would not traverse A2 or A3, and so indicate a violation of the specification. To fix this, I can simply pause after updating T1 until any packets it previously forwarded have left the network. It is thus necessary to have a command "wait" that pauses the controller for a sufficient period of time to ensure that in-flight packets have exited the network. Hence, the correct update sequence for this example would be as above, with a "wait" between T1 and C1. Note that two-phase updates also need to wait, once per update, since one must ensure that all in-flight packets have left the network before deleting the old version of the rules on switches. Other approaches have traded off control-plane waiting for stronger consistency, e.g. [79] performs updates in "rounds" that are analogous to "wait" commands, and Consensus Routing [62] relies on timers to obtain wait-like functionality. Note that the single-switch update time can be on the order of seconds [59, 73], whereas typical datacenter transit time (the time for a packet to traverse the network) is much lower, even on the order of microseconds [1]. Hence, waiting for in-flight packets has a negligible overall effect. In addition, I provide a reachability-based heuristic eliminates most waits in practice.

Summary. This part of the thesis presents a sound and complete algorithm and implementation for synthesizing a large class of ordering updates efficiently and automatically. The network updates it generates initially modify each switch at most once and "wait" between updates to switches, but a heuristic removes an

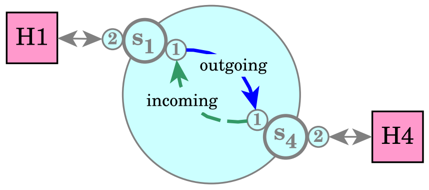


Figure 1.3: Topology for simple Stateful Firewall.

overwhelming majority of unnecessary waits in practice. For example, in switching from the red path to the blue path (while preserving connectivity from H1 to H3, and making sure that each packet visits either A3 or A4), my tool produces the following sequence: update A2, then A4, then T1, then wait, then update C1. The resulting update can be executed using the Frenetic SDN platform and used with OpenFlow switches—e.g., I generated Figure 1.2 (a-b) using my tool. I ran experiments on a suite of real-world topologies, configurations, and properties—my results demonstrate the effectiveness of synthesis, which scales to over one-thousand switches, and incremental model checking, which outperforms a popular symbolic model checker used in **batch** mode, and a state-of-the-art network model checker used in incremental mode.

1.3.2 Event-Driven Network Programming

In practice, many network programs are more complex than a network update. For example, various applications of interest are **dynamic** in that they produce changes to the global network configuration in response to **events**, such as arrival of packets at certain switches, etc. In this part of the thesis, I seek to determine what is the "best" way to specify event-driven network programs, and how to reason about and ensure correctness in this context.

Example: Stateful Firewall. To illustrate the challenges that arise when implementing dynamic applications, consider a topology as in Figure 1.3, where an internal host H_1 is connected to switch s_1, an external host H_4 is connected to a switch s_4, and switches s_1 and s_4 are connected to each other. Suppose I wish to implement a stateful firewall: at all times, host H_1 is allowed to send packets to host H_4, but H_4 should only be allowed to send packets to H_1 if H_1 previously initiated a connection. Implementing even this simple application turns out to be difficult, because it involves coordinating behavior across multiple devices and packets. The basic idea is that upon receiving a packet from H_1 at s_4, the program will need to issue a command to install a forwarding rule on s_4 allowing traffic to flow from H_4 back to H_1. There are two straightforward (but incorrect) implementation strategies on current SDN controllers.

(1) The outgoing request from H_1 is diverted to the controller, which sets up flow tables for the incoming path and also forwards the packet(s) to H_4. Reconfiguring flow tables takes time, so H_4's response will likely be processed by the default drop rule. Even worse, if the response is the SYN-ACK in a TCP handshake, normal retransmission mechanisms will not help—the client will have to wait for a timeout and initiate another TCP connection. In practice, this greatly increases the latency of setting up a connection, and potentially wreaks havoc on application performance.

(2) The outgoing request is buffered at the controller, which sets up the flow tables for the incoming path but waits until the rules are installed before forwarding the packet(s). This avoids the problem in (1), but places extra load on the controller and also implements the firewall incorrectly, since incoming traffic is allowed before the outgoing request is delivered. Leaving the network unprotected (even briefly) can be exploited by a malicious attacker.

Thus, while it is tempting to think that reliability mechanisms built into protocols such as TCP already prevent (or at least reduce) these types of errors, this is not the case. While it is true that some applications can tolerate long latencies, dropped packets, and weak consistency, problems with updates **do** lead to serious problems in practice. As another example, consider an intrusion detection system that monitors suspicious traffic—inadvertently dropping or allowing even a few packets due to a reconfiguration would weaken the protection it provides. The root of these problems is that **existing SDN frameworks do not provide strong guarantees during periods of transition between configurations in response to events**. An eventual guarantee is not strong enough to implement the stateful firewall correctly, and even a (per-packet) consistent update would not suffice, since consistent updates only dictate what must happen to individual packets.

Existing Approaches. Experienced network operators may be able to use existing tools/methods to correctly implement event-driven configuration changes. However, as seen above, this requires thinking carefully about the potential interleavings of events and updates, delegating atomic operations to the controller (incurring a performance hit), etc.

As mentioned, there are stateful programming systems that attempt to make this process easier for the programmer, but update strategies in these systems either offer no consistency guarantees during dynamic updates, rely on expensive processing via the controller, and/or require the programmer to craft an update protocol by hand. In this thesis, I group these approaches together, using the term **uncoordinated update** to describe their lack of support for coordinating local updates in a way that ensures global consistency.

Event-Driven Consistent Update. I propose a new correctness condition with clear guarantees about updates triggered by events. This enables specification of how the network should behave during updates, and enables precise formal reasoning about dynamic network programs.

An **event-driven consistent update** is denoted as a triple $C_i \xrightarrow{e} C_f$, where C_i and C_f are the initial and final configurations respectively, and e is an event. Intuitively, these configurations describe the forwarding behaviors of the network before/after the update, while the event describes a phenomenon, such as the receipt of a packet at a particular switch, that triggers the update itself. Semantically, an event-triggered consistent update ensures that for each packet:

(1) **the packet is forwarded consistently,** i.e. it must be processed entirely by a **single configuration** C_i or C_f, and

(2) **the update does not happen too early,** meaning that if every switch traversed by the packet has **not** heard about the event, then the packet must be processed by C_i, and

(3) **the update does not happen too late,** meaning that if every switch traversed by the packet **has** heard about the event, then the packet must be processed by C_f.

The first criterion requires that updates are consistent, which is analogous to a condition proposed previously by Reitblatt et al. [105]. However, a consistent update alone would not provide the necessary guarantees for the stateful firewall example, as it applies only to a single packet, and not to multiple packets in a bidirectional flow. The last two criteria relate the packet-processing behavior on each switch to the events it has "heard about." Note that these criteria leave substantial flexibility for implementations: packets that do not satisfy the second or third condition can be processed by either the preceding or following configuration. It remains to define what it means for a switch s to have "heard about" an event e that occurred at switch t (assuming $s \neq t$). I use a causal model and say that s hears about e when a packet, which was processed by t after e occurred, is received at s. This can be formalized using a "happens-before" relation.

Returning to the stateful firewall, it is not hard to see that the guarantees offered by event-driven consistent updates are sufficient to ensure correctness of the overall application. Consider an update $C_i \xrightarrow{e} C_f$. In C_i, H_1 can send packets to H_4, but not vice-versa. In C_f, additionally H_4 can send packets to H_1. The event e is the arrival at s_4 of a packet from H_1 to H_4. Before e occurs, can H_4 send a packet to H_1, as is possible in C_f? No, since **none** of the switches along the necessary path have heard about the event. Now, imagine that the event e occurs, and H_4 wants to send a packet to H_1 afterwards. Can s_4 drop the new packet, as it would have done in the initial configuration C_i? No, because the **only** switch the packet would traverse is s_4, and s_4 has heard about the event, meaning that the only possible correct implementation should process this new packet in C_f.

Event-Driven Transition Systems. To specify event-driven network programs, I use labeled transition systems called **event-driven transition systems** (ETSs). In an ETS, each node is annotated with a network configuration and each edge is annotated with an event. For example, the stateful firewall application would be described as a two-state ETS, one state representing the initial configuration before H_1 has sent a packet to H_4, and another representing the configuration after this communication has occurred. There would be a transition between the states corresponding to receipt of a packet from H_1 to H_4 at s_4. This model is similar to the finite state machines used in Kinetic [68] and FAST [91]. However, whereas Kinetic uses uncoordinated updates, I impose additional constraints on the ETSs which allow them to be implemented **correctly** with respect to my consistency property. For example, I extend event-triggered consistent updates to **sequences**, requiring each sequence of transitions in the ETS to satisfy the property. For simplicity, in Chapter 3, I focus on finite-state systems and events corresponding to **packet delivery**. However, these are not fundamental assumptions—my design extends naturally to other notions of events, as well as infinite-state systems.

Network Event Structures. The key challenge in implementing event-driven network programs stems from the fact that at any time, the switches may have different views of the global set of events that have occurred. Hence, for a given ETS, several different updates may be enabled at a particular moment of time, necessitating a way to resolve conflicts. I turn to the well-studied model of event structures [123], which allows me to constrain transitions in two ways: (1) **causal dependency**, which requires that an event

e_1 happens before another event e_2 may occur, and (2) **compatibility**, which forbids sets of events that are in some sense incompatible with each other from occurring in the same execution. I present an extension called **network event structure** (NES), and show how an ETS can be encoded as an NES.

Locality. While event-driven consistent updates require immediate responses to **local** events (as in the firewall), they do not require immediate reactions to events “at a distance.” This is achieved by two aspects of my definitions.

The first defining aspect of the locality requirements involves the happens-before (“heard-about”) relation in event-driven consistent update. For example, receipt of a packet in New York can not immediately affect the behavior of switches in London. Intuitively, this makes sense: requiring “immediate” reaction to remote events would force synchronization between switches and buffering of packets, leading to unacceptable performance penalties. Event-driven consistent update only requires the switches in London to react **after** they have heard about the event in New York.

The second defining aspect of the locality requirements involves the compatibility constraints in NESs. Suppose that New York sends packets to London and Paris, but the program requires transitioning to a different global state based on who received a packet first. Clearly, it would be impossible to implement this behavior without significant coordination. However, suppose New York and Philadelphia are sending packets to London, and the program requires transitioning to a different global state based on whose packet was received first in London. This behavior is easily implementable since the choice is local to London. I use NESs to rule out non-local incompatible events—specifically, I require that incompatible events must occur at the same switch.

This approach gives consistency guarantees even when an event occurs at a switch different from the one that will be updated. The change will not happen “atomically” with the event that triggered it, but (a) every packet is processed by a single configuration, and (b) the configuration change occurs as dictated by event-driven consistent update (happens-before) requirements. I show that these requirements can be implemented with minimal performance penalty.

Locality issues are an instance of the tension between **consistency** and **availability** in distributed systems, which motivates existing SDN languages to favor availability (avoiding synchronization and packet buffering) over consistency (offering strong guarantees when state changes). I demonstrate that it is possible to provide the same level of availability as existing systems, while providing a natural consistency condition that is powerful enough to build many applications. I also show that weakening the locality requirement would necessitate weakening availability, meaning that my model is in some sense the "best" model for event-driven network programming.

Together, this provides a new programming abstraction based on (i) a notion of causal consistency requiring that events are propagated between nodes, (ii) per-packet consistency governing how packets are forwarded through the network, and (iii) locality requirements. I believe this is a powerful combination that is a natural fit for building many applications.

Implementing Network Programs. NESs also provide a natural formalism for guiding an implementation technique for stateful programs. Intuitively, this requires switches that can record the set of events that have been seen locally, make decisions based on those events, and transmit events to other switches. Fortunately, in the networking industry there is a trend toward more programmable data planes: mutable state is already supported in most switch ASICs (e.g. MAC learning tables) and is also being exposed to SDN programmers in next-generation platforms such as OpenState [13] and P4 [16]. Using these features, an NES can be implemented as follows.

(1) Encode sets of events in the NES as tags that can be carried by packets and tested on switches.

(2) Compile the configurations contained in the NES to a collection of forwarding tables.

(3) Add "guards" to each configuration's forwarding rules to test for the tag enabling the configuration.

(4) Add rules to "stamp" incoming packets with tags corresponding to the current set of events.

(5) Add rules to "learn" which events have happened by reading tags on incoming packets and adding the tags in the local state to outgoing packets, as required to implement the happens-before relation.

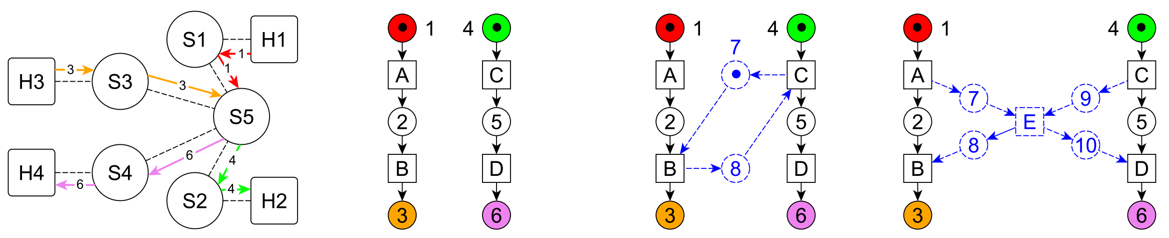
In this thesis, I show that a system implemented in this way **correctly** implements an NES.

Evaluation. To evaluate my design, I built a prototype of the system described here. I have used this to build a number of event-driven network applications: (a) a stateful firewall, which I have already described; (b) a learning switch that floods packets going to unknown hosts along a spanning tree, but uses point-to-point forwarding for packets going to known hosts; (c) an authentication system that initially blocks incoming traffic, but allows hosts to gain access to the internal network by sending packet probes to a predefined sequence of ports; (d) a bandwidth cap that disables access to an external network after seeing a certain number of packets; and (e) an intrusion detection system that allows all traffic until seeing a sequence of internal hosts being contacted in a suspicious order. I have also built a synthetic application that forwards packets around a ring topology, to evaluate update scalability. I developed these applications in an extended version of NetKAT which I call **Stateful NetKAT**. My experiments show that my implementation technique provides competitive performance on several important metrics while ensuring important consistency properties. I draw several conclusions. (1) Event-driven consistent update allow programmers to easily write real-world network applications and get the correct behavior, whereas approaches relying only on uncoordinated consistency guarantees do not. (2) The performance overhead of maintaining state and manipulating tags (measured in bandwidth) is within 6% of an implementation that uses only uncoordinated update. (3) There is an optimization that exploits common structure in rules across states to reduce the number of rules installed on switches. In my experiments, a basic heuristic version of this optimization resulted in a 32-37% reduction in the number of rules required on average.

1.3.3 Synchronization Synthesis for Event-Driven Network Programs

Software-defined networking (SDN) enables programmers or **network operators** to more easily implement important applications such as traffic engineering, distributed firewalls, network virtualization, etc. These applications are typically **event-driven**, in the sense that the packet-processing behavior can change in response to network events such as topology changes, shifts in traffic load, or arrival of packets at various network nodes. SDN enables this type of event-driven behavior via a **controller machine** that manages the network **configuration**, i.e., the set of **forwarding rules** installed on the network **switches**. The programmer can write code which runs on the controller, as well as instruct the switches to install custom forwarding rules, which inspect incoming packets and move them to other switches or send them to the controller for custom processing.

Concurrency in Network Programs. Although SDN provides the abstraction of a **centralized** controller machine, in reality, network control is often physically distributed, with controller processes run-



(a) Configurations (b) Input Net (c) Iteration 1 (d) Output Net

Figure 1.4: Example #1

ning on multiple network nodes [30]. The fact that these distributed programs control a network which is **itself** a distributed packet-forwarding system means that event-driven network applications can be especially difficult to write and debug. In particular, there are two types of races that can occur, resulting in incorrect behavior. First, there are races between updates of forwarding rules at individual switches, or between packets that are in-flight during updates. Second, there are races among the different processes of the distributed controller. I call the former **packet races**, and the latter **controller races**. Bugs resulting from either of these types of races can lead to serious problems such as packet loss and security violations.

Illustrative Example. I will begin by examining the difficulties of writing distributed controller programs, in regards to the two types of races. Consider the network topology in Figure 4.1a. In the initial configuration, packets entering at $H1$ are forwarded through $S1, S5, S2$ to $H2$. There are two controllers (not shown), $C1$ and $C2$—controller $C1$ manages the upper part of the network ($H1, S1, S5, S3, H3$), and $C2$ manages the lower part ($H2, S2, S5, S4, H4$). Now imagine that the network operator wants to take down the forwarding rules that send packets from $H1$ to $H2$, and instead install rules to forward packets from $H3$ to $H4$. Furthermore, the operator wants to ensure that the following property ϕ holds at all times: **all packets entering the network from** $H1$ **must exit at** $H2$. When developing the program to do this, the network operator must consider the following:

- Packet race: If $C1$ removes the rule that forwards from $S1$ to $S5$ before removing the rule that forwards from $H1$ to $S5$, then a packet entering at $H1$ will be dropped at $S1$, violating ϕ.
- Controller race: Suppose $C1$ makes no changes, and $C2$ adds rules that forward from $S5$ to $S4$, and from $S4$ to $H4$. In the resulting configuration, a packet entering at $H1$ can be forwarded to $H4$, again violating ϕ.

Synthesis Approach. I present a program synthesis approach that makes it easier to write distributed controller programs. The programmer can specify each sequential process (e.g., $C1$ and $C2$ in the previous example), and add a declarative specification of paths that packets are allowed to take (e.g., ϕ in the previous example). The synthesizer then inserts **synchronization constructs** that constrain the interactions among the controller processes to ensure that the specification is always satisfied by any packets traversing the network. Effectively, this allows the programmer to reduce the amount of effort spent on keeping track of possible interleavings of controller processes and inserting low-level synchronization constructs, and instead focus on writing a declarative specification which describes allowed packet paths. In the examples I have considered, I find these specifications to be a clear and easy way to write desired correctness properties.

Network Programming Model. In my approach, similar to network programming languages like OpenState [13], and Kinetic [68], I allow a network program to be described as a set of concurrently-operating finite state machines (FSMs) consisting of event-driven transitions between global network states. I generalize this by allowing the input network program to be a set of **event nets**, which are 1-safe Petri nets where each transition corresponds to a network event, and each place corresponds to a set of forwarding rules. This model extends network event structures [86] to enable straightforward modeling of programs with loops. An advantage of extending this particular programming model is that its programs can be efficiently implemented without packet races (see Section 4.2 for details).

Problem Statement. My synthesizer has two inputs: (1) a set of event nets representing sequential processes of the distributed controller, and (2) a linear temporal logic (LTL) specification of paths that packets are allowed to take. For example, the programmer can specify properties such as “packets from $H1$ must pass through Middlebox $S5$ before exiting the network.” The output is an event net consisting of the input event nets and added synchronization constructs, such that all packets traversing the network satisfy the specification. In other words, the added synchronization eliminates problems caused by controller races. Since I use event nets, which can be implemented without packet races, both types of races are eliminated in the final implementation of the distributed controller.

Algorithm. My main contribution is a counterexample-guided inductive synthesis (CEGIS) algorithm for event nets. This consists of (1) a **repair engine** that synthesizes a candidate event net from the input event nets and a finite set of known counterexample traces, and (2) a **verifier** that checks whether the candidate satisfies the LTL property, producing a counterexample trace if not. The repair engine uses SMT to produce a candidate event net by adding synchronization constructs which ensure that it does not contain the counterexample traces discovered so far. Repairs are chosen from a variety of constructs (barriers, locks, condition variables). Given a candidate event net, the verifier checks whether it is deadlock-free (i.e., there is an execution where all processes can proceed without deadlock), whether it is 1-safe, and whether it satisfies the LTL property. I encode this as an LTL model-checking problem—the check fails (and returns a counterexample) if the event net exhibits an incorrect interleaving.

Evaluation. I have implemented this technique, and evaluated the tool on examples from the SDN literature. I show that the prototype implementation can fix realistic concurrency bugs, and can readily scale to problems featuring network topologies of 1000+ switches.

Contributions. This chapter contains the following contributions:

- I describe **event nets**, a new model for representing concurrent network programs, which extends several previous approaches, enables using and reasoning about many synchronization constructs, and admits an efficient distributed implementation (Sections 4.1-4.2).
- I present **synchronization synthesis for event nets**. To my knowledge, this is the first counterexample-guided technique that automatically adds synchronization constructs to Petri-net based programs. My solution includes a **model checker for event nets**, and an SMT-based **repair engine for event nets** which can insert a variety of synchronization constructs (Section 4.3).
- I show the usefulness and efficiency of my prototype implementation, using several examples featuring network topologies of 1000+ switches (Section 4.4).

1.3.4 Data-Plane Mechanisms for Distributed Network Programming

One of the goals of software-defined networking (SDN) is to make networks more programmable. In practice, real SDN implementations such as OpenFlow require complicated stateful functionality to be handled on the **controller** machine—the switches simply serve to hold static forwarding tables, which are (re)populated by the controller. This model is beginning to change. The SDN **data-plane** (packet-processing functionality) is becoming more advanced, with powerful devices emerging which are able to perform computations and update local state based on packet contents, all at line rate [13, 16, 110]. This has fueled an increased interest in pushing functionality which normally occurred in the control-plane into the data-plane. Instead of viewing a **network program** as simply a process that runs on the controller and interacts with switches, it can now be viewed as a **distributed system**, running atop the networking hardware.

My event nets formalism provides the ability to write network programs which fit this paradigm. However, the language does not deal with **global state** in a fully general way. The focus of this chapter is to build a network programming language and runtime that extends event nets to give programmers a convenient and correct way of managing global state.

More specifically, in this chapter, I provide a declarative callback-based language known as **callback nets** for describing network behavior. Instead of the event nets model, in which events change state between static configurations, callback nets allow events to trigger a set of modifications to **global variables**. These global variables are built from conflict-free replicated datatype (CRDT) [109] registers, and updates to these are propagated lazily by piggybacking on data packets. In this way, causality is maintained, i.e., devices that have received a data packet which passed through a switch with newer state will also see the new state.

The second key contribution of this chapter is a compiler that produces executable code from callback nets. In this work, I target the Barefoot P4 architecture [16], but my compiler is structured into **stages**, which allows it to be easily extended with back-ends for different switch architectures. I introduce an intermediate representation (IR) for data-plane programs which allows switch-level packet-processing behavior to be specified using a simple C-like syntax. The compiler performs the following transformations:

- **L3:** Callback net (callbacks that read/write global state)

  ↓

- **L2:** (per-switch) IR programs with global variables

  ↓

- **L1:** IR programs that read/write only local state

  ↓

- P4 programs that read/write only local state

This approach can be used to efficiently implement the event-driven network programs described in this thesis using real state-of-the-art SDN hardware.

1.4 Thesis Outline

This thesis has the following basic structure: Chapter 2 contains the work described in Section 1.3.1; Chapter 3 contains the work described in Section 1.3.2; Chapter 4 contains the work described in Section 1.3.3; Chapter 5 contains the work described in Section 1.3.4.

In particular, in Chapter 2 (published in [85]), I investigate using program synthesis to automatically generate network updates. The main contributions of that chapter are: (1) I present a simple operational model of SDN and formalize the network update problem precisely (§2.1), (2) I present a counterexample-guided search algorithm that solves instances of the network update problem, and prove the algorithm to be correct (§2.2), (3) I present an incremental LTL model checker for loop-free models (§2.3), and (4) I describe an OCaml implementation with backends to third-party model checkers and present experimental results on real-world networks and properties, demonstrating strong performance improvements (§2.4). This chapter supports Hypothesis 1, because it provides an approach for synthesizing network updates which are correct with respect to customizable LTL properties, and experimental results show that the approach is efficient.

Chapter 3 (published in [86]), contains the following contributions: (1) I propose a new semantic correctness condition for dynamic network programs called **event-driven consistent update** that balances the need for immediate response with the need to avoid costly synchronization and buffering of packets. My consistency property generalizes the guarantees offered by consistent updates, and is as strong as possible without sacrificing availability; (2) I propose **network event structures** to capture causal dependencies and compatibility between events, and show how to implement these using SDN functionality; (3) I describe a compiler based on a stateful extension of NetKAT, and present optimizations that reduce the overhead of implementing such stateful programs; (4) I present experimental result showing that my approach gives well-defined consistency guarantees, while avoiding expensive synchronization and packet buffering. This chapter supports Hypothesis 2, because it provides a new language which supports event-driven network programming, and I demonstrate the intuitiveness of the approach through several real-world case studies.

Additionally, I show how to provide a useful consistency property that enables reasoning about how events and data packets should interleave in an event-driven program.

Chapter 4 (published in CAV 2017 [84]) contains the following contributions: (1) I describe **event nets**, a new model for representing concurrent network programs, which extends several previous approaches (including my network event structures), enables using and reasoning about many synchronization constructs, and admits an efficient distributed implementation (Sections 4.1-4.2). (2) I present **synchronization synthesis for event nets**. To my knowledge, this is the first counterexample-guided technique that automatically adds synchronization constructs to Petri-net based programs. This solution includes a **model checker for event nets**, and an SMT-based **repair engine for event nets** which can insert a variety of synchronization constructs (Section 4.3). (3) I demonstrate the usefulness and efficiency of this approach through several real-world examples and large network topologies (Section 3.4). This chapter supports Hypothesis 3, because it provides a programmer-friendly approach for (1) writing event-driven distributed network programs and high-level correctness properties, and (2) automatically adding synchronization to fix concurrency bugs in the resulting application. I also present experimental results showing the scalability.

Chapter 5 generalizes and extends the work on event nets, providing a general and intuitive network programming language which allows operators to write correct-by-construction data-plane programs with global state, and a compiler which in turn produces efficient executable code to run on modern SDN switches. The chapter supports Hypothesis 4, because it develops general mechanisms built on top of the P4 hardware switch platform, allowing many different applications (and previously-described consistency models) to be efficiently implemented in a real SDN.

Chapter 2

Efficient Synthesis of Network Updates

2.1 Preliminaries and Network Model

To facilitate precise reasoning about networks during updates, I develop a formal model in the style of Chemical Abstract Machine [12]. This model captures key network features using a simple operational semantics. It is similar to the one used by [48], but is streamlined to model features most relevant to updates.

2.1.1 Network Model

Basic structures. Each **switch** sw, **port** pt, or **host** h is identified by a natural number. A **packet** pkt is a record of fields containing header values such as source and destination address, protocol type, and so on. I write $\{f_1; \ldots; f_k\}$ for the type of packets having fields f_i and use "dot" notation to project fields from records. The notation $\{r \text{ with } f = v\}$ denotes functional update of $r.f$.

Forwarding Tables. A switch configuration is defined in terms of forwarding rules, where each rule has a **pattern** pat specified as a record of optional packet header fields and a port, a list of **actions** act that either forward a packet out a given port ($fwd\ pt$) or modify a header field ($f{:=}n$), and a priority that disambiguates rules with overlapping patterns. I write $\{pt?; f_1?; \ldots; f_k?\}$ for the type of patterns, where the question mark denotes an option type. A set of such rules $ruls$ forms a forwarding table tbl. The semantic function $[\![tbl]\!]$ maps packet-port pairs to multisets of such pairs, finding the highest-priority rule whose pattern matches the packet and applying the corresponding actions. If there are multiple matching rules with the same priority, the function is free to pick any of them, and if there are no matching rules, it drops the packet. The forwarding tables collectively define the network's **data plane**.

Commands. The **control plane** modifies the data plane by issuing commands that update forwarding tables. The command (sw, tbl) replaces the forwarding table on switch sw with tbl (I call this a **switch-granularity** update). I model this command as an atomic operation (it can be implemented with OpenFlow **bundles** [97]). Sometimes switch granularity is too coarse to find an update sequence, in which case one can update individual rules (**rule-granularity**). My tool supports this finer-grained mode of operation, but since it is not conceptually different from switch granularity, I frame most of my discussion in terms of **switch-granularity**.

To synchronize updates involving multiple switches, I include a $wait$ command. In the model, the controller maintains a natural-number counter known as the current epoch ep. Each packet is annotated with the epoch on ingress. The control command $incr$ increments the epoch so that subsequent incoming packets are annotated with the next epoch, and $flush$ blocks the controller until all packets annotated with the previous epoch have exited the network. I introduce a command $wait$ defined as $incr; flush$. The epochs are included in my model solely to enable reasoning. They do not need to be implemented in a real network—all that is needed is a mechanism for blocking the controller to allow a flush of all packets currently in the network. For example, given a topology, one could compute a conservative delay based on the maximum hop count, and then implement $wait$ by sleeping, rather than synchronizing with each switch. Note that I implicitly assume failure-freedom and packet-forwarding fairness of switches and links, i.e. there is an upper bound on each element's packet-processing time.

Elements. The elements E of the network model include switches S_i, links L_j, and a single controller element C, and a **network** N is a tuple containing these. Each switch S_i is encoded as a record comprising a unique identifier sw, a table tbl of prioritized forwarding rules, and a multiset prs of pairs (pkt, pt) of buffered packets and the ports they should be forwarded to respectively. Each link L_j is represented by a record consisting of two **locations** l and l' and a list of queued packets $pkts$, where a location is either a host or a switch-port pair. Finally, controller C is represented by a record containing a list of commands $cmds$ and an epoch ep. I assume that commands are totally-ordered. The controller can ensure this by using OpenFlow **barrier** messages.

Operational semantics. Network behavior is defined by small-step operational rules in Figure 2.1.

Switch	sw	$\in$	$\mathbb{N}$	*Packet*	pkt	$::=$	$\{f_1; ..; f_k\}$	*Location*	l	$::=$	$h \mid (sw, pt)$
Port	pt	$\in$	$\mathbb{N}$	*Pair*	pr	$::=$	(pkt, pt)	*Command*	cmd	$::=$	$(sw, tbl) \mid incr \mid flush$
Host	h	$\in$	$\mathbb{N}$	*Pattern*	pat	$::=$	$\{pt?; f_1?; ..; f_k?\}$	*Switch*	S	$::=$	$\{sw; tbl; prs\}$
Priority	pri	$\in$	$\mathbb{N}$	*Action*	act	$::=$	$fwd\ pt \mid f{:=}n$	*Link*	L	$::=$	$\{l; pkts; l'\}$
Epoch	ep	$\in$	$\mathbb{N}$	*Rule*	rul	$::=$	$\{pri; pat; acts\}$	*Controller*	C	$::=$	$\{cmds; ep\}$
Field	f	$::=$	$src \mid dst \mid typ \mid ..$	*Table*	tbl	$::=$	$ruls$	*Element*	E	$::=$	$S \mid L \mid C$

Data Plane

$$\frac{L.l = h \quad L.l' = (sw', pt') \quad L.pkts = pkts \quad C.ep = ep}{C,\ L \to C,\ \{L \text{ with } pkts = pkt^{ep}{::}pkts\}}\ \textsc{In} \qquad \frac{L.l = (sw, pt) \quad L.l' = h \quad L.pkts = (pkt^{ep}{::}pkts)}{L \xrightarrow{(sw,pt,pkt)} \{L \text{ with } pkts = pkts\}}\ \textsc{Out}$$

$$\frac{L.loc' = (sw, pt) \quad L.pkts = (pkt^{ep}{::}pkts) \quad S.sw = sw \quad [\![S.tbl]\!](pkt, pt) = \{(pkt_1, pt_1), .., (pkt_n, pt_n)\}}{L, S \xrightarrow{(sw,pt,pkt)} \{L \text{ with } pkts = pkts\}, \{S \text{ with } prs = S.prs \uplus \{(pkt_1^{ep}, pt_1), .., (pkt_n^{ep}, pt_n)\}\}}\ \textsc{Process}$$

$$\frac{S.sw = sw \quad S.prs = \{(pkt^{ep}, pt)\} \uplus prs \quad L.l = (sw, pt)}{S,\ L \to \{S \text{ with } prs = prs\},\ \{L \text{ with } pkts = L.pkts @ [pkt^{ep}]\}}\ \textsc{Forward}$$

Control Plane and Abstract Machine

$$\frac{C.cmds = ((sw, tbl){::}cmds) \quad S.sw = sw}{C,\ S \to \{C \text{ with } cmds = cmds\},\ \{S \text{ with } tbl = tbl\}}\ \textsc{Update} \qquad \frac{C.cmds = (incr{::}cmds)}{C \to \{C \text{ with } cmds = cmds; ep = C.ep + 1\}}\ \textsc{Incr}$$

$$\frac{C.cmds = (flush{::}cmds) \quad ep(S_1, .., S_k,\ L_1, .., L_m) = C.ep}{S_1, .., S_k,\ L_1, .., L_m,\ C \to S_1, .., S_k,\ L_1, .., L_m,\ \{C \text{ with } cmds = cmds\}}\ \textsc{Flush} \qquad \frac{Es_1 \xrightarrow{o} Es_1'}{Es_1 \uplus Es_2 \xrightarrow{o} Es_1' \uplus Es_2}\ \textsc{Congruence}$$

Figure 2.1: Network model.

These define interactions between subsets of elements, based on OpenFlow semantics [87]. States of the model are given by multisets of elements. I write $\{x\}$ to denote a singleton multiset, and $m_1 \uplus m_2$ for the union of multisets m_1 and m_2. I write $[x]$ for a singleton list, and $l_1 @ l_2$ for concatenation of l_1 and l_2. Each transition $N \xrightarrow{o} N'$ is annotated, with o being either an empty annotation, or an **observation** (sw, pt, pkt) indicating the location and packet being processed.

The first rules describe date-plane behavior. The IN rule admits arbitrary packets into the network from a host, stamping them with the current controller epoch. The OUT rule removes a packet buffered on a link adjacent to a host. PROCESS processes a single packet on a switch, finding the highest priority rule with matching pattern, applying the actions of that rule to generate a multiset of packets, and adding those packets to the output buffer. FORWARD moves a packet from a switch to the adjacent link. The final rules describe control-plane behavior. UPDATE replaces the table on a single switch. INCR increments the epoch on the controller, and FLUSH blocks the controller until all packets in the network are annotated with **at least** the current epoch ($ep(Es)$ denotes the smallest annotation on any packet in Es). Finally, CONGRUENCE, allows any sub-collection of network elements to interact.

2.1.2 Network Update Problem

In order to define the network update problem, I need to first carefully define **traces** of packets flowing through the network.

Packet traces. Given a network N, my operational rules can generate sequences of observations. However, the network can process many packets concurrently, and I want observations generated by a single packet. I define a successor relation $\sqsubseteq$ for observations. Intuitively $o \overset{ep}{\sqsubseteq} o'$ if the network can directly produce the packet in o' by processing o in the epoch ep.

To formalize this, I first define what it means for a table to be active, i.e. the controller contains an update that will eventually produce that table.

Definition 1 (Active Forwarding Table)**.** Let N be a network. The forwarding table tbl is **active in the epoch** ep for the switch sw if

(1) $ep = 0$ and tbl is the initial table of sw in N, or

(2) $ep > 0$ and either (a) if there exists a command $(sw', tbl') \in C.cmds$ such that $sw = sw'$ and the number of $wait$ commands preceding (sw, tbl) in $C.cmds$ is ep, then $tbl = tbl'$, or (b) if there does not exist such a command, then tbl is the table active for the switch sw in epoch $ep - 1$.

Next I define what it means for an observation o' to succeed o.

Definition 2 (Successor Observation)**.** Let N be a network and let $o = (sw, pt, pkt)$ and $o' = (sw', pt', pkt')$ be observations. The observation o' is a **successor of** o **in** ep, written $o \overset{ep}{\sqsubseteq} o'$, if either:

- there exists a switch S_i and link L_j such that $S_i.sw = sw$ and $S_i.tbl$ is active in ep and $L_j.l = (sw, pt_j)$ and $L_j.l' = (sw', pt')$ and $(pt_j, pkt') \in [\![S_i.tbl]\!](pt, pkt)$, or
- there exists a switch S_i, a link L_j, and a host h such that $S_i.sw = sw$ and $S_i.tbl$ is active in ep and $L_j.l = (sw, pt')$ and $L_j.l' = h$ and $(pt', pkt') \in [\![S_i.tbl]\!](pt, pkt)$.

Intuitively $o \overset{ep}{\sqsubseteq} o'$ if the packet in o could have directly produced the packet in o' in ep by being processed on some switch. The two cases correspond to an internal and egress processing steps.

Now, I can use these definitions to formalize single-packet traces.

Definition 3 (Single-Packet Trace)**.** Let N be a network. A sequence $(o_1 \cdots o_l)$ is a **single-packet trace of** N if $N \xrightarrow{o'_1} \ldots \xrightarrow{o'_k} N_k$ such that $(o_1 \cdots o_l)$ is a subsequence of $(o'_1 \cdots o'_k)$ for which

- every observation is a successor of the preceding observation in monotonically increasing epochs, and
- if $o_1 = o'_j = (sw, pt, pkt)$, then $\exists o'_i \in \{o'_1, \cdots, o'_{j-1}\}$ such that the o'_i transition is an IN moving pkt from host to (sw, pt) and none of $o'_i, \cdots, o'_{j-1}$ is a predecessor of o_1, and
- the o_l transition is an OUT terminating at a host.

Intuitively, single-packet traces are end-to-end paths through the network. I write $\mathcal{T}(N)$ for the set of single-packet traces generated by N. A trace $(o_1 \cdots o_k)$ is **loop-free** if $o_i \neq o_j$ for all distinct i and j between 1 and k. I consider only loop-free traces, since a network that forwards packets around a loop is generally considered to be misconfigured. In the worst case, forwarding loops can cause a packet storm, wasting bandwidth and degrading performance. My tool automatically detects/rejects such configurations.

LTL formulas. Many important network properties can be understood by reasoning about the traces that packets can take through the network. For example, reachability requires that all packets starting at src eventually reach dst. Temporal logics are an expressive and well-studied language for specifying such trace-based properties. Hence, I use Linear Temporal Logic (LTL) to describe traces in my network model. Let AP be atomic propositions that test the value of a switch, port, or packet field: $f_i = n$. I call elements of the set 2^{AP} **traffic classes**. Intuitively, each traffic class T identifies a set of packets that agree on the values of particular header fields. An LTL formula φ in negation normal form (NNF) is either $true$, $false$, atomic proposition p in AP, negated proposition $\neg p$, disjunction $\varphi_1 \vee \varphi_2$, conjunction $\varphi_2 \wedge \varphi_2$, next $X\varphi$, until $\varphi_1 U \varphi_2$, or release $\varphi_1 R \varphi_2$, where φ_1 and φ_2 are LTL formulas in NNF. The operators F and G can be defined using other connectives. Since (finite) single-packet traces can be viewed as infinite sequences of packet observations where the final observation repeats indefinitely, the semantics of the LTL formulas can be defined in a standard way over traces. I write $t \models \varphi$ to indicate that the single-packet trace t satisfies the formula φ and $\mathcal{T} \models \varphi$ to indicate that $t \models \varphi$ for each t in $\mathcal{T}$. Given a network N and a formula φ, I write $N \models \varphi$ if $\mathcal{T}(N) \models \varphi$.

Problem Statement. Recall that my network model includes commands for updating a single switch, incrementing the epoch, and waiting until all packets in the preceding epoch have been flushed from the network. At a high-level, my goal is to identify a sequence of commands to transition the network between configurations without violating specified invariants. First, I need a bit of notation. Given a network N, I write $N[sw \leftarrow tbl]$ for the **switch update** obtained by updating the forwarding table for switch sw to tbl. I call N **static** if $C.cmds$ is empty. If static networks N_1, N_n have the same traces $\mathcal{T}(N_1) = \mathcal{T}(N_n)$, then I say they are trace-equivalent, $N_1 \simeq N_n$.

Definition 4 (Network Update)**.** Let N_1 be a static network. A command sequence $cmds$ induces a sequence $N_1, \dots, N_n$ of static networks if $c_1 \cdots c_{n-1}$ are the update commands in $cmds$, and for each $c_i = (sw, tbl)$, I have $N_i[sw \leftarrow tbl] \simeq N_{i+1}$.

I write $N_1 \xrightarrow{cmds} N_n$ if there exists such a sequence of static networks induced by $cmds$ which ends with N_n.

I call N **stable** if all packets in N are annotated with the same epoch. Intuitively, a stable network is one with no in-progress update, i.e. any preceding update command was finalized with a **wait**. Consider the set of **unconstrained** single-packet traces generated by removing the requirement that traces start at an ingress. This includes $\mathcal{T}(N)$ as well as traces of packets initially present in N. I call this $\bar{\mathcal{T}}(N)$.

Definition 5 (Unconstrained Single-Packet Trace)**.** Let N be a network. Then $(o_1 \cdots o_l)$ is a **unconstrained single-packet trace of** N if $N \xrightarrow{o'_1} \dots \xrightarrow{o'_k} N_k$ such that $(o_1 \cdots o_l)$ is a subsequence of $(o'_1 \cdots o'_k)$ for which

- every observation is a successor of the preceding one in monotonically increasing epochs, and
- if $o_1 = o'_j = (sw, pt, pkt)$, i.e. $N \xrightarrow{o'_1} \dots \xrightarrow{o'_j = o_1} N_j \xrightarrow{o'_{j+1}} \dots \xrightarrow{o'_k} N_k$, then no $o'_i \in \{o'_1, \cdots, o'_{j-1}\}$ precedes o_1, and
- the o_l transition is an OUT terminating at a host.

Unconstrained single-packet traces are not required to begin at a host. I write $\bar{\mathcal{T}}(N)$ for the set of unconstrained single-packet traces generated by N, and note that $\mathcal{T}(N) \subseteq \bar{\mathcal{T}}(N)$.

Note that for a **stable** network N, $\bar{\mathcal{T}}(N)$ is equal to $\mathcal{T}(N)$.

Lemma 1 (Traces of a Stable Network)**.** Let N be a stable network. Then for each trace $t \in \bar{\mathcal{T}}(N)$, there exists a trace $t' \in \mathcal{T}(N)$ such that t is a **suffix** of t'.

Definition 6 (Update Correctness)**.** Let N be a stable static network and let φ be an LTL formula. The command sequence $cmds$ is **correct** with respect to N and φ if $\hat{N} \models \phi$ where $\hat{N}$ is obtained from N by setting $C.cmds = cmds$.

A **network configuration** is a static network which contains no packets. I can now present the problem statement.

Definition 7 (Update Synthesis Problem)**.** Given stable static network N, network configuration N', and LTL specification φ, construct a sequence of commands $cmds$ such that (i) $N \overset{cmds}{\longrightarrow} N''$ where $N'' \simeq N'$, and (ii) $cmds$ is correct with respect to φ.

2.1.3 Efficiently Checking Network Properties

To facilitate efficient checking of network properties via LTL model checkers, I show how to model a network as a Kripke structure.

Kripke structures. A **Kripke structure** is a tuple $(Q, Q_0, \delta, \lambda)$, where Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, $\delta \subseteq Q \times Q$ is a transition relation, and $\lambda : Q \to 2^{AP}$ labels each state with a set of atomic propositions drawn from a fixed set AP. A Kripke structure is **complete** if every state has at least one successor. A state $q \in Q$ is a **sink state** if for all states q', $\delta(q, q')$ implies that $q = q'$, and I call a Kripke structure **DAG-like** if the only cycles are self-loops on sink states. In this thesis, I will consider complete and DAG-like Kripke structures. A **trace** t is an infinite sequence of states, $t_0 t_1 \ldots$ such that $\forall i \geq 0 : \delta(t_i, t_{i+1})$. Given a trace t, I write t^i for the suffix of t starting at the i-th position—i.e., $t^i = t_i t_{i+1} \ldots$. Given a set of traces $\mathcal{T}$, I let $\mathcal{T}^i$ denote the set $\{t^i \mid t \in \mathcal{T}\}$. Given a state q of a Kripke structure K, let $traces_K(q)$ be the set of traces of K starting from q and $succ_K(q)$ be the set of states defined by $q' \in succ_K(q)$ if and only if $\delta(q, q')$. I will omit the subscript K when it is clear form the context. A Kripke structure $K = (Q, Q_0, \delta, \lambda)$ satisfies an LTL formula φ if for all states $q_0 \in Q_0$ I have that $traces(q_0) \models \varphi$.

Network Kripke structures. For every static N, I can generate a Kripke structure $\mathcal{K}(N)$ containing traces which correspond according to an intuitive trace relation $\lesssim$.

Definition 8 (Network Kripke Structure)**.** Let N be a static network. I define a Kripke structure $\mathcal{K}(N) = (Q, Q_0, \delta, \lambda)$ as follows. The set of states Q comprises tuples of the form (sw, pt, T_k). The set Q_0 contains states (sw, pt, T_k) where sw and pt are adjacent to an ingress link—i.e., there exists a link L_j and host h such that $L_j.l = h$ and $L_j.l' = (sw, pt)$. Transition relation δ contains all pairs of states (sw, pt, T_k) and (sw', pt', T_k') where there exists a switch S and a link L such that $S.sw = sw$ and either:

- there exists a link L_j and packets $pkt \in T_k$ and $pkt' \in T_k'$ such that $L.l' = (sw, pt)$ and $L_j.l = (sw, pt_j)$ and $L_j.l' = (sw', pt')$ and $(pkt', pt_j) \in [\![S.tbl]\!](pkt, pt)$.
- there exists a link L_j, a host h, and packets $pkt \in T_k$ and $pkt' \in T_k'$ such that $L.l' = (sw, pt)$ and $L_j.l = (sw, pt')$ and $L_j.l' = h$ and $(pkt', pt') \in [\![S.tbl]\!](pkt, pt)$.
- $(sw, pt, T_k) = (sw', pt', T_k')$ and there exists a packet $pkt \in T_k$ such that $L.l' = (sw, pt)$ and $[\![S.tbl]\!](pkt, pt) = \{\}$.
- $(sw, pt, T_k) = (sw', pt', T_k')$ and there exists a link L_j and host h such that $L_j.l = (sw, pt)$ and $L_j.l' = h$.

Finally, the labeling function λ maps each state (sw, pt, T_k) to T_k, which captures the set of all possible header values of packets located at switch sw and port pt.

The four cases of the δ relation correspond to forwarding packets to an internal link, forwarding packets out an egress, dropping packets on a switch, or reaching an egress (inducing a self-loop).

I now relate observations generated by a network and traces of the Kripke structure generated from it.

Definition 9 (Trace Relation)**.** Let N be a static network and K a Kripke structure. Define $\lesssim$ on observations of N and states of K as $(sw, pt, pkt) \lesssim (sw, pt, T_k)$ if and only if $pkt \in T_k$. Lift $\lesssim$ to a relation on (finite) sequences of observations and (infinite) traces by repeating the final observation and requiring $\lesssim$ to hold pointwise: $o_1 \cdots o_k \lesssim t$ if and only if $o_i \lesssim t_i$ for i from 1 to k and $o_k \lesssim t_j$ for all $j > k$.

I currently do not reason about packet modification, so the Kripke structure has disjoint parts corresponding to the traffic classes. It is straightforward to enable packet modification, by adding transitions between the parts of the Kripke structure, but I leave this for future work. I now show that the generated Kripke structure faithfully encodes the network semantics.

Lemma 2 (Network Kripke Structure Soundness)**.** Let N be a static network and $K = \mathcal{K}(N)$ a network Kripke structure. For every single-packet trace t in $\mathcal{T}(N)$ there exists a trace t' of K from a start state such that $t \lesssim t'$, and vice versa.

Proof. I proceed by induction over k, the length of the (finite prefix of the) trace. The base case $k = 1$ is easy to see, since the lone observation in t must be on an ingress link, meaning the corresponding state in K will be an initial state with a self-loop (case 3 of Definition 8), and these are equivalent via Definition 9.

For the inductive step ($k > 1$), I wish to show both directions of subtrace relation $\lesssim$ to conclude equivalence. First, let $t = o_1, \cdots, o_{k+1}$ be a single-packet trace of length $k+1$ in $\mathcal{T}(N)$, and I must show that $\exists t' \in \mathcal{K}(N)$ such that $t \lesssim t'$. Let t^k be the prefix of t having length k. By my induction hypothesis, there exists $t'^k = s_1, \cdots, s_{k-1}, s_k, s_k, \cdots \in \mathcal{K}(N)$ such that $t^k \lesssim t'^k$. I have the successor relation $o_k \sqsubseteq o_{k+1}$, so Definition 2 and 8 tells me that I have a transition $s_k \to s'$ for some $s' \in K$. I see that this s' is exactly what I need to construct $t' = s_1, \cdots, s_k, s', s', \cdots$ which satisfies the relation $t \lesssim t'$.

Now, let $t' = s_1, \cdots, s_k, s_{k+1}, s_{k+1}, \cdots$ be a trace in $\mathcal{K}(N)$ for which the finite prefix has length $k+1$. I must show that $\exists t \in \mathcal{T}(N)$ such that $t \lesssim t'$. Let $t'^k = s_1, \cdots, s_{k-1}, s_k, s_k, \cdots$, and by my induction hypothesis, and there exists $t^k = o_1, \cdots, o_k$ such that $t^k \lesssim t'^k$. Consider transition $s_k \to s_{k+1}$. If $s_k = s_{k+1}$, then $t' = t'^k$, so I can let $t = t^k$, and conclude that $t \lesssim t'$. Otherwise, if $s_k \neq s_{k+1}$, then I have one of the first two cases in Definition 8, which correspond to the cases in Definition 2, allowing me to construct an o_{k+1} such that $o_k \sqsubseteq o_{k+1}$. I let $t = o_1, \cdots, o_k, o_{k+1}$, and conclude that $t \lesssim t'$. $\square$

This means that checking LTL over single-packet traces can be performed via LTL model-checking of Kripke structures.

Checking network configurations. One key challenge arises because the network is a distributed system. Packets can "see" an inconsistent configuration (some switches updated, some not), and reasoning about possible interleavings of commands becomes intractable in this context. I can simplify the problem by ensuring that each packet traverses at most one switch that was updated after the packet entered the network.

Definition 10 (Careful Command Sequences)**.** A sequence of commands $(cmd_1 \cdots cmd_n)$ is **careful** if every pair of switch updates is separated by a $wait$ command.

In the rest of this chapter, I consider careful command sequences, and develop a sound and complete algorithm that finds them efficiently. Section 2.2 describes a technique for removing wait commands that works well in practice, but I leave **optimal** wait removal for future work. Recall that $\mathcal{T}(N)$ denotes the sequence of all traces that a packet could take through the network, regardless of when the commands in $N.cmds$ are executed. This is a superset of the traces induced by each static N_i in a solution to the network update problem. However, if $cmds$ is careful, then each packet only encounters a single configuration, allowing the correctness of the sequence to be reduced to the correctness of each N_i. To show this, I begin by present the following auxiliary lemma.

Lemma 3 (Traces of a Careful Network)**.** Let N be a stable network with $C.cmds$ careful, and consider a sequence of static networks induced by $C.cmds$. For every trace $t \in \mathcal{T}(N)$ there exists a stable static network N_i in the sequence s.t. $t \in \mathcal{T}(N_i)$.

Proof. I. First, I show that at most one update transition can be involved in the trace. In other words, if $N \xrightarrow{o'_1} \dots \xrightarrow{o'_k} N_k$ where $t = o_1 \cdots o_n$ is a subsequence of $o'_1 \cdots o'_k$, and if $f : \mathbb{N} \to \mathbb{N}$ is a bijection between o_i indices and o'_i indices, then at most one of the transitions $o'_{f(1)}, \cdots, o'_{f(n)}$ is an UPDATE transition.

Assume to the contrary that there are more than one such transitions, and consider two of them, o'_i, o'_j where $i, j \in \{f(1), \cdots, f(n)\}$, assuming without loss of generality that $i < j$. Now, since the sequence $C.cmds$ is careful, I must have both an INCR and FLUSH transition between o'_i and o'_j. This means that the second update o'_j cannot happen while the trace's packet is still in the network, i.e. $j > f(n)$, and I have reached a contradiction.

II. Now, if there are zero update transitions, I am done, since the trace is contained in the first static N. If there is one update transition $N_{k+1} = N_k[sw \leftarrow tbl]$, and this update occurs before the packet reaches sw in the trace, then the trace is fully contained in N_{k+1}. Otherwise, the trace is fully contained in N_k. □

I can now use this lemma to prove my claim about the correctness of careful command sequences.

Lemma 4 (Careful Correctness)**.** Let N be a stable network with $C.cmds$ careful and let φ be an LTL formula. If $cmds$ is careful and $N_i \models \phi$ for each static network in any sequence induced by $cmds$, then $cmds$ is correct with respect to φ.

Proof. Consider a trace $t \in \mathcal{T}(N)$. From Lemma 3, I have $t \in \mathcal{T}(N_i)$ for some N_i in the induced sequence. Thus $t \models \varphi$, since my hypothesis tells me that $N_i \models \varphi$. Since this is true for an arbitrary trace, I have shown that $\mathcal{T}(N) \models \varphi$, i.e. $N \models \varphi$, meaning that $cmds$ is correct with respect to φ. □

In the following lemmas, I show that checking the **unique sequence of network configurations** induced by $cmds$ is equivalent to the above.

Lemma 5 (Trace-Equivalence)**.** Let N_1, N_n be static networks where $N_1 \to \cdots \to N_n$ and no transition is an update command. For a single-packet trace t, I have $t \in \mathcal{T}(N_1) \iff t \in \mathcal{T}(N_n)$.

Lemma 6 (Induced Sequence of Networks)**.** Let N_1 be a static network, and let N_1' be the network obtained by emptying all packets from N_1. Let $cmds$ be a sequence of commands, and let $c_1 \cdots c_{n-1}$ be the subsequence of update commands. Construct sequence $N_1' \to \cdots \to N_n'$ of empty networks by executing the update commands in order. Now, given any $N_1 \to \cdots \to N_n$ induced by $cmds$, I have $N_i \simeq N_i'$ for all i.

In other words, any induced sequence of static networks is pointwise trace-equivalent to the unique sequence of **network configurations** generated by running the update commands in order.

Next I will develop a sound and complete algorithm that solves the update synthesis problem for careful sequences by checking configurations.

2.2 Update Synthesis Algorithm

This section presents a synthesis algorithm that searches through the space of possible solutions, using counterexamples to detect wrong configurations and exploiting several optimizations.

2.2.1 Algorithm Description

ORDERUPDATE (Algorithm 2.1.1) returns a **simple** sequence of updates (one in which each switch appears at most once), or fails if no such sequence exists. Note that I could broaden my **simple** definition, e.g. **k-simple**, where each switch appears at most k times, but I have found the above restriction to work well in practice. The core procedure is DFSFORORDER, which manages the search and invokes the model checker (I use DFS because I expect common properties/configurations to admit many update sequences). It

Algorithm 2.1.1: ORDERUPDATE Algorithm.

```
Procedure ORDERUPDATE($N_i, N_f, \varphi$)
    Input: Initial static network $N_i$, final static configuration $N_f$, formula $\varphi$.
    Output: update sequence $L$, or error $\epsilon$ if no update sequence exists.
    $W \leftarrow false$                    // Formula encoding wrong configurations.
    $V \leftarrow false$                    // Formula encoding visited configurations.
    $(ok, L) \leftarrow$ DFSFORORDER($N_i, \mathcal{K}(N_i), \bot, \varphi, \lambda_0$)
    if $ok$ then return $L$
    else return $\epsilon$                  // Failure---no update exists.
Procedure DFSFORORDER($N$,$K$,$s$,$\varphi$,$\lambda$)
    Input: Static network $N$ and Kripke structure $K$, next switch to update $s$, formula $\varphi$, and
           labeling $\lambda$.
    Output: Boolean ok if a correct update exists; correct update sequence $L$.
    if $N \models V \vee W$ then return $(false, [\,])$
    if $s = \bot$ then $(ok, cex, \lambda) \leftarrow modelCheck(K, \varphi)$
    else
        $(N, K, S) \leftarrow swUpdate(N, s)$
        $(ok, cex, \lambda) \leftarrow incrModelCheck(K, \varphi, S, \lambda)$
    $V \leftarrow V \vee makeFormula(N)$
    if $\neg ok$ then
        $W \leftarrow W \vee makeFormula(cex)$
        return $(false, [\,])$
    if $N = N_f$ then return $(true, [s])$
    for $s' \in possibleUpdates(N)$ do
        $(ok, L) \leftarrow$ DFSFORORDER($N, K, s', \varphi, \lambda$)
        if $ok$ then return $(true, (upd\ s') :: wait :: L)$
    return $(false, [\,])$
```

attempts to add a switch s to the current update sequence, yielding a new network configuration. I maintain two formulas, V and W, tracking the set of configurations that have been visited so far, and the set of configurations excluded by counterexamples.

To check whether all packet traces in this configuration satisfy the LTL property φ, I use my (incremental) model checking algorithm (discussed in Section 2.3). First, I call a full check of the model (line 7). The model checker labels the Kripke structure nodes with information about what formulas hold for paths starting at that state. The labeling (stored in λ) is then re-used in the subsequent model checking calls for related Kripke structures (line 10). The parameters passed in the incremental model checking call are: updated Kripke structure K, specification φ, set of nodes S in K whose transition function has changed by the update of the switch s, and correct labeling λ of the Kripke structure before the update. Note that before the initial model checking, I convert the network configuration N to a Kripke structure K. The update of K is performed by a function $swUpdate$ that returns a triple (N', S, K'), where N' is the new static network, K' is the updated Kripke structure obtained as $\mathcal{K}(N')$, and S is the set of nodes that have different outgoing transitions in K'.

If the model checker returns **true**, then N is safe and the search proceeds recursively, after adding $(upd\ s')$ to the current sequence of commands. If the model checker returns **false**, the search backtracks, using the counterexample-learning approach below.

2.2.2 Optimizations

I now present optimizations improving synthesis (**pruning with counterexamples**, **early search termination**), and improving efficiency of synthesized updates (**wait removal**).

Counterexamples. Counterexample-based pruning learns network configurations that do not satisfy the specification to avoid making future model checking calls that are certain to fail. The function $makeFormula(cex)$ (Line 13) returns a formula representing the set of switches that occurred in the counterexample trace cex, with flags indicating whether each switch was updated. This allows equivalent future configurations to be eliminated without invoking the model checker. Recall the red-green example in Section 1.3.1 and suppose that I update A1 and then C2. At the intermediate configuration obtained by updating just A1, packets will be dropped at C2, and the specification will not be satisfied. The formula for the unsafe set of configurations that have A1 updated and C2 not updated will be added to W. In practice, many counterexamples are small compared to network size, and this greatly prunes the search space.

Early search termination. The early search termination optimization speeds up termination of the search when no (switch-granularity) update sequence is possible. Recall how I use counterexamples to prune **configurations**. With similar reasoning, I can use counterexamples for pruning possible **sequences of updates**. Consider a counterexample trace which involves three nodes A, B, C, with A updated, B updated, and C not updated. This can be seen as requiring that C must be updated before A, or C must be updated before B. Early search termination involves collecting such constraints on possible updates, and terminating if these constraints taken together form a contradiction. In my tool, this is done efficiently using an (incremental) SAT solver. If the solver determines that no update sequence is possible, the search terminates. For simplicity, early search termination is not shown in Algorithm 2.1.1.

Wait removal. This heuristic eliminates waits that are unnecessary for correctness. Consider an update sequence $L = cmd_0 cmd_1 \cdots cmd_n$, and consider some switch update $cmd_k = (upd\ s)$. In the configuration resulting from executing the sequence $cmd_0 cmd_1 \cdots cmd_{k-1}$, if the switch s cannot possibly receive a packet which passed through some switch s_0 before an update $cmd_j{=}(upd\ s_0)$ where $j < k$, then I can update s without waiting. Thus, I can remove some unnecessary waits if I can maintain reachability-between-switches information during the update. Wait removal is not shown in Algorithm 2.1.1, but in my tool, it operates as a post-processing pass once an update sequence is found. In practice, this removes a majority of unnecessary waits (see § 2.4).

2.2.3 Formal Properties

The following two theorems show that my algorithm is sound for careful updates, and complete if I limit my search to **simple** update sequences.

Theorem 1 (Soundness)**.** Given initial network N_i, final configuration N_f, and LTL formula φ, if ORDERUPDATE returns a command sequence $cmds$, then $N_i \xrightarrow{cmds} N'$ s.t. $N' \simeq N_f$, and $cmds$ is correct with respect to φ and N_i.

Proof. It is easy to show that if ORDERUPDATE returns $cmds$, then $N_i \xrightarrow{cmds} N'$ where $N' \simeq N_f$. Each update in the returned sequence changes a switch configuration of one switch sw to the configuration $N_f(sw)$, and termination occurs when all (and only) switches sw such that $N_i(sw) \neq N_f(sw)$ have been updated.

Observe that if ORDERUPDATE returns $cmds$, the sequence can be made careful by choosing an adequate delay between each update command, and for all $j \in \{0, \cdots, n\}$, $N_j \models \varphi$. This is ensured by the model checker call (Line 7). I use Lemma 4 to conclude that $cmds$ is correct with respect to φ and N_i. $\square$

To show that ORDERUPDATE is complete with respect to simple and careful command sequences, I observe that ORDERUPDATE searches through all simple and careful sequences.

Theorem 2 (Completeness)**.** Given initial network N_i, final configuration N_f, and specification φ, if there exists a simple, careful sequence $cmds$ with $N_i \xrightarrow{cmds} N'$ s.t. $N' \simeq N_f$, then ORDERUPDATE returns one such sequence.

2.3 Incremental Model Checking

I now present an incremental algorithm for model checking Kripke structures. This algorithm is central to my synthesis tool, which invokes the model checker on many closely related structures. The algorithm makes use of the fact that the only cycles in the Kripke structure are self-loops on sink nodes—something that is true of structures encoding loop-free network configurations—and re-labels the states of a previously-labeled Kripke structure with the (possibly different) formulas that hold after an update.

2.3.1 State Labeling

I begin with an algorithm for labeling states of a Kripke structure with sets of formulas, following the approach of [124] (WVS) and [120]. The WVS algorithm translates an LTL formula φ into a local automaton and an eventuality automaton. The local automaton checks consistency between a state and its predecessor, and handles labeling of all formulas except $\varphi_1\ U\ \varphi_2$, which is checked by the eventuality automaton. The two automata are composed into a single Büchi automaton whose states correspond to subsets of the set of subformulas of φ and their negations. Hence, I label each Kripke state by a set L of sets of formulas such that if a state q is labeled by L, then for each set of formulas S in L, there exists a trace t starting from q satisfying all the formulas in S.

I now describe state labeling precisely. Let φ be an LTL formula in NNF. The **extended closure** of φ, written $ecl(\varphi)$, is the set of all subformulas of φ and their negations:

- $true \in ecl(\varphi)$
- $\varphi \in ecl(\varphi)$
- If $\psi \in ecl(\varphi)$, then $\neg\psi \in ecl(\varphi)$

  (I identify ψ with $\neg\neg\psi$, for all ψ).
- If $\varphi_1 \vee \varphi_2 \in ecl(\varphi)$, then $\varphi_1 \in ecl(\varphi)$ and $\varphi_2 \in ecl(\varphi)$.

- If $\varphi_1 \wedge \varphi_2 \in ecl(\varphi)$, then $\varphi_1 \in ecl(\varphi)$ and $\varphi_2 \in ecl(\varphi)$.
- If $X\,\varphi_1 \in ecl(\varphi)$, then $\varphi_1 \in ecl(\varphi)$.
- If $\varphi_1\, U\, \varphi_2 \in ecl(\varphi)$, then $\varphi_1 \in ecl(\varphi)$ and $\varphi_2 \in ecl(\varphi)$
- If $\varphi_1\, R\, \varphi_2 \in ecl(\varphi)$, then $\varphi_1 \in ecl(\varphi)$ and $\varphi_2 \in ecl(\varphi)$.

A subset $M \subset ecl(\varphi)$ of the extended closure is said to be **maximally consistent** if it contains $true$ and is simultaneously closed and consistent under boolean operations:

- $true \in M$
- $\psi \in M$ iff $\neg\psi \notin M$ (I identify ψ with $\neg\neg\psi$, for all ψ)
- $\varphi_1 \vee \varphi_2 \in M$ iff ($\varphi_1 \in M$ or $\varphi_2 \in M$)
- $\varphi_1 \wedge \varphi_2 \in M$ iff ($\varphi_1 \in M$ and $\varphi_2 \in M$)

Likewise, the relation $follows(M_1, M_2)$ captures the notion of successor induced by temporal operators, lifted to maximally-consistent sets. I say $follows(M_1, M_2)$ holds if and only if all of the following hold:

- $X\,\varphi_1 \in M_1$ iff $\varphi_1 \in M_2$
- $\varphi_1\, U\, \varphi_2 \in M_1$ iff $\big(\varphi_2 \in M_1 \vee (\varphi_1 \in M_1 \wedge \varphi_1\, U\, \varphi_2 \in M_2)\big)$
- $\varphi_1\, R\, \varphi_2 \in M_1$ iff $\big(\varphi_1 \in M_1 \vee (\varphi_2 \in M_1 \wedge \varphi_1\, R\, \varphi_2 \in M_2)\big)$

Given a trace t and a maximally-consistent set M, I write $t \models M$ if and only if for all $\psi \in M$, I have $t \models \psi$.

For the rest of this section, I fix a Kripke structure $K = (Q, Q_0, \delta, \lambda)$, a state q in Q, an LTL formula φ in NNF, and a maximally-consistent set $M \subset ecl(\varphi)$.

To compute the label of a state q, there are two cases depending on whether it is a sink state or a non-sink state. If q is a sink state, the function $HoldsSink(q, M)$ computes a predicate that is true if and only if, for all $\psi \in M$ and the unique trace t starting from q, I have $t \models \psi$. More formally, $HoldsSink(q, M)$ is defined to be $(\forall \psi \in M : Holds_0(q, \psi))$, where $Holds_0$ is defined as in Figure 2.2. The function $Holds_0$ computes a predicate that is true if and only if ψ holds at q. For example, $Holds_0(q, \phi_1 \text{ U } \phi_2)$ is defined as $Holds_0(q, \phi_2)$ because the only transition from q is a self-loop.

For the second case, suppose q is a non-sink state. If I am given a labeling for $succ_K(q)$ (the successors of the node q), I can extend it to a labeling for q. Let $V \subseteq Q$ be a set of vertices. A function $labGr_K$ is a **correct labeling of K with respect to φ and V** if for every $v \in V$, it returns a set

$$\begin{array}{rcl}
\mathit{Holds}_0(q, p) & = & q \models p \\
\mathit{Holds}_0(q, \neg p) & = & q \not\models p \\
\mathit{Holds}_0(q, \phi_1 \wedge \phi_2) & = & \mathit{Holds}_0(q, \phi_1) \wedge \mathit{Holds}_0(q, \phi_2) \\
\mathit{Holds}_0(q, \phi_1 \vee \phi_2) & = & \mathit{Holds}_0(q, \phi_1) \vee \mathit{Holds}_0(q, \phi_2) \\
\mathit{Holds}_0(q, \mathrm{X}\phi) & = & \mathit{Holds}_0(q, \phi) \\
\mathit{Holds}_0(q, \phi_1 \,\mathrm{U}\, \phi_2) & = & \mathit{Holds}_0(q, \phi_2) \\
\mathit{Holds}_0(q, \phi_1 \,\mathrm{R}\, \phi_2) & = & \mathit{Holds}_0(q, \phi_1) \vee \mathit{Holds}_0(q, \phi_2)
\end{array}$$

Figure 2.2: The Holds_0 function

L of maximally consistent sets such that (a) $M \in L$ if and only if $M \subset \mathit{ecl}(\varphi)$, and (b) there exists a trace t in $\mathit{traces}(v)$ such that $t \models M$. Suppose that labGr_K is a correct labeling of K with respect to φ and $\mathit{succ}_K(q)$. The function $\mathit{Holds}_K(q, M, \mathit{labGr}_K)$ computes a predicate that is true if and only if there exists a trace t in $\mathit{traces}_K(q)$ with $t \models M$. Formally, $\mathit{Holds}_K(q, M, \mathit{labGr}_K)$ is defined as $(\lambda(q) = (AP \cap M)) \wedge \exists q' \in \mathit{succ}_K(q), M' \in \mathit{labGr}_K(q') : \mathit{follows}(M, M')$.

The following captures the correctness of labeling:

Lemma 7. First, $\mathit{HoldsSink}(q, M) \Leftrightarrow \exists t {\in} \mathit{traces}(q) : t \models M$ for sink states q. Second, if labGr_K is a correct labeling with respect to φ and $\mathit{succ}_K(q)$, then $\mathit{Holds}_K(q, M, \mathit{labGr}_K) \Leftrightarrow \exists t {\in} \mathit{traces}_K(q) : t \models M$.

Proof. First, for sink states, observe that there is a unique trace t in $\mathit{traces}(q)$, as q is a sink state. I first prove that $t \models \varphi$ iff $\mathit{Holds}_0(q, \varphi)$. I prove this by induction on the structure of the LTL formula. Then I observe that there is a unique maximally-consistent set M such that $t \models M$. This is the set $\{\psi \mid t \models \psi \wedge \psi \in \mathit{ecl}(\varphi)\}$. I then use the definition of $\mathit{HoldsSink}(q, M)$ for sink states to conclude the proof.

Now consider non-sink states: I first prove soundness, i.e., if $\mathit{Holds}_K(q, M, \mathit{labGr}_K)$, then there exists $t \in \mathit{traces}(q)$ such that $t \models M$. I have $\mathit{Holds}_K(q, M, \mathit{labGr}_K)$ iff $(\lambda(q) = (AP \cap M))$ and there exists $q' \in \mathit{succ}_K(M)$, and $M' \in \mathit{labGr}_K(q')$ such that $\mathit{follows}(M, M')$. By assumption of the theorem, I have that if $M' \in \mathit{labGr}_K(q')$, then there exists a trace t' in $\mathit{traces}(q')$ such that $t' \models M'$. Consider a trace t such that $t_0 = q$ and $t^1 = t'$. For each $\psi \in M$, I can prove that $t \models \psi$ as follows. The base case of the proof by induction is implied by the fact that $q \models (AP \cap M)$. The inductive cases are proven using the definitions of maximally-consistent set and the function $\mathit{follows}$. I now prove completeness, i.e., that if there exists a trace t in $\mathit{traces}_K(q)$ such that $t \models M$, then $\mathit{Holds}_K(q, M, \mathit{labGr}_K)$ is true. Let t be the trace $qq_1q_2\ldots$. It is easy to see that if M is a maximally-consistent set, and $t \models M$, then $M = \{\psi \mid \psi \in \mathit{ecl}(\varphi) \wedge t \models \psi\}$.

Consider the set of formulas $S = \{\psi \mid \psi \in \mathit{ecl}(\varphi) \wedge t^1 \models \psi\}$. Observe that S is a maximally-consistent set. By assumption of the theorem, I have that S is in $\mathit{labGr}_K(q_1)$. It is easy to verify that $\mathit{follows}(M, S)$. $\square$

Finally, I define $\mathit{labelNode}_K(\varphi, q, \mathit{labGr}_K)$, which computes a label L for q such that $M \in L$ if and only if there exists a trace $t \in \mathit{traces}_K(q)$ such that $t \models M$ for all $M \subset \mathit{ecl}(\varphi)$. I assume that labGr_K is a correct labeling of K with respect to φ and $\mathit{succ}(q)$. For sink states, $\mathit{labelNode}_K(\varphi, q, \mathit{labGr}_K)$ returns $\{M \mid M \in \mathit{ecl}(\varphi) \wedge \mathit{HoldsSink}(q, M)\}$, while for non-sink states it returns $\{M \mid M \in \mathit{ecl}(\varphi) \wedge \mathit{Holds}_K(q, M, \mathit{labGr}_K)\}$.

2.3.2 Incremental Algorithm

To incrementally model check a modified Kripke structure, I must re-label its states with the formulas that hold after the update.

Consider two Kripke structures $K = (Q, Q_0, \delta, \lambda)$ and $K' = (Q', Q'_0, \delta', \lambda')$, such that $Q_0 = Q'_0$. Furthermore, assume that $Q = Q'$, and there is a set $U \subseteq Q$ such that δ and δ' differ only on nodes in U. I call such a triple (K, K', U) an **update** of K.

An update (K, K', U) might add or remove edges connected to a (small) set of nodes, corresponding to a change in the rules on a switch. Suppose that labGr_K is a correct labeling of K with respect to φ and Q. The incremental model checking problem is defined as follows: I am given an update (K, K', U), and labGr_K, and I want to know whether K' satisfies φ. The naïve approach is to model check K' without using the labeling labGr_K. I call this the **monolithic** approach. In contrast, the **incremental** approach uses labGr_K (and thus intuitively re-uses the results of model checking K to efficiently verify K').

Example. Consider the left side of Figure 2.3, with H the only initial state. Suppose that the update modifies J, and the δ' relation applied to J only contains the pair (J, N), and consider labeling the structure with formulas $F\ a$, $F\ b$, and $F\ a \vee F\ b$. To simplify the example, I label a node by all those formulas which hold for at least one path starting from the node (note that in the algorithm, a node is labeled by a set of sets of formulas, rather than a set of formulas). I will have that all the nodes are labeled by $F\ a \vee F\ b$, and in addition the nodes K, I, H, M, J contain label $F\ a$, and the nodes L, I, H, N contain $F\ b$. Now I want to relabel the structure after the update (right-hand side). Given that the update changes only node J, the

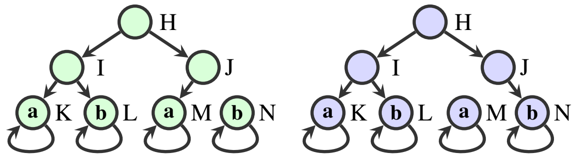


Figure 2.3: Incremental labeling—Initial (left), Final (right)

labeling can only change for J and its ancestors. I therefore start labeling node J, and find that it will now be labeled with $F\ b$ instead of $F\ a$. Labeling proceeds to H, whose label does not change (still labeled by all of $F\ a$, $F\ b$, $F\ a \vee F\ b$). The labeling process could then stop, even if H has ancestors.

Re-labeling states. Let $\mathit{ancestors}_K(V)$ be the ancestors of V in K—i.e., a set of vertices s.t. $\mathit{ancestors}_K(V) \subseteq Q$ and $q \in \mathit{ancestors}_K(V)$, if some node $v \in V$ is reachable from q. To define incremental model checking for φ, I need a function accepting a property φ, set of vertices V, labeling labGr_K that is correct for K with respect to φ and $Q \setminus \mathit{ancestors}_K(V)$, and returns a correct labeling of K with respect to φ and Q. This function is:

$$\mathit{relbl}_K(\varphi, \mathit{labGr}_K, V) = \begin{cases} \mathit{labGr}_K & \text{if } V = \emptyset \\ \mathit{relbl}_K(\varphi, \mathit{labGr}'_K, V') & \text{otherwise} \end{cases}$$

where $\mathit{labGr}'_K(v)$ is $\mathit{labelNode}_K(\varphi, v, \mathit{labGr}_K)$ if $v \in V$, and it is $\mathit{labGr}_K(v)$ if $v \notin V$. The set V' is $\{q \mid \exists v \in V : v \in \mathit{succ}_K(q)\}$.

Theorem 3. Let $V \subseteq Q$ be a set of vertices and labGr_K a correct labeling with respect to φ and $Q \setminus \mathit{ancestors}_K(V)$. Then $\mathit{relbl}_K(\varphi, \mathit{labGr}_K, V)$ is a correct labeling w.r.t. φ and Q.

Proof. I first note that only ancestors of nodes in V are re-labeled—all the other nodes are correctly labeled by assumption on labGr. I say that a node q is at level k w.r.t. a set of vertices T iff the longest simple path from q to a node in T is k. Let H_k be the set of nodes at level k from V. I prove by induction on k that at k-th iteration, I have a correct labeling of K w.r.t. φ and $(S \setminus \mathit{ancestors}_K(V)) \cup H_k$, where S is the set of states of K. I can prove the inductive claim using Lemma 7. □

Given a labeling that is correct with respect to φ and Q, it is easy to check whether φ is true for all the traces starting in the initial states: the predicate $\mathit{checkInitStates}_K(\mathit{labGr}_K, \varphi)$ is defined as $\forall q_0 \in Q_0, M \in$

$labGr_K(q_0) : \varphi \in M$. Next, let Q_f be the set of all sink states of K. Then $ancestors_K(Q_f)$ is the set Q of all states K. Therefore, for any initial labeling $labGr^0_K$, $relbl(\varphi, labGr^0_K, Q_f)$ is a correct labeling with respect to φ and Q. The function $modelCheck_K(\varphi)$ is defined to be equal to $checkInitStates_K(relbl_K(\varphi, labGr^0_K, Q_f), \varphi)$, where I can set $labGr^0_K$ to be the empty labeling $\lambda v.\emptyset$.

I now define my incremental model checking function. Let (K, K', U) be an update, and $labGr_K$ a previously-computed correct labeling of K with respect to φ and Q, where Q is the set of states of K. The function $incrModelCheck(K, \varphi, U, labGr_K)$ is defined as $checkInitStates_{K'}(relbl_{K'}(\varphi, labGr_K, U), \varphi)$. The following shows the correctness of my model checking functions.

Corollary 1. First, $modelCheck_K(\varphi) = true \iff K \models \varphi$. Second, for (K, K', U) and $labGr_K$ as above, I have $incrModelCheck(K, \varphi, U, labGr_K) = true \iff K \models \varphi$.

Proof. Using Theorem 3, and the fact that the set $ancestors_K(S_f)$ is the set S of all states K, I obtain that $labGr_K = relbl_K(\varphi, labGr^0_K, S_f)$ is a correct labeling of K with respect to φ and S. In particular, for all initial states q_0, I have that for all $M \subset ecl(\varphi)$, $m \in labGr_K(q_0)$ iff there exists a trace $t \in traces_K(q_0)$ such that $t \models M$. I now use the definition of $checkInitStates$ to show that if $checkInitStates$ returns true, then there is no initial state q_0 such that there exists $M \in labGr_K(q_0)$ such that $\neg\varphi \in M$. Thus for all initial states q_0, for all traces t in $traces(t_0)$, I have that $t \models \varphi$.

The proof for incremental model checking is similar. $\square$

The runtime complexity of the $modelCheck_K$ function is $O(|K| \times 2^{|\varphi|})$. The runtime complexity of the $incrModelCheck$ function is $O(|ancestors_K(U)| \times 2^{|\varphi|})$, where U is the set of nodes being updated.

Counterexamples. This incremental algorithm can generate counterexamples in cases where the formula does not hold. A formula $\neg\varphi$ does not hold if an initial state is labeled by L, such that there exists a set $M \in L$, such that $\neg\varphi \in M$. Examining the definition of $labelNode_K$, I find that in order to add a set M to the label L of a node q, there is a set M' in the label of one a child q' of q that explains why M is in L. The first node of the counterexample trace starting from q is one such child q'.

2.4 Implementation and Experiments

I have built a prototype tool that implements the algorithms described in this thesis. It consists of 7K lines of OCaml code. The system works by building a Kripke structure (§2.1) and then repeatedly interacting with a model checker to synthesize an update. I currently provide four checker backends: **Incremental** uses incremental relabeling to check and recheck formulas, **Batch** re-labels the entire graph on each call, **NuSMV** queries a state-of-the-art symbolic model checker in batch mode, and **NetPlumber** queries an incremental network model checker [64]. All tools except NetPlumber provide counterexample traces, so my system learns from counterexamples whenever possible (§2.2).

Experiments. To evaluate performance, I generated configurations for a variety of real-world topologies and ran experiments in which I measured the amount of time needed to synthesize an update (or discover that no order update exists). These experiments were designed to answer two key questions: (1) how the performance of my Incremental checker compares to state-of-the-art tools (NuSMV and Net-Plumber), and (2) whether my synthesizer scales to large topologies. I used the **Topology Zoo** [69] dataset, which consists of 261 actual wide-area topologies, as well as synthetically constructed **Small-World** [93] and **FatTree** [40] topologies. I ran the experiments on a 64-bit Ubuntu machine with 20GB RAM and a quad-core Intel i5-4570 CPU (3.2 GHz) and imposed a 10-minute timeout for each run. I ignored runs in which the solver died due to an out-of-memory error or timeout—these are infrequent (less than 8% of the 996 runs for Figure 2.4), and my Incremental solver only died in instances where other solvers did too.

Configurations and properties. A recent paper [75] surveyed data-center operators to discover common update scenarios, which mostly involve taking switches on/off-line and migrating traffic between switches/hosts. I designed experiments around a similar scenario. To create configurations, I connected random pairs of nodes (s, d) via disjoint initial/final paths W_i, W_f, forming a "diamond", and asserted one of the following properties for each pair:

- **Reachability:** traffic from a given source must reach a certain destination: $(\text{port}{=}s) \Rightarrow \text{F}\,(\text{port}{=}d)$
- **Waypointing:** traffic must traverse a waypoint w:

  $(\text{port}{=}s) \Rightarrow \big((\text{port}{\neq}d)\,\text{U}\,((\text{port}{=}w) \wedge \text{F}\,(port{=}d))\big)$

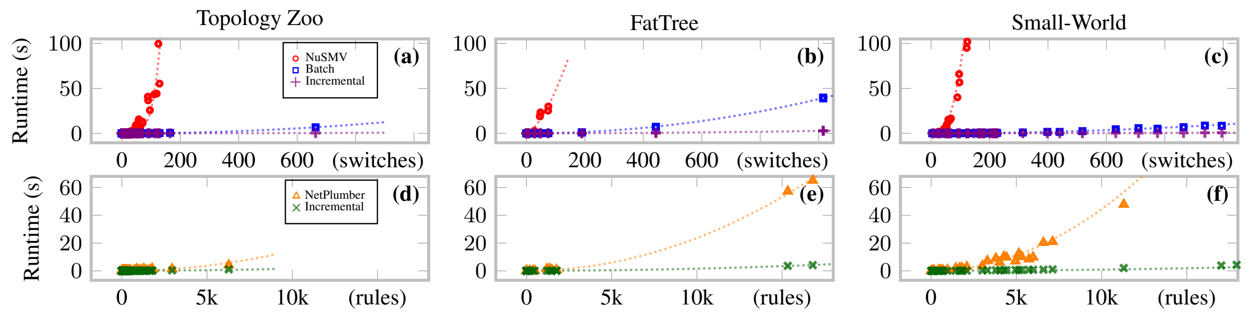


Figure 2.4: Relative performance results: **(a-c)** Performance of Incremental vs. NuSMV, Batch, NetPlumber solvers on Topology Zoo, FatTree, Small-World topologies (columns); **(d-f)** Performance of Incremental vs. NetPlumber (rule-granularity).

- **Service chaining:** traffic must waypoint through several intermediate nodes: $(\text{port} = s) \Rightarrow way(W, d)$, where

$$\begin{aligned} way([\,], d) &\equiv \mathrm{F}\,(\text{port} = d) \\ way(w_i :: W, d) &\equiv \big((\textstyle\bigwedge_{w_k \in W} \text{port}{\neq}w_k \wedge \text{port}{\neq}d) \\ &\qquad \mathrm{U}\,((\text{port} = w_i) \wedge way(W, d))\big). \end{aligned}$$

Incremental vs. NuSMV/Batch. Figure 2.4 (a-c) compares the performance of Incremental and NuSMV backends for the reachability property. Of the 247 Topology Zoo inputs that completed successfully, my tool solved all of them faster. The measured speedups were large, with a geometric mean of 447.23x. For the 24 FatTree examples, the mean speedup was 465.03x, and for the 25 Small-World examples, the mean speedup was 4484.73x. I also compared the Incremental and Batch solvers on the same inputs. Incremental performs better on almost all examples, with mean speedup of 4.26x, 5.27x, 11.74x on the datasets shown in Figure 2.4(a-c) and maximum runtimes of 0.36s, 2.80s, and 0.92s respectively. The maximum runtimes for Batch were 6.71s, 39.75s, and 12.50s.

Incremental vs. NetPlumber. I also measured the performance of Incremental versus the network property checker NetPlumber (Figure 2.4(d-f)). Note that NetPlumber uses rule-granularity for updates, so I enabled this mode in my tool for these experiments. For the three datasets, my checker is faster on all experiments, with mean speedups of (6.41x, 4.90x, 17.19x). NetPlumber does not report counterexamples, putting it at a disadvantage in this end-to-end comparison, so I also measured total Incremental versus NetPlumber runtime on the same set of model-checking questions posed by Incremental for the Small-World example. My tool is still faster on all instances, with a mean speedup of 2.74x.

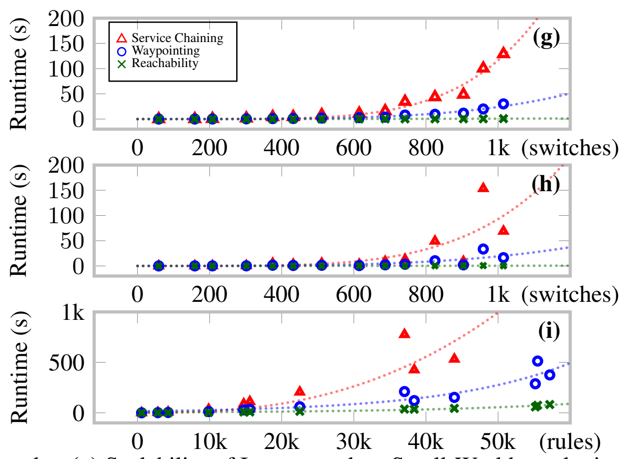


Figure 2.5: Overall scalability results: **(g)** Scalability of Incremental on Small-World topologies of increasing size; **(h)** Scalability when no correct switch-granularity update exists (i.e. algorithm reports "impossible"), and **(i)** Scalability of fine-grained (rule-granularity) approach for solving switch-impossible examples in (h).

Scalability. To quantify my tool's scalability, I constructed Small World topologies with up to 1500 switches, and ran experiments with large diamond updates—the largest has 1015 switches updating. The results appear in Figure 2.5(g). The maximum synthesis times for the three properties were 129.04s, 30.11s, and 0.85s, which shows that my tool scales to problems of realistic size.

Infeasible Updates. I also considered examples for which there is no switch-granular update. Figure 2.5(h) shows the results of experiments where I generated a second diamond atop the first one, requiring it to route traffic in the opposite direction. Using switch-granularity, the inputs are reported as unsolvable in maximum time 153.48s, 33.48s, and 0.69s. Using rule-granularity, these inputs are solved successfully for up to 1000 switches with maximum times of 776.13s, 512.84s, and 82.00s (see Figure 2.5(i)).

Waits. I also separately measured the time needed to run the wait-removal heuristic for the Figure 2.5 experiments. For (g), the maximum wait-removal runtime was 0.89s, resulting in 2 needed waits for each instance. For (i), the maximum wait-removal runtime was 103.87s, resulting in about 2.6 waits on average (with a maximum of 4). For the largest problems in (g) and (i), this corresponds to removal of 1397/1399 and 55823/55826 waits (about 99.9%).

2.5 Related Work

This chapter extends preliminary work reported in a workshop paper [94]. I present a more precise and realistic network model, and replace expensive calls to an external model checker with calls to a new built-in **incremental** network model checker. I extend the DFS search procedure with optimizations and heuristics that improve performance dramatically. Finally, I evaluate my tool on a comprehensive set of benchmarks with real-world topologies.

Synthesis of concurrent programs. There is much previous work on synthesis for concurrent programs [121, 113, 52]. In particular, work by Solar-Lezama et al. [113] and Vechev et al. [121] synthesizes sequences of instructions. However, traditional synthesis and synthesis for networking are quite different. First, traditional synthesis is a game against the environment which (in the concurrent programming case) provides inputs and schedules threads; in contrast, my synthesis problem involves reachability on the space of configurations. Second, my space of configurations is very rich, meaning that checking configurations is itself a model checking problem.

Network updates. There are many protocol- and property-specific algorithms for implementing network updates, e.g. avoiding packet/bandwidth loss during planned maintenance to BGP [43, 104]. Other work avoids routing loops and blackholes during IGP migration [118]. Work on network updates in SDN proposed the notion of **consistent updates** and several implementation mechanisms, including two-phase updates [105]. Other work explores propagating updates incrementally, reducing the space overhead on switches [63]. As mentioned in Section 1.3.1, recent work proposes ordering updates for specific properties [59], whereas I can handle combinations and variants of these properties. Furthermore, SWAN and zUpdate add support for bandwidth guarantees [54, 75]. Zhou et al. [126] consider customizable trace properties, and propose a dynamic algorithm to find order updates. This solution can take into account unpredictable delays caused by switch updates. However, it may not always find a solution, even if one exists. In contrast, I obtain a completeness guarantee for my static algorithm. Ludwig et al. [79] consider ordering updates for waypointing properties.

Model checking. Model checking has been used for network verification [108, 81, 65, 66, 82]. The closest to my work is the incremental checker NetPlumber [64]. Surface-level differences include the specification languages (LTL vs. regular expressions), and NetPlumber's lack of counterexample output. The main difference is incrementality: Netplumber restricts checking to "probe nodes," keeping track of "header-space" reachability information for those nodes, and then performing property queries based on this. In contrast, I look at the **property**, keeping track of **portions of the property** holding at each node, which keeps incremental rechecking times low. The empirical comparison (Section 2.4) showed better performance of my tool as a back-end for synthesis.

Incremental model checking has been studied previously, with [112] presenting the first incremental model checking algorithm, for alternation-free μ-calculus. I consider LTL properties and specialize my algorithm to exploit the no-forwarding-loops assumption. The paper [26] introduced an incremental algorithm, but it is specific to the type of partial results produced by IC3 [17].

Chapter 3

Event-Driven Network Programming

3.1 Event-Driven Network Behavior

This chapter presents my new consistency model for stateful network programs: **event-driven consistent update**.

Preliminaries. A **packet** pkt is a record of fields $\{f_1; f_2; \cdots; f_n\}$, where fields f represent properties like source/destination address, protocol type, etc. The (numeric) values of fields are accessed via the notation $pkt.f$, and field updates are denoted $pkt[f \leftarrow n]$. A **switch** sw is a node in the network with one or more **ports** pt. A **host** is a switch that can be a source or a sink of packets. A **location** l is a switch-port pair $n{:}m$. Locations may be connected by (unidirectional) physical links (l_{src}, l_{dst}) in the topology.

Packet forwarding is dictated by a **network configuration** C. A **located packet** $lp = (pkt, sw, pt)$ is a tuple consisting of a packet and a location $sw{:}pt$. I model C as a relation on located packets: if $C(lp, lp')$, then the network maps lp to lp', possibly changing its location and rewriting some of its fields. Since C is a relation, it allows multiple output packets to be generated from a single input. In a real network, the configuration only forwards packets between ports within each individual switch, but for convenience, I assume that my C also captures link behavior (forwarding between switches), i.e. $C((pkt, n_1, m_1), (pkt, n_2, m_2))$ holds for each link $(n_1{:}m_1, n_2{:}m_2)$. I refer to a sequence of located packets that starts at a host and can be produced by C as a **packet trace**, using $\mathit{Traces}(C)$ to denote the set of all such packet traces. I let $\mathcal{C}$ be the set of all configurations.

Consider a tuple $ntr = (lp_0 lp_1 \cdots, T)$, where the first component is a sequence of located packets, and each $t \in T$ is an increasing sequence of indices corresponding to located packets in the sequence. I call such a tuple a **network trace** if and only if the following conditions hold:

(1) for each lp_j, I have $j \in t$ for some $t \in T$, and

(2) for each $t{=}(k_0 k_1 \cdots){\in}T$, lp_{k_0} is at a host, and $\exists C{\in}\mathcal{C}$ such that $C(lp_{k_i}, lp_{k_{i+1}})$ holds for all i, and

(3) if I consider the graph G with nodes $\{k : (\exists t \in T : k \in t)\}$ and edges $\{(k_i, k_{i+1}) : (\exists t \in T : t = k_0 k_1 \cdots k_i k_{i+1} \cdots)\}$, then G is a family of **trees** rooted at $K = \{k_0 : (\exists t \in T : t = k_0 \cdots)\}$.

I will use $ntr{\downarrow}k$ to denote the set $\{t \in T : k \in t\}$, and when $t = (k_0 k_1 \cdots) \in T$, I can use similar notation $ntr{\downarrow}t$ to denote the packet trace $lp_{k_0} lp_{k_1} \cdots$. Intuitively, I have defined a network trace to be an interleaving of these packet traces (the packet traces form the family of **trees** because, as previously mentioned, the configuration allows multiple output packets from a single input packet). Ultimately, I will introduce a consistency definition that dictates which interleavings of packet traces are correct.

I now define how the network changes its configuration in response to events. An **event** e is a tuple $(\varphi, sw, pt)_{eid}$, where eid is an (optional) event identifier and φ is a first-order formula over fields. Events model the arrival of a packet satisfying φ (denoted $pkt \models \varphi$) at location $sw{:}pt$. Note that I could have other types of events—anything that a switch can detect could be an event—but for simplicity, I focus on packet events. I say that a located packet $lp = (pkt, sw', pt')$ **matches** an event $e = (\varphi, sw, pt)$ (denoted by $lp \models e$) if and only if $sw = sw' \wedge pt = pt' \wedge pkt \models \varphi$.

Definition 11 (Happens-before relation $\prec_{ntr}$)**.** Given a network trace $ntr = (lp_0 lp_1 \cdots, T)$, the happens-before relation $\prec_{ntr}$ is the least partial order on located packets that

- respects the total order induced by ntr at switches, i.e., $\forall i, j : lp_i \prec lp_j \Leftarrow i < j \wedge lp_i = (pkt, sw, pt) \wedge lp_j = (pkt', sw, pt')$, and
- respects the total order induced by ntr for each packet, i.e., $\forall i, j : lp_i \prec lp_j \Leftarrow i < j \wedge \exists t \in T : i \in t \wedge j \in t$.

Event-Driven Consistent Update. In Section 1.3.2, I informally defined an event-driven consistent update as a triple $C_i \xrightarrow{e} C_f$ consisting of an initial configuration C_i, event e, and final configuration C_f. Here, I formalize that definition in a way that describes **sequences** of events and configurations (in the single-event case, this formal definition is equivalent to the informal one). I denote an **event-driven consistent update** as a pair $(U, \mathcal{E})$, where U is a sequence $C_0 \xrightarrow{e_0} C_1 \xrightarrow{e_1} \cdots \xrightarrow{e_n} C_{n+1}$, and $\{e_0, \cdots, e_n\} \subseteq \mathcal{E}$.

Let $ntr = (lp_0 lp_1 \cdots, T)$ be a network trace. Given an event-driven consistent update $(U, \mathcal{E})$, I need

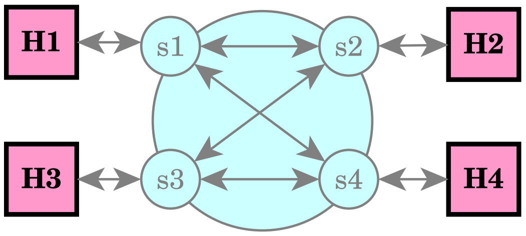


Figure 3.1: Example topology with four switches and hosts.

the indices where the events from U first occurred. Specifically, I wish to find the sequence $k_0, \cdots, k_n$ where lp_j does not match any $e \in \mathcal{E}$ for any $j > k_n$, and the following properties hold for all $0 \leq i \leq n$ (assuming $k_{(-1)} = -1$ for convenience):

- $k_i > k_{i-1}$, and
- lp_{k_i} matches e_i, and for all j, if $k_{i-1} < j < k_i$ then lp_j does not match e_i (i.e., k_i is the first occurrence of e_i after the index k_{i-1}), and
- $\exists t \in ntr{\downarrow}k_i$ such that t is in $\mathit{Traces}(C_i)$ (intuitively, the event e_i can be triggered only by a packet processed in the immediately preceding configuration).

If such a sequence exists, it is unique, and I denote it by $FO(ntr, U)$, shorthand for “first occurrences.”

Definition 12 (Event-driven consistent update correctness)**.** A network trace $ntr = (lp_0 lp_1 \cdots, T)$ is **correct** with respect to an event-driven consistent update $U = C_0 \xrightarrow{e_0} C_1 \xrightarrow{e_1} \cdots \xrightarrow{e_n} C_{n+1}$, if $FO(ntr, U) = k_0, \cdots, k_n$ exists, and for all $0 \leq i \leq n$, the following holds for each $ntr{\downarrow}t = lp'_0 lp'_1 \cdots$ where $t \in T$:

- $ntr{\downarrow}t$ is in $\mathit{Traces}(C)$ for some $C \in \{C_0, \cdots, C_{n+1}\}$ (packet is processed entirely by one configuration), and
- if $\forall j : lp'_j \prec lp_{k_i}$, then $ntr{\downarrow}t$ is in $\mathit{Traces}(C)$ for some $C \in \{C_0, \cdots, C_i\}$ (the packet is processed entirely in a preceding configuration), and
- if $\forall j : lp_{k_i} \prec lp'_j$, then $ntr{\downarrow}t$ is in $\mathit{Traces}(C)$ for some $C \in \{C_{i+1}, \cdots, C_{n+1}\}$ (the packet is processed entirely in a following configuration).

To illustrate, consider Figure 3.1. I describe an update $C_i \xrightarrow{e} C_f$. In the initial configuration C_i, the host H_1 can send packets to H_2, but not vice-versa. In the final configuration C_f, traffic from H_2 to H_1 is allowed. Event e models the arrival to s_4 of a packet from H_1 (imagine s_4 is part of a distributed firewall). Assume that e occurs, and immediately afterwards, H_2 wants to send a packet to s_1. Can s_2 drop the packet (as it would do in configuration C_i)? Event-driven consistent updates allow this, as otherwise I would require s_2 to react immediately to the event at s_4, which would be an example of action at a distance. Formally, the occurrence of e is not in a happens-before relation with the arrival of the new packet to s_2. On the other hand, if e.g. s_4 forwards some packets to s_1 and s_2 before the new packet from H_2 arrives, s_1 and s_2 would be required to change their configurations, and the packet would be allowed to reach H_1.

Network Event Structures. As I have shown, event-driven consistent updates specify how the network should behave during a sequence of updates triggered by events, but additionally, I want the ability to capture constraints between the events themselves. For example, I might wish to say that e_2 can only happen after e_1 has occurred, or that e_2 and e_3 cannot both occur in the same network trace.

To model such constraints, I turn to the **event structures** model introduced by Winskel [123]. Intuitively, an event structure endows a set of events $\mathcal{E}$ with (a) a **consistency predicate** (con) specifying which events are allowed to occur in the same sequence, and (b) an **enabling relation** ($\vdash$) specifying a (partial) order in which events can occur. This is formalized in the following definition (note that I use $\subseteq_{fin}$ to mean "finite subset," and $\mathcal{P}_{fin}(X) = \{Y : Y \subseteq_{fin} \mathcal{P}(X)\}$).

Definition 13 (Event structure)**.** An event structure is a tuple $(\mathcal{E}, con, \vdash)$ where:

- $\mathcal{E}$ is a set of events,
- $con : (\mathcal{P}_{fin}(\mathcal{E}) \to Boolean)$ is a consistency predicate that satisfies $con(X) \wedge Y \subseteq X \Rightarrow con(Y)$,
- $\vdash : (\mathcal{P}(\mathcal{E}) \times \mathcal{E} \to Boolean)$ is an enabling relation that satisfies $(X \vdash e) \wedge X \subseteq Y \implies (Y \vdash e)$.

An event structure can be seen as defining a transition system whose states are subsets of $\mathcal{E}$ that are consistent and reachable via the enabling relation. I term these subsets **event-sets** (called "configurations" in [123]).

Definition 14 (Event-set of an event structure)**.** Given an event structure $N = (\mathcal{E}, con, \vdash)$, an **event-set** of N is any subset $X \subseteq \mathcal{E}$ which is: (a) consistent: $\forall Y \subseteq_{fin} X$, $con(Y)$ holds, and (b) reachable via the enabling relation: for each $e \in X$, there exists $e_0, e_1, \cdots, e_n \in X$ where $e_n = e$ and $\emptyset \vdash \{e_0\}$ and $\{e_0, \cdots, e_{i-1}\} \vdash e_i$ for all $1 \leq i \leq n$.

I want to be able to specify which network configuration should be active at each event-set of the event structure. Thus, I need the following extension of event structures.

Definition 15 (NES)**.** A **network event structure** is a tuple $(\mathcal{E}, con, \vdash, g)$ where $(\mathcal{E}, con, \vdash)$ is an event structure, and $g : (\mathcal{P}(\mathcal{E}) \to \mathcal{C})$ maps each event-set of the event structure to a network configuration.

Correct Network Traces. I now define what it means for a network trace ntr to be **correct** with respect to an NES $N = (\mathcal{E}, con, \vdash, g)$. I begin by constructing a sequence S of events that is allowed by N. A sequence $S = e_0 e_1 \cdots e_n$ is **allowed by** N, if $\emptyset \vdash \{e_0\} \wedge con(\{e_0\})$, and $\forall 1 \leq i \leq n : (\{e_0, e_1, \cdots, e_{i-1}\} \vdash e_i \wedge con(\{e_0, e_1, \cdots, e_i\}))$.

Intuitively, I say that ntr is correct if there is a sequence of events allowed by N which would cause ntr to satisfy the event-driven consistent update condition.

Definition 16 (Correct network trace)**.** Let $\mathcal{S}$ be the set of all sequences allowed by N. Formally, a network trace $ntr = (lp_0 lp_1 \cdots, T)$ is **correct** with respect to N if

- no lp_j matches any $e \in \mathcal{E}$, and for all packet traces $ntr{\downarrow}t$ where $t \in T$, I have $ntr{\downarrow}t$ is in $Traces(g(\emptyset))$, or
- there exists some $e_0 e_1 \cdots e_n \in \mathcal{S}$ such that ntr is correct with respect to event-driven consistent update $(g(\emptyset) \xrightarrow{e_0} g(\{e_0\}) \xrightarrow{e_1} \cdots \xrightarrow{e_n} g(\{e_0, \cdots, e_n\}), \mathcal{E})$.

Locality Restrictions for Incompatible Events. I now show how NESs can be used to impose reasonable locality restrictions. A set of events E is called **inconsistent** if and only if $con(E)$ does not hold. I use the term **minimally-inconsistent** to describe inconsistent sets where all proper subsets are not inconsistent. An NES N is called **locally-determined** if and only if for each of its minimally-inconsistent sets E, all events in E happen at the same switch (i.e., $\exists sw \forall e_i \in E : e_i = (\varphi_i, sw, pt_i)$). To illustrate the need for the locally-determined property, I consider the following two programs, P_1 and P_2.

- **Program** P_1: Recall that two events are inconsistent if either of them can happen, but both cannot happen in the same execution. Consider the topology shown in Figure 3.1 and suppose this program requires that H_2 and H_4 can both receive packets from H_1, but only the first one to receive a packet is allowed to respond. There will be two events e_1 and e_2, with e_1 the arrival of a packet from H_1 at s_2, and e_2 the arrival of a packet from H_1 at s_4. These events are always enabled, but the set

$\{e_1, e_2\}$ is not consistent, i.e. $con(\{e_1, e_2\})$ does not hold. This models the fact that at most one of the events can take effect. These events happen at different switches—making sure that **at most one** of the events takes effect would necessitate information to be propagated instantaneously "at a distance." In implementations, this would require using inefficient mechanisms (synchronization and/or packet buffering). My locality restriction is a clean condition which ensures that the NES is efficiently implementable.

- **Program** P_2: Consider a different program where H_2 can send traffic to one of the two hosts H_1, H_3 that sends it a packet first. The two events (a packet from H_1 arriving at s_2, and a packet from H_3 arriving at s_2) are still inconsistent, but inconsistency does not cause problems in this case, because both events happen at the same switch (the switch can determine which one was first).

In contrast to my approach, an uncoordinated update approach improperly handles locality issues, mainly because it does not guarantee **when** the configuration change occurs. Consider the program P_1 again, and consider the (likely) scenario where events e_1 and e_2 happen nearly simultaneously. In an uncoordinated approach, this could result in switch s_2 hearing about e_1, e_2 (in that order), and s_4 hearing about e_2, e_1 (in that order), meaning the two switches would have conflicting ideas of which event was "first" (i.e. the switches would be in conflicting states, and this conflict cannot be resolved). In my implementation, I would require e_1 and e_2 to occur at the same switch, guaranteeing that I never see such a conflicting mix of states.

Strengthening Consistency. I now show that strengthening the consistency conditions imposed by NESs would lead to lower availability, as it would lead to the need for expensive synchronization, packet buffering, etc. First, I will try to remove the locally-determined condition, and second, I will try to obtain a strengthened consistency condition. The proof of the following theorem is an adaptation of the proof of the CAP theorem [18], as presented in [45]. The idea is that in asynchronous network communication, a switch might need to wait arbitrarily long to hear about an event.

Lemma 8. In general, it is impossible to implement an NES that does not have the locally-determined condition while guaranteeing that switches process each packet within an a priori given time bound.

Proof Sketch. Consider a simple NES, with event sets $\emptyset$, $\{e_1\}$, $\{e_2\}$, and where $\{e_1\}$ and $\{e_2\}$ are both enabled from $\emptyset$. Assume that $con(\{e_1, e_2\})$ does not hold, and that e_1 can happen at switch A and e_2 can happen at switch B (i.e., the locally-determined condition does not hold).

Because the communication is asynchronous, there is no a priori bound on how long the communication between switches can take. When a packet p that matches e_2 arrives at the switch B, the switch must distinguish the following two cases: (#1) event e_1 has occurred at A (and thus p does not cause e_2), or (#2) event e_1 has not occurred at A (and thus p causes e_2). No matter how long B waits, it cannot distinguish these two cases, and hence, when a packet that matches e_2 arrives to B, the switch B cannot correctly decide whether to continue as if e_2 has happened. It has the choice to either eventually decide (and risk the wrong decision), or to buffer the packet that matches e_2. $\square$

I now ask whether I can strengthen the event-driven consistent update definition. I define **strong update** as an update $C_1 \xrightarrow{e} C_2$ such that immediately after e occurred, the network processes all incoming packets in C_2. I obtain the following lemma by the same reasoning as the previous one.

Lemma 9. In general, it is impossible to implement strong updates and guarantee that switches process each packet within an a priori given time bound.

Proof Sketch. Let A be the switch where e can happen, and let B be a switch on which the configurations C_1, C_2 differ. For A and B, the same argument as in the previous lemma shows that B must either risk the wrong decision on whether to process packets using C_1 or C_2, or buffer packets. $\square$

3.2 Programming with Events

The correctness condition I described in the previous section offers useful application-level guarantees to network programmers. The programmer is freed from thinking about interleavings of packets/events and responses to events (configuration updates). She can think in terms of my consistency model—each packet is processed in a single configuration, and packets entering "after" an event will be processed in the new configuration (similar to causal consistency). An important consequence is that the response to an event is immediate with respect to a given flow if the event is handled at that flow's ingress switch.

With this consistency model in mind, programmers can proceed by specifying the desired event-driven program behavior using network event structures. This section introduces an intuitive method for building

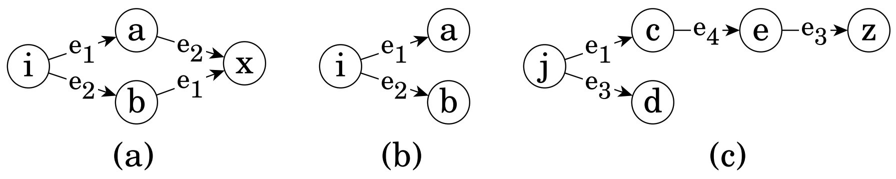


Figure 3.2: Event-driven transition systems.

NESs using simple transition systems where nodes correspond to configurations and edges correspond to events. I also present a network programming language based on NetKAT that provides a compact notation for specifying both the transition system and the configurations at the nodes.

3.2.1 Event-Driven Transition Systems

Definition 17 (Event-driven Transition System)**.** An **event-driven transition system (ETS)** is a graph (V, D, v_0), in which V is a set of vertices, each labeled by a configuration; $D \subseteq V \times V$ is a set of edges, each labeled by an event e; and v_0 is the initial vertex.

Consider the ETSs shown in Figure 3.2 (a-b). In (a), the two events are intuitively **compatible**—they can happen in any order, so I obtain a correct execution if both happen in different parts of the network, and different switches can have a different view of the order in which they happened. In (b), the two events are intuitively **incompatible**—only one of them can happen in any particular execution. Therefore, even if they happen nearly simultaneously, only one of them should take an effect. To implement this, I require the locality restriction—I need to check whether the two events happen at the same switch. I thus need to distinguish between ETSs such as (a) and (b) in Figure 3.2, to determine where locality restrictions must be imposed in the conversion from an ETS to an NES.

From ETSs to NESs. Given an ETS, I first form the event sets (Definition 14) and then construct the enabling relation and consistency predicate. Given an ETS T, consider the set $W(T)$ of sequences of events in T from the initial node to any vertex (including the empty sequence). For each sequence $p \in W(T)$, let $E(p)$ be the set of events collected along the sequence. The set $F(T) = \{E(p) \mid p \in W(T)\}$ is my candidate collection of event sets. I now define conditions under which $F(T)$ gives rise to an NES.

(1) I require that each set E in $F(T)$ must correspond to exactly one network configuration. This holds

if all paths in $W(T)$ corresponding to E end at states labeled with the same configuration.

(2) I require that $F(T)$ is **finite-complete**, i.e. for any sets $E_1, E_2, \cdots, E_n$ where each $E_i \in F(T)$, if there is a set $E' \in F(T)$ which contains every E_i (an upper bound for the sets E_i), then the set $E_{lub} = \cup_i E_i$ (the least upper bound for the E_i) must also be in $F(T)$. For example, consider the ETS in Figure 3.2(c), which violates this condition since the event-sets $E_1 = \{e_1\}$ and $E_2 = \{e_3\}$ are both subsets of $\{e_1, e_4, e_3\}$, but there is no event-set of the form $E_1 \cup E_2 = \{e_1, e_3\}$.

In [123], such a collection $F(T)$ is called a **family of configurations**. My condition (2) is condition (i) in Theorem 1.1.9 in [123] (conditions (ii)-(iii) are satisfied by construction).

Given an ETS T, it is not difficult to confirm the above conditions statically. They can be checked using straightforward graph algorithms, and any problematic vertices or edges in T can be indicated to the programmer. The development of **efficient** checking algorithms is left for future work.

I build the con and $\vdash$ relations of an NES from the family $F(T)$, using Theorem 1.1.12. of [123]. Specifically, predicate con can be defined by declaring all sets in $F(T)$ as consistent, and for $\vdash$, I take the smallest relation satisfying the constraints $\emptyset \vdash e \iff \{e\} \in F(T)$ and $X' \vdash e \iff (X' \in con) \wedge ((X' \cup \{e\}) \in F(T) \vee (\exists X \subseteq X' : X \vdash e))$.

After obtaining an NES, deciding whether it satisfies the locality restriction is easy: I check whether the NES is locally determined (see Section 3.1), verifying for each minimally-inconsistent set that the locality restriction holds. Again, I leave the **efficiency** of this check for future work.

Loops in ETSs. If there are loops in the ETS T, the previous definition needs to be slightly modified, because I need to "rename" events encountered multiple times in the same execution. This gives rise to an NES where each event-set is finite, but the NES itself might be infinite (and thus can only be computed lazily). If I have the ability to store and communicate unbounded (but finite) event-sets in the network runtime, then no modifications are needed to handle infinite NESs in the implementation (which is described in Section 3.3). Otherwise, there are various correct overapproximations I could use, such as computing the strongly-connected components (SCCs) of the ETS, enforcing the locality restriction on events in each (non-singleton) SCC, and requiring the implementation to attach timestamps on occurrences of events in those SCCs. For simplicity of the presentation, I will consider only loop-free ETSs in this chapter.

3.2.2 Stateful NetKAT

NetKAT [3] is a domain-specific language for specifying network behavior. It has semantics based on Kleene Algebra with Tests (KAT), and a sound and complete equational theory that enables formal reasoning about programs. Operationally, a NetKAT program behaves as a function which takes as input a single packet, and uses tests, field-assignments, sequencing, and union to produce a set of "histories" corresponding to the packet's traces.

Standard NetKAT does not support mutable **state**. Each packet is processed in isolation using the function described by the program. In other words, I can use a standard NetKAT program for specifying individual network configurations, but not event-driven configuration changes. I describe a stateful variant of NetKAT which allows me to compactly specify a **collection** of network configurations, as well as the event-driven relationships between them (i.e. an ETS). This variant preserves the existing equational theory of the individual static configurations (though it is not a KAT itself), but also allows packets to affect processing of future packets via assignments to (and tests of) a global **state**. The syntax of Stateful NetKAT is shown in Figure 3.3. A Stateful NetKAT program is a **command**, which can be:

- a **test**, which is a formula over packet header fields (there are special fields **sw** and **pt** which test the switch- and port-location of the packet respectively),
- a **field assignment** $x{\leftarrow}n$, which modifies the (numeric) value stored in a packet's field,
- a **union** of commands $p + q$, which unions together the packet-processing behavior of p and q,
- a command **sequence** $p\,;q$, which runs packet-processing program q on the result of p,
- an **iteration** $p{*}$, which is equivalent to **true** $+\, p + (p\,;p) + (p\,;p\,;p) + \cdots$,
- or a **link definition** $(n_1{:}m_1){\rightarrow\!\triangleright}(n_2{:}m_2)$, which forwards a packet from port m_1 at switch n_1 across a physical link to port m_2 at switch n_2.

The functionality described above is also provided by standard NetKAT [111]. The key distinguishing feature of my **Stateful** NetKAT is a special global vector-valued variable called **state**, which allows the programmer to represent a collection of NetKAT programs. The function shown in Figure 3.4 gives the standard NetKAT program $[\![p]\!]_{\vec{k}}$ corresponding to each value $\vec{k}$ of the state vector (for conciseness, I only show the non-trivial cases). I can use the NetKAT compiler [111] to generate forwarding tables (i.e. configurations)

$$\begin{array}{lcll}
f & \in & \mathit{Field} & \textbf{(packet field name)}\\
n & \in & \mathbb{N} & \textbf{(numeric value)}\\
x & ::= & f \mid \textbf{pt} & \textbf{(modifiable field)}\\
a,b & ::= & \textbf{true} \mid \textbf{false} \mid x = n \mid \textbf{sw}{=}n \mid \textbf{state}(n) = n \mid a \vee b \mid a \wedge b \mid \neg a & \textbf{(test)}\\
p,q & ::= & a \mid x \leftarrow n \mid p + q \mid p \mathbin{;} q \mid p* \mid (n{:}n) \twoheadrightarrow (n{:}n) \mid (n{:}n) \twoheadrightarrow (n{:}n) \twoheadrightarrow \langle \textbf{state}(n) \leftarrow n \rangle & \textbf{(command)}
\end{array}$$

Figure 3.3: Stateful NetKAT: syntax.

corresponding to these, which I denote $C([\![p]\!]_{\vec{k}})$.

3.2.3 Converting Stateful NetKAT Programs to ETSs

Now that I have the $[\![\cdot]\!]_{\vec{k}}$ function to extract the static configurations (NetKAT programs) corresponding to the vertices of an ETS, I define $(\!|\cdot|\!)_{\vec{k}}$, which produces the event-edges (Figure 3.5). This collects (using parameter φ) the conjunction of all tests seen up to a given program location, and records a corresponding event-edge when a state assignment command is encountered. The function returns a tuple (D, P), where D is a set of event-edges, and P is a set of updated conjunctions of tests. In the figure, the $\sqcup$ operator denotes pointwise union of tuples, i.e. $(A_1, B_1, \cdots) \sqcup (A_2, B_2, \cdots) = (A_1 \cup A_2, B_1 \cup B_2, \cdots)$. The $\blacksquare$ operator denotes (pointwise) Kleisli composition, i.e. $(f \blacksquare g) \triangleq \bigsqcup \{g\ y : y \in f\ x\}$, and function F is as follows.

$$\begin{array}{rcl}
F_p^0\ (\varphi, \vec{k}) & \triangleq & (\{\}, \{\varphi\})\\
F_p^{j+1}\ (\varphi, \vec{k}) & \triangleq & ((\!|p|\!)_{\vec{k}} \blacksquare F_p^j)\ \varphi
\end{array}$$

The $\ominus$ is either equality "$=$" or inequality "$\neq$", and $\not\ominus$ is the opposite symbol with respect to $\ominus$. Given any conjunction φ and a header field f, the formula $(\exists f : \varphi)$ strips all predicates of the form $(f \ominus n)$ from φ.

Using *fst* to denote obtaining the first element of a tuple, I can now produce the event-driven transition system for a Stateful NetKAT program p with the initial state $\vec{k}_0$:

$$\begin{array}{rcl}
[\![\textbf{state}(m){=}n]\!]_{\vec{k}} & \triangleq & \begin{cases} [\![\textbf{true}]\!]_{\vec{k}} & \text{if } \vec{k}(m){=}n \\ [\![\textbf{false}]\!]_{\vec{k}} & \text{otherwise} \end{cases}\\
[\![(a{:}b) \twoheadrightarrow (c{:}d) \twoheadrightarrow \langle \textbf{state}(m) \leftarrow n \rangle]\!]_{\vec{k}} & \triangleq & [\![(a{:}b) \twoheadrightarrow (c{:}d)]\!]_{\vec{k}}
\end{array}$$

Figure 3.4: Stateful NetKAT: extracting NetKAT Program (state $\vec{k}$).

$$\begin{aligned}
\llparenthesis f \circledequal n \rrparenthesis_{\vec{k}}\ \varphi &\triangleq (\{\}, \{\varphi \wedge f \circledequal n\}) \\
\llparenthesis \textbf{sw} \circledequal n \rrparenthesis_{\vec{k}}\ \varphi &\triangleq \llparenthesis \textbf{true} \rrparenthesis_{\vec{k}}\ \varphi \\
\llparenthesis \textbf{port} \circledequal n \rrparenthesis_{\vec{k}}\ \varphi &\triangleq \llparenthesis \textbf{true} \rrparenthesis_{\vec{k}}\ \varphi \\
\llparenthesis state(m) \circledequal n \rrparenthesis_{\vec{k}}\ \varphi &\triangleq \begin{cases} \llparenthesis \textbf{true} \rrparenthesis_{\vec{k}}\ \varphi & \text{if } \vec{k}(m) \circledequal n \\ \llparenthesis \textbf{false} \rrparenthesis_{\vec{k}}\ \varphi & \text{otherwise} \end{cases} \\
\llparenthesis f \leftarrow n \rrparenthesis_{\vec{k}}\ \varphi &\triangleq (\{\}, \{(\exists f : \varphi) \wedge f{=}n\}) \\
\llparenthesis p + q \rrparenthesis_{\vec{k}}\ \varphi &\triangleq (\llparenthesis p \rrparenthesis_{\vec{k}}\ \varphi) \sqcup (\llparenthesis q \rrparenthesis_{\vec{k}}\ \varphi) \\
\llparenthesis p\ ;\ q \rrparenthesis_{\vec{k}}\ \varphi &\triangleq (\llparenthesis p \rrparenthesis_{\vec{k}} \blacksquare \llparenthesis q \rrparenthesis_{\vec{k}})\ \varphi \\
\llparenthesis p* \rrparenthesis_{\vec{k}}\ \varphi &\triangleq \textstyle\bigsqcup_j F_p^j\ (\varphi, \vec{k})
\end{aligned}
\qquad
\begin{aligned}
\llparenthesis a \wedge b \rrparenthesis_{\vec{k}}\ \varphi &\triangleq \llparenthesis a\ ;\ b \rrparenthesis_{\vec{k}}\ \varphi \\
\llparenthesis a \vee b \rrparenthesis_{\vec{k}}\ \varphi &\triangleq \llparenthesis a + b \rrparenthesis_{\vec{k}}\ \varphi \\
\llparenthesis \textbf{true} \rrparenthesis_{\vec{k}}\ \varphi &\triangleq (\{\}, \{\varphi\}) \\
\llparenthesis \textbf{false} \rrparenthesis_{\vec{k}}\ \varphi &\triangleq (\{\}, \{\}) \\
\llparenthesis \neg\textbf{true} \rrparenthesis_{\vec{k}}\ \varphi &\triangleq \llparenthesis \textbf{false} \rrparenthesis_{\vec{k}}\ \varphi \\
\llparenthesis \neg\textbf{false} \rrparenthesis_{\vec{k}}\ \varphi &\triangleq \llparenthesis \textbf{true} \rrparenthesis_{\vec{k}}\ \varphi \\
\llparenthesis \neg(v \circledequal n) \rrparenthesis_{\vec{k}}\ \varphi &\triangleq \llparenthesis v \not\circledequal n \rrparenthesis_{\vec{k}}\ \varphi \\
\llparenthesis \neg\neg a \rrparenthesis_{\vec{k}}\ \varphi &\triangleq \llparenthesis a \rrparenthesis_{\vec{k}}\ \varphi \\
\llparenthesis \neg(a \wedge b) \rrparenthesis_{\vec{k}}\ \varphi &\triangleq \llparenthesis \neg a \vee \neg b \rrparenthesis_{\vec{k}}\ \varphi \\
\llparenthesis \neg(a \vee b) \rrparenthesis_{\vec{k}}\ \varphi &\triangleq \llparenthesis \neg a \wedge \neg b \rrparenthesis_{\vec{k}}\ \varphi
\end{aligned}$$

$$\begin{aligned}
\llparenthesis (s_1{:}p_1) \rightarrowtriangle (s_2{:}p_2) \rrparenthesis_{\vec{k}}\ \varphi &\triangleq (\{\}, \{\varphi\}) \\
\llparenthesis (s_1{:}p_1) \rightarrowtriangle (s_2{:}p_2) \rightarrowtriangle \langle \textbf{state}(m) \leftarrow n \rangle \rrparenthesis_{\vec{k}}\ \varphi &\triangleq (\{(\vec{k}, (\varphi, s_2, p_2), \vec{k}[m \mapsto n])\}, \{\varphi\})
\end{aligned}$$

Figure 3.5: Stateful NetKAT: extracting event-edges from state $\vec{k}$.

$$\begin{aligned}
ETS(p) \triangleq&\quad (V, D, v_0) \\
&\text{where } V \triangleq \textstyle\bigcup_{\vec{k}} \{(\vec{k}, C(\llbracket p \rrbracket_{\vec{k}}))\} \\
&\text{and } D \triangleq fst\left(\textstyle\bigsqcup_{\vec{k}} \llparenthesis p \rrparenthesis_{\vec{k}}\ \textbf{true}\right) \\
&\text{and } v_0 \triangleq (\vec{k}_0, C(\llbracket p \rrbracket_{\vec{k}_0}))
\end{aligned}$$

3.3 Implementing Event-Driven Programs

Next, I show one method of implementing NESs in a real SDN, and I prove that this approach is correct—i.e., all traces followed by actual packets in the network are correct with respect to Definition 16 in Section 3.1. At a high level, the basic idea of my implementation strategy can be understood as follows. I assume that the switches in the network provide mutable state that can be read and written as packets are processed. Given an NES, I assign a tag to each event-set and compile to a collection of configurations whose rules are “guarded” by the appropriate tags. I then add logic that (i) updates the mutable state to record local events, (ii) stamps incoming packets with the tag for the current event-set upon ingress, and (iii) reads the tags carried by packets, and updates the event-set at subsequent switches.

3.3.1 Implementation Building Blocks

Static Configurations. The NES contains a set of network configurations that need to be installed as flow tables on switches. In addition, I must be able to transition to a new configuration in response to a local event. I do this **proactively**, installing all of the needed rules on switches in advance, with each rule guarded by its configuration's ID. This has a disadvantage of being less efficient in terms of rule-space usage, but an advantage of allowing quick configuration changes. In Section 3.4.3, I discuss an approach for addressing the space-usage issue by sharing rules between configurations. My implementation strategy encodes each event-set in the NES as an integer, so a single unused packet header field (or single register on switches) can be used. This keeps the overhead low, even for very large programs.

Stateful Switches. Emerging data-plane languages such as P4 [16] and OpenState [13] are beginning to feature advanced functionality such as customizable parsing, stateful memories, etc. I assume that my switches support (1) modifying a local register (e.g. an integer on a switch) appropriately upon receipt of a packet, and (2) making packet forwarding decisions based on the value of a register. This allows each switch to maintain a local view of the global state. Specifically, the register records the set of events the device knows have occurred. At any time, the device can receive a packet (from the controller or another device) informing it of new event occurrences, which are "unioned" into the local register (by performing a table lookup based on integer values). Currently, P4 data planes support this type of functionality.

I also assume that the switch atomically processes each packet in the order in which it was received. Such "atomic" switch operations are proposed by the "Packet Transactions" P4 extension [110]. Because the P4 switch platform is attracting considerable attention (even spawning its own highly-attended workshop), I feel that my assumptions are realistic for the current state-of-the-art in regards to switches.

Packet Processing. Each packet entering the network is admitted from a host to a port on an edge switch. The configuration ID j corresponding to the device's view of the global state is assigned to the packet's **version number** field. The packet will processed only by j-guarded rules throughout its lifetime. Packets also carry a **digest** encoding the set of events the packet has heard about so far (i.e. the packet's view of the global state). If the packet passes through a device which has heard about additional events, the packet's digest is updated accordingly. Similarly, if the packet's digest contains events not yet heard about by the device, the latter adds them to its view of the state. When a packet triggers an event, that event is immediately added to the packet's digest, as well as to the state of the device where the event was detected. The controller is then notified about the event. Optionally (as an optimization), the controller can

				Packet	pkt	$::=$	$\{f_1; \cdots; f_k; C; digest\}$	*Configuration*	C	$::=$	$\{(lp, lp), \cdots\}$
				Located Packet	lp	$::=$	(pkt, l)	*Enabling Rel.*	$\vdash$	$::=$	$\{(E, e), \cdots\}$
Switch ID	n	$\in$	$\mathbb{N}$	*Queue Map*	qm	$::=$	$\{n \mapsto pkts, \cdots\}$	*Consist. Pred.*	con	$::=$	$\{E, \cdots\}$
Port ID	m	$\in$	$\mathbb{N}$	*Link*	lk	$::=$	(l, l)	*Config. Map*	g	$::=$	$\{E \mapsto C, \cdots\}$
Host ID	h	$\in$	$\mathbb{N}$	*Links*	L	$::=$	$\{lk, \cdots\}$	*Switch*	sw	$::=$	(n, qm, E, qm)
Location	l	$::=$	$n : m$	*Event*	e	$::=$	(φ, l)	*Queue, Control.*	Q, R	$::=$	E
				Event-set	E	$::=$	$\{e, \cdots\}$	*Switches*	S	$::=$	$\{sw, \cdots\}$

$$\frac{(h, n{:}m) \in L \quad S = S' \cup \{(n, qm[m \mapsto pkts], E, qm_2)\}}{(Q, R, S) \to (Q, R, S' \cup \{(n, qm[m \mapsto pkts @ [pkt[C \leftarrow g(E)]]], E, qm_2)\})}\ \textsc{In} \qquad \frac{(n{:}m, h) \in L \quad S = S' \cup \{(n, q_1, E, qm[m \mapsto pkt{::}pkts])\}}{(Q, R, S) \to (Q, R, S' \cup \{(n, q_1, E, qm[m \mapsto pkts])\})}\ \textsc{Out}$$

$$\frac{\begin{array}{c} E' = \{e : (E \cup pkt.digest) \vdash e \wedge con(E \cup pkt.digest \cup \{e\}) \wedge (pkt, n{:}m) \models e\} \\ \{lp : pkt.C((pkt, n{:}m), lp)\} = \{(pkt_1, n{:}m_1), \cdots\} \quad S = S' \cup \{(n, qm[m \mapsto pkt{::}pkts], E, qm_2[m_1 \mapsto pkts_1, \cdots])\} \end{array}}{(Q, R, S) \to (Q \cup E', R, S' \cup \{(n, qm[m \mapsto pkts], E \cup E' \cup pkt.digest, qm_2[m_1 \mapsto pkts_1 @ [pkt_1[digest \leftarrow pkt_1.digest \cup E \cup E']], \cdots])\})}\ \textsc{Switch}$$

$$\frac{(n_1{:}m_1, n_2{:}m_2) \in L \quad S = S' \cup \{(n_1, qm_1, E_1, qm_2[m_1 \mapsto pkt{::}pkts]), (n_2, qm_3[m_2 \mapsto pkts'], E_2, qm_4)\}}{(Q, R, S) \to (Q, R, S' \cup \{(n_1, qm_1, E_1, qm_2[m_1 \mapsto pkts]), (n_2, qm_3[m_2 \mapsto pkts' @ [pkt]], E_2, qm_4)\})}\ \textsc{Link}$$

$$\frac{Q = Q' \cup \{e\}}{(Q, R, S) \to (Q', R \cup \{e\}), S)}\ \textsc{CtrlRecv} \qquad \frac{R = R' \cup \{e\} \quad S = S' \cup \{(n, qm, E, qm_2)\}}{(Q, R, S) \to (Q, R, S' \cup \{(n, qm, E \cup \{e\}, qm_2)\})}\ \textsc{CtrlSend}$$

Figure 3.6: Implemented program semantics.

periodically broadcast its view of the global state to all switches, in order to speed up state dissemination.

3.3.2 Operational Model

I formalize the above via operational semantics for the global behavior of the network as it executes an NES. Each state in Figure 3.6 has the form (Q, R, S), with a controller queue Q, a controller R, and set of switches S. Both the controller queue and controller are a set of events, and initially, $R{=}Q{=}\emptyset$. Each switch $s \in S$ is a tuple $(n, qm_{in}, E, qm_{out})$, where n is the switch ID, qm_{in}, qm_{out} are the input/output queue maps (mapping port IDs to packet queues). Map updates are denoted $qm[m \mapsto pkts]$. The event-set E represents a switch's view of what events have occurred. A packet's digest is denoted $pkt.digest$, and the configuration corresponding to its version number is denoted $pkt.C$. The rules in Figure 3.6 can be summarized as follows.

- IN/OUT: move a packet between a host and edge port.
- SWITCH: process a packet by first adding new events from the packet's digest to the local state, then checking if the packet's arrival matches an event e enabled by the NES and updating the state and packet digest if so, and finally updating the digest with other local events.
- LINK: move a packet across a physical link.
- CTRLRECV: bring an event from the controller queue into the controller.

- CtrlSend: update the local state of the switches.

3.3.3 Correctness of the Implementation

I now prove the correctness of my implementation. Formally, I show that the operational semantics generates correct traces, as defined in Section 3.1.

Lemma 10 (Global Consistency)**.** Given a locally-determined network event structure N, for an execution of the implementation $(Q_1, R_1, S_1)(Q_2, R_2, S_2) \cdots (Q_m, R_m, S_m)$, the event-set $Q_i \cup R_i$ is consistent for all $1 \leq i \leq m$.

Proof Sketch. I first show that if an **inconsistent** set Y where $|Y| > 1$ satisfies the locality restriction (i.e. all of its events are handled at the same switch), then $Y \subseteq R_i \cup Q_i$ is not possible for any i (the Switch rule ensures that multiple events from Y could not have been sent to the controller).

I proceed by induction over m, the trace length, noting that the base case $Q_0 \cup R_0 = \emptyset$ is consistent. Assume that the implementation adds an e (via Switch) to some consistent event-set $Q_m \cup R_m$, producing an inconsistent set. I look at the minimally-inconsistent set $Y \subseteq (Q_m \cup R_m \cup \{e\})$, and notice that the locality restriction requires all events in Y to be detected at the same switch, so by the previous paragraph, I must have $|Y| \leq 1$. This generates a contradiction, since it would mean that either $Y = \{e_0\}$ or $Y \subseteq Q_m \cup R_m$, either of which would make Y consistent. □

Traces of the Implementation. Note that I can readily produce the network trace (Section 3.1) that corresponds to an implementation trace, since a single packet pkt is processed at each step of Figure 3.6. I now present the main result of this section—executions of the implementation correspond to correct network traces (Definition 16).

Theorem 4 (Implementation Correctness)**.** For an NES N, and an execution $(Q_1, R_1, S_1)(Q_2, R_2, S_2) \cdots (Q_m, R_m, S_m)$ of the implementation, corresponding network trace ntr is correct with respect to N.

Proof Sketch. The proof is by induction over the length m of the execution. In the induction step, I show that (1) the Switch rule can only produce consistent event-sets (this follows directly from Lemma 10), and

(2) when the IN rule tags a packet pkt based on the local event-set E, that E consists of exactly the events that happened before pkt arrived (as ordered by the happens-before relation). $\square$

3.4 Implementation and Evaluation

I built a full-featured prototype implementation in OCaml.

- I implemented the compiler described in Section 3.2. This tool accepts a Stateful NetKAT program, and produces the corresponding NES, with a standard NetKAT program representing the configuration at each node. I interface with Frenetic's NetKAT compiler to produce flow-table rules for each of these NetKAT programs.
- I modified the OpenFlow 1.0 reference implementation to support the custom switch/controller needed to realize the runtime described in Section 3.3.
- I built tools to automatically generate custom Mininet scripts to bring up the programmer-specified network topology, using switches/controller running the compiled NES. I can then realistically simulate the whole system using real network traffic.

Research Questions. To evaluate my approach, I sought answers to the following questions.

(1) How useful is my approach? Does it allow programmers to easily write real-world network programs, and get the behavior they want?

(2) What is the performance of my tools (compiler, etc.)?

(3) How much does my correctness guarantee help? For instance, how do the running network programs compare with uncoordinated event-driven strategies?

(4) How efficient are the implementations generated by my approach? For instance, what about message overhead? State-change convergence time? Number of rules used?

I address #1-3 through case studies on real-world examples, and #4 through quantitative performance measurements on simple automatically-generated programs. For the experiments, I assume that the programmer has first confirmed that the program satisfies the conditions allowing proper compilation to an NES, and I assume that the ETS has no loops. My tool could be modified to perform these checks via basic algorithms operating on the ETS, but they have not yet been implemented in the current prototype (as mentioned in

Section 3.2.1, developing efficient algorithms for these checks is left for future work). Experiments were run on an Ubuntu machine with 20GB RAM and a quad-core Intel i5-4570 CPU (3.2 GHz).

To choose a representative set of realistic examples, I first studied the examples addressed in other recent stateful network programming approaches, such as SNAP [5], FlowLog [92], Kinetic [68], NetEgg [125], and FAST [91], and categorized them into three main groups:

- **Protocols/Security**: accessing streaming media across subnets, ARP proxy, **firewall with authentication**, FTP monitoring, **MAC learning**, **stateful firewall**, TCP reassembly, Virtual Machine (VM) provisioning.
- **Measurement/Performance**: heavy hitter detection, **bandwidth cap management (uCap)**, connection affinity in load balancing, counting domains sharing the same IP address, counting IP addresses under the same domain, elephant flows detection, link failure recovery, load balancing, network information base (NIB), QoS in multimedia streaming, rate limiting, sampling based on flow size, Snort flowbits, super spreader detection, tracking flow-size distributions.
- **Monitoring/Filtering**: application detection, DNS amplification mitigation, DNS TTL change tracking, DNS tunnel detection, **intrusion detection system (IDS)**, optimistic ACK attack detection, phishing/spam detection, selective packet dropping, sidejack attack detection, stolen laptop detection, SYN flood detection, UDP flood mitigation, walled garden.

As I will show in the following section, my current prototype system is best suited for writing programs such as the ones in the **Protocols/Security** category, since some of the **Measurement/Performance** programs require timers and/or integer counters, and some of the **Monitoring/Filtering** programs require complex pattern matching of (and table lookups based on) sequences of packets—functionality which I do not (yet) natively support, Thus, I have selected three examples from the first category, and one from each of the latter two, corresponding to the boldface applications in the list. I believe that these applications are representative of the basic types of behaviors seen in the other listed applications.

3.4.1 Case Studies

In the first set of experiments, I compare **correct** behavior (produced by my implementation strategy)

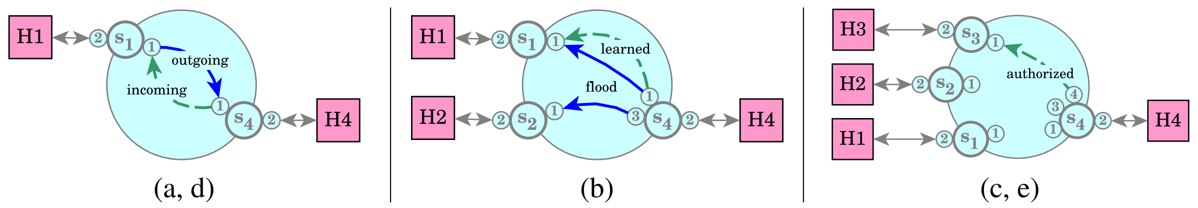


Figure 3.7: Topologies: (a) Firewall, (b) Learning Switch, (c) Authentication, (d) Bandwidth Cap, (e) Intrusion Detection System.

(a)

```
pt=2 ∧ ip_dst=H4; pt←1; (state=[0]; (1:1)→(4:1)→⟨state←[1]⟩ + state≠[0]; (1:1)→(4:1)
    ); pt←2
+ pt=2 ∧ ip_dst=H1; state=[1]; pt←1; (4:1)→(1:1); pt←2
```

(b)

```
pt=2 ∧ ip_dst=H1; (pt←1; (4:1)→(1:1) + state=[0]; pt←3; (4:3)→(2:1)); pt←2
+ pt=2 ∧ ip_dst=H4; pt←1; (1:1)→(4:1)→⟨state←[1]⟩; pt←2
+ pt=2; pt←1; (2:1)→(4:3); pt←2
```

(c)

```
state=[0] ∧ pt=2 ∧ ip_dst=H1; pt←1; (4:1)→(1:1)→⟨state←[1]⟩; pt←2
+ state=[1] ∧ pt=2 ∧ ip_dst=H2; pt←3; (4:3)→(2:1)→⟨state←[2]⟩; pt←2
+ state=[2] ∧ pt=2 ∧ ip_dst=H3; pt←4; (4:4)→(3:1); pt←2
+ pt=2; pt←1; ((1:1)→(4:1) + (2:1)→(4:3) + (3:1)→(4:4)); pt←2
```

(d)

```
pt=2 ∧ ip_dst=H4;
   pt←1; (
      state=[0]; (1:1)→(4:1)→⟨state←[1]⟩
      + state=[1]; (1:1)→(4:1)→⟨state←[2]⟩
      + state=[2]; (1:1)→(4:1)→⟨state←[3]⟩
                  ⋮
      + state=[10]; (1:1)→(4:1)→⟨state←[11]⟩
      + state=[11]; (1:1)→(4:1)
   ); pt←2
+ pt=2 ∧ ip_dst=H1; state≠[11]; pt←1; (4:1)→(1:1); pt←2
```

(e)

```
pt=2 ∧ ip_dst=H1; pt←1; (state=[0]; (4:1)→(1:1)→⟨state←[1]⟩ + state≠[0]; (4:1)→(1:1)
    ); pt←2
+ pt=2 ∧ ip_dst=H2; pt←3; (state=[1]; (4:3)→(2:1)→⟨state←[2]⟩ + state≠[1];
    (4:3)→(2:1)); pt←2
+ pt=2 ∧ ip_dst=H3; pt←4; state≠[2]; (4:4)→(3:1); pt←2
+ pt=2; pt←1; ((1:1)→(4:1) + (2:1)→(4:3) + (3:1)→(4:4)); pt←2
```

Figure 3.8: Programs: (a) Firewall, (b) Learning Switch, (c) Authentication, (d) Bandwidth Cap, (e) Intrusion Detection System.

with that of an **uncoordinated** update strategy. I simulate an uncoordinated strategy in the following way: events are sent to the controller, which pushes updates to the switches (in an unpredictable order) after a few-seconds time delay. I believe this delay is reasonable because heavily using the controller and frequently updating switches can lead to delays between operations of several seconds in practice (e.g. [61] reports up to 10s for a single switch update).

To show that problems still arise for smaller delays, in the firewall experiment described next, I varied the time delay in the uncoordinated strategy between 0ms and 5000ms (in increments of 100ms), running

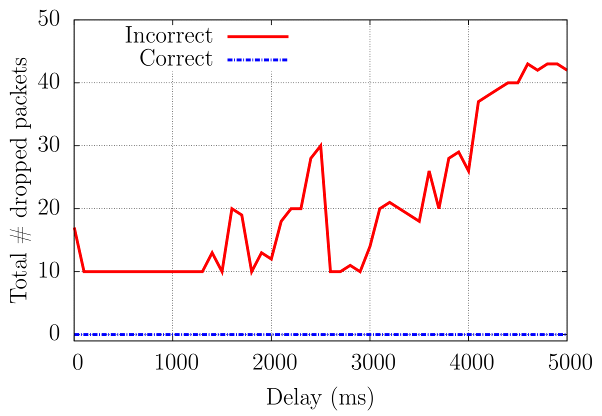


Figure 3.9: Stateful Firewall: impact of delay.

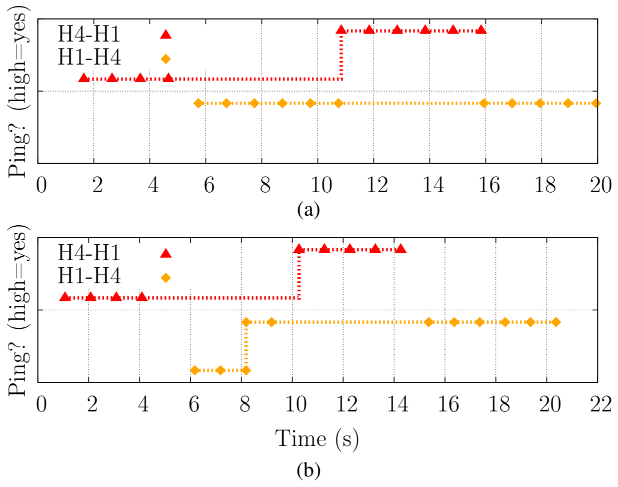


Figure 3.10: Stateful Firewall: (a) correct vs. (b) incorrect.

the experiment 10 times for each. I then plotted the total number of incorrectly-dropped packets with respect to delay. The results are shown in Figure 3.9. Note that even with a very small delay, the uncoordinated strategy still always drops at least one packet.

Stateful Firewall. The example in Figures 3.7-3.8(a) is a simplified stateful firewall. It always allows "outgoing" traffic (from H1 to H4), but only allows "incoming" traffic (from H4 to H1) after the outside network has been contacted, i.e. "outgoing" traffic has been forwarded to H4.

Program p corresponds to configurations $C_{[0]} = [\![p]\!]_{[0]}$ and $C_{[1]} = [\![p]\!]_{[1]}$. In the former, only outgoing traffic is allowed, and in the latter, both outgoing and incoming are allowed. The ETS has the form $\{\langle[0]\rangle \xrightarrow{(dst=H4,\,4:1)} \langle[1]\rangle\}$. The NES has the form $\{E_0{=}\emptyset \to E_1{=}\{(dst{=}H4,\ 4{:}1)\}\}$, where the g is given by $g(E_0) = C_{[0]}$, $g(E_1) = C_{[1]}$.

The Stateful Firewall example took 0.013s to compile, and produced a total of 18 flow-table rules. In Figure 3.10(a), I show that the running firewall has the expected behavior. I first try to ping H1 from H4 (the "H4-H1"/red points), which fails. Then I ping H4 from H1 (the "H1-H4"/orange points), which succeeds. Again I try H4-H1, and now this succeeds, since the event-triggered state change occurred.

For the uncoordinated strategy, Figure 3.10(b) shows that some of the H1-H4 pings get dropped (i.e. H1 does not hear back from H4), meaning the state change did not behave as if it was caused immediately upon arrival of a packet at S4.

Learning Switch. The example in Figures 3.7-3.8(b) is a simple learning switch. Traffic from H4

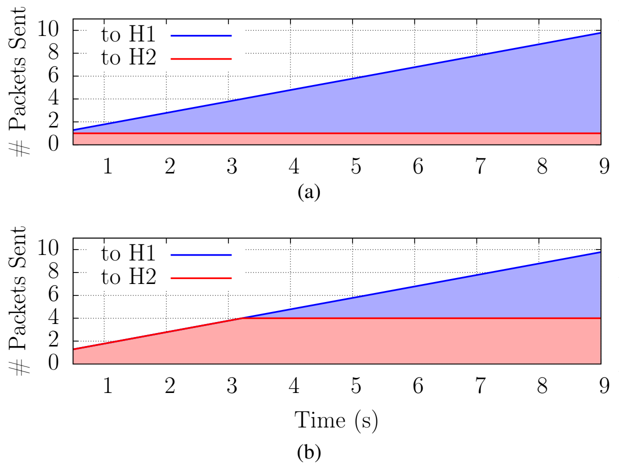


Figure 3.11: Learning Switch: (a) correct vs. (b) incorrect.

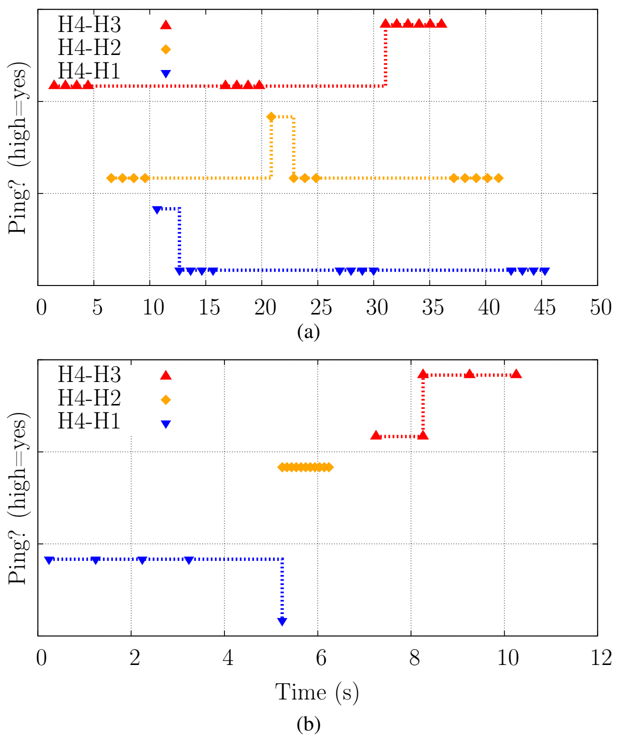


Figure 3.12: Authentication: (a) correct vs. (b) incorrect.

to H1 is flooded (sent to both H1 and H2), until H4 receives a packet from H1, at which point it "learns" the address of H1, and future traffic from H4 to H1 is sent only to H1.

This program p corresponds to two configurations $C_{[0]} = [\![p]\!]_{[0]}$ and $C_{[1]} = [\![p]\!]_{[1]}$. In the former, flooding occurs from H4, and in the latter, packets from H4 are forwarded directly to H1. The ETS has the form $\{\langle[0]\rangle \xrightarrow{(dst=H4,\,4:1)} \langle[1]\rangle\}$. The NES has the form $\{E_0{=}\emptyset \to E_1{=}\{(dst{=}H4,\ 4{:}1)\}\}$, where the g is given by $g(E_0) = C_{[0]}$, $g(E_1) = C_{[1]}$.

This only allows learning for a single host (H1), but I could easily add learning for H2 by using a different index in the vector-valued **state** field: I could replace **state** in Figure 3.8(b) with **state**(0), and union the program (using the NetKAT "$+$" operator) with another instance of Figure 3.8(b) which learns for H2 and uses **state**(1).

The Learning Switch took 0.015s to compile, and produced a total of 43 flow-table rules. I again compare the behavior of my correct implementation with that of an implementation which uses an uncoordinated update strategy. I first ping H1 from H4. Expected behavior is shown in Figure 3.11(a), where the first packet is flooded to both H1 and H2, but then H4 hears a reply from H1, causing the state change (i.e.

learning H1's address), and all subsequent packets are sent only to H1. In Figure 3.11(b), however, since the state change can be delayed, multiple packets are sent to H2, even after H4 has seen a reply from H1.

Authentication. In this example, shown in Figures 3.7-3.8(c), the untrusted host H4 wishes to contact H3, but can only do so after contacting H1 and then H2, in that order.

This program p corresponds to three configurations: $C_{[0]} = [\![p]\!]_{[0]}$ in which only H4-H1 traffic is enabled, $C_{[1]} = [\![p]\!]_{[1]}$ in which only H4-H2 traffic is enabled, and $C_{[2]} = [\![p]\!]_{[2]}$ which finally allows H4 to communicate with H3. The ETS has the form $\{\langle[0]\rangle \xrightarrow{(dst=H1,\,1{:}1)} \langle[1]\rangle \xrightarrow{(dst=H2,\,2{:}1)} \langle[2]\rangle\}$. The NES has the form $\{E_0{=}\emptyset \to E_1{=}\{(dst{=}H1,\ 1{:}1)\} \to E_2{=}\{(dst{=}H1,\ 1{:}1), (dst{=}H2,\ 2{:}1)\}\}$, where the g function is given by $g(E_0) = C_{[0]}$, $g(E_1) = C_{[1]}$, $g(E_2) = C_{[2]}$.

The Authentication example took 0.017s to compile, and produced a total of 72 flow-table rules. In Figure 3.12(a) I demonstrate the correct behavior of the program, by first trying (and failing) to ping H3 and H2 from H4, then successfully pinging H1, again failing to ping H3 (and H1), and finally succeeding in pinging H3. The incorrect (uncoordinated) implementation in Figure 3.12(b) allows an incorrect behavior where I can successfully ping H1 and then H2, but then fail to ping H3 (at least temporarily).

Bandwidth Cap. The Figure 3.7-3.8(d) example is a simplified bandwidth cap implementation. It allows "outgoing" traffic (H1-H4), but only until the limit of n packets has been reached, at which point the service provider replies with a notification message, and disallows the "incoming" path. In this experiment, I use a bandwidth cap of $n = 10$ packets.

Program p corresponds to configurations $C_{[0]}{=}[\![p]\!]_{[0]}, \cdots, C_{[n]}{=}[\![p]\!]_{[n]}$, which all allow incoming/outgoing traffic, and a configuration $C_{[n+1]}{=}[\![p]\!]_{[n+1]}$ which disallows the incoming traffic. The ETS has the form $\{\langle[0]\rangle \xrightarrow{(dst=H4,\,4{:}1)} \langle[1]\rangle \xrightarrow{(dst=H4,\,4{:}1)} \cdots \xrightarrow{(dst=H4,\,4{:}1)} \langle[n+1]\rangle\}$. The NES has the form $\{E_0{=}\emptyset \to E_1{=}\{(dst{=}H4,\ 4{:}1)\} \to \cdots \to E_{n+1}{=}\{(dst{=}H4,\ 4{:}1)_0, \cdots, (dst{=}H4,\ 4{:}1)_n\}\}$, where the g is given by $g(E_0) = C_{[0]}, \cdots, g(E_{n+1}) = C_{[n+1]}$. Note that the subscripts on events in the NES event-sets (e.g. the ones in E_{n+1}) indicate "renamed" copies of the same event (as described in Section 3.2.1).

The Bandwidth Cap example took 0.023s to compile, and produced a total of 158 flow-table rules. In Figure 3.13(a), I show that the running example has the expected behavior. I send pings from H1 to H4, of which exactly 10 succeed, meaning I have reached the bandwidth cap. Using the uncoordinated update

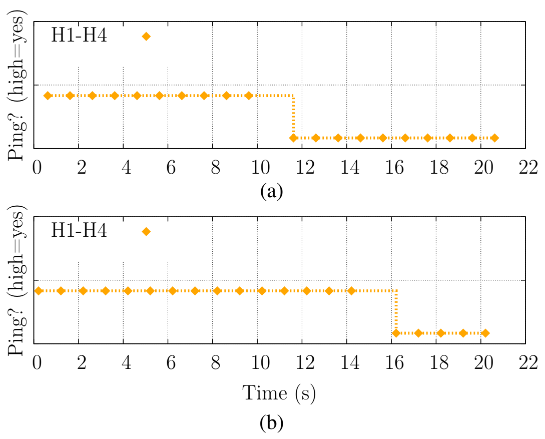


Figure 3.13: Bandwidth Cap: (a) correct vs. (b) incorrect.

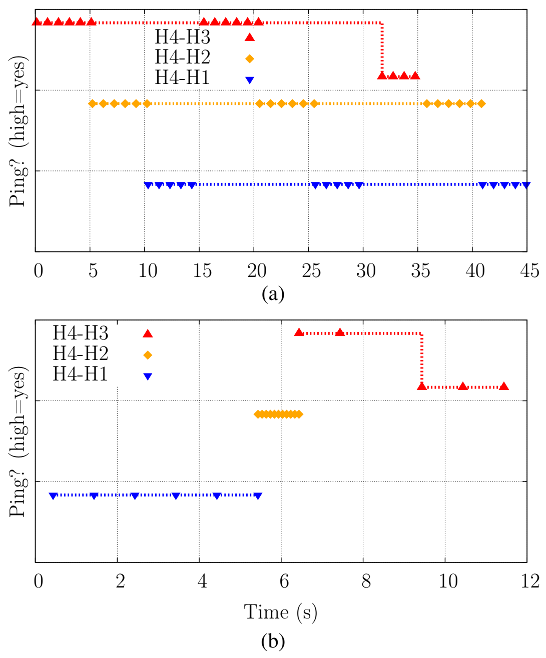


Figure 3.14: Intrusion Detection System: (a) correct vs. (b) incorrect.

strategy in Figure 3.13(b), I again send pings from H1 to H4, but in this case, 15 are successful, exceeding the bandwidth cap.

Intrusion Detection System. In this example, shown in Figures 3.7-3.8(e), the external host H4 is initially free to communicate with the internal hosts H1, H2, and H3. However, if H4 begins engaging in some type of suspicious activity (in this case, beginning to scan through the hosts, e.g. contacting H1 and then H2, in that order), the activity is thwarted (in this case, by cutting off access to H3).

Program p corresponds to three configurations: $C_{[0]} = [\![p]\!]_{[0]}$ and $C_{[1]} = [\![p]\!]_{[1]}$, in which all traffic is enabled, and $C_{[2]} = [\![p]\!]_{[2]}$ in which H4-H3 communication is disabled. The ETS has the form $\{\langle[0]\rangle \xrightarrow{(dst{=}H1,\,1{:}1)} \langle[1]\rangle \xrightarrow{(dst{=}H2,\,2{:}1)} \langle[2]\rangle\}$. The NES has the form $\{E_0{=}\emptyset \rightarrow E_1{=}\{(dst{=}H1,\ 1{:}1)\} \rightarrow E_2{=}\{(dst{=}H1,\ 1{:}1), (dst{=}H2,\ 2{:}1)\}\}$, where g is given by $g(E_0) = C_{[0]}$, $g(E_1) = C_{[1]}$, $g(E_2) = C_{[2]}$.

This IDS example took 0.021s to compile and produced 152 flow-table rules. In Figure 3.14(a), I demonstrate the correct behavior of the program, by first successfully pinging H3, H2, H1, H3, H2, H1 (in that order) from H4. This results in a situation where I have contacted H1 and then H2, causing the third attempt to contact H3 to be blocked (H4-H3 pings dropped). The incorrect (uncoordinated) implementation

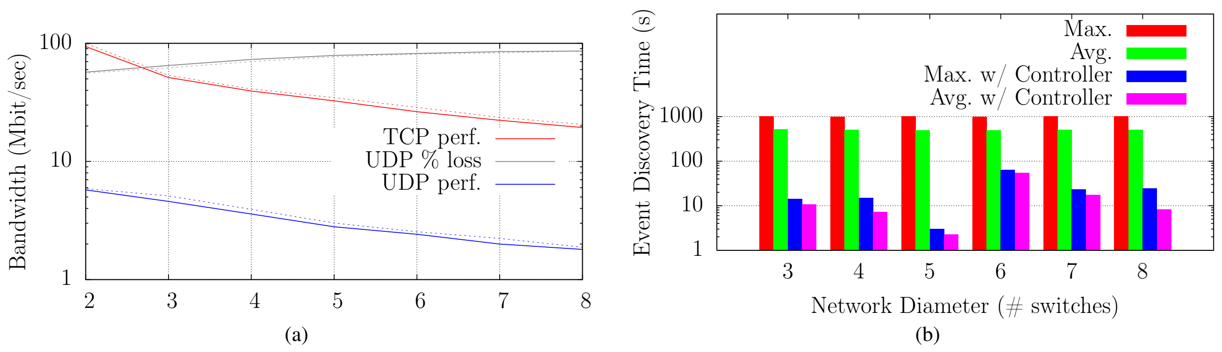


Figure 3.15: Circular Example: (a) bandwidth (solid line is mine, dotted line is reference implementation) and (b) convergence.

in Figure 3.14(b) allows a faulty behavior where I can successfully ping H1 and then H2 (in that order), but subsequent H4-H3 traffic is still enabled temporarily.

3.4.2 Quantitative Results

In this experiment, I automatically generated some event-driven programs which specify that two hosts H1 and H2 are connected to opposite sides of a ring of switches. Initially, traffic is forwarded clockwise, but when a specific switch detects a (packet) event, the configuration changes to forward counterclockwise. I increased the "diameter" of the ring (distance from H1 to H2) up to 8, as shown in Figure 3.15, and performed the following two experiments.

(1) I used `iperf` to measure H1-H2 TCP/UDP bandwidth, and compared the performance of my running event-driven program, versus that of the initial (static) configuration of the program running on un-modified OpenFlow 1.0 reference switches/controller. Figure 3.15(a) shows that my performance (solid line) is very close to the performance of a system which does not do packet tagging, event detection, etc. (dashed line)—I see around 6% performance degradation on average (note that the solid and dashed lines almost coincide).

(2) I measured maximum and average time needed for a switch to learn about the event. The "Max." and "Avg." bars in Figure 3.15(b) are these numbers when the controller does not assist in disseminating events (i.e. only the packet digest is used), and the other columns are the maximum and average when the controller does so.

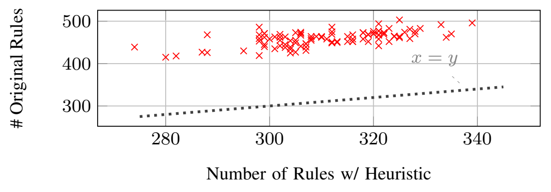


Figure 3.16: Heuristic: reducing the number of rules.

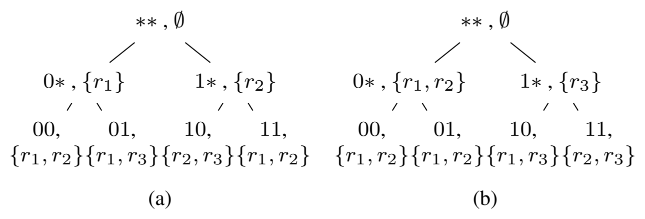


Figure 3.17: Heuristic: two different tries for the same configurations.

3.4.3 Optimizations

When a configuration change occurs, the old and new configurations are often similar, differing only in a subset of flow-table rules. Tables are commonly stored in TCAM memory on switches, which is limited/costly, so it is undesirable to store duplicate rules. As mentioned in Section 3.3.1, each of my rules is guarded by its configuration's numeric ID. If the same rule occurs in several configurations having IDs with the same (binary) high-order bits, intuitively I can reduce space usage by keeping a single copy of the rule, and guarding it with a configuration ID having the shared high-order bits, and **wildcarded** low-order bits. For example, if rule r is used in two different configurations having IDs 2 (binary 10) and 3 (binary 11), I can wildcard the lowest bit $(1*)$, and keep a single rule $(1*)r$ having this wildcarded guard, instead of two copies of r, with the "10" and "11" guards. Ideally, I would like to (re)assign numeric IDs to the configurations, such that maximal sharing of this form is achieved.

I formalize the problem as follows. Assume there is a set of all possible rules $\mathcal{R}$. A configuration C is a subset of these rules $C \subseteq \mathcal{R}$. Assume there are k bits in a configuration ID. Without loss of generality I assume there are exactly 2^k configurations (if there are fewer, I can add dummy configurations, each containing all rules in $\mathcal{R}$). For a given set of configurations, I construct a **trie** having all of the configurations at the leaves. This trie is a complete binary tree in which every node is marked with (1) a wildcarded mask that represents the configuration IDs of its children, and (2) the intersection of the rule-sets of its children.

Consider configurations $C_0 = \{r_1, r_2\}$, $C_1 = \{r_1, r_3\}$, $C_2 = \{r_2, r_3\}$, $C_3 = \{r_1, r_2\}$. Figure 3.17 shows two different assignments of configurations to the leaves of tries. The number of rules for trie (a) is 6: $(0*)r_1$, $(00)r_2$, $(01)r_3$, $(1*)r_2$, $(10)r_3$, $(11)r_1$. The number of rules for trie (b) is 5: $(0*)r_1$, $(0*)r_2$, $(1*)r_3$, $(10)r_1$, $(11)r_2$. Intuitively, this is because the trie (b) has larger sets in the interior. My polynomial heuristic follows that basic intuition: it constructs the trie from the leaves up, at each level pairing nodes in a way that maximizes the sum of the cardinalities of their sets. This does not always produce the global maximum rule sharing, but I find that it produces good results in practice.

As indicated by the Figure 3.16 result (64 randomly-generate configurations w/ 20 rules), on average, rule savings was about 32% of the original number of rules. I also ran this on the previously-discussed Firewall, Learning Switch, Authentication, Bandwidth Cap, and IDS examples, and got rule reductions of $18 \rightarrow 16$, $43 \rightarrow 27$, $72 \rightarrow 46$, $158 \rightarrow 101$, and $152 \rightarrow 133$ respectively.

3.5 Related Work

Network Updates, Verification, and Synthesis. I already briefly mentioned an early approach known as consistent updates [105]. This work was followed by update techniques that respect other correctness properties [79] [61] [126] [85]. These approaches for expressing and verifying correctness of network updates work in terms of **individual** packets.

In event-driven network programs, it is necessary to check properties which describe interactions between **multiple** packets. There are several works which seek to perform network updates in the context of multi-packet properties [44] [76]. There are also proposals for synthesizing SDN controller programs from multi-packet examples [125] and from first-order specifications [98]. Lopes et al. presented techniques for verifying reachability in stateful network programs [78], using a variant of Datalog. This is a complimentary approach which could be used as a basis for verifying reachability properties of my stateful programs.

Network Programming Languages. Network programs can often be constructed using high-level languages. The Frenetic project [42] [89] [41] allows higher-level specification of network policies. Other related projects like Merlin [115] and NetKAT [111] [10] provide high-level languages/tools to compile such programs to network configurations. Works such as Maple [122] and FlowLog [92] seek to address the **dynamic** aspect of network programming.

None of these systems and languages provide both (1) event-based constructs, and (2) strong semantic guarantees about consistency during updates, while my framework enables both. Concurrently with this thesis, an approach called SNAP [5] was developed, which enables event-driven programming, and allows the programmer to ensure consistency via an **atomic** language construct. Their approach offers a more expressive language than my Stateful NetKAT, but in my approach, I enable correct-by-construction event-based behavior and provide a dynamic correctness property, showing (formally) that is strong enough for easy reasoning, yet flexible enough to enable efficient implementations. I also prove the correctness of my implementation technique.

Routing. The consistency/availability trade-off is of interest in routing outside the SDN context as well. In [62], a solution called consensus routing is presented, based on a notion of causality between **triggers** (related to my events). However, the solution is different in many aspects, e.g. it allows a transient phase without safety guarantees.

High-Level Network Functionality. Some recent work has proposed building powerful high-level features into the network itself, such as fabrics [22], intents [96], and other virtualization functionality [70]. Pyretic [90] and projects built on top of it such as PyResonance [67], SDX [51], and Kinetic [68] provide high-level operations on which network programs can be built. These projects do not guarantee consistency during updates, and thus could be profitably combined with an approach such as mine.

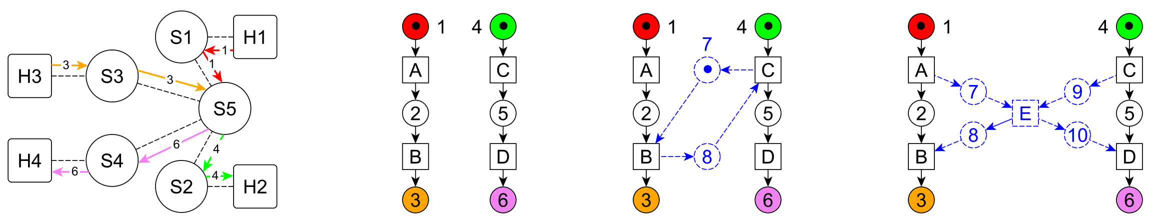
Chapter 4

Synchronization Synthesis for Network Programs

4.1 Network Programming using Event Nets

Network programs change the network's global forwarding behavior in response to events. Recently proposed approaches such as OpenState [13] and Kinetic [68] allow a network program to be specified as a set of finite state machines, where each state is a static configuration (i.e., a set of forwarding rules at switches), and the transitions are driven by network events (packet arrivals, etc.). In this case, support for concurrency is enabled by allowing FSMs to execute in parallel, and any conflicts of the global forwarding state due to concurrency are avoided by either requiring the FSMs to be restricted to **disjoint** types of traffic, or by ignoring conflicts entirely. Neither of these options solves the problem—as I will show here (and in the Evaluation), serious bugs can arise due to unexpected interleavings. Overall, network programming languages typically do not have strong support for handling (and reasoning about) **concurrency**, and this is increasingly important, as SDNs are moving to distributed or multithreaded controllers.

Event Nets for Network Programming. I introduce a new approach which extends the finite-state view of network programming with support for concurrency and synchronization. My model is called **event nets**, an extension of **1-safe Petri nets**, a well-studied framework for concurrency. An event net is a set of **places** (denoted as circles) which are connected via **directed edges** to **events** (denoted as squares). The current state of the program is indicated by a **marking** which assigns at most one **token** to each place, and an event can change the current marking by consuming a token from each of its input places and emitting a token to each of its output places. Since event nets model network programs, each place is labeled with a static network configuration, and at any time, the global configuration is taken as the union of the configurations at the marked places.



(a) Configurations (b) Input Net (c) Iteration 1 (d) Output Net

Figure 4.1: Example #1

Figure 4.1b shows an example event net. I will use integer IDs (and alternatively, colors) to distinguish static configurations. Figure 4.1a shows the network topology corresponding to this example. In a given topology, the configurations associated with the event net are drawn in the color of the **places** which contain them, and also labeled with the corresponding place IDs. For example, place 3 in Figure 4.1b is orange, and this corresponds to enabling forwarding along the orange path $H3, S3, S5$ (labeled with “3”) in the topology shown in Figure 4.1a. In the initial state of this event net, places $1, 4$ contain a token, meaning forwarding is initially enabled along the red (1) and green (4) paths.

Event Nets and Synchronization. Event nets allow me to specify synchronization easily. In Figure 4.1c, I have added places $7, 8$—this makes event C unable to fire initially (since it does not have a token on input place 8), forcing it to wait until event B fires (B consumes a token from places $2, 7$ and emits a token at 8). Ultimately, I will show how these types of **synchronization skeletons** can be produced automatically. In Figure 4.1(b-d), the original event net is shown in black (solid), and synchronization constructs produced by my tool are shown in blue (dashed). I will now demonstrate by example how my tools works.

Example—Tenant Isolation in a Datacenter. Koponen et al. [70] describe an approach for providing **virtual networks** to **tenants** (users) of a datacenter, allowing them to connect virtual machines (VMs) using virtualized networking functionality (middleboxes, etc.). An important aspect is **isolation** between tenants—one tenant intercepting another tenant’s traffic would be a severe security violation.

I will begin by extending the example described in the Introduction. In Figure 4.1a, $S5$ is a physical device initially being used as a virtual middlebox processing Tenant X’s traffic, which is being sent along the red (1) and green (4) paths. I wish to perform an update in the datacenter which allows Tenant Y to use $S5$, and moves the processing of Tenant X’s traffic to a different physical device. For efficiency, I will use two controllers to execute this update—path 1 is taken down and path 3 is brought up by $C1$, and path 4 is taken down and path 6 is brought up by $C2$. The event net for this program is shown in Figure 4.1b. The combinations of configurations $1, 6$ and $4, 3$ both allow traffic to flow between tenants, violating isolation. I can formalize the isolation specification as follows:

(1) ϕ_1: **no packet originating at** $H1$ **should arrive at** $H4$, and

(2) ϕ_2: **no packet originating at** $H3$ **should arrive at** $H2$.

Properties like these which describe **single-packet traces** can be encoded straightforwardly in linear temporal logic (LTL) (note that instead of LTL, I could use the more user-friendly PDL). Given an LTL specification, I ask a **verifier** whether the event net has any reachable marking whose configuration violates the specification. If so, a **counterexample trace** is provided, i.e., a sequence of events (starting from the initial state) which allows the violation. For example, using the specification $\phi_1 \wedge \phi_2$ and the Figure 4.1b event net, my verifier informs me that the sequence of events C, D leads to a property violation—in particular, when the tokens are at places $6, 1$, traffic is allowed along the path $H1, S1, S5, S4, H4$, violating ϕ_1. Next, I ask a **repair engine** to suggest a fix for the event net which disallows the trace C, D, and in this case, my tool produces 4.1c. Again, I call the verifier, which now gives me the counterexample trace A, B (when the tokens are at $4, 3$, traffic is allowed along the path $H3, S3, S5, S2, H2$, violating property ϕ_2). When I ask the repair engine to produce a fix which avoids **both** traces C, D and A, B, I obtain the event net shown in 4.1d. A final call to the verifier confirms that this event net satisfies both properties.

The synchronization skeleton produced in Figure 4.1d functions as a **barrier**—it prevents tokens from arriving at 6 or 3 until **both** tokens have moved from $4, 1$. This ensures that $1, 4$ must **both** be taken down before bringing up paths $3, 6$. The following sections detail this synchronization synthesis approach.

4.2 Synchronization Synthesis for Event Nets

Before describing my synthesis algorithm in detail, I first need to formally define the concepts/terminology mentioned so far.

SDN Preliminaries. A **packet** pkt is a record of fields $\{f_1; f_2; \cdots; f_n\}$, where fields f represent properties such as source and destination address, protocol type, etc. The (numeric) values of fields are accessed via the notation $pkt.f$, and field updates are denoted $pkt[f \leftarrow n]$, where n is a numeric value. A **switch** sw is a node in the network with one or more **ports** pt. A **host** is a switch that can be a source or a sink of packets. A **location** l is a switch-port pair $n{:}m$. Locations may be connected by (bidirectional) physical **links** (l_1, l_2). The graph formed using the locations as nodes and links as edges is referred to as the **topology**. I fix the topology for the remainder of this section.

A **located packet** $lp = (pkt, sw, pt)$ is a packet and a location $sw{:}pt$. A **packet-trace** (**history**) h is a non-empty sequence of located packets. Packet forwarding is dictated by a **network configuration** C. I model C as a relation on located packets: if $C(lp, lp')$, then the network maps lp to lp', possibly changing its location and rewriting some of its fields. Since C is a relation, it allows multiple output packets to be generated from a single input. In a real network, the configuration only forwards packets between ports within each individual switch, but for convenience, I assume that C also captures link behavior (forwarding between switches), i.e. $C((pkt, n_1, m_1), (pkt, n_2, m_2))$ and $C((pkt, n_2, m_2), (pkt, n_1, m_1))$ hold for each link $(n_1{:}m_1, n_2{:}m_2)$. Consider a packet-trace $h = lp_0 lp_1 lp_2 \cdots lp_n$. I say that h is **allowed by** configuration C if and only if $\forall 1 \leq k \leq n.\ C(lp_{k-1}, lp_k)$, and I denote this as $h \in C$. I use $h(i)$ to denote lp_i, i.e., the i-th packet in the trace, and h^i to denote the corresponding suffix of the trace, i.e., $lp_i lp_{i+1} \cdots lp_n$.

Petri Net Preliminaries. As I have shown, a Petri net is a transition system where one or more tokens can move between **places**, as dictated by **transitions**. Petri nets provide a flexible framework for concurrency that I can utilize. For example, the Petri net in Figure 4.2(a) shows how **sequencing** can be modeled—transition a must fire first (moving the token to place 2), before transition b can fire. Figure 4.2(b) shows how **conflict** can be modeled—either c can fire (moving the token to place 5), or d can fire, but not both. Figure 4.2(c) shows how **concurrency** can be modeled—transition e can fire (moving the token from place 7 to place 8), and f can fire independently.

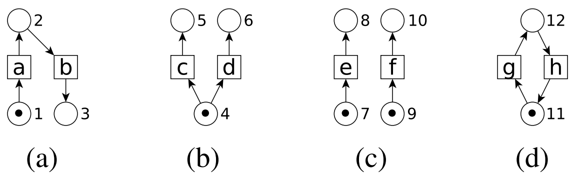


Figure 4.2: Petri nets: (a) sequencing, (b) conflict, (c) concurrency, (d) loops.

My treatment of Petri nets closely follows that of Winskel [123] (Chapter 3). A **Petri net** N is a tuple (P, T, F, M_0), where P is a set of **places** (shown as circles), T is a set of **transitions** (shown as squares), $F \subseteq (P \times T) \cup (T \times P)$ is a set of **directed edges**, and M_0 is multiset of places denoting the **initial marking** (shown as dots on places). For notational convenience, I can view a multiset as a mapping from places to integers, i.e., $M(p)$ denotes the number of times place p appears in multiset M. I require that $P \neq \emptyset$, and $\forall p \in P. \, (M_0(p) > 0 \vee (\exists t \in T. \, ((p, t) \in F \vee (t, p) \in F)))$, and $\forall t \in T. \, \exists p_1, p_2 \in P. \, ((p_1, t) \in F \wedge (t, p_2) \in F)$. Given a transition t, I define its post- and pre-places as $t^\bullet = \{p \in P : (t, p) \in F\}$ and ${}^\bullet t = \{p \in P : (p, t) \in F\}$ respectively. This can be extended in the obvious way to $T'^\bullet$ and ${}^\bullet T'$, for subsets T' of T.

A marking indicates the number of **tokens** at each place. I say that a transition $t \in T$ is **enabled** by a marking M if and only if $\forall p \in P. \, ((p, t) \in F \implies M(p) > 0)$, and I use the notation $T' \subseteq M$ to mean that all $t \in T'$ are enabled by M. A marking M_i can transition into another marking M_{i+1} as dictated by the **firing rule**: $M_i \xrightarrow{T'} M_{i+1} \iff T' \subseteq M_i \wedge M_{i+1} = M_i - {}^\bullet T' + T'^\bullet$, where the $-/+$ operators denote multiset difference/union respectively. The **state graph** of a Petri net is a graph where each node is a marking (the initial node is M_0), and an edge $(M_i \xrightarrow{t} M_j)$ is in the graph if and only if I have $M_i \xrightarrow{\{t\}} M_j$ in the Petri net. A **trace** τ of a Petri net is a sequence $t_0 t_1 \cdots t_n$ such that there exist $M_i \xrightarrow{t_i} M_{i+1}$ in the Petri net's state graph, for all $0 \leq i \leq n$. I define $\mathit{markings}(t_0 t_1 \cdots t_n)$ to be the sequence $M_0 M_1 \cdots M_{n+1}$, where $M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \cdots \xrightarrow{t_n} M_{n+1}$ is in the state graph. I can **project** a trace onto a Petri net (denoted $\tau \rhd N$) by removing any transitions in τ which are not in N. A **1-safe** Petri net is a Petri net in which for any marking M_j reachable from the initial marking M_0, I have $\forall x \in N. \, (0 \leq M_j(x) \leq 1)$, i.e., there is no more than 1 token at each place.

Event Nets. An **event** is a tuple (ψ, l), where l is a location, and ψ can be any predicate over network state, packet locations, etc. For instance, in [86], an event encodes an arrival of a packet with a header matching a given predicate to a given location. A **labeled net** L is a pair (N, λ), where N is a Petri net, and λ labels each place with a network configuration, and each transition with an event. An **event net** is a labeled net (N, λ) where N is 1-safe.

Semantics of Event Nets. Given event net marking M, I denote the **global configuration** of the network $C(M)$, given as $C(M) = \bigcup_{p \in M} \lambda(p)$. Given event net $E = (N, \lambda)$, let $\mathit{Tr}(E)$ be its set of traces (the set of traces of the underlying N). Given trace τ of an event net, I use $\mathit{Configs}(\tau)$ to denote $\{C(M) : M \in \mathit{markings}(\tau)\}$, i.e., the set of global configurations reachable along that trace.

Given event net E and trace τ in $\mathit{Tr}(E)$, I define $\mathit{Traces}(E, \tau)$, the packet traces allowed by τ and E, i.e., $\mathit{Traces}(E, \tau) = \{h : \exists C \in \mathit{Configs}(\tau). (h \in C)\}$. Note that labeling λ is not used here—I could define a more precise semantics by specifying consistency guarantees on how information about event occurrences propagates (as in [86]), but I instead choose an overapproximate semantics, to be independent of the precise definition of events and consistency guarantees.

Distributed Implementations of Event Nets. In general, an implementation of a network program specifies the initial network configuration, and dictates how the configuration changes (e.g., in response to events). I abstract away the details, defining the semantics of an **implemented network program** Pr as the set $W(Pr)$ of **program traces**, each of which is a set of packet traces. A program trace models a full execution, captured as the packet traces exhibited by the network as the program runs. I do not model packet trace interleavings, as this is not needed for the correctness notion I define. I say that Pr **implements** event net E if $\forall tr \in W(Pr). \exists \tau \in \mathit{Tr}(E). (tr \subseteq \mathit{Traces}(E, \tau))$. Intuitively, this means that each program trace can be explained by a trace of the event net E.

I now sketch a **distributed** implementation of event nets, i.e., one in which decisions and state changes are made locally at switches (and not, e.g., at a centralized controller). In order to produce a (distributed) implementation of event net E, I need to solve two issues (both related to the definition of $\mathit{Traces}(E, \tau)$).

First, I ensure that each packet is processed by a single configuration (and not a mixture of several). This is solved by **edge** switches—those where packets enter the network from a host. An edge switch fixes the configuration in which a packet pkt will be processed, and attaches a corresponding tag to pkt.

Second, I must ensure that for each program trace, there exists a trace of E that explains it. The difficulty here stems from the possibility of **distributed conflicts** when the global state changes due to events. For example, in an application where two different switches listen for the same event, and **only the first** switch to detect the event should update the state, I can encounter a conflict where both switches think they are first, and both attempt to update the state. One way to resolve this is by using expensive

$$\begin{array}{rcll}
f & \in & \mathit{Field} & \textbf{(packet field name)} \\
n & \in & \mathbb{N} & \textbf{(numeric value)} \\
x, y & ::= & \mathit{pkt}.f \mid \mathit{pkt}.\textbf{loc} \mid n \mid x + y \mid x - y \mid x * y \mid x \div y & \textbf{(numeric expression)} \\
a & ::= & \textbf{true} \mid \textbf{false} \mid x = y \mid x > y \mid x < y \mid x \geq y \mid x \leq y & \textbf{(atomic proposition)} \\
\varphi, \psi & ::= & a \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \Rightarrow \psi \mid \varphi \Leftrightarrow \psi \mid \mathbf{X}\, \varphi \mid \varphi\, \mathbf{U}\, \psi \mid \varphi\, \mathbf{R}\, \psi \mid \mathbf{G}\, \varphi \mid \mathbf{F}\, \varphi & \textbf{(formula)}
\end{array}$$

Figure 4.3: LTL syntax.

coordination to reach agreement on which was "first." Another way is to use the following constraint. I define **local event net** to be an event net in which for any two events $e_1 = (\psi_1, l_1)$ and $e_2 = (\psi_2, l_2)$, I have $({}^\bullet e_1 \cap {}^\bullet e_2 \neq \emptyset) \Rightarrow (l_1 {=} l_2)$, i.e., events sharing a common input place must be handled at the same location (**local labeled net** can be defined similarly). A local event net can be implemented without expensive coordination [86].

Theorem 5 (Implementability)**.** Given a local event net E, there exists a (distributed) implemented network program that implements E.

The theorem implies that there are no packet races in the implementation, since it guarantees that each packet is never processed in a mix of configurations.

Packet-Trace Specifications. Beyond simply freedom from packet races, I wish to rule out **controller races**, i.e., unwanted interleavings of concurrent events in an event net. In particular, I use LTL to specify formulas that should be satisfied by each packet-trace possible in each global configuration. I use LTL because it is a very natural language for constructing formulas that describe **traces**. For example, if I want to describe traces for which some condition φ **eventually** holds, I can construct the LTL formula **F** φ, and if I want to describe traces where φ holds **at each step (globally)**, I can construct the LTL formula **G** φ.

My LTL formulas are over a single packet pkt, which has a special field $\mathit{pkt.loc}$ denoting its current location. For example, the property $(\mathit{pkt.loc}{=}H_1 \wedge \mathit{pkt.dst}{=}H_2 \implies \mathbf{F}\ \mathit{pkt.loc}{=}H_2)$ means that any packet located at Host 1 destined for Host 2 should eventually reach Host 2. Given a trace τ of an event net, I use $\tau \models \varphi$ to mean that φ holds in each configuration $C \in \mathit{Configs}(\tau)$.

LTL syntax is shown in Figure 4.3. The basic formula is an **atomic proposition**, which is either **true**, **false**, or a comparison between **numeric expressions** over the variable pkt. Formulas can be extended using

$h \models a$	$\triangleq$	$h(0) \models a$	atomic proposition a holds in the first step
$h \models \neg\varphi$	$\triangleq$	$\neg(h \models \varphi)$	φ does not hold
$h \models (\varphi \wedge \psi)$	$\triangleq$	$(h \models \varphi) \wedge (h \models \psi)$	both φ and ψ hold
$h \models (\varphi \vee \psi)$	$\triangleq$	$(h \models \varphi) \vee (h \models \psi)$	either φ or ψ holds
$h \models (\varphi \Rightarrow \psi)$	$\triangleq$	$h \models (\neg\varphi \vee \psi)$	if φ holds, then ψ holds
$h \models (\varphi \Leftrightarrow \psi)$	$\triangleq$	$h \models ((\varphi \Rightarrow \psi) \vee (\psi \Rightarrow \varphi))$	φ holds if and only if ψ holds
$h \models \mathbf{X}\,\varphi$	$\triangleq$	$h^1 \models \varphi$	φ holds at the next step
$h \models \varphi\,\mathbf{U}\,\psi$	$\triangleq$	$\exists i \geq 0 : (h^i \models \psi \wedge (\forall 0 \leq j < i : h^j \models \varphi))$	eventually ψ holds, and φ holds until ψ holds
$h \models \varphi\,\mathbf{R}\,\psi$	$\triangleq$	$\neg(\neg\varphi\,\mathbf{U}\,\neg\psi)$	ψ holds until both φ and ψ hold
$h \models \mathbf{G}\,\varphi$	$\triangleq$	$\mathbf{false}\,\mathbf{R}\,\varphi$	φ always holds
$h \models \mathbf{F}\,\varphi$	$\triangleq$	$\mathbf{true}\,\mathbf{U}\,\varphi$	eventually φ holds

Figure 4.4: LTL semantics.

the standard logical operators negation ($\neg\varphi$), conjunction ($\varphi \wedge \psi$), disjunction ($\varphi \vee \psi$), implication ($\varphi \Rightarrow \psi$), and equality ($\varphi \Leftrightarrow \psi$). Additionally, LTL provides the **next** operator $\mathbf{X}\ \varphi$, the **until** operator $\varphi\ \mathbf{U}\ \psi$, the **release** operator $\varphi\ \mathbf{R}\ \psi$, the **always (globally)** operator $\mathbf{G}\ \varphi$, and the **eventually (future)** operator $\mathbf{F}\ \varphi$. Given a packet-trace h and an LTL formula φ, I define the notion of h satisfying the formula (denoted $h \models \varphi$) using the recursive definition in Figure 4.4. I can extend this to a configurations by letting $C \models \varphi$ mean that all packet-traces $h \in C$ satisfy φ.

Note that the above packet-traces are assumed to be **infinite**, so for the purposes of the definition, I consider a finite trace to be an infinite one where the last step repeats indefinitely.

For efficiency, I forbid the next operator. I have found this restricted form of LTL (usually referred to as **stutter-invariant** LTL) to be sufficient for expressing many properties about network configurations.

Processes and Synchronization Skeletons. The input to my algorithm is a set of disjoint local event nets, which I call **processes**—I can use simple pointwise-union of the tuples (denoted as $\bigsqcup$) to represent this as a single local event net $E = \bigsqcup\{E_1, E_2, \cdots, E_n\}$. Given an event net $E = ((P, T, F, M_0), \lambda)$, a **synchronization skeleton** S for E is a tuple (P', T', F', M_0'), where $P \cap P' = \emptyset$, $T \cap T' = \emptyset$, $F \cap F' = \emptyset$, and $M_0 \cap M_0' = \emptyset$, and where $((P \cup P', T \cup T', F \cup F', M_0 \cup M_0'), \lambda)$ is a labeled net, which I denote $\bigsqcup\{E, S\}$.

Deadlock Freedom and 1-Safety. I want to avoid adding synchronization which fully deadlocks any process E_i. Let $L = \bigsqcup\{E, S\}$ be a labeled net where $E = \bigsqcup\{E_1, E_2, \cdots, E_n\}$, and let P_i, T_i be the places and transitions of each E_i. I say that L is **deadlock-free** if and only if there exists a trace $\tau \in L$

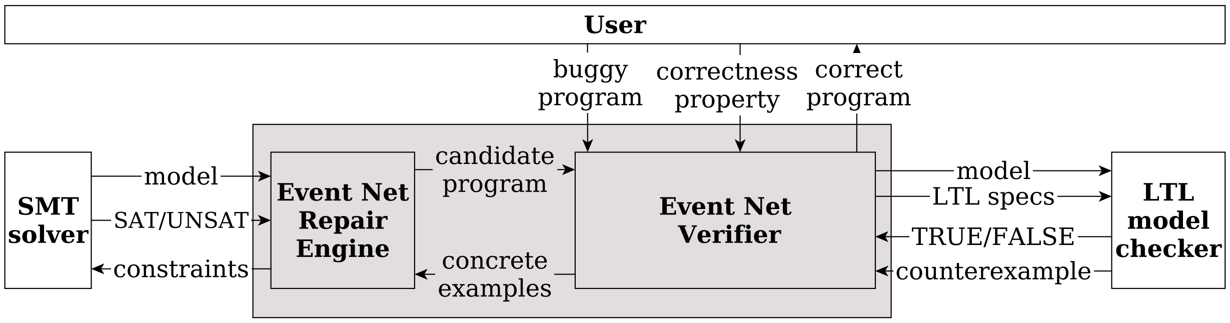


Figure 4.5: Synchronization Synthesis—System Architecture

such that $\forall 0 \leq i \leq n, M_j \in \mathit{markings}(\tau), t \in T_i. (((^{\bullet}t \cap P_i) \subseteq M_j) \Rightarrow (\exists M_k \in \mathit{markings}(\tau). (k \geq j \wedge (t^{\bullet} \cap P_i) \subseteq M_k)))$, i.e. a trace of L where transitions t of each E_i fire as if they experienced no interference from the rest of L. I encode this as an LTL formula, obtaining a **progress** constraint φ_{progr} for E. Similarly, I want to avoid adding synchronization which produces a labeled net that is not 1-safe. I can also encode this as an LTL constraint φ_{1safe}.

Synchronization Synthesis Problem. Given φ and local event net $E = \bigsqcup\{E_1, E_2, \cdots, E_n\}$, find a local labeled net $L = \bigsqcup\{E, S\}$ which **correctly synchronizes** E:

(1) $\forall \tau \in \mathit{Tr}(L). ((\tau \rhd E) \in \mathit{Tr}(E))$, i.e., each τ of L (modulo added events) is a trace of E, and

(2) $\forall \tau \in \mathit{Tr}(L). (\tau \models \varphi)$, i.e., all reachable configurations satisfy φ, and

(3) $\forall \tau \in \mathit{Tr}(L). (\tau \models \varphi_{1safe})$, i.e., L is 1-safe (L is an event net), and

(4) $\exists \tau \in \mathit{Tr}(L). (\tau \models \varphi_{progr})$, i.e., L deadlock-free.

4.3 Fixing and Checking Synchronization in Event Nets

Figure 4.5 shows the architecture of my solution—an instance of the CEGIS algorithm in [50], set up to solve problems of the form $\exists L. ((\forall \tau \in L. (\phi(\tau, E, \varphi, \varphi_{1safe}))) \wedge \neg(\forall \tau \in L. (\tau \not\models \varphi_{progr})))$, where E, L are input/output event nets, and ϕ captures 1-3 of the above specification. My **event net repair engine** (§4.3.1) performs synthesis (producing candidates for $\exists$), and my **event net verifier** (§4.3.2) performs verification (checking $\forall$). Algorithm 4.3.1 shows the pseudocode of my synthesizer. The function *makeProperties* produces the $\varphi_{1safe}, \varphi_{progr}$ formulas discussed in §4.2. The following sections describe the other components of the algorithm.

Algorithm 4.3.1: Synchronization Synthesis Algorithm

```
Input: local event net $E = \bigsqcup\{E_1, E_2, \cdots, E_n\}$, LTL property $\varphi$, upper bound $Y$ on the number of
        added places, upper bound $X$ on the number of added transitions, upper bound $I$ on the
        number of synchronization skeletons
Result: local labeled net $L$ which correctly synchronizes $E$
1  $\mathit{initRepairEngine}(E_1, E_2, \cdots, E_n, X, Y, I)$ ;        // initialize repair engine (§4.3.1)
2  $L \leftarrow E$; $(\varphi_{1safe}, \varphi_{progr}) \leftarrow \mathit{makeProperties}(E_1, E_2, \cdots, E_n)$;
3  while $\mathit{true}$ do
4      $\mathit{ok} \leftarrow \mathit{true}$; $\mathit{props} \leftarrow \{\varphi, \varphi_{1safe}, \varphi_{progr}\}$;
5      for $\varphi' \in \mathit{props}$ do
6          $\tau_{ctex} \leftarrow \mathit{verify}(L, \varphi')$ ;                  // check the property (§4.3.2)
7          if $(\tau_{ctex} = \emptyset \wedge \varphi' = \varphi_{progr}) \vee (\tau_{ctex} \neq \emptyset \wedge \varphi' = \varphi_{1safe})$ then
8              $\mathit{differentRepair}()$; $\mathit{ok} \leftarrow \mathit{false}$ ;          // try different repair (§4.3.1)
9          else if $\tau_{ctex} \neq \emptyset \wedge \varphi' \neq \varphi_{progr}$ then
10             $\mathit{assertCtex}(\tau_{ctex})$; $\mathit{ok} \leftarrow \mathit{false}$ ;        // record counterexample (§4.3.1)
11     if $\mathit{ok}$ then
12         return $L$ ;                    // return correctly-synchronized event net
13     $L \leftarrow \mathit{repair}(L)$ ;                              // generate new candidate
14     if $L = \bot$ then
15         return $\mathit{fail}$ ;                                            // cannot repair
```

4.3.1 Repairing Event Nets Using Counterexample Traces

I use SMT to find synchronization constructs to fix a finite set of bugs (given as unwanted event-net traces). Figure 4.6 shows **synchronization skeletons** which my repair engine adds between processes of the input event net. The **barrier** prevents events b, d from firing until **both** a, c have fired, **condition variable** requires a to fire before c can fire, and **mutex** ensures that events between a and b (inclusive) cannot interleave with the events between c and d (inclusive). My algorithm explores different combinations of these skeletons, up to a given set of bounds.

Repair Engine Initialization. Algorithm 4.3.1 calls *initRepairEngine*, which initializes the function symbols shown in Figure 4.7 and asserts well-formedness constraints. Labels in bold/blue are function symbol names, and cells are the corresponding values. For example, *Petri* is a 2-ary function symbol, and *Loc* is a 1-ary function symbol. Note that there is a separate *Ctex*, *Acc*, *Trans* for each k (where k is a counterexample index, as will be described shortly). The return type (i.e., the type of each cell) is indicated in parentheses after the name of each function symbol. For example, letting $\mathbb{B}$ denote the Boolean

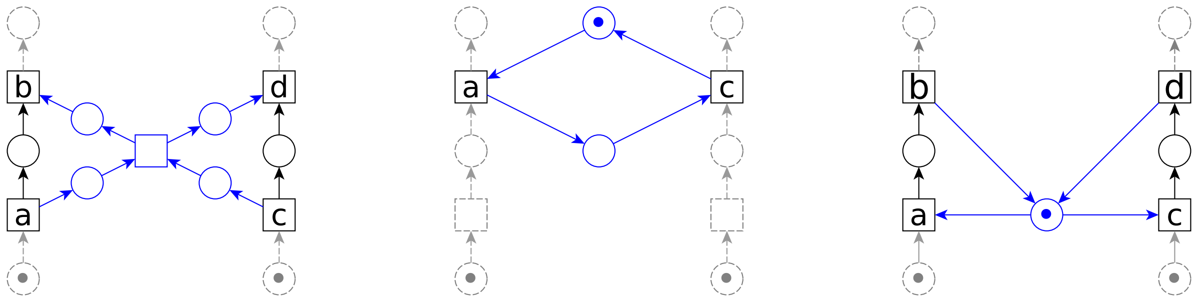


Figure 4.6: Synchronization skeletons: (1) Barrier, (2) Condition Variable, (3) Mutex.

type $\{true, false\}$, the types of the function symbols are: $Petri : \mathbb{N} \times \mathbb{N} \to \mathbb{B} \times \mathbb{B}$, $Mark : \mathbb{N} \to \mathbb{N}$, $Loc : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$, $Type : \mathbb{N} \to \mathbb{N}$, $Pair : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, $Range : \mathbb{N} \to \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, $Len : \mathbb{N} \to \mathbb{N}$, $Ctex_k : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, $Acc_k : \mathbb{N} \to \mathbb{B}$, $Trans_k : \mathbb{N} \to \mathbb{N}$, $Len : \mathbb{N} \to \mathbb{N}$ (note that Len is not shown in the figure).

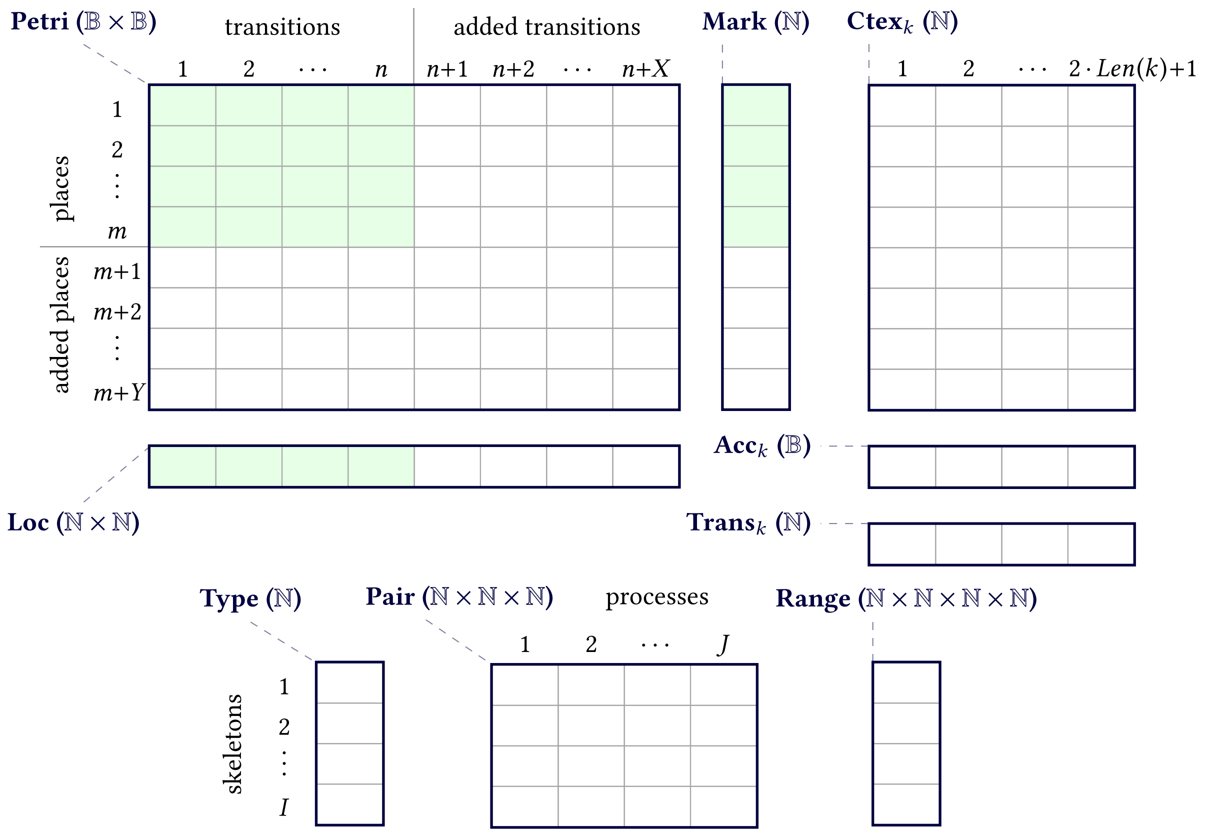


Figure 4.7: SMT function symbols.

The regions highlighted in Figure 4.7 are “set” (asserted equal) to values matching the input event net.

In particular, $Petri(y, x)$ is of the form (b_1, b_2), where I set b_1 if and only if there is an edge from place y to transition x in E, and similarly set b_2 if and only if there is an edge from transition x to place y. $Mark(y)$ is set to 1 if and only if place y is marked in E. $Loc(x)$ is set to the location (switch/port pair) of the event at transition x. The bound Y limits how many places can be added, and X limits how many transitions can be added.

Bound I limits how many skeletons can be used simultaneously. Each "row" i of the $Type, Pair, Range$ symbols represents a single added skeleton. More specifically, $Type(i)$ identifies one of the three types of skeletons. Up to J processes can participate in each skeleton (Figure 4.6 shows the skeletons for 2 processes, but they generalize to $j \geq 2$), and by default, J is set to the number of processes. Thus, $Pair(i, j)$ is a tuple (id, fst, snd), where id identifies a process, and fst, snd is a pair of events in that process. $Range(i)$ is a tuple $(pMin, pMax, tMin, tMax)$, where $pMin, pMax$ reserve a range of rows in the **added places** section of Figure 4.7, and similarly, $tMin, tMax$ reserve a range of columns in the **added transitions**.

I assert a conjunction ϕ_{global} of well-formedness constraints to ensure that proper values are used to fill in the empty (un-highlighted) cells of Figure 4.7. The primary constraint forces the $Petri$ cells to be populated as dictated by any synchronization skeletons appearing in the $Type, Pair, Range$ rows. For example, given a row i where $Type(i) = 1$ (**barrier** synchronization skeleton), I would require that $Range(i) = (y_1, y_2, x_1, x_2)$, where $(y_2 - y_1) + 1 = 4$ and $(x_2 - x_1) + 1 = 1$, meaning 4 new places and 1 new transition would be reserved. Additionally, the values of $Petri$ for rows y_1 through y_2 and columns x_1 through x_2 would be set to match the edges for the **barrier** construct in Figure 4.6. Several other constraints are captured by ϕ_{global}—due to space limitations, I will not present the full details, but the following list summarizes the high-level descriptions of the ϕ_{global} constraints:

(1) For each active cell (id, fst, snd) in $Pair$, I require that fst, snd are from the same input process, and the events between fst and snd (inclusive) in E form a simple chain (i.e., no branching behavior). Additionally, different cells on the same row of $Pair$ are from different processes, i.e., they have different id values.

(2) Cells are in decreasing order of id in each row of $Pair$.

(3) No two active rows of $Pair$ are equal.

(4) No two intervals represented in the $Range$ cells are overlapping.

(5) Each interval in the $Range$ cells stays within the **added places/transitions** area of $Petri$.

(6) Each row of $Type$ is between 0 and 3 (**no skeleton** (inactive row), or one of the 3 skeleton types respectively).

(7) Un-used places/transitions in the **added places/transitions** area of $Petri$ are set to zero.

(8) As described above, interval lengths in $Range$ and corresponding $Petri$/$Mark$ cells are set based on $Type$.

(9) $Mark$ values are between 0 and 1 (enforcing 1-safety).

(10) Two transitions having a common input place have equal corresponding values of Loc (enforcing locality).

(11) Each Loc value is a valid location in the network topology.

Asserting Counterexample Traces. Once the repair engine has been initialized, Algorithm 4.3.1 can add counterexample traces by calling $assertCtex(\tau_{ctex})$. To add the k-th counterexample trace $\tau_k = t_0 t_1 \cdots t_{n-1}$, I assert the conjunction ϕ_k of the following constraints. In essence, these constraints make the columns of $Ctex_k$ correspond to the sequence of markings of the current event net in $Petri$ if it were to fire the sequence of transitions τ_k. Let $Ctex_k(*, x)$ denote the x-th "column" of $Ctex_k$. I define $Ctex_k$ inductively as $Ctex_k(*, 1) = Mark$ and for $x > 1$, $Ctex_k(*, x)$ is equal to the marking that would be obtained if t_{x-2} were to fire in $Ctex_k(*, x-1)$. The symbol Acc_k is similarly defined as $Acc_k(1) = true$ and for $x > 1$, $Acc_k(x) \iff (Acc_k(x-1) \wedge (t_{x-2} \text{ is enabled in } Ctex_k(*, x-1)))$. I also assert a constraint requiring that Acc_k must become false at some point.

An important adjustment must be made to handle **general** counterexamples. Specifically, if a trace of the event net in $Petri$ is equal to τ_k modulo transitions added by the synchronization skeletons, that trace should be rejected just as τ_k would be. I do this by instead considering the trace $\tau'_k = \epsilon\ t_0\ \epsilon\ t_1\ \cdots\ \epsilon\ t_{n-1}$ (where ϵ is a placeholder transition used only for notational convenience), and for the ϵ transitions, I set $Ctex_k(*, x)$ as if I fired any enabled **added transitions** in $Ctex_k(*, x-1)$, and for the t transitions, I update $Ctex_k(*, x)$ as described previously. More specifically, the adjusted constraints ϕ_k are as follows:

(1) $Ctex_k(*, 1) = Mark$.

(2) $\mathit{Len}(k){=}n \land \mathit{Acc}_k(1) \land \neg \mathit{Acc}_k(2 \cdot \mathit{Len}(k) + 1)$.

(3) For $x \geq 2$, $\mathit{Acc}_k(x) \iff (\mathit{Acc}_k(x-1) \land (\mathit{Trans}_k(x){=}\epsilon \lor (\mathit{Trans}_k(x)$ is enabled in $\mathit{Ctex}_k(*, x-1))))$.

(4) For **odd** indices $x \geq 3$, $\mathit{Trans}_k(x) = t_{(x-3)/2}$, and $\mathit{Ctex}_k(*, x)$ is set as if $\mathit{Trans}_k(x)$ fired in $\mathit{Ctex}_k(*, x-1)$.

(5) For **even** indices $x \geq 2$, $\mathit{Trans}_k(x) = \epsilon$, and $\mathit{Ctex}_k(*, x)$ is set as if all enabled **added transitions** fired in $\mathit{Ctex}_k(*, x-1)$.

The last constraint works because for my synchronization skeletons, any added transitions that occur immediately after each other in a trace can also occur in parallel. The negated acceptance constraint $\neg \mathit{Acc}_k(2 \cdot \mathit{Len}(k)+1)$ makes sure that any synchronization generated by the SMT solver will not allow the counterexample trace τ_k to be accepted.

Trying a Different Repair. The $\mathit{differentRepair}()$ function in Algorithm 4.3.1 makes sure the repair engine does not propose the current candidate again. When this is called, I prevent the current set of synchronization skeletons from appearing again by taking the conjunction of the Type and Pair values, as well as the values of Mark corresponding to the places reserved in Range, and asserting the negation. I denote the current set of all such assertions ϕ_{skip}.

Obtaining an Event Net. When the synthesizer calls $\mathit{repair}(L)$, I query the SMT solver for satisfiability of the current constraints. If satisfiable, values of $\mathit{Petri}, \mathit{Mark}$ in the model can be used to add synchronization skeletons to L. I can use **optimizing** functionality of the SMT solver (or a simple loop which asserts progressively smaller bounds for an objective function) to produce a minimal number of synchronization skeletons.

Note that formulas $\phi_{global}, \phi_{skip}, \phi_1, \cdots$ have polynomial size in terms of the input event net size and bounds Y, X, I, J, and are expressed in the decidable fragment `QF_UFLIA` (quantifier-free uninterpreted function symbols and linear integer arithmetic). I found this to scale well with modern SMT solvers (§4.4).

Lemma 11 (Correctness of the Repair Engine)**.** If the SMT solver finds that $\phi = \phi_{global} \land \phi_{skip} \land \phi_1 \land \cdots \land \phi_k$ is satisfiable, then the event net represented by the model does not contain any of the seen counterexample traces $\tau_1, \cdots, \tau_k$. If the SMT solver finds that ϕ is unsatisfiable, then all synchronization skeletons within

Algorithm 4.3.2: Event Net Verifier (PROMELA Model)

```
1  $marked \leftarrow initMarking()$ ;                  // initial marking from input event net
2  **run** $singlePacket, transitions$ ;                        // start both processes
3  **Process** $singlePacket$**:**
4      $lock()$; $status \leftarrow 1$; $pkt \leftarrow pickPacket()$; $n \leftarrow pickHost()$;
5      **do**
6          $pkt \leftarrow movePacket(pkt, marked)$ ;      // move according to current config.
7      **while** $pkt.loc \neq drop \wedge \neg isHost(pkt.loc)$;
8      $status \leftarrow 2$; $unlock()$;
9  **Process** $transitions$**:**
10     **while** $true$ **do**
11         $lock()$;
12         $t \leftarrow pickTransition(marked)$; $marked \leftarrow updateMarking(t, marked)$;
13         $unlock()$;
```

the bounds fail to prevent some counterexample trace.

4.3.2 Checking Event Nets

I now describe $verify(L, \varphi')$ in Algorithm 4.3.1. From L, I produce a PROMELA model for LTL model checking. Algorithm 4.3.2 shows the model pseudocode, which is an efficient implementation of the semantics described in Section 4.2. Variable $marked$ is a list of boolean flags, indicating which places currently contain a token. The $initMarking$ macro sets the initial values based on the initial marking of L. The $singlePacket$ process randomly selects a packet pkt and puts it at a random host, and then moves pkt until it either reaches another host, or is dropped ($pkt.loc = drop$). The $movePacket$ macro modifies/moves pkt according to the current marking's configuration. The $pickTransition$ macro randomly selects a transition $t \in L$, and $updateMarking$ updates the marking to reflect t firing.

I ask the model checker for a **counterexample trace** demonstrating a violation of φ'. This gives the sequence of transitions t chosen by $pickTransition$. I **generalize** this sequence by removing any transitions which are not in the original input event nets. This sequence is returned as τ_{ctex} to Algorithm 4.3.1.

Lemma 12 (Correctness of the Verifier)**.** If the verifier returns counterexample τ, then L violates φ in one of the global configurations in $Configs(\tau)$. If the verifier does not return a counterexample, then all traces of L satisfy φ.

benchmark	#number					time (sec.)				
	switch	iter	ctex	skip	SMT	build	verify	synth	misc	total
ex01-isolation	5	2	2	0	318	0.48	0.43	0.04	0.52	1.47
ex02-conflict	3	10	3	6	349	0.28	0.94	0.61	1.14	2.98
ex03-loop	4	2	1	0	257	0.48	0.43	0.01	0.45	1.37
ex04-composition	4	2	1	0	305	0.48	0.74	0.03	0.50	1.75
ex05-exclusive	3	5	3	3	583	5.17	4.48	0.10	1.00	10.74

Table 4.1: Performance of Examples 1-5.

4.3.3 Overall Correctness Results

The proofs of the following theorems use Lemmas 11, 12, and Theorem 5.

Theorem 6 (Soundness of Algorithm 4.3.1)**.** Given E, φ, if an L is returned, then it is a local labeled net which correctly synchronizes E with respect to φ.

Theorem 7 (Completeness of Algorithm 4.3.1)**.** If there exists a local labeled net $L = \bigsqcup\{E, S\}$, where $|S| \leq I$, and synchronization skeletons in S are each of the form shown in Figure 4.6, and S has fewer than X total transitions and fewer than Y total places, and L correctly synchronizes E, then my algorithm will return such an L. Otherwise, the algorithm returns "$fail$."

4.4 Implementation and Evaluation

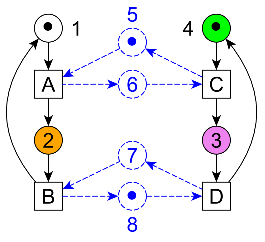
I have implemented a prototype of my synthesizer. The repair engine (§4.3.1) utilizes the Z3 SMT solver, and the verifier (§4.3.2) utilizes the SPIN LTL model checker. In this section, I evaluate my system by addressing the following:

(1) Can I use my approach to model a variety of real-world network programs?

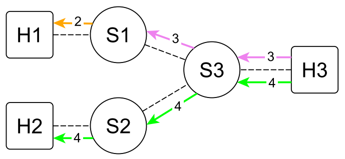
(2) Is my tool able to fix realistic concurrency-related bugs?

(3) Is the performance of my tool reasonable when applied to real networks?

I address #1 and #2 via case studies based on real concurrency bugs described in the networking literature, and #3 by trying increasingly-large topologies for one of the studies. Table 4.1 shows quantitative results for the case studies. The first group of columns denote number of switches (**switch**), CEGIS iterations (**iter**), SPIN counterexamples (**ctex**), event nets "skipped" due to a deadlock-freedom or 1-safety violation (**skip**), and formulas asserted to the SMT solver (**smt**). The remaining columns report runtime of the SPIN verifier



(a) Event net.



(b) Configurations.

Figure 4.8: Inputs for Example #2.

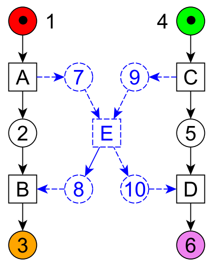
generation/compilation (**build**), SPIN verification (**verify**), repair engine (**synth**), various auxiliary/initialization functionality (**misc**), and overall execution (**total**). My experimental platform had 20GB RAM and a 3.2 GHz 4-core Intel i5-4570 CPU.

Example #1—Tenant Isolation in a Datacenter. I used my tool on the example described in Section 4.1. I formalize the isolation property using the following LTL properties: $\phi_1 \triangleq \mathbf{G}(loc{=}H1 \implies \mathbf{G}(loc{\neq}H4))$ and $\phi_2 \triangleq \mathbf{G}(loc{=}H3 \implies \mathbf{G}(loc{\neq}H2))$. My tool finds the **barrier** in Figure 4.1d, which properly synchronizes the event net to avoid isolation violations, as described in Section 4.1.

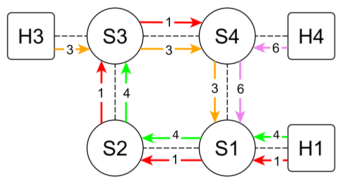
Example #2—Conflicting Controller Modules. In a real bug (El-Hassany et al. [36]) encountered using the POX SDN controller, two concurrent controller modules **Discovery** and **Forwarding** made conflicting assumptions about which forwarding rules should be deleted, resulting in packet loss. Figure 4.8a shows a simplified version of such a scenario, where the left side $(1, A, 2, B)$ corresponds to the Discovery module, and the right side $(4, C, 3, D)$ corresponds to the Forwarding module. In this example, Discovery is responsible for ensuring that packets can be forwarded to H1 (i.e., that the configuration labeled with 2 is active), and Forwarding is responsible for choosing a path for traffic from H3 (either the path labeled 3 or 4). In all cases, I require that traffic from H3 is not dropped.

I formalize this requirement using the LTL property $\phi_3 \triangleq \mathbf{G}(loc{=}H3 \implies \mathbf{G}(loc{\neq}drop))$. My tool finds the **two condition variables** which properly synchronize the event net. As shown in Figure 4.8a, this requires the path corresponding to place 2 to be brought up **before** the path corresponding to place 3 (i.e., event C can only occur after A), and only allows it to be taken down **after** the path 3 is moved back to path 4 (i.e., event B can only occur after D).

Example #3—Discovery Forwarding Loop. In a real bug scenario (Scott et al. [107]), the NOX SDN controller's discovery functionality attempted to learn the network topology, but an unexpected interleaving of packets caused a small forwarding loop to be created. I show how such a forwarding loop can arise due to an unexpected interleaving of controller modules. In Figure 4.9a, the **Forwarding/Discovery** modules are the left/right sides respectively. Initially, **Forwarding** knows about the red (1) path in Figure 4.9b, but will delete these rules, and later set up the orange (3) path. On the other hand, **Discovery** first learns that the green (4) path is going down, and then later learns about the violet (6) path. Since these modules both modify the same forwarding rules, they can create a forwarding loop when configurations $1, 6$ or $4, 3$ are active simultaneously.



(a) Event net.



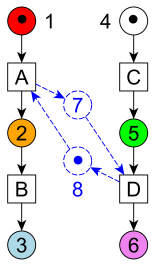
(b) Configurations.

Figure 4.9: Inputs for Example #3.

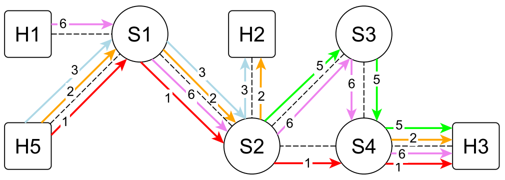
I wish to disallow such loops, formalizing this using the following property: $\phi_4 \triangleq \mathbf{G}(status{=}1 \implies \mathbf{F}(status{=}2))$. As discussed in Section 4.3.2, $status$ is set to 1 when the packet is injected into the network, and set to 2 when/if the packet subsequently exits or is dropped. My tool enforces this by inserting a **barrier** (Figure 4.9a), preventing the unwanted combinations of configurations.

Example #4—Policy Composition. In an update scenario (Canini et al. [20]) involving **overlapping** policies, one policy enforces HTTP traffic monitoring and the other requires traffic from a particular hosts(s) to **waypoint** through a device (e.g., an intrusion detection system or firewall). Problems arise for traffic processed by the **intersection** of these policies (e.g., HTTP packets from a particular host), causing a policy violation.

Figure 4.10b shows such a scenario. The left process of 4.10a is traffic monitoring, and the right process is waypoint enforcement. HTTP traffic is initially enabled along the red (1) path. Traffic monitoring



(a) Event net.



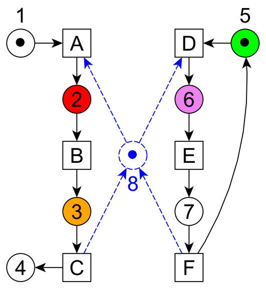
(b) Configurations.

Figure 4.10: Inputs for Example #4.

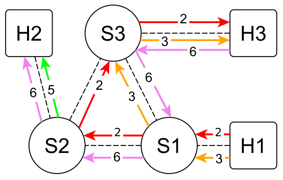
intercepts this traffic and diverts it to $H2$ by setting up the orange (2) path and subsequently bringing it down to form the blue path (3). Waypoint enforcement initially sets up the green path (5) through the waypoint $S3$, and finally allows traffic to enter by setting up the violet (6) path from $H1$. For **HTTP traffic from** $H1$ **destined for** $H3$, if traffic monitoring is not set up **before** waypoint enforcement enables the path from $H1$, this traffic can circumvent the waypoint (on the $S2 \rightarrow S4$ path), violating the policy.

I can encode this specification using the following LTL properties: $\phi_6 \triangleq \mathbf{G}((pkt.type{=}HTTP \wedge pkt.loc{=}H5) \Rightarrow \mathbf{F}(pkt.loc{=}H2 \vee pkt.loc{=}H3))$ and $\phi_7 \triangleq (\neg(pkt.src{=}H1 \wedge pkt.dst{=}H3 \wedge pkt.loc{=}H3)$ $\mathbf{W}$ $(pkt.src{=}H1 \wedge pkt.dst{=}H3 \wedge pkt.loc{=}S3))$, where **W** is **weak until**. My tool finds Figure 4.10a, which forces traffic monitoring to divert traffic **before** waypoint enforcement proceeds.

Example #5—Topology Changes during Update. Peresíni et al. [99] describe a scenario in which a controller attempts to set up forwarding rules, and concurrently the topology changes, resulting in a forwarding loop being installed. Figure 4.11b, examines a similar situation where the processes in Figure 4.11a



(a) Event net.



(b) Configurations.

Figure 4.11: Inputs for Example #5.

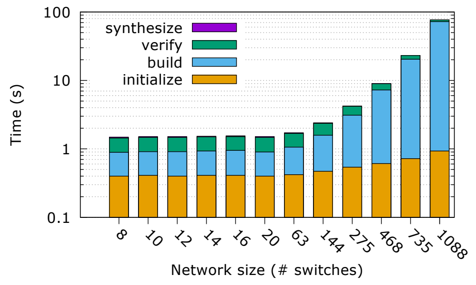


Figure 4.12: Performance results: scalability of Example #1 using Fat Tree topology.

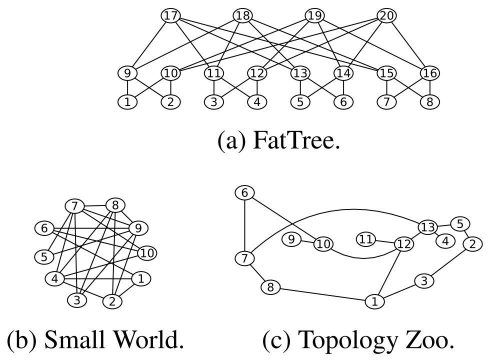


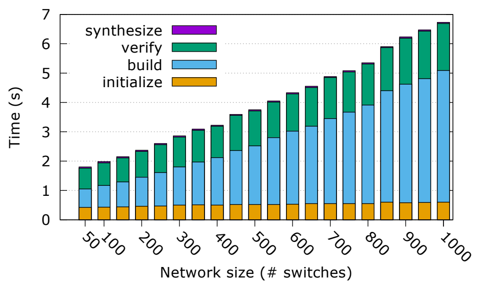
Figure 4.13: Example network topologies.

interleave improperly, resulting in a forwarding loop. The left process updates from the red (2) to the orange (3) path, and the right process extends the green (5) to the violet (6) path (potential forwarding loops: $S1, S3$ and $S1, S2, S3$).

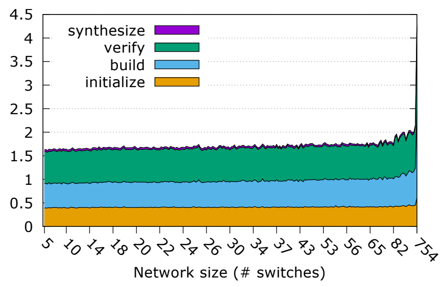
I use the loop-freedom property ϕ_4 from Example #3. My tool finds a **mutex** synchronization skeleton (Figure 4.11a). Note that both places $2, 3$ are protected by the mutex, since either would interact with place 6 to form a loop.

Scalability Experiments.

Recall Example #1 (Figure 4.1a). Instead of the short paths between the pairs of hosts $H1, H2$ and $H3, H4$, I gathered a large set of real network topologies, and randomly selected long host-to-host paths with a single-switch intersection, corresponding to Example #1. I used datacenter FatTree topologies (e.g., Figure 4.13a), scaling up the **depth** (number of layers) and **fanout** (number of links per switch) to achieve a maximum size of 1088 switches, which would support a datacenter with 4096 hosts. I also used highly-connected (“small-world”) graphs, such as the one shown in Figure 4.13b, and I scaled up the number of switches (**ring size** in the Watts-Strogatz model) to 1000. Additionally, I used 240 wide-area network topologies from the Topology Zoo dataset—as an example, Figure 4.13c shows the **NSFNET** topology, featuring physical nodes across the United States. The results of these experiments are shown in Figure 4.12, 4.14a, and 4.14b.



(a) using Small World topologies.



(b) using Topology Zoo topologies.

Figure 4.14: Performance results: scalability of Example #1 (continued).

4.5 Related Work

Network Repair and Network Update Synthesis. Saha et al. [106] and Hojjat et al. [53] present approaches for repairing a buggy network configuration using SMT and a Horn-clause-based synthesis algorithm respectively. Instead of repairing a static configuration, my event net repair engine must repair a network **program**.

A **network update** is a simple network program—a situation where the global forwarding state of the network must change. In the networking community, there are several proposals for packet- and flow-level consistency properties that should be preserved during an update. For example, per-packet and per-flow consistency [105, 80], and inter-flow consistency [76]. Many approaches solve the problem with respect to different variants of these consistency properties [54, 63, 79, 61, 85, 126]. In contrast, I provide a new language for succinctly describing how multiple updates can be composed, as well as an approach for synthesizing a composition which respects customizable LTL properties over packet traces.

Concurrent Programming for Networks. Dudycz et al. [31] present an algorithm to **compose** network updates correctly with respect to loop freedom, and show that the problem of **optimally** doing so is NP-hard. Beyond network updates, there has been work on composing network programs. For example, Pyretic has a programming language which allows sequential/parallel composition of static policies—dynamic behavior can be obtained via a sequence of policies [90]. NetKAT is a mathematical formalism and compiler which also allows composition of static policies [3, 111]. CoVisor is a hypervisor that allows multiple controllers to run concurrently (sequential or parallel composition). It can incrementally update the configuration based on intercepted messages from controllers, and does not need to recompile the full composed policy [60]. The PGA system addresses the issue of how to handle distributed conflicts, via customizable constraints between different portions of the policies, allowing them to be composed correctly [102]. Bonatti et al. [15] present an algebra for properly composing access-control policies. Canini et al. [20] use an approach based on software transactional networking to handle conflicts. I deal with conflicts automatically, by producing **local event nets**.

Handling persistent **state** properly in network programming is a difficult problem. Although basic support is provided by switch-level mechanisms for stateful behavior [16, 13, 110], global coordination still needs to be handled carefully at the language/compiler level. FAST [91], OpenState [13], and Kinetic [68] provide a finite-state-machine-based approach to stateful network programming. Arashloo et al. [5] present SNAP, a high-level language for writing network programs. SNAP has a language with support for sequential/parallel composition of stateful policies, as well as built-in features beyond what I provide (such as atomic blocks). However, none of these approaches examine how to avoid/handle (or even analyze) distributed conflicts. McClurg et al. [86] present an approach which formalizes event-driven network programs using event structures, and show how to deal with distributed conflicts. I extend this to a flexible model which has a more natural notion of loops, while retaining the ability to utilize the consistency properties presented there. I also present a synchronization synthesis framework that helps users properly compose several such structures into a single correct network program.

Synthesis/Verification of Concurrent Network Programs. Padon et al. [98] show how to “decentralize” a network program to work properly on distributed switches. My work on the other hand takes an improperly-decentralized program and inserts the necessary synchronization to make it correct. El-Hassany et al. [36] present SDNracer, a tool for discovering concurrency bugs in network programs. My work instead seeks to repair a buggy concurrent network program to make it satisfy a high-level correctness property. Yuan et al. [125] present NetEgg, pioneering the approach of using examples to write network programs. Similar to my event net repair engine, they produce a policy compatible with a set of finite traces. However, NetEgg does not support negative examples, limiting its ability to rule out incorrect interleavings. Additionally, in contrast with my SMT-based strategy, NetEgg uses a backtracking search which may limit its scalability when applied to large real-world networks.

Petri Net Synthesis. Ehrenfeucht et al. [35] introduce the "net synthesis" problem, i.e., producing a net whose state graph is **isomorphic to a given DFA**, and present the "regions" construction on which Petri net synthesis algorithms are based. Desel et al. [29] present an algorithm for synthesizing **all** nets isomorphic to a given DFA, in order to find "small" ones. Cortadella et al. [27] produce elementary nets, minimize the number of places, and use label splitting when the region-based synthesis method fails. Badouel et al. [7] show that synthesizing **elementary nets** (essentially 1-safe Petri nets without self-loops) is NP-complete.

For **general** Petri nets, the synthesis problem is polynomial-time solvable. Badouel et al. [6] present a polynomial algorithm (based on linear programming) for pure (no self-loops) bounded nets. Badouel et al. [8] present a polynomial-time linear-algebra-based algorithm for synthesizing **distributable nets** [55]. Distributable nets are **local** like my event nets, but not necessarily 1-safe.

The above work is not directly applicable in my context because the definition of "net synthesis" is very different than what is needed for my repair engine. A more closely-related type of synthesis is presented by Bergenthum et al. [11] and Cabasino et al. [19], who synthesize minimal Petri nets consistent with positive and positive/negative examples respectively. My programming model relies on 1-safe Petri nets, so I cannot directly apply these approaches either.

Process Mining. Process mining looks at an **event log** and produces an event structure which generalizes the traces in the log [32]. This approach can also synthesize a Petri net—Ponce de León et al. [101] use the log to produce an event structure, and then generalize the event structure by "folding" it into an equivalent bounded net via SMT, using negative traces to constrain the amount of generalization. This is different than my approach in that (1) their generalization adds potentially **more** behaviors not seen in the positive traces, meaning it would not work for synthesis of synchronization constructs, and (2) they have a strong well-formedness assumption on negative examples, while I allow arbitrary traces.

A related area is **process enhancement** (repair) [39]. This computes a minimal number of changes to the original Petri net such that certain properties are satisfied (such as agreement with the event log). Quality metrics are used to maintain closeness to the original model, and the degree of conformance with the event log. For example, Martínez-Araiza et al. [83] use a backtracking algorithm that modifies the Petri net while checking a CTL property, producing a repaired Petri net which satisfies the property. This changes the semantics of the Petri net, while I want **semantics-preserving transformations**. In other words, I do not generate arbitrary repairs—I **restrict** behaviors by adding new events/places (synchronization skeletons). Basile et al. [9] preserve the semantics of the original (buggy) Petri net, but they are restricted to the context of **time petri nets** (they modify the timing, not the net structure). These do not correspond well to network programs, because careful timing can require expensive synchronization/buffering in the network.

Automata Learning. My approach is essentially an abstract learning framework [77], where my event net repair engine is the learner. Automata learning is conceptually similar, producing a DFA instead of a Petri net, and has been used for verification/synthesis [119]. **Offline** approaches to automata learning (such as RPNI [95]) produce an automaton which agrees with a set of labeled (positive/negative) example traces. **Online** approaches such as L^* [4] actively pose queries to the user asking whether certain traces are contained in the target language. For my purposes, an offline approach is desirable, since I wish to provide a fully automatic tool. Learning a minimal DFA from positive/negative examples is known to be an NP-complete problem [46], but under various restrictions on the example traces, a polynomial algorithm can be obtained [34, 33]. It would be interesting to investigate an RPNI-style formalization for learning Petri nets, although (1) in my case, I would need to **modify a given Petri net in a minimal way, such that a set of negative traces are rejected**, rather than producing a **general** Petri net from a set of positive/negative examples, and (2) it is possible that there is not an efficient solution, due to the NP-completeness of both DFA learning and elementary net synthesis. Additionally, it would be interesting to examine the usefulness of an online approach for learning Petri nets (e.g. [38]) in my context, but both of these directions are left for future work.

Synthesis/Repair for Synchronization. Emerson et al. [37] use a decision procedure for satisfiability of CTL to synthesize "synchronization skeletons." The processes themselves are specified using CTL, and the synchronization skeleton is extracted from the model. Chatterjee et al. [25] present complexity results for distributed LTL synthesis, i.e., synthesizing a set of processes such that their behavior satisfies an LTL specification. My approach is a similar idea, but I exploit the speed of SMT solvers on quantifier-free linear integer arithmetic.

PSketch [114] extends Sketch to synthesize concurrent programs. They add constructs for statement reordering, as well as concurrency primitives for forking threads, atomic sections, etc. The SAT-based synthesis component produces a candidate program which avoids a finite set of buggy traces, and a Spin verifier checks that all interleavings of the candidate are correct, and if not, a counterexample representing a new buggy trace is returned. This is conceptually similar to my approach, but PSketch encodes all possible programs as a SAT formula, while I utilize the SMT solver to **add a repair** to the original program.

There are various other SAT/SMT-based approaches, such as synthesis of memory-order [88] and insertion of fences [71] in a relaxed memory model, instruction reordering [24, 23], atomic section insertion [121, 14], etc. Additionally, there are program-analysis-based approaches which look at a bug report, and perform semantics-preserving reorderings, thread join/lock, etc., producing a **patch** to fix the bug [57, 58, 74]. Raychev et al. [103] present an approach to "determinize" a concurrent program by synthesizing order relationships between statements. In my approach, I model programs as Petri nets, resulting in a general framework for synthesis of synchronization where many different types of synchronization constructs can be readily described and synthesized.

Synthesis from Examples. Programming by examples is an active research area, and has been applied in many contexts where individual behaviors are easier to specify than full programs [49, 100]. My approach uses a new example-based Petri repair engine to synthesize programs which respect certain high-level properties.

Transit [117, 2] allows programmers to synthesize a distributed program (protocol) using both concrete and symbolic partial examples of the program's desired behavior (concolic snippets). This approach uses CEGIS—the synthesizer enumeratively "fills in" program expressions, and uses an SMT solver to check that the resulting candidate protocol agrees with the concolic examples (and invariants). If not, a counterexample is provided as a new concrete example. My approach is similar, except that rather than enumeration, I use an SMT solver (guided by negative traces) to produce a candidate, and my candidates are Petri nets rather than program expressions.

Chapter 5

Data-Plane Mechanisms for Distributed SDN Programming

This chapter describes a general mechanism for implementing the event-driven network programs described in previous chapters. The key features of this mechanism are a high-level language which extends event nets with the ability to access arbitrary global registers, and a compiler that produces corresponding executable code to be run on modern switches. The basic architecture of this system is shown in Figure 5.1.

Before examining these contributions in detail, I give a brief introduction to the P4 switch-programming language, which is currently used in the backend for my compiler. Although I cover P4 specifically, the techniques I use in building my compiler apply to other platforms as well, and other compiler backends could be added relatively easily.

P4 Stateful Data-Plane. The P4 SDN platform provides the ability to store state on the switches using **registers**, which can be read/written with values computed from the header fields of incoming packets. The packet-processing functionality on P4 switches can be customized using the P4 language. This is a "schema"-like language, which allows a sequence of **tables** to be (conditionally) applied to the packet, and the tables must be separately populated with **forwarding rules** to achieve the desired behavior. The switch

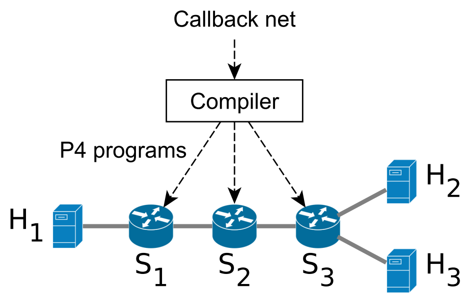


Figure 5.1: System architecture.

is able to achieve line rate by using a **pipelined** architecture which executes the P4 program. At a high level, the switch is structured as an **ingress pipeline**, followed by a queueing mechanism, followed by an **egress pipeline**. Each arriving packet is processed by the ingress pipeline (with special packet **metadata** fields set to indicate which port the packet arrived on), and the P4 program can set metadata fields which tell the subsequent queueing mechanism which output port(s) to send the packet to. The queueing mechanism duplicates the packet if needed, and sends each copy through the egress pipeline, where the P4 program can make additional modifications to the packet (except to the output port, which is now fixed).

My first goal is to make the dataplane programming process more accessible, by moving away from this forwarding-table-based model towards a more familiar imperative programming language.

Data-Plane Intermediate Representation. The IR is a simple imperative programming language used in the intermediate stages of the compiler. Conceptually, the language is similar to P4, in that it provides a way of describing basic packet-processing functions—functions which accept a packet, optionally perform some modifications and/or update local switch registers, and then send the packet to another port(s) on the switch. The main purpose of the IR is to provide a concise and straightforward way of encoding *both* the control flow and the table contents of a P4 program.

For the purposes of experimentation and rapid prototyping, the IR can also specify the desired network topology, and the backend emits a custom Mininet harness which implements it. For example, the following IR code specifies the topology shown in Figure 5.1.

```
let topology = [
  link(S1:2, S2:1),
  link(S2:2, S3:1),
  host(H1, 00:00:00:00:00:11, 10.0.0.1, S1:1),
  host(H2, 00:00:00:00:00:22, 10.0.0.2, S3:2),
  host(H3, 00:00:00:00:00:33, 10.0.0.3, S3:3),
  switch(S1, main),
  switch(S2, main),
  switch(S3, main)
]
```

The full syntax of the IR is shown in Figure 5.2—the syntax and semantics are similar to (a subset

$$
\begin{array}{lcll}
id & \in & \mathit{Ident} & \textbf{(identifier)} \\
n & \in & \mathbb{Z} & \textbf{(numeric constant)} \\
e & ::= & \textbf{true} \mid \textbf{false} \mid n \mid id \mid id(e, \cdots) \quad \mid [e, \cdots] \mid e[e] \mid (e, \cdots) \mid \{id : e, \cdots\} & \textbf{(expression)} \\
 & \mid & e.e \quad \mid e + e \mid e - e \mid e * e \mid -e \mid \{s;\ \cdots\} \mid \textbf{if}(c)\ e\ \textbf{else}\ e & \\
c & ::= & e \neq e \mid e = e \mid e > e \mid e < e \mid \neg c \mid c \wedge c \mid c \vee c & \textbf{(condition)} \\
s & ::= & e \mid \textbf{skip} \mid \textbf{let}\ id = e \mid \textbf{let mut}\ id = e \mid e = e \mid \textbf{push\_output}(id, e, n, id) \mid \textbf{for}(id\ \textbf{in}\ n\ ..\ n)\ s & \textbf{(statement)} \\
m & ::= & \textbf{fn}\ id(id, \cdots)\ s & \textbf{(function)}
\end{array}
$$

Figure 5.2: Intermediate representation (IR) syntax.

of) Rust. In IR programs, the base data are `true` and `false` (of type `bool`), and fixed-width integers of a certain size (e.g., `int32`, `int64`, etc.). Data can be structured into **tuples** such as $(\texttt{false}, 1, 2, \cdots)$, fixed-length **arrays** such as $[1, 2, \cdots]$, and **records** such as $\{\texttt{field1}{:}100, \texttt{field2}{:}200\}$.

Variables can be created using `let` bindings, and later destructively modified using assignment (when created as mutable using `let mut`). There is a `for` loop, where I require that the loop bounds evaluate to a constant at compile time (for compatibility with the bounded programming model provided by P4 and other hardware switches). There is also an `if` statement (standard `else if` syntax is also supported, but is omitted for conciseness). Note that, as in Rust, the IR is an expression-based language, and "return values" are simply the last expression in a block. For example, the following code

```
let a = {
  let x = 1;
  let y = 2;
  x + y
}
```

sets the value of `a` to $1 + 2 = 3$.

A packet is modeled as having a record type, i.e., $\{\texttt{ip\_proto}{:}\texttt{int8}, \texttt{ip\_dst}{:}\texttt{int32}, \cdots, \texttt{data}{:}\cdots\}$, where `ip_proto` etc. are the standard header fields (TCP/IP, etc.), and the `data` field(s) can hold a custom payload (of custom type). These fields are stored in the packet, and are readable/writable at switches. Similarly, a switch is modeled as having a record type, and in this case, the fields represent stateful registers on the switch, which can be read/written when packets arrive. Custom packet headers can also contain bounded **stack** data structures, which provide `push` and `pop` operations, but I elide discussion of this, and instead use arrays for clarity of the presentation.

An ingress function m is associated with each switch:

$$\texttt{ingress}(\texttt{pk}, \texttt{sw}, \texttt{sw\_id}, \texttt{input\_port}, \texttt{clone\_id}, \texttt{is\_edge})$$

(in the above topology declaration, the function `main` is such a callback, associated with each of the three switches). The parameters `pk` and `sw` are the records representing the current packet and switch respectively, `sw_id` and `input_port` identify the current switch and port, and `is_edge` is a flag which is set when the packet has arrived at this port directly from a host (and also when a packet will be delivered directly to a host after leaving this port. This function is applied to each incoming packet pk, and makes modifications to the packet (and also potentially reads from and/or writes to the current switch sw), as well as setting the output ports(s) for the packet, via $\texttt{push\_output}(\texttt{pk}, \texttt{port\_id}, \texttt{unique\_flag}, \texttt{egress})$. Here, `egress` is an egress callback which is called on that copy of the packet before it is transmitted from the switch (the `clone_id` parameter of the callback will be set to the value of `unique_flag`, to distinguish multiple copies of the same packet, if needed). Conceptually, the `push_output` function adds the packet to a stack, and I configure P4's queueing mechanism to send the packet (and any created copies) to the proper output port, where it processed by the egress callback and transmitted.

There is an initialization function for packets

$$\texttt{init\_packet}(\texttt{pkt}, \texttt{swt}, \texttt{swt\_id}, \texttt{input\_port})$$

which is called on each packet newly entering the network from a host. This allows packet header fields to be set to default values. There is also an initialization function for switches $\texttt{init\_switch}(\texttt{swt}, \texttt{swt\_id})$, which is called once per switch, and is used to initialize switch registers to their desired initial values. Unless initialized, all custom header fields and all registers are set to zero.

Typechecking IR Programs. Before attempting to translate an IR program to P4, I perform type checking—this is useful for preventing tricky bugs due to unexpected bit-width conversions, etc. The IR is strongly-typed, and as mentioned, has boolean, fixed-width integer, and array, tuple, and record types. Although not shown in the syntax, values can be affixed with a type annotation, such as `let x = 123 int48`. The integer types can be signed or unsigned, and the width can be an **expression**, as long as it

```
let mut x = [1,2,3,4,5][0];     let mut x = 1;
let mut y = (5,(6,7),8).1.0;    let mut y = 6;
let z = 123;                    let mut w = 128;
let mut w = z+5;                let mut r = 124;
let mut r = 124;                let mut s = r
let mut s = (1,r,3).1
```

Figure 5.3: Constant Propagation

evaluates to a constant integer at compile time, e.g., `let n = 63; let y = 124 int(n+1)`, which gives `y` type `int64`. The typechecking functionality first performs **constant propagation**, eliminating non-constant `let` bindings, and replacing the name with the corresponding value, as shown in Figure 5.3.

A simple bottom-up typechecking algorithm is then employed to confirm that all type annotations are correct. By default, integer constants are taken to be signed 64-bit integers. Explicit type conversions can be performed between integer types, e.g., `let x = 123 int32; let b = (x as int64)`. The compiler makes sure that the proper code is emitted to handle this conversion cleanly (sign extension, etc.). Parameter type annotations are required on function parameters, but the function's return type can be **inferred** by my algorithm.

Callback Nets. In Section 1.3.4, I described callback nets at a high level, and based on that idea, I will formalize the definition. I define a **location** to be a switch-port pair (sw, pt), and an **event** to be a pair (l, φ), where l is a location and φ is a property over packet header fields. In this chapter, for conciseness I will require $\varphi = \mathit{true}$, so that I can identify an event simply by its location (signifying arrival of some packet at that location). I define a **global variable** as a Conflict-free Replicated Data Type (CRDT) register. I define a **callback** to be a function which takes the event-triggering packet and location as parameters, and performs reads/writes to global variables. In this chapter, my callbacks will have the same signature as the ingress/egress functions described previously. Finally, I will define a **callback net** to be an event net where each transition is labeled with a callback.

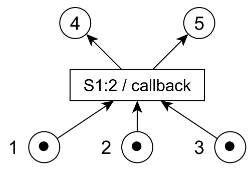


Figure 5.4: Callback net.

An example callback net is shown in Figure 5.4. In this case, the event is `S1:2` (arrival of a packet at port 2 of switch $S1$), and the callback is a function named `callback`. Updates to CRDT registers are propagated **lazily** by piggybacking on data packets. The marking of each place in the callback net also behaves as a CRDT register. For example, the packet that triggers the event in Figure 5.4 will "see" the value of places $1, 2, 3$ decremented and the value of places $4, 5$ incremented, and any switches that subsequently receive the packet will also see these updates.

Compilation: L3 $\rightarrow$ L2. The first stage of the compilation translates callback nets into IR programs with global registers. A callback net is specified along with the topology declaration:

```
let topology = [
  // ... declare topology ...
  // declare callback net:
  event([1,2,3], S1:2, [4,5], callback, 123),
  marking([1,2,3])
]
```

This example defines the simple callback net discussed previously. The places $1, 2, 3$ are initially marked, and a packet arrival at location `S1:2` fires the transition, moving the tokens to places $4, 5$, and calling `callback` with the `clone_id` parameter set to 123. This unique ID can be used to distinguish multiple events using the same callback.

I translate the callback net into global registers as follows. For each place, I generate a single-bit global register. I generate a custom `init_switch` function to initialize these globals to match the specified initial marking. At the beginning of each ingress callback, I read the globals and check the current marking. I then insert a sequence of `if` statements to check whether the current marking and current switch and port match a transition in the callback net. Within the body of each of these, I first insert the statements of the corresponding `callback`, and then insert code to update the globals to match the new marking.

Compilation: L2 $\rightarrow$ L1. The second stage of the compilation translates IR programs with global registers into into IR programs with local registers. Global registers are declared along with the topology declaration. Currently, I support two types of CRDT global registers: **Increment** (unsigned) and **Last-writer wins (LWW)** (signed). Bit-width of the registers can be specified.

```
let topology = [
  // ... declare topology ...
  // declare globals:
  global("counter", 32, "inc"),
  global("test", 64, "lww")
]
```

These global registers are accessed in the following way:

```
// read the counter
let x = counter::read(swt, swt_id);
// increment the counter by 2
counter::inc(swt, swt_id, 2 uint32);

// read the LWW register
let y = test::read(swt, swt_id);
// write 123 to the LWW register
test::write(swt, swt_id, 123 int64)
```

In general, CRDT registers rely on causal ordering, so for this I use Lamport timestamps [72]. I add a new custom header field `time` to packets, and add a new register `time` to each switch. For each global, I store a data structure in the packet header fields, and in registers on each switch. The type of this data structure differs for each type of CRDT register. For example, an increment register is stored as an array of (per-switch) counters, and an LWW register is stored as a register value along with a timestamp [109].

At the beginning of each ingress callback, I insert the following code, where $merge$ is code for the state-based merge for that CRDT type, and max computes the maximum:

```
// update the local timestamp
swt.time = max(swt.time, pkt.time) + 1;
// update local copy of globals
swt.count = merge(swt.count, pkt.count);
swt.test = merge(swt.test, pkt.test)
```

At the end of each egress callback, I insert the following:

```
// send out local timestamp
swt.time = swt.time + 1;
```

```
      let a = {                   let x = 1;
        let x = 1;                let y = 2;
(a)     let y = 2;                let a = x + y
        x + y
      }
-------------------------------------------------------
      let b = if(a > 1) {         let umut t = 0;
          123                     if(a > 1) {
(b)     } else {                    t = 123
          124                     } else {
        }                           t = 124
                                  }
-------------------------------------------------------
      let c =                     let t1 = swt.one;
        swt.one + swt.two;        let t2 = swt.two;
(c)   let d =                     let c =
        pkt.one + pkt.two           t1 + t2;
                                  let d =
                                    pkt.one + pkt.two
```

Figure 5.5: Step 1: Flattening IR statements

```
pkt.time = swt.time;
// send out local copy of globals
pkt.count = merge(swt.count, pkt.count);
pkt.test = merge(swt.test, pkt.test)
```

At this point, I have IR code with only local reads/writes, so I can now proceed to emitting P4 code.

Compilation: L1 → P4. The final stage of the compiler produces P4 code from an IR program. The P4_14 language does not have the expression-level `if` construct, `let` bindings, or data structures like arrays/tuples. Thus, I perform several transformations on the code to simplify it before P4 code generation.

The first step is to flatten all expression-level blocks into statement-level blocks. This is shown in Figure 5.5. In particular, I first eliminate blocks appearing in a `let` binding, by pulling out all statements, and then `let`-binding the final expression (Figure 5.5(a)). I then eliminate `if` expressions appearing in a `let` binding, by introducing a temporary mutable variable, and assigning the final expression in each branch to this variable (Figure 5.5(b)). Finally, I pull reads/writes of **switch** fields out of expressions, so that the appear at the statement level (Figure 5.5(c)). This is because P4 only has statement-level read/write functionality for registers. After applying these transformations, the control-flow in the resulting IR code is implementable using P4's `if` blocks and statements like `register_write` etc.

In order to translate IR data structures into flat integer types which can be handled by P4, I need to

```
     let a =          let a_0_0 = 1;
       [[1,2],        let a_0_1 = 2;
(a)     [3,4],        let a_1_0 = 3;
        [5,6]]        let a_1_1 = 4;
                      let a_2_0 = 5;
                      let a_2_1 = 6
----------------------------------------
     let b =          let b_foo = 123;
       {foo:123,      let b_bar = true;
(b)     bar:true,     let b_baz_0 = 1;
        baz:[1,2]}    let b_baz_1 = 2
```

Figure 5.6: Step 2: Flattening IR assignments/datatypes

perform the transformations shown in Figure 5.6. For example, I recursively flatten arrays into lists of flat integers (Figure 5.6(a)). Similarly, I (recursively) flatten records in a similar way (Figure 5.6(b)), and tuples follow a similar pattern.

The last step of the translation before emitting P4 code involves eliminating `let` bindings, as shown in Figure 5.7. Since P4 does not have support for such a construct, I translate all `let` bindings into packet metadata-field writes. These metadata fields essentially function as "temporary variables" stored in the packet during processing on the switch. They are separate from the packet's custom header fields, and are not transmitted with the packet.

After the Figure 5.5-5.7 transformations are performed, the body of each callback function only contains `if` statements, and straight-line code containing only reads/writes to packet or switch fields. Each field is of a flat integer type. This maps readily into P4 code: the `if` blocks can be emitted directly, and each executable statement (field read/write) can be emitted as a call to `apply(table)`, where `table` is an empty P4 table whose default action is the executable statement. For example, writes to switch fields such as `swt.field = e` become `register_write(field, 0, e)`, reads from switch fields such as `pkt.meta.field = swt.field` become `register_read(routing_metadata.field, field, 0)`, and accesses to custom packet fields such as `pkt.one` become `data.one`, where `data`

```
pkt.one = 1;          pkt.one = 1;
let mut two = 2;      pkt.meta.two = 2;
pkt.one =             pkt.one =
  pkt.one + two;        pkt.one +
                        pkt.meta.two
```

Figure 5.7: Step 3: Flattening IR variables

is a custom P4 header holding all the (now flat integer) custom fields. The IR `push_output` function is implemented by conceptually "pushing" the desired output port, egress callback, and unique ID to a bounded "stack" contained in the packet metadata fields.

A P4 parser is built for the `data` header. The `ingress` block of the P4 program (entrypoint which processes packets from the ingress queue) first checks whether an incoming packet contains this custom header—for simplicity, I indicate this with a special flag in the PCP bits of a VLAN header. If the packet is not flagged, I add the custom header (and the VLAN header, if necessary). I apply tables which set the `sw_id`, `input_port`, and, `is_edge` callback parameters, and then emit the P4 code for the callback's body as described above.

I use the **multicast group** feature of P4's "simple switch" model to make sure that the packet is sent to each of the ports contained in the output-port stack. At the end of the ingress block, I apply tables which match on the contents of the output-port stack, and set the `intrinsic_metadata.mcast_grp` field accordingly. I map a unique multicast group ID to each potential combination of port IDs in the stack. The number of multicast groups is kept low by limiting the size of the stack (multicasting many packets is not common in my applications).

The `egress` block of the P4 program processes each packet after the ingress pipeline has moved it to a specific output port queue (as dictated by the multicast group). I have set up my multicast group assignments such that the multicast mechanism sets `intrinsic_metadata.egress_rid` to correspond to the index of this packet in the output-port stack. Thus, I apply tables which set metadata fields `clone_id` and `callback` to the unique ID and specified egress callback used in the `push_output` call. I then emit an `if` block for each possible egress callback in the IR program, and match on the `callback` ID to determine which one should handle the packet. Finally, the egress queue ends with tables which strip the VLAN and custom `data` header when the packet is being transmitted directly to a host.

Chapter 6

Conclusion and Future Work

In this thesis, I have outlined a new approach to network programming, based on program synthesis. This includes a practical tool for automatically **synthesizing correct network update** sequences from formal specifications. The thesis also presents a full framework for correct event-driven programming. The approach provides a way of rigorously defining **correct event-driven behavior** without the need for specifying logical formulas. Additionally, I presented an approach for **synthesis of synchronization** to produce event-driven programs which satisfy correctness properties when operating in parallel. Finally, I described an **efficient implementation mechanism** for event-driven programs.

There are many promising directions for continuing this research. In particular, I am especially interested in investigating the following.

(1) **Consistently "Updating" a Running Network Program:** I address the problem of performing (event-driven) updates which are correct with respect to customizable properties, but the (dynamic) implementations my techniques produce are meant to "run" in the network indefinitely. I want to investigate ways to update the running dynamic program itself in some consistent way.

(2) **Multi-Packet Properties:** The techniques in this thesis consider only **single-packet properties**, meaning they cannot precisely capture behavior arising from several different interacting packets. Research is needed to determine (1) what is the right formalism (such as LTL) for specifying multi-packet properties, and (2) how my verifiers can be extended to efficiently check such properties.

(3) **Obtaining Event Nets from Controller Applications:** In this thesis, I show how various networking applications can be (manually) written using my event-driven abstractions. However, it

seems likely that event-driven code written in other SDN frameworks (e.g., the FloodLight controller), could be automatically translated to event nets. This would enable existing applications to more easily be migrated to my framework.

(4) **Formalizing the Event Nets Language:** It would be interesting to consider formal reasoning and automated verification for the event nets language, as has been done for, e.g., NetKAT.

(5) **Generality of the Approach:** The event-driven SDN update problem considered in this thesis is an instance of a more general distributed-systems programming problem, namely **how to write correct and efficient programs for distributed systems**. I provide a PL approach (consistency property, programming language, and compiler/runtime) which ensures that the programmer need not reason about interleavings of events and updates for each application, and I show that my consistency model and implementation technique work well in the context of SDN programs, but I do not believe they are limited to that specific arena. The approach could also possibly be extended to other distributed systems in which availability is prioritized, and consistency can be relaxed in a well-defined way, as in my event-driven consistent updates. Example domains include wireless sensor networks or other message-passing systems where the nodes have basic stateful functionality.

Bibliography

[1] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. “Data Center TCP (DCTCP)”. In **SIGCOMM**. 2010, pp. 63–74.

[2] Rajeev Alur, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. “Automatic Completion of Distributed Protocols with Symmetry”. In **CAV** (2015).

[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. “NetKAT: Semantic Foundations for Networks”. In **POPL** (2014).

[4] D. Angluin. “Learning Regular Sets from Queries and Counterexamples”. In **Inf. Comput.** 75.2 (1987), pp. 87–106.

[5] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. “SNAP: Stateful Network-Wide Abstractions for Packet Processing”. In **SIGCOMM** (2016).

[6] Eric Badouel, Luca Bernardinello, and Philippe Darondeau. “Polynomial Algorithms for the Synthesis of Bounded Nets”. In **TAPSOFT**. Vol. 915. Lecture Notes in Computer Science. Springer, 1995, pp. 364–378.

[7] Eric Badouel, Luca Bernardinello, and Philippe Darondeau. “The Synthesis Problem for Elementary Net Systems is NP-Complete”. In **Theor. Comput. Sci.** 186.1-2 (1997), pp. 107–134.

[8] Eric Badouel, Benoît Caillaud, and Philippe Darondeau. “Distributing Finite Automata Through Petri Net Synthesis”. In **Formal Asp. Comput.** 13.6 (2002), pp. 447–470.

[9] F. Basile, P. Chiacchio, and J. Coppola. “Model repair of Time Petri Nets with temporal anomalies”. In **IFAC-PapersOnLine** 48.7 (2015). 5th {IFAC} International Workshop on Dependable Control of Discrete SystemsDCDS 2015, pp. 85–90. ISSN: 2405-8963.

[10] Ryan Beckett, Michael Greenberg, and David Walker. “Temporal NetKAT”. In **PLVNET** (2015).

[11] Robin Bergenthum, Jörg Desel, Robert Lorenz, and Sebastian Mauser. “Synthesis of Petri Nets from Finite Partial Languages”. In **Fundam. Inform.** 88.4 (2008), pp. 437–468.

[12] G. Berry and G. Boudol. “The Chemical Abstract Machine”. In **POPL**. 1990, pp. 81–94.

[13] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. “OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch”. In **ACM SIGCOMM CCR** (2014).

[14] Roderick Bloem, Georg Hofferek, Bettina Könighofer, Robert Könighofer, Simon Ausserlechner, and Raphael Spork. "Synthesis of synchronization using uninterpreted functions". In **FMCAD**. IEEE, 2014, pp. 35–42.

[15] Piero A. Bonatti, Sabrina De Capitani di Vimercati, and Pierangela Samarati. "A modular approach to composing access control policies". In **ACM Conference on Computer and Communications Security**. ACM, 2000, pp. 164–173.

[16] Pat Bosshart et al. "P4: Programming Protocol-independent Packet Processors". In **ACM SIGCOMM CCR** (2014).

[17] A. Bradley. "SAT-Based Model Checking without Unrolling". In **VMCAI**. 2011.

[18] E. Brewer. "Towards robust distributed systems (abstract)". In **PODC** (2000), p. 7.

[19] Maria Paola Cabasino, Alessandro Giua, and Carla Seatzu. "Identification of Petri Nets from Knowledge of Their Language". In **Discrete Event Dynamic Systems** 17.4 (2007), pp. 447–474.

[20] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. "Software transactional networking: concurrent and consistent policy composition". In **HotSDN**. ACM, 2013, pp. 1–6.

[21] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. "Ethane: Taking Control of the Enterprise". In **SIGCOMM** (2007).

[22] Martin Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. "Fabric: A Retrospective on Evolving SDN". In **HotSDN** (2012).

[23] Pavol Cerný, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. "Regression-Free Synthesis for Concurrency". In **CAV**. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 568–584.

[24] Pavol Černý, Thomas A Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. "Efficient Synthesis for Concurrency by Semantics-preserving Transformations". In **CAV** (2013).

[25] Krishnendu Chatterjee, Thomas A Henzinger, Jan Otop, and Andreas Pavlogiannis. "Distributed Synthesis for LTL Fragments". In **Formal Methods in Computer-Aided Design (FMCAD)**. IEEE. 2013, pp. 18–25.

[26] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo. "Incremental Formal Verification of Hardware". In **FMCAD**. 2011, pp. 135–143.

[27] Jordi Cortadella, Michael Kishinevsky, Luciano Lavagno, and Alexandre Yakovlev. "Synthesizing Petri nets from state-based models". In **ICCAD**. IEEE, 1995, pp. 164–171.

[28] "Data Center Outage Costs Continue to Rise". In **Electrical Construction and Maintenance** (2016). URL: `http://ecmweb.com/power-quality/data-center-outage-costs-continue-rise`.

[29] Jörg Desel and Wolfgang Reisig. "The Synthesis Problem of Petri Nets". In **Acta Inf.** 33.4 (1996), pp. 297–315.

[30] Advait Abhay Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman, and Ramana Kompella. “ElastiCon: An Elastic Distributed Sdn Controller”. In **ANCS**. Los Angeles, California, USA, 2014.

[31] Szymon Dudycz, Arne Ludwig, and Stefan Schmid. “Can’t Touch This: Consistent Network Updates for Multiple Policies”. In **DSN**. IEEE Computer Society, 2016, pp. 133–143.

[32] Marlon Dumas and Luciano García-Bañuelos. “Process Mining Reloaded: Event Structures as a Unified Representation of Process Models and Event Logs”. In **Petri Nets**. Vol. 9115. Lecture Notes in Computer Science. Springer, 2015, pp. 33–48.

[33] Pierre Dupont. “Incremental Regular Inference”. In **Grammatical Interference: Learning Syntax from Sentences**. Springer, 1996, pp. 222–237.

[34] Pierre Dupont, Laurent Miclet, and Enrique Vidal. “What is the search space of the regular inference?” In **Grammatical Inference and Applications**. Springer, 1994, pp. 25–37.

[35] Andrzej Ehrenfeucht and Grzegorz Rozenberg. “Partial (Set) 2-Structures. Part II: State Spaces of Concurrent Systems”. In **Acta Inf.** 27.4 (1990), pp. 343–368.

[36] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin T. Vechev. “SDNRacer: concurrency analysis for software-defined networks”. In **PLDI**. ACM, 2016, pp. 402–415.

[37] E. Allen Emerson and Edmund M. Clarke. “Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons”. In **Sci. Comput. Program.** 2.3 (1982), pp. 241–266.

[38] Javier Esparza, Martin Leucker, and Maximilian Schlund. “Learning Workflow Petri Nets”. In **Petri Nets**. Vol. 6128. Lecture Notes in Computer Science. Springer, 2010, pp. 206–225.

[39] Dirk Fahland and Wil M. P. van der Aalst. “Repairing Process Models to Reflect Reality”. In **BPM**. Vol. 7481. Lecture Notes in Computer Science. Springer, 2012, pp. 229–245.

[40] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. “A Scalable, Commodity Data Center Network Architecture”. In **SIGCOMM**. 2008.

[41] Nate Foster et al. “Languages for Software-Defined Networks”. In **Communications Magazine, IEEE** 51.2 (2013), pp. 128–134.

[42] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. “Frenetic: A Network Programming Language”. In **ICFP** (2011).

[43] Pierre Francois, Olivier Bonaventure, Bruno Decraene, and P-A Coste. “Avoiding Disruptions during Maintenance Operations on BGP Sessions”. In **IEEE Transactions on Network and Service Management** 4.3 (2007), pp. 1–11.

[44] Soudeh Ghorbani and Brighten Godfrey. “Towards Correct Network Virtualization”. In **HotSDN** (2014).

[45] S. Gilbert and N. Lynch. “Perspectives on the CAP Theorem”. In **IEEE Computer** 45.2 (2012), pp. 30–36.

[46] E. Mark Gold. “Complexity of Automaton Identification from Given Data”. In **Information and Control** 37.3 (1978), pp. 302–320.

[47] “Google apologizes for cloud outage that one person describes as a ‘comedy of errors’”. In **Business Insider** (2016). URL: `http://www.businessinsider.com/google-apologizes-for-cloud-outage-2016-4`.

[48] Arjun Guha, Mark Reitblatt, and Nate Foster. “ Machine-Verified Network Controllers ”. In **PLDI**. June 2013.

[49] Sumit Gulwani. “Automating String Processing in Spreadsheets Using Input-Output Examples”. In **POPL** (2011).

[50] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. “Synthesis of Loop-free Programs”. In **PLDI** (2011).

[51] Arpit Gupta et al. “SDX: A Software Defined Internet Exchange”. In **SIGCOMM** (2014).

[52] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. “Concurrent Data Representation Synthesis”. In **PLDI**. June 2012, pp. 417–428.

[53] Hossein Hojjat, Philipp Ruemmer, Jedidiah McClurg, Pavol Cerny, and Nate Foster. “Optimizing Horn Solvers for Network Repair”. In **FMCAD** (2016).

[54] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. “Achieving High Utilization with Software-driven WAN”. In **SIGCOMM** (2013).

[55] Richard P. Hopkins. “Distributable nets”. In **Applications and Theory of Petri Nets**. Vol. 524. Lecture Notes in Computer Science. Springer, 1990, pp. 161–187.

[56] Sushant Jain et al. “B4: Experience with a Globally-deployed Software Defined WAN”. In **SIGCOMM**. Hong Kong, China, 2013.

[57] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. “Automated atomicity-violation fixing”. In **PLDI**. ACM, 2011, pp. 389–400.

[58] Guoliang Jin, Wei Zhang, and Dongdong Deng. “Automated Concurrency-Bug Fixing”. In **OSDI**. USENIX Association, 2012, pp. 221–236.

[59] X. Jin, H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. “Dynamic Scheduling of Network Updates”. In **SIGCOMM**. 2014, pp. 539–550.

[60] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. “CoVisor: A Compositional Hypervisor for Software-Defined Networks”. In **NSDI** (2015).

[61] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. “Dynamic Scheduling of Network Updates”. In **SIGCOMM** (2014).

[62] J. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani. “Consensus Routing: The Internet as a Distributed System”. In **NSDI** (2008).

[63] Naga Praveen Katta, Jennifer Rexford, and David Walker. “Incremental Consistent Updates”. In **HotSDN**. ACM. 2013, pp. 49–54.

[64] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. “Real Time Network Policy Checking Using Header Space Analysis”. In. NSDI. 2013, pp. 99–112.

[65] Peyman Kazemian, George Varghese, and Nick McKeown. “Header Space Analysis: Static Checking for Networks”. In **NSDI**. 2012.

[66] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Godfrey. “VeriFlow: Verifying Network-wide Invariants in Real Time”. In **ACM SIGCOMM CCR** (2012).

[67] Hyojoon Kim, Arpit Gupta, Muhammad Shahbaz, Joshua Reich, Nick Feamster, and Russ Clark. **Simpler Network Configuration with State-Based Network Policies**. Tech. rep. Georgia Institute of Technology, 2013.

[68] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. “Kinetic: Verifiable Dynamic Network Control”. In **NSDI** (2015).

[69] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. “The Internet Topology Zoo”. In **IEEE Journal on Selected Areas in Communications** 29.9 (Oct. 2011), pp. 1765–1775.

[70] Teemu Koponen et al. “Network Virtualization in Multi-tenant Datacenters”. In **NSDI** (2014).

[71] Michael Kuperstein, Martin T. Vechev, and Eran Yahav. “Automatic inference of memory fences”. In **FMCAD**. IEEE, 2010, pp. 111–119.

[72] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In **Commun. ACM** 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: http://doi.acm.org/10.1145/359545.359563.

[73] A. Lazaris, D. Tahara, X. Huang, L. Li, A. Voellmy, Y. Yang, and M. Yu. “Tango: Simplifying SDN Programming with Automatic Switch Behavior Inference, Abstraction, and Optimization”. In. 2014.

[74] Haopeng Liu, Yuxi Chen, and Shan Lu. “Understanding and generating high quality patches for concurrency bugs”. In **SIGSOFT FSE**. ACM, 2016, pp. 715–726.

[75] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. “zUpdate: Updating Data Center Networks with Zero Loss”. In **SIGCOMM**. ACM, 2013, pp. 411–422.

[76] Weijie Liu, Rakesh B Bobba, Sibin Mohan, and Roy H Campbell. “Inter-Flow Consistency: Novel SDN Update Abstraction for Supporting Inter-Flow Constraints”. In **NDSS** (2015).

[77] Christof Löding, P. Madhusudan, and Daniel Neider. “Abstract Learning Frameworks for Synthesis”. In **TACAS**. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 167–185.

[78] Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. “Checking Beliefs in Dynamic Networks”. In **NSDI** (2015).

[79] A. Ludwig, M. Rost, D. Foucard, and S. Schmid. "Good Network Updates for Bad Packets: Waypoint Enforcement Beyond Destination-Based Routing Policies". In **HotNets**. 2014.

[80] Ratul Mahajan and Roger Wattenhofer. "On Consistent Updates in Software Defined Networks". In **HotNets**. Nov. 2013.

[81] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Godfrey, and Samuel Talmadge King. "Debugging the Data Plane with Anteater". In **SIGCOMM**. 2011.

[82] R. Majumdar, S. Tetali, and Z. Wang. "Kuai: A Model Checker for Software-defined Networks". In **FMCAD**. 2014.

[83] Ulises Martínez-Araiza and Ernesto López-Mellado. "{CTL} Model Repair for Bounded and Deadlock Free Petri Nets". In **IFAC-PapersOnLine** 48.7 (2015). 5th {IFAC} International Workshop on Dependable Control of Discrete SystemsDCDS 2015, pp. 154–160. ISSN: 2405-8963.

[84] Jedidiah McClurg, Hossein Hojjat, and Pavol Cerny. "Synchronization Synthesis for Network Programs". In **CAV** (2017).

[85] Jedidiah McClurg, Hossein Hojjat, Pavol Cerny, and Nate Foster. "Efficient Synthesis of Network Updates". In **PLDI** (2015).

[86] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Cerny. "Event-driven Network Programming". In **PLDI** (2016).

[87] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: Enabling Innovation in Campus Networks". In **SIGCOMM Computing Communications Review** 38.2 (2008), pp. 69–74.

[88] Yuri Meshman, Noam Rinetzky, and Eran Yahav. "Pattern-based Synthesis of Synchronization for the C++ Memory Model". In **FMCAD**. IEEE, 2015, pp. 120–127.

[89] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. "A Compiler and Run-time System for Network Programming Languages". In **POPL** (2012).

[90] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. "Composing Software Defined Networks". In **NSDI** (2013).

[91] Masoud Moshref, Apoorv Bhargava, Adhip Gupta, Minlan Yu, and Ramesh Govindan. "Flow-level State Transition as a New Switch Primitive for SDN". In **HotSDN**. 2014.

[92] Tim Nelson, Andrew D Ferguson, MJ Scheer, and Shriram Krishnamurthi. "Tierless Programming and Reasoning for Software-Defined Networks". In **NSDI** (2014).

[93] Mark EJ Newman, Steven H Strogatz, and Duncan J Watts. "Random Graphs with Arbitrary Degree Distributions and their Applications". In (2001).

[94] Andrew Noyes, Todd Warszawski, and Nate Foster. "Toward Synthesis of Network Updates". In **SYNT**. July 2013.

[95] José Oncina and Pedro García. “Identifying Regular Languages in Polynomial Time”. In **Advances in Structural and Syntactic Pattern Recognition** (1992).

[96] “ONOS Intent Framework”. In (2014).

[97] Open Networking Foundation. **OpenFlow 1.4 Specification**. 2013.

[98] Oded Padon, Neil Immerman, Aleksandr Karbyshev, Ori Lahav, Mooly Sagiv, and Sharon Shoham. “Decentralizing SDN Policies”. In **POPL** (2015).

[99] Peter Peresíni, Maciej Kuzniar, Nedeljko Vasic, Marco Canini, and Dejan Kostic. “OF.CPP: consistent packet processing for openflow”. In **HotSDN**. ACM, 2013, pp. 97–102.

[100] Oleksandr Polozov and Sumit Gulwani. “FlashMeta: A Framework for Inductive Program Synthesis”. In **OOPSLA** (2015).

[101] Hernán Ponce de León, César Rodríguez, Josep Carmona, Keijo Heljanko, and Stefan Haar. “Unfolding-Based Process Discovery”. In **ATVA**. Vol. 9364. Lecture Notes in Computer Science. Springer, 2015, pp. 31–47.

[102] Chaithan Prakash et al. “PGA: Using Graphs to Express and Automatically Reconcile Network Policies”. In **SIGCOMM**. ACM, 2015, pp. 29–42.

[103] Veselin Raychev, Martin T. Vechev, and Eran Yahav. “Automatic Synthesis of Deterministic Concurrency”. In **SAS**. Vol. 7935. Lecture Notes in Computer Science. Springer, 2013, pp. 283–303.

[104] Saqib Raza, Yuanbo Zhu, and Chen-Nee Chuah. “Graceful Network State Migrations”. In **IEEE/ACM Transactions on Networking** 19.4 (2011), pp. 1097–1110.

[105] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. “Abstractions for Network Update”. In **SIGCOMM** (2012).

[106] Shambwaditya Saha, Santhosh Prabhu, and P. Madhusudan. “NetGen: synthesizing data-plane configurations for network policies”. In **SOSR**. ACM, 2015, 17:1–17:6.

[107] Colin Scott et al. “Troubleshooting blackbox SDN control software with minimal causal sequences”. In **SIGCOMM**. ACM, 2014, pp. 395–406.

[108] Ehab Al-Shaer and Saeed Al-Haj. “FlowChecker: Configuration Analysis and Verification of Federated OpenFlow Infrastructures”. In **SafeConfig**. 2010.

[109] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. **A comprehensive study of Convergent and Commutative Replicated Data Types**. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011, p. 50. URL: `https://hal.inria.fr/inria-00555588`.

[110] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. “Packet Transactions: High-Level Programming for Line-Rate Switches”. In **SIGCOMM** (2016).

[111] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. “A Fast Compiler for NetKAT”. In **ICFP** (2015).

[112] O. Sokolsky and S. Smolka. “Incremental Model Checking in the Modal Mu-Calculus”. In **CAV**. 1994, pp. 351–363.

[113] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. “Sketching Concurrent Data Structures”. In **PLDI**. 2008, pp. 136–148.

[114] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. “Sketching concurrent data structures”. In **PLDI**. ACM, 2008.

[115] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. “Merlin: A Language for Provisioning Network Resources”. In **CoNEXT** (2014).

[116] “Twitter Went Down Because of an ‘Internal Code Change’”. In **Recode** (2016). URL: `http://www.recode.net/2016/1/19/11588920/twitter-went-down-because-of-an-internal-code-change`.

[117] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. “Transit: Specifying Protocols with Concolic Snippets”. In **PLDI** (2013).

[118] Laurent Vanbever, Stefano Vissicchio, Cristel Pelsser, Pierre Francois, and Olivier Bonaventure. “Seamless Network-wide IGP Migrations”. In **SIGCOMM**. 2011.

[119] Abhay Vardhan, Koushik Sen, Mahesh Viswanathan, and Gul Agha. “Learning to Verify Safety Properties”. In **ICFEM**. Vol. 3308. Lecture Notes in Computer Science. Springer, 2004, pp. 274–289.

[120] Moshe Y. Vardi and Pierre Wolper. “An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report)”. In **LICS**. 1986.

[121] Martin Vechev, Eran Yahav, and Greta Yorsh. “Abstraction-guided Synthesis of Synchronization”. In **POPL** (2010).

[122] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. “Maple: Simplifying SDN Programming Using Algorithmic Policies”. In **SIGCOMM** (2013).

[123] Glynn Winskel. **Event Structures**. Springer, 1987.

[124] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. “Reasoning about Infinite Computation Paths (Extended Abstract)”. In **FOCS**. 1983.

[125] Yifei Yuan, Dong Lin, Rajeev Alur, and Boon Thau Loo. “Scenario-based Programming for SDN Policies”. In **CoNEXT** (2015).

[126] Wenxuan Zhou, Dong Jin, Jason Croft, Matthew Caesar, and P. Brighten Godfrey. “Enforcing Generalized Consistency Properties in Software-Defined Networks”. In **NSDI** (2015).