2017

# Extending Provenance For Deep Diagnosis Of Distributed Systems

Yang Wu
*University of Pennsylvania*, wuyangjack1991@gmail.com

# Extending Provenance For Deep Diagnosis Of Distributed Systems

**Abstract**
Diagnosing and repairing problems in complex distributed systems has always been challenging. A wide variety of problems can happen in distributed systems: routers can be misconfigured, nodes can be hacked, and the control software can have bugs. This is further complicated by the complexity and scale of today's distributed systems. Provenance is an attractive way to diagnose faults in distributed systems, because it can track the causality from a symptom to a set of root causes. Prior work on network provenance has successfully applied provenance to distributed systems. However, they cannot explain problems beyond the presence of faulty events and offer limited help with finding repairs.

In this dissertation, we extend provenance to handle diagnostics problems that require deeper investigations. We propose three different extensions: negative provenance explains not just the presence but also the absence of events (such as missing packets); meta provenance can suggest repairs by tracking causality not only for data but also for code (such as bugs in control plane programs); temporal provenance tracks causality at the temporal level and aims at diagnosing timing-related faults (such as slow requests). Compared to classical network provenance, our approach tracks richer causality at runtime and applies more sophisticated reasoning and post-processing. We apply the above techniques to software-defined networking and the border gateway protocol. Evaluations with real world traffic and topology show that our systems can diagnose and repair practical problems, and that the runtime overhead as well as the query turnarounds are reasonable.

**Degree Type**
Dissertation

**Degree Name**
Doctor of Philosophy (PhD)

**Graduate Group**
Computer and Information Science

**First Advisor**
Andreas Haeberlen

**Second Advisor**
Jonathan M. Smith

**Keywords**
Debugging, Diagnostics, Distributed systems, Networks, Provenance

**Subject Categories**
Computer Sciences

# EXTENDING PROVENANCE FOR DEEP DIAGNOSIS OF DISTRIBUTED SYSTEMS

Yang Wu

A DISSERTATION
in
Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2017

Andreas Haeberlen
Associate Professor of Computer and Information Science
Supervisor of Dissertation

Jonathan M. Smith
Olga and Alberico Pompa Professor of Computer and Information Science
Co-Supervisor of Dissertation

Lyle Ungar
Professor of Computer and Information Science
Graduate Group Chairperson

**Dissertation Committee**:
Boon Thau Loo, Professor of Computer and Information Science (Chair)
Zachary G. Ives, Professor of Computer and Information Science
Insup Lee, Cecilia Fitler Moore Professor of Computer and Information Science
Wenchao Zhou, Assistant Professor of Computer Science, Georgetown University

**EXTENDING PROVENANCE FOR DEEP DIAGNOSIS OF DISTRIBUTED SYSTEMS**

## Acknowledgments

First and foremost, I thank my advisor, Andreas Haeberlen. He offered me constant guidance and support. His selfless time and care kept me going forward. I also thank my advisor, Jonathan M. Smith, for his continuous encouragement and wisdom over the years. Without their mentorship, I would not be here.

Besides my advisors, I would like to thank the rest of my dissertation committee, Boon Thau Loo, Zack Ives, Insup Lee, and Wenchao Zhou, for their insightful feedback and valuable questions.

I would like to express special appreciation to Boon and Wenchao. Their mentorship over the years have been instrumental to my journey. My sincere thanks also goes to Linh Thi Xuan Phan for her inspiration and support.

Most of the technical material in this dissertation has been previously published. Negative provenance was published in SIGCOMM 2014 [134] and Meta provenance in NSDI 2017 [133]. This would have not been possible without my collaborators. I feel fortunate to have worked with Andreas Haeberlen, Boon Thau Loo, Wenchao Zhou, and Linh Thi Xuan Phan, as well as, fellow students Ang Chen and Mingchen Zhao. I thank Nate Foster, Vincent Liu, Behnaz Arzani, and Alexander J. T. Gurney for feedback on papers drafts. I thank Antonis Papadimitriou, Arjun Narayan, Hanjun Xiao, Yifei Yuan, Behnaz Arzani, Steven Wu, Wenrui Meng, Shaohui Wang, Meng Xu, Chen Chen, Yang Li, Nikos Vasilakis, Joel Hypolite, Ben Karel, John Sonchack, Max Demoulin, Qizhen Zhang, Raja Sambasivan, and Colin Scott for enlightening discussions.

A special thanks to my parents. Words cannot describe how much they have loved me and taught me. Likewise, I thank my parents-in-law for their care to our family. Last but not least, I would like express appreciation to my beloved wife, Qingxiao Dong, who has always been there for me through good times and bad times. I dedicate this dissertation to her.

# ABSTRACT

# EXTENDING PROVENANCE FOR DEEP DIAGNOSIS OF DISTRIBUTED SYSTEMS

Yang Wu

Andreas Haeberlen

Jonathan M. Smith

Diagnosing and repairing problems in complex distributed systems has always been challenging. A wide variety of problems can happen in distributed systems: routers can be misconfigured, nodes can be hacked, and the control software can have bugs. This is further complicated by the complexity and scale of today's distributed systems. Provenance is an attractive way to diagnose faults in distributed systems, because it can track the causality from a symptom to a set of root causes. Prior work on network provenance has successfully applied provenance to distributed systems. However, they cannot explain problems beyond the presence of faulty events and offer limited help with finding repairs.

In this dissertation, we extend provenance to handle diagnostics problems that require deeper investigations. We propose three different extensions: *negative* provenance explains not just the presence but also the absence of events (such as missing packets); *meta* provenance can suggest repairs by tracking causality not only for data but also for code (such as bugs in control plane programs); *temporal* provenance tracks causality at the temporal level and aims at diagnosing timing-related faults (such as slow requests). Compared to classical network provenance, our approach tracks richer causality at runtime and applies more sophisticated reasoning and post-processing. We apply the above techniques to software-defined networking and the border gateway protocol. Evaluations with real world traffic and topology show that our systems can diagnose and repair practical problems, and that the runtime overhead as well as the query turnarounds are reasonable.

# Contents

# List of Tables

# List of Figures

# 1 Introduction

Finding problems in complex distributed systems is notoriously hard, as the substantial literature on diagnostic tools [39, 58, 66, 89, 68, 120, 95, 125, 10, 27, 19, 42, 29, 142, 4, 76] can attest. A wide variety of problems can happen in distributed systems: routers can be misconfigured [129], nodes can be hacked [40], and the control software can have bugs [118]. Diagnosing problems can be difficult because distributed systems are often constructed from many different hardware and software components, where problems can manifest in subtle ways that have no obvious connection with the original root cause. This is further complicated by the complexity and scale of today's distribute systems – even a campus network can contain as many as hundreds of thousands of routing entries and access control rules [138], commercial data centers can have tens of thousands of devices [14], and the configurations and devices can interact with each other in subtle ways [18, 118].

Furthermore, diagnosing the problem is merely the first step. Once the root cause of a problem is known, the operator must find an effective fix that not only solves the problem at hand, but also avoids creating new problems elsewhere in the network. Given the complexity of modern control software and configuration files, finding a good fix can be as challenging as – or perhaps even more challenging than – diagnostics, and it often requires considerable expertise on the part of the operator.

Recent research in *data provenance* [17] has introduced a promising approach to diagnostics in distributed systems. In essence, provenance tracks causality. It works by linking each event or state to its direct causes, such as the corresponding inputs and the operation that was performed on them. By applying this idea recursively, it is possible to "explain" a particular result by showing its direct causes, and their own causes, and so on, until a set of base inputs is reached. The result is a comprehensive causal explanation of how the result came to exist: it is a tree whose vertexes represent events or states and whose edges represent direct causal relationships. This idea originated in the database community, and it has found many uses such as diagnosing queries [21, 126] and tracking unexpected results [92]. The provenance concept itself is not database-specific: it can broadly help in diagnostic situations where an unexpected behavior must be tracked down to a set of "root causes". Provenance has previously been adapted for distributed systems as *network provenance*, and it has been used, e.g., in a number of debuggers and forensic tools such as ExSPAN [147], DTaP [146], SNP [145], SPP [22], and differential provenance [24].

However, prior work has considered network provenance only in terms of data causality between events that did occur. While this has already been useful in explaining the presence of unexpected events (or configurations), there are important classes of problems that it cannot handle.

- **Missing events:** The problem is not the presence of an unexpected event, but rather the absence of an expected event, for instance: a certain server is no longer receiving any requests of a particular type. Classical provenance systems cannot diagnose such problems: it is not immediately clear where to start the investigation; the debugger does not know where the missing requests would normally come from, or how they would be generated.

- **Software bugs:** Operators must be able to explain and repair an observed faulty behavior even when the root cause is a bug in the program (e.g., a copy-and-paste typo). A fundamental challenge is that prior work primarily focus on tracking dependencies between messages or configurations and fall short of capturing fine-grain causality within programs; this is insufficient for exposing software bugs and for suggesting possible repairs, because it is not even clear which elements in the program contributed to the faulty behavior.

- **Timing-related faults:** The problem is not the presence or absence of an event, but rather the wrong timing of an event. For instance, suppose that an administrator observes that a virtual machine takes unusually long to boot. The reason is that a misconfigured machine is overloading the storage backend. The administrator can inspect the provenance of the booting request and identify the bottleneck (the storage backend). However, this is only the first step. The operator must then find the causes of the bottleneck in order to fix the problem. Provenance provides little help with the latter step – in fact, the actual root cause (the misconfigured machine) would not even appear in it! Classical provenance only captures *functional* causality, that is, events which directly contributed to the occurrence of the observed symptom, this will miss the actual root causes that contributed only in terms of timing.

For network provenance to be usable more broadly in diagnosing networks and distributed systems, we need to build systems that can effectively and efficiently address the three classes of problems above.

## 1.1 Approach

In this dissertation, we argue that extensions to classical network provenance can support deeper diagnosis of networks and distributed systems. We demonstrate this by developing systems that address the problems identified above.

First, in order to diagnose missing events, we need to use the concept of *negative provenance*. The key insight is counterfactual reasoning, that is, to examine all the ways in which a missing event could have occurred, and then show, as a "root cause", the reason why each of them did not come to pass.

Second, to use provenance for correcting software bugs, we must reason about the provenance of data not only in terms of the data it was computed from, but also in terms of the parts of the program it was computed with. In addition, the provenance must allow us to infer how the observed symptom might be affected by changes to the program. Our key insight is to treat the program as just another kind of data. We called the resulting provenance as *meta provenance*.

Third, in order for provenance to handle timing-related faults, we need to track not only functional causality but also temporal causality, which is any event that has contributed to the timing of the observed symptom, regardless of whether it is functionally-related. The key technique to track temporal causality is capturing and analyzing event orderings on distributed nodes. We referred to the resulting provenance as *temporal provenance*.

## 1.2 Contribution and roadmap

In chapter 2, we provide some background material on network provenance and discuss related work on diagnosis of networks and distributed systems. We then make the following contributions:

- In chapter 3, we present *negative provenance* – a network provenance extension that is able to generate causal explanations for not only the presence of events but also the absence of events.

  We present a formal model of positive and negative provenance in distributed systems, as well as a concrete algorithm for tracking such provenance. In addition, we developed a set of heuristics for simplifying the resulting explanations and for making them more readable to a human investigator. We present the design and implementation of Y! (pronounced "Why not?"), a system for tracking positive and negative provenance and for answering queries about it, as well as two case studies in the context of software-defined networking (SDN) and border- gateway protocols (BGP).

- In chapter 4, we present *meta provenance*, which extends network provenance to both program and data.

  We present a concrete algorithm that can generate meta provenance, as well as a prototype system that can collect the necessary data in SDNs and suggest repairs. We have applied our approach to three different environments, covering both declarative and imperative SDN languages. We report our experience from diagnosing practical faults in a real-world network setting, as well as scalability and runtime overhead results.

- In chapter 5, we propose *temporal provenance* – a network provenance extension that allows us to find all causes to unexpected timing, regardless of whether it is functionally-related.

  We present a concrete algorithm that generates temporal provenance as well as postprocessing techniques that improves the readability of temporal provenance. We present Zeno, a prototype debugger for collecting temporal provenance in both declarative and imperative environments and for diagnosing timing-related faults. We present results of using Zeno to explain faulty temporal behaviors observed in real-world incidents.

# 2

# Background and related work

For ease of exposition, we first briefly review Network Datalog [83] in Section 2.1. We then introduce provenance-based diagnosis using an example in Section 2.2. We conclude by discussing the literature on provenance as well as on diagnosis of networks and distributed systems in Section 2.3.

## 2.1 Network Datalog

For ease of exposition, we sometimes assume that the distributed system is written in Network Datalog (NDlog) [83] while describing our approach.Note that our approach is not specific to NDlog, or even to declarative languages; indeed, our case studies have applied the approach both to NDlog and to imperative environments (such as Pyretic [96], Trema [127], and Zipkin [148]). Next we briefly review the features of NDlog that are relevant here.

In NDlog, the state of a node (switch, controller, or server) is modeled as a set of *tables*. Each table contains a number of *tuples*. For instance, a SDN switch might

```
r1 packet(@S',H) :- packet(@S,H),flowTable(@S,H,A,P), link(@S,S',P), A == Fwd.
r2 flowTable(@S,H,A,P) :- missHandler(@C,H,S,A,P), packet(@C,H).
```

Figure 2.1: Example NDlog rules that describe a SDN.

contain a table called `flowTable`, and each tuple in this table might represent a flow entry, or a SDN switch might have a table called `packet` that contains the packets it has received from neighboring switches. Tuples can be manually inserted, or they can be programmatically derived from other tuples; the former are called *base tuples*, and the latter are referred to as *derived tuples*.

Figure 2.1 shows parts of a NDlog program that describe a (simplistic version of) SDN. The program consists of *rules* that describe how tuples should be derived from each other. In an NDlog rule, the *head* on the left is derived when all *predicates* on the right are satisfied. For instance, rule `r1` says that a `packet(@S',H)` tuple should be derived on switch `S'` whenever there is also a incoming `packet(@S,H)` tuple, a matching `flowTable(@S,H,A,P)` tuple, and a `link(@S,S',P)` tuple on the last-hop switch `S`. Here, S and H are variables that must be instantiated with values when the rule is applied; a `packet(@S,80)` tuple would create an `packet(@S',80)` tuple. The `@` operator specifies the node on which the tuple resides. (NDlog supports other operators – e.g., arithmetic or aggregation operators – as well as user-defined functions, but we do not consider these here.) A key advantage of a declarative formulation is that causality is very easy to see: if a tuple `packet(@S2,80)` was derived using the rule above, then `packet(@S2,80)` exists simply because `packet(@S1,80)`, `flowTable(@S1,80,Fwd,1)`, and `link(@S1,S2,1)` exist.

To execute a NDlog program, the runtime compiles rules into chains of relational operators called *strands*: a strand takes as input inserted, derived, or received tuples that match any table predicate (e.g., `packet(@S',H)` or `flowTable(@S,H,A,P)` in rule `r1`); such tuples are pushed into a equijoin between all table predicates; matches from the equijoin go through selection filters, which evaluate selection predicates (e.g., `A == Fwd` in rule `r1`); aggregate operations are translated after equijoins and

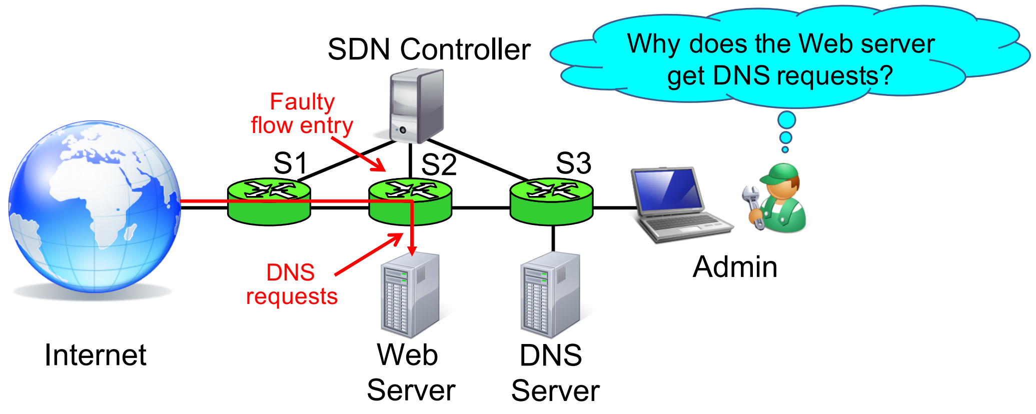


Figure 2.2: Scenario: A faulty switch forwards DNS requests to the web server.

selections; finally, a projection constructs the tuple to match the head of the rule (e.g., `packet(@S',H)` in rule `r1`). In order to execute NDlog rules on distributed nodes, the runtime sends tuples with remote location identifiers through the network stack to their respective destinations.

## 2.2 Provenance-based diagnosis

Before discussing the literature on provenance, we use a basic example to sketch how provenance-based diagnosis works. Figure 2.2 illustrates a example scenario. An operator manages a small SDN that connects a DNS server, a web server, and the Internet. DNS requests arrive at the network and are forwarded by flow entries on the SDN switches (rule `r1` in Figure 2.1). Flow entries are installed by the SDN controller (rule `r2` in Figure 2.1). However, a switch with a faulty flow entry forwards all DNS requests to the port that connects to the web server. The operator observes that the web server is receiving DNS requests. This is formulated as a provenance query, such as, "What is the provenance of the DNS query?". Ideally, the provenance-based debugger should a) explain the observed symptoms (such as, how the DNS request was initiated and forwarded), b) reveal the corresponding root cause (such as, a recent switch configuration change), and c) propose remedies or repairs to the problem (such as, reverting the configuration change).

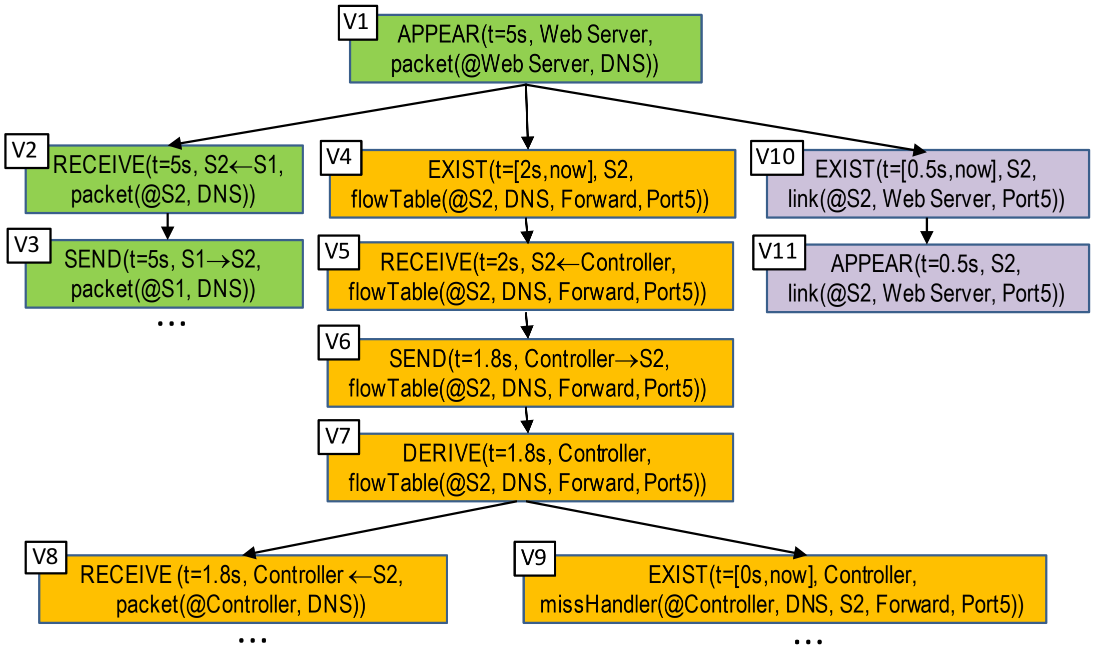

V1
APPEAR(t=5s, Web Server, packet(@Web Server, DNS))
V2
RECEIVE(t=5s, S2←S1, packet(@S2, DNS))
V3
SEND(t=5s, S1→S2, packet(@S1, DNS))
...
V4
EXIST(t=[2s,now], S2, flowTable(@S2, DNS, Forward, Port5))
V5
RECEIVE(t=2s, S2←Controller, flowTable(@S2, DNS, Forward, Port5))
V6
SEND(t=1.8s, Controller→S2, flowTable(@S2, DNS, Forward, Port5))
V7
DERIVE(t=1.8s, Controller, flowTable(@S2, DNS, Forward, Port5))
V8
RECEIVE (t=1.8s, Controller ←S2, packet(@Controller, DNS))
...
V9
EXIST(t=[0s,now], Controller, missHandler(@Controller, DNS, S2, Forward, Port5))
...
V10
EXIST(t=[0.5s,now], S2, link(@S2, Web Server, Port5))
V11
APPEAR(t=0.5s, S2, link(@S2, Web Server, Port5))

Figure 2.3: Classical network provenance example, explaining the symptom from the scenario in Figure 2.2 (how a DNS packet made its way to the web server).

As we have discussed in Section 2.1, provenance models the distributed system as a giant database: the state of each node are stored in tables, and the programs are modeled as declarative rules. The provenance of a tuple (or packet, or data item) consists of the tuples from which it was directly derived. By applying this idea recursively, it is possible to trace the provenance of a tuple in the output of a query all the way to the "base tuples" in the underlying databases. The result is a *provenance graph* – a comprehensive causal explanation of how the tuple came to exist. In provenance graphs, vertexes represent events and edges indicate a direct causal relationships. Figure 2.3 shows an example that explains why the DNS request from the scenario in Figure 2.2 appeared at the web server at time $t = 5$ (V1). The web server had received the packet from switch S2, which in turn had received it from S1, and ultimately from the Internet (V2–V3); the switch was connected to the web server via port #5 (V10–V11) and had a flow entry that directed DNS packets to that port (V4). The flow entry had been installed at $t = 2$ (V5–V7) because the switch had forwarded an earlier DNS packet to the controller (V8), which had generated the flow entry based on its configuration (V9).

## 2.3 Related work

### 2.3.1 Provenance

Provenance originated in the database community [17], which is defined as the description of the origins of data and the process by which it arrived at the database. There are several common notions of database provenance [28]. *Why provenance* explains why a tuple in the query result was produced; it collects "witnesses" from input records, which are input tuples that are sufficient to ensure the existence of the output tuple. *How provenance* explains how an output tuple was computed step by step; for instance, provenance semirings [52] represent the derivation process using a polynomial, where operators represent transformations and variables represent input tuples or intermediate results. *Where provenance* describes the relationship between source and output locations, where a location is a column of a tuple.

Provenance has found many interesting uses, such as estimating data quality [63], building replication recipes [43], diagnosing query answers [21], data integration [53], managing probalistic data [111, 131], and reverse data management [93, 94, 62]. For instance, Why-Not [21] provide query-based explanations for SQL queries, which reveal over-constrained conditions in the queries and suggest modifications to them; Meliou et al. [94] focuses on instance-based explanations for missing answers, that is, how to obtain the missing answers by making modifications to the value of base instances (tuples).

The provenance concept itself is not specific to databases: it can potentially help in any situation where a system has shown some unexpected or suspicious behavior that must now be investigated. For instance, provenance has been used for diagnostics and forensics in operating systems [11, 59, 98, 44, 32].

### 2.3.2 Network provenance

Recently, there has been a line of work on *network provenance*, which adapt the concept of provenance for diagnostics and forensics in distributed systems. These systems sometime exhibit unexpected behaviors such as usual routes or dropped packets, which may have happened due to configurations errors [129], software bugs [118], or worse intentional attacks [40].

To adapt provenance for distributed systems, network provenance has dealt with a number of challenges. ExSPAN [147] adapts provenance to the distributed nature of networks; it partitions and distributes the provenance such that each node maintains only a portion of the information; diagnostic queries are evaluated via running distributed queries. DTaP [146] adds a temporal dimension to provenance; this is useful because network states tends to be short-lived; to answer diagnostic queries, the provenance must also maintain past states; however, this massively increases the amount of provenance information that must be kept; therefore, DTaP optimizes the overhead by exploiting trade-offs between maintenance overhead and query latency. SNP [145] adapts provenance for forensics. In adversarial settings, provenance components are prone to attacks. A hacked node can fabricate its local provenance records or can lie to other nodes. To address this, SNP guarantees that each piece of provenance information attributes to a single node, and use cryptographic primitives to validate such information. SPP [22] considers the problem of providing secured provenance information of high-speed network traffic. In such environments, cryptographic operations (that are used for securing provenance) cause enormous overhead. To address this challenge, SPP uses a lightweight security protocol. Differential provenance [24], given a symptom event, find its root cause by reasoning about the differences between its provenance and the provenance of a similar "reference" event. Chen et al. [26] compresses provenance data in distributed environments for storage savings. This dissertation generalizes network provenance to addresses several open challenges, as explained in Section 1.2.

### 2.3.3 Verification and testing

Verification techniques have been used to ensure the correctness of networks and distributed systems. Some systems statically analyze protocols or configurations and check for correctness violations, as in ConfigChecker [6], FlowChecker [5], Batfish [41], Header Space Analysis [68], NetPlumber [67], Anteater [89], VeriFlow [69], Libra [139], rcc [38], Cocoon [113], VMN [107], Delta-Net [61], and Minesweeper [12]. There are also domain-specific languages for writing distributed systems with verifiable guarantees [100, 71, 7, 96, 85, 75].

Testing is another effective approach to diagnostics. Hubble [66] uses probing to find AS-level reachability problems. ATPG [138] generates test packets for ensuring the liveness, reachability, and performance of a network. NICE [18] uses model checking to test whether a given SDN program has specific correctness properties. MCS [118] and DEMi [117] can extract a minimal execution trace that triggers a faulty behavior. Buzz [37] uses symbolic execution to generate test packets.

These systems pro-actively find and eliminate certain types of bugs before actual deployment. Consequently, operators can entirely avoid post-facto diagnosis of covered bugs. In comparison, provenance focuses on diagnosing unforeseen problems at runtime. Furthermore, verification is a powerful technique and provides strong guarantees, but it usually cannot handle dynamic or stateful interactions like the ones where provenance is targeted at. For example, Header Space Analysis [68] can find erroneous properties in a network by verifying its configurations. However, it cannot handle a dynamic node whose behavior depends on past traffic.

### 2.3.4 Program analysis

Program analysis has been used to ensure the correctness of programs. The software engineering community has used genetic programming [79] and symbolic execution [102] to fix programs. ClearView [109] mines invariants in programs, correlates violations with failures, and generates fixes at runtime; ConfDiagnoser [140] compares correct and undesired executions to find suspicious predicates in the program; and Sidiroglou et al. [119] runs attack vectors on instrumented applications and then generates fixes automatically. Given a specification of the output, program slicing [3, 130, 108] captures relevant parts of the program by generating a reduced program, which is obtained by eliminating statements from the original program.

These systems work well in explaining bugs or generating specific kinds of fixes for programs. Network provenance focuses on diagnosing and repairing faults in distributed systems, which is an orthogonal problem. Furthermore, these systems primarily rely on heuristics, whereas our proposed approach uses provenance to track causality. In comparison, tracking causality can incur higher cost at runtime, but provenance can pinpoint concrete root causes, and therefore using provenance is usually more precise than relying on heuristics.

### 2.3.5 Program synthesis

Recent work has applied program synthesis to distributed systems: NetEgg [137] synthesizes SDN programs from example scenarios; Condor [116] synthesizes network topologies; McClurg et al. [90], Genesis [122], Hojjat et at. [60], and NetGen [114] synthesize network updates and configurations to satisfy invariants.

These systems often aim to derive a complete configuration of the network, and thus require a full test suite or a formal specification, which are not easily available in distributed systems. Network provenance is less ambitious: the goal is to explain and sometimes repair a specific problem in the network. Consequently, network provenance does not burden the operator with providing a comprehensive description of her intent. Indeed, a simple description of the faulty event is sufficient. Furthermore, provenance handles more dynamic executions. For instance, both Hojjat et at. [60] and NetGen [114] fix problems on the data plane, i.e., a snapshot of the network configuration at a particular time; where as meta provenance repairs control programs and considers dynamic network configuration changes triggered by network traffic. At the same time, program synthesis do usually provide stronger guarantees: both Hojjat et at. [60] and NetGen [114] are proven to find the optimal change to the data plane.

### 2.3.6 Tracing and profiling

These systems can handle complex and dynamic interactions, usually by using tools such as statistical analysis, data mining, or custom heuristics. Distributed tracing systems track the path of individual requests, and then explain the execution step by step. Some of these systems *infer* causal relationships, usually from log messages and configurations [29, 142, 129, 65, 8, 2, 39], or from external annotations [4, 76]. This has the advantage of being minimally intrusive to the runtime system. For instance, the mystery machine [29] relies on "big data" techniques: it generates a large number of potential hypotheses about program behavior and then rejects those that are contradicted by empirical observations. CPI [141] learns normal and anomalous behaviors by aggregating data from similar tasks and use statistical correlation to identify culprits. These systems tend to work well when there is abundant training data, but its power is limited when diagnosing rare anomalies or occasional glitches, which are often the trickiest and most time-consuming problems to debug. X-ray [9] can infer which root causes are most costly during an entire application execution; it record costs of low-level operations at runtime and rely on dynamic information flow analysis to associate these costs with potential root causes. Diagnosis can also be done by comparing "good" and "bad" instances [104, 112, 99, 115] and analyzing their differences. This tends to work well when both types of instances are available, but this is not always the case.

In comparison, provenance is a white-box approach that explicitly records causal relationships rather than inferring or mining them; this helps us avoid false positives (e.g., when events are correlated but not causally related) and false negatives (e.g., in the case of sporadic problems for which the system does not have enough data to mine). This approach is similar to tracing systems that rely on instrumentation [120, 1, 95, 125, 10, 27, 19, 123, 135, 42, 58, 57, 121, 64]. For example, Dapper [120] produces trace tress which describes RPCs triggered by a request and the causal dependencies between them; Canopy [64] annotates traces with performance data (such as, counters or stack traces), which enables engineers to perform performance diagnosis. However, tracing systems usually rely on ad-hoc or protocol-specific ways of capturing causality whereas provenance systematically tracks the full detail of the execution and is generally applicable to distributed systems.

# 3 Negative Provenance

## 3.1 Introduction

Provenance can be a useful tool for debugging complex interactions, but there are cases that it cannot handle. For instance, suppose that the administrator observes that a certain server is *no longer receiving any requests* of a particular type. The key difference to the earlier scenario is that the observed symptom is not a positive event, such as the arrival of a packet, that could serve as a "lead" and point the administrator towards the root cause. Rather, the observed symptom is a negative event: the *absence* of packets of a certain type. Negative events can be difficult to debug: provenance does not help, and even a manual investigation can be difficult if the administrator does not know where the missing packets would normally come from, or how they would be generated.

Nevertheless, it is possible to construct an explanation for negative events, using the concept of *negative provenance* [62]. The key insight is to use counterfactual reasoning, that is, to examine all possible causes that *could have* produced the missing

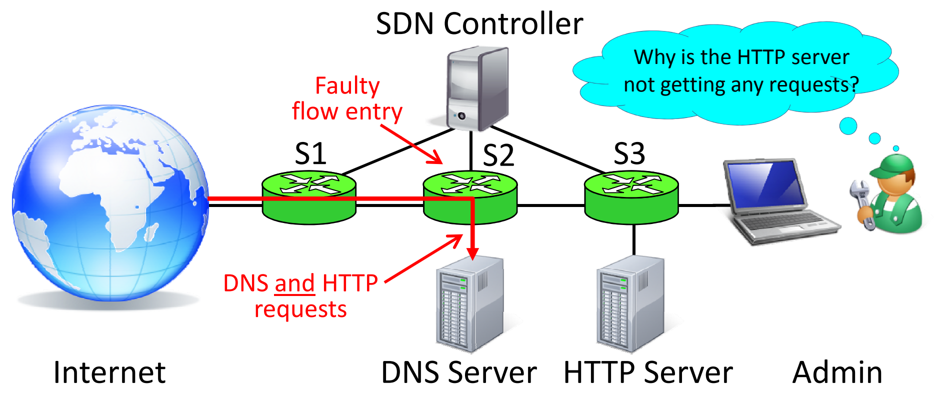


Figure 3.1: Negative event scenario: Web requests from the Internet are no longer reaching the web server because a faulty program on the controller has installed an overly general flow entry in the switch in the middle (S2).

effect. For instance, it might be the case that the missing packets could only have reached the server through one of two upstream switches, and that one of them is missing a flow entry that would match the packets. Based on the controller program, we might then establish that the missing entry could only have been installed if a certain condition had been satisfied, and so on, until we either reach a positive event (such as the installation of a conflicting flow entry with higher priority) that can be traced with normal provenance, or a negative root cause (such as a missing entry in a configuration file).

Negative provenance could be a useful debugging tool for networks and distributed systems in general, but so far it has not been explored very much. A small number of papers from the database community [62, 94, 21] have used negative provenance to explain why a given database query did *not* return a certain tuple, but we are not aware of any previous applications in the networking domain.

In Section 3.2, we provide an overview of positive and negative provenance. We then make the following contributions:

- A formal model of positive and negative provenance in distributed systems, as well as a concrete algorithm for tracking such provenance (Section 3.3);

- A set of heuristics for simplifying the resulting provenance graphs and for making them more readable to a human investigator (Section 3.4);
- The design of Y! (pronounced "Why not?"), a system for tracking positive and negative provenance and for answering queries about it (Section 3.5);
- Two case studies of Y!, in the context of software-defined networks and BGP (Section 3.6); and
- An experimental evaluation of an Y! prototype, based on Mininet, Trema [127] and RapidNet [84] (Section 3.7).

We discuss related work in Section 3.8 and conclude this chapter in Section 3.9.

## 3.2 Overview

In this section, we take a closer look at negative provenance, and we discuss some of the key challenges. To distinguish classical provenance 2.3.2 from negative provenance, we will refer to it as *positive provenance*.

### 3.2.1 Scenario: Network debugging

Figure 3.1 shows a simple example scenario that illustrates the problem we are focusing on. A network administrator manages a small network that includes a DNS server, a web server, and a connection to the Internet. At some point, the administrator notices that the web server is no longer receiving any requests from the Internet. The administrator strongly suspects that the network is somehow misconfigured, but the only observable symptom is a *negative event* (the absence of web requests at the server), so there is no obvious starting point for an investigation.

Today, the only way to resolve such a situation is to manually inspect the network until some positive symptom (such as requests arriving at the wrong server) is found. In the very simple scenario in Figure 3.1, this is not too difficult, but in a data center or large corporate network, it can be a considerable challenge. It seems preferable

| Mailing list | Posts related to diagnostics | Initial symptoms Positive | Initial symptoms Negative |
|---|---|---|---|
| NANOG-user | 29/144 | 14 (48%) | 15 (52%) |
| floodlight-dev | 19/154 | 5 (26%) | 14 (74%) |
| Outages [105] | 46/60 | 8 (17%) | 38 (83%) |

Table 3.1: Survey of networking problems and their symptoms, as discussed on three mailing lists over a two-month period, starting on November 22, 2013.

for the administrator to directly ask the network for an explanation of the negative event, similar to a "backtrace" in a conventional debugger. This is the capability we seek to provide.

### 3.2.2 Case study: Broken flow entry

In the scenario from Figure 3.1, one possible reason for this situation is that the administrator has configured the controller to produce a generic, low-priority flow entry for DNS traffic and a specific, high-priority flow entry for HTTP traffic. If both entries are installed, the system works as expected, but if the low-priority entry is installed first, it matches HTTP packets as well; thus, these packets are not forwarded to the controller and cannot trigger the installation of the high-priority entry. This subtle race condition might manifest only at runtime, e.g., when both entries expire simultaneously during an occasional lull in traffic; thus, it could be quite difficult to find.

Positive provenance is not helpful here because, as long as requests are still arriving at the HTTP server, their provenance contains only the high-priority entry, and when the requests stop arriving, there is no longer anything to generate the provenance *of*!

### 3.2.3 How common are negative symptoms?

To get a sense of how common this situation is, we surveyed diagnostics-related posts on three mailing lists that covers a mix of different diagnostic situations: NANOG-user and Outages [105] (for faults and misconfigurations), and floodlight-dev (for software bugs). To get a good sample size, we examined a two-month period for each list, starting on November 22, 2013. In each post, we looked for the description of the initial symptoms and classified them as either positive (something bad happened) or negative (something good failed to happen).

Table 3.1 shows our results. While the proportion of positive and negative symptoms varies somewhat between lists, we find that the negative symptoms are consistently *in the majority* – that is, it seems more common for problems to initially manifest as the absence of something (e.g., a route, or a response to a probe packet) than as the presence of something (e.g., high latencies on a path, or a DDoS attack).

Many of the problems we surveyed were eventually diagnosed, but we observe that the process seems comparatively harder: there were significantly more (and lengthier) email threads where negative symptoms resulted in inconclusive identification of root causes. Moreover, troubleshooting negative symptoms often required exploratory "guesswork" by the mailing list participants. Since this trial-and-error approach requires lots of time and effort, it seems useful to develop better tool support for this class of problems.

### 3.2.4 Negative provenance

Our approach towards such a tool is to extend provenance to negative events. Although these cannot be explained directly with positive provenance, there is a way to construct a similar "backtrace" for negative events: instead of explaining how an actual event *did* occur, as with positive provenance, we can simply find all the ways in which a missing event *could have* occurred, and then show, as a "root cause", the reason why each of them did not come to pass.

Intuitively, we can use a kind of counterfactual reasoning to recursively generate the explanations, not unlike positive provenance: for a web request to arrive at the web server, a request would have had to appear at the rightmost switch (S3), which did not happen. Such a request could only have come from the switch in the middle (S2), and, eventually, from the switch on the left (S1). But S2 would only have sent the request if there had been 1) an actual request, 2) a matching flow entry with a forward action to S3, and 3) no matching higher-priority flow entry. Conditions 1) and 2) were satisfied, but condition 3) was not (because of the DNS server's flow entry). We can then ask where the higher-priority flow entry came from, which can be answered with positive provenance. We refer to such a counterfactual explanation as *negative provenance*.

### 3.2.5 Challenges

To explain the key challenges, we consider two strawman solutions. First, it may seem that there is a simpler way to investigate the missing HTTP requests from Section 3.2.2: why not simply compare the system state before and after the requests stopped arriving, and return any differences as the likely cause? This approach may indeed work in some cases, but in general, there are way too many changes happening in a typical system: even if we could precisely pinpoint the time where the problem appeared, chances are that most of the state changes at that time would be unrelated. Moreover, if the problem was caused by a *chain* of events, this method would return only the last step in the chain. To identify the relevant events reliably, and to trace them back to the root cause, we must have a way to track *causality*, which is, in essence, what provenance represents.

Second, it may seem that, in order to track negative provenance, we can simply take an existing provenance system, like ExSPAN or SNP, and associate each positive provenance vertex with a negative "twin". However, the apparent similarity between positive and negative provenance does not go very deep. While positive provenance considers only *one specific* chain of events that led to an observed event, negative provenance must consider *all possible* chains of events that *could have* caused the observed event. This disparity between existential and universal quantifiers has profound consequences: for instance, negative provenance graphs are often infinite and cannot be materialized, and responses to negative queries tend to be a lot more complex, and thus need more sophisticated post-processing before they can be shown to a human user. These are some of the challenges we address in Y!.

## 3.3 Basic Negative Provenance

In this section, we show how to derive a simple provenance graph for both positive and negative events. For ease of exposition, we will assume that the distributed system is written in Network Datalog (NDlog) [83], since this representation makes provenance particularly easy to see. However, our approach is not specific to NDlog, or even to declarative languages; indeed, our case studies in Section 3.6.1 apply it to Pyretic [96], an existing imperative programming language for SDNs, as well as to BGP debugging.

### 3.3.1 Goals

Before we define our provenance graph, we first state, somewhat informally, the properties we would like to achieve. One way to describe what "actually happened" in an execution of the system is by means of a *trace*: a sequence of message transmissions and arrivals, as well as base tuple insertions and deletions. (Other events, such as derivations, follow deterministically from these.) Following [146], we can then think of the provenance $G(e,\mathscr{E})$ of an event $e$ in a trace $\mathscr{E}$ as describing a series of trace properties, which, in combination, cause $e$ to appear – or, in the case of a negative event, prevent $e$ from appearing. We demand the following properties:

- **Soundness:** $G(e,\mathscr{E})$ must be consistent with $\mathscr{E}$;
- **Completeness:** There must not be another execution $\mathscr{E}'$ that is also consistent with $G(e,\mathscr{E})$ but does not contain the event $e$; and

- **Minimality:** There must not be a subset of $G(e,\mathcal{E})$ that is also sound and complete.

Informally, soundness means that $G(e,\mathcal{E})$ must describe events that actually happened in $\mathcal{E}$ – we cannot explain the absence of a tuple with the presence of a message that was never actually sent. Completeness means that $G(e,\mathcal{E})$ must be sufficient to explain $e$, and minimality means that all events in $G(e,\mathcal{E})$ must actually be relevant (though there could be more than one provenance that is minimal in this sense). We will state these properties formally in Section 3.3.7.

### 3.3.2 The provenance graph

Provenance can be represented as a DAG in which the vertices are events and the edges indicate direct causal relationships. Thanks to NDlog's simplicity, it is possible to define a very simple provenance graph for it, with only ten types of event vertices (based on [145]):

- EXIST$([t_1,t_2],N,\tau)$: Tuple $\tau$ existed on node $N$ from time $t_1$ to $t_2$;
- INSERT$(t,N,\tau)$: Base tuple $\tau$ was inserted on node $N$ at time $t$;
- DELETE$(t,N,\tau)$: Base tuple $\tau$ was deleted on node $N$ at time $t$;
- DERIVE$(t,N,\tau)$: Derived tuple $\tau$ acquired support on $N$ at time $t$;
- UNDERIVE$(t,N,\tau)$: Derived tuple $\tau$ lost support on $N$ at time $t$;
- APPEAR$(t,N,\tau)$: Tuple $\tau$ appeared on node $N$ at time $t$;
- DISAPPEAR$(t,N,\tau)$: Tuple $\tau$ disappeared on node $N$ at time $t$;
- SEND$(t,N\rightarrow N',\pm\tau)$: $\pm\tau$ was sent by node $N$ to/from $N'$ at $t$;
- RECEIVE$(t,N\leftarrow N',\pm\tau)$: $\pm\tau$ was received by node $N$ to/from $N'$ at $t$; and
- DELAY$(t,N\rightarrow N',\pm\tau,d)$: $\pm\tau$, sent from $N$ to $N'$ at $t$, took time $d$ to arrive at $N'$.

The edges between the vertices correspond to their intuitive causal connections: tuples can appear on a node because they a) were inserted as base tuples, b) were derived from other tuples, or c) were received in a message from another node (for cross-node rules). Messages are received because they were sent, and tuples exist because they appeared. Note that vertices are annotated with the node on which they occur, as well as with the relevant time; the latter will be important for negative provenance because we will often need to reason about past events.

This model can be extended to support negative provenance by associating each vertex with a negative "twin":

- NEXIST($[t_1,t_2],N,\tau$): Tuple $\tau$ never existed on node $N$ in time interval $[t_1,t_2]$;
- NINSERT($[t_1,t_2],N,\tau$): Tuple $\tau$ was never inserted on $N$ in $[t_1,t_2]$;
- NDELETE($[t_1,t_2],N,\tau$): Tuple $\tau$ was never removed on $N$ in $[t_1,t_2]$;
- NDERIVE($[t_1,t_2],N,\tau$): $\tau$ was never derived on $N$ in $[t_1,t_2]$;
- NUNDERIVE($[t_1,t_2],N,\tau$): $\tau$ was never underived on $N$ in $[t_1,t_2]$;
- NAPPEAR($[t_1,t_2],N,\tau$): Tuple $\tau$ never appeared on $N$ in $[t_1,t_2]$;
- NDISAPPEAR($[t_1,t_2],N,\tau$): Tuple $\tau$ never disappeared on $N$ in $[t_1,t_2]$;
- NSEND($[t_1,t_2],N,\tau$): $\tau$ was never sent by node $N$ in $[t_1,t_2]$;
- NRECEIVE($[t_1,t_2],N,\tau$): $\tau$ was never received by node $N$ in $[t_1,t_2]$; and
- NARRIVE($[t_1,t_2],N_1\rightarrow N_2,t_3,\tau$): $\tau$ was sent from $N_1$ to $N_2$ at $t_3$ but did not arrive within $[t_1,t_2]$.

Again, the causal connections are the intuitive ones: tuples never existed because they never appeared, they never appeared because they were never inserted, derived, or received, etc. However, note that, unlike their positive counterparts, *all* negative vertices are annotated with time intervals: unlike positive provenance, which can

```
function QUERY(EXIST([$t_1$, $t_2$],N,$\tau$))
    S ← $\emptyset$
    for each (+$\tau$,N,t,r,c) ∈ Log:   $t_1 \le t \le t_2$
        S ← S ∪ { APPEAR(t,N,$\tau$,r,c) }
    for each (−$\tau$,N,t,r,c) ∈ Log:   $t_1 \le t \le t_2$
        S ← S ∪ { DISAPPEAR(t,N,$\tau$,r,c) }
    RETURN $S$
function QUERY(APPEAR(t,N,$\tau$,r,c))
    if BaseTuple($\tau$) then
        RETURN { INSERT(t,N,$\tau$) }
    else if LocalTuple(N,$\tau$) then
        RETURN { DERIVE(t,N,$\tau$,r) }
    else RETURN{RECEIVE(t,$N \leftarrow r.N$,$\tau$)}
function QUERY(INSERT(t,N,$\tau$))
    RETURN $\emptyset$
function QUERY(DERIVE(t,N,$\tau$,$\tau$:-$\tau_1$,$\tau_2$...))
    S ← $\emptyset$
    for each $\tau_i$: if (+$\tau_i$,N,t,r,c) ∈ Log:
        S ← S ∪ { APPEAR(t,N,$\tau_i$,c) }
    else
        $t_x$ ← max $t' < t$: (+$\tau$,$N$,$t'$,r,1) ∈ Log
        S ← S ∪ { EXIST([$t_x$,t],N,$\tau_i$,c) }
    RETURN $S$
function QUERY(NRECEIVE([$t_1$,$t_2$],N,+$\tau$))
    S ← $\emptyset$, $t_0 \leftarrow t_1 - \Delta$max
    for each $N'$ ∈ SENDERS($\tau$,N):
        X← {$t_0 \le t \le t_2$|(+$\tau$,$N'$,t,r,1)∈Log}
        $t_x \leftarrow t_0$
        for (i=0; i< |$X$|; i++)
            S←S∪{NSEND(($t_x$,$X_i$),$N'$,+$\tau$),
           NARRIVE(($t_1$,$t_2$),$N' \rightarrow N$,$X_i$,+$\tau$)}
            $t_x \leftarrow X_i$
        S ← S ∪ { NSEND([$t_x$,$t_2$],$N'$,+$\tau$) }
    RETURN $S$
```

```
function QUERY(RECEIVE(t,$N_1 \leftarrow N_2$,+$\tau$))
    $t_s$ ← max $t' < t$: (+$\tau$,$N_2$,$t'$,r,1) ∈ Log
    RETURN { SEND($t_s$,$N_1 \rightarrow N_2$,+$\tau$),
        DELAY($t_s$,$N_2 \rightarrow N_1$,+$\tau$,$t - t_s$) }
function QUERY(SEND(t,$N \rightarrow N'$,+$\tau$))
    FIND (+$\tau$,N,t,r,c) ∈ Log
    RETURN { APPEAR(t,$N$,$\tau$,r) }
function QUERY(NEXIST([$t_1$,$t_2$],N,$\tau$))
    if $\exists t < t_1$ : (-$\tau$,N,t,r,1) ∈ Log then
        $t_x$ ← max $t < t_1$: (-$\tau$,N,t,r,1) ∈ Log
        RETURN { DISAPPEAR($t_x$,N,$\tau$),
            NAPPEAR(($t_x$,$t_2$],N,$\tau$) }
    else RETURN { NAPPEAR([0,$t_2$],N,$\tau$) }
function QUERY(NDERIVE([$t_1$,$t_2$],N,$\tau$,r))
    S ← $\emptyset$
    for ($\tau_i$,$I_i$) ∈ PARTITION([$t_1$,$t_2$],N,$\tau$,r)
        S ← S ∪ { NEXIST($I_i$,N,$\tau_i$) }
    RETURN $S$
function QUERY(NSEND([$t_1$,$t_2$],N,+$\tau$))
    if $\exists t_1 < t < t_2$ : (-$\tau$,N,t,r,1) ∈ Log then
        RETURN { EXIST([$t_1$,$t$],N,$\tau$),
            NAPPEAR(($t$,$t_2$],N,$\tau$) }
    else RETURN { NAPPEAR([$t_1$,$t_2$],N,$\tau$) }
function QUERY(NAPPEAR([$t_1$,$t_2$],N,$\tau$))
    if BaseTuple($\tau$) then
        RETURN { NINSERT([$t_1$,$t_2$],N,$\tau$) }
    else if LocalTuple(N,$\tau$) then
        RETURN $\bigcup_{r \in \text{Rules(N):Head}(r)=\tau}$
            { NDERIVE([$t_1$,$t_2$],N,$\tau$,r) }
    else RETURN {NRECEIVE([$t_1$,$t_2$],N,+$\tau$)}
function Q(NARRIVE([$t_1$,$t_2$],$N_1 \rightarrow N_2$,$t_0$,+$\tau$))
    FIND (+$\tau$,$N_2$,$t_3$,($N_1$,$t_0$),1) ∈ Log
    RETURN { SEND($t_0$,$N_1 \rightarrow N_2$,+$\tau$),
        DELAY($t_0$,$N_1 \rightarrow N_2$,+$\tau$,$t_3 - t_0$) }
```

Figure 3.2: Graph construction algorithm. Some rules have been omitted; for instance, the handling of $+\tau$ and $-\tau$ messages is analogous, and the rules for INSERT/DELETE, APPEAR/DISAPPEAR, and DERIVE/UNDERIVE are symmetric.

refer to specific events at specific times, negative provenance must explain the *absence* of events in certain intervals.

### 3.3.3 Handling multiple explanations

Sometimes the absence of an event can have more than one cause. For instance, suppose there is a rule `A:-B,C,D` and, at some time $t$, none of the tuples B, C, or D exist. How should we explain the absence of A in this case? One possible approach would be to include the absence of all three tuples; this would be useful, for instance, if our goal was recovery – i.e., if we wanted to find a way make A appear. However, for diagnostic purposes, the resulting provenance is somewhat verbose, since the absence of each individual tuple is already sufficient to explain the absence of A. For this reason, we opt to include only a *sufficient reason* in our provenance trees.

In cases where there is more than one sufficient reason, the question arises which one we should choose. Since we aim for compact provenance trees, *we try to find the reason that can be explained with the fewest vertices.* For instance, if B and D are derived tuples whose absence is due to a complex sequence of events on several other nodes, whereas C is a base tuple that simply was never inserted, we would choose the explanation that is based on C, which only requires a single NINSERT vertex. In practice, it is not always easy to see which explanation is simplest (at least not without fully expanding the corresponding subtree), but we can use heuristics to find a good approximation, e.g., based on a look-ahead of a few levels down the tree.

### 3.3.4 Graph construction

Provenance systems like ExSPAN [147] rely on a materialized provenance graph: while the distributed system is executing, they build some representation of the vertices and edges in the graph, and they respond to queries by projecting out the relevant subtree. This approach does not work for negative provenance because the provenance graph is typically infinite: for instance, it contains NEXIST vertices for every tuple that could *potentially* exist, and, for each vertex with that contains a time interval $I$, it also contains vertices with intervals $I' \subseteq I$.

For this reason, we adopt a top-down procedure for constructing the provenance of a given (positive or negative) event "on demand", without materializing the entire graph. We define a function QUERY(v) that, when called on a vertex $v$ in the provenance graph, returns the immediate children of $v$. Thus, the provenance of a negative event $e$ can be found by constructing a vertex $v_e$ that describes $e$ (e.g., a NEXIST vertex for an absent tuple) and then calling QUERY recursively on $v_e$ until leaf vertices are reached. The interval in $v_e$ can simply be some interval in which $e$ was observed; it does not need to cover the entire duration of $e$, and it does not need to contain the root cause(s).

QUERY needs access to a log of the system's execution to date. We assume that the log is a sequence of tuples $(\pm\tau, N, t, r, c)$, which indicate that $\tau$ was derived ($+\tau$) or underived ($-\tau$) on node $N$ at time $t$ via rule $r$. Since some tuples can be derived in more than one way, we include a derivation counter $c$, which is 1 when a tuple first appears, and is increased by one for each further derivation. For tuples that node $N$ received from node $N'$, we set $r = N'$, and for base tuples, we set $r = \perp$ and $c = 1$.

Figure 3.2 shows part of the algorithm we use to construct positive and negative provenance. There are several points worth noting. First, the algorithm uses functions BaseTuple($\tau$) and LocalTuple(N,$\tau$) to decide whether a missing tuple $\tau$ is a base tuple that was not inserted, a local tuple on node $N$ that was not derived, or a remote tuple that was not received. The necessary information is a byproduct of the compilation of any NDlog program and is thus easily obtained. Second, to account for propagation delays, the algorithm uses a constant $\Delta_{\text{max}}$ that denotes the maximum time a message can spend in the network and still be accepted by the recipient; this is used to narrow down the time interval during which a missing message could have been sent. Third, the algorithm can produce the same vertex more than once, or semantically identical vertices with adjacent or overlapping intervals;

in these cases, it is necessary to coalesce the vertices using the union of their intervals in order to preserve minimality. Finally, the algorithm uses two functions PARTITION and SENDERS, which we explain next.

### 3.3.5 Explaining nonderivation

The PARTITION function encodes a heuristic for choosing among several possible explanations of a missing derivation. When explaining why a rule with multiple preconditions did not derive a certain tuple, we must consider a potentially complex parameter space. For instance, if `A(@X,p):-B(@X,p,q,r),C(@X,p,q)` did not derive `A(@X,10)`, we can explain this with the absence of `B(@X,10,q,r)`, `C(@X,10,q,r)`, or a combination of both – e.g., by dividing the possible $q$ and $r$ values between the two preconditions. Different choices can result in explanations of dramatically different sizes once the preconditions themselves have been explained; hence, we would prefer a partition of the parameter space (here, $Q \times R$) that results in an explanation that is as small as possible. In general, finding the optimal partition is at least as hard as the SetCover problem, which is NP-hard; hence the need for a heuristic. In our experiments, we use a simple greedy heuristic that always picks the largest available subspace; if there are multiple subspaces of the same size, it explores both for a few steps and then picks the one with the simplest subgraph.

### 3.3.6 Missing messages

The SENDERS($\pm\tau$,N) function is used to narrow down the set of nodes that could have sent a specific missing message $\pm\tau$ to node $N$. One valid choice is to simply return the set of all nodes in the system that have a rule for deriving $\tau$; however, the resulting provenance can be complex, since it must explain why *each* of these nodes did not send $\pm\tau$. Hence, it is useful to enhance SENDERS with other information that may be available. For instance, in a routing protocol, communication is restricted by the network topology, and messages can come only from direct neighbors.

In some cases, further nodes can be ruled out based on the specific message that is missing: for instance, a BGP message whose AS path starts with 7 should come from a router in AS 7. We do not pursue this approach here, but we hypothesize that static analysis of the NDlog program could be used for inferences of this type.

### 3.3.7 Formal properties

We now briefly present the key definitions from our formal model (see Appendix A.1). An *event* $d@n = (m, r, t, c, m')$ represents that rule $r$ was triggered by message $m$ and generated a set of (local or remote) messages $m'$ at time $t$, given the precondition $c$ (a set of tuples that existed on node $n$ at time $t$). Specifically, we write $d@n_{recv} = (m@n_{send}, -, t, 1, m@n_{recv})$ to denote a message $m$ (from $n_{send}$ is delivered at $n_{recv}$ at $t$). A *trace* $\mathscr{E}$ of a system execution is an ordered sequence of events from an initial state $\mathscr{S}_0$, $\mathscr{S}_0 \xrightarrow{d_1@n_1} \mathscr{S}_1 \xrightarrow{d_2@n_2} \ldots \xrightarrow{d_x@n_x} \mathscr{S}_x$. We say a trace $\mathscr{E}$ is *valid*, if (a) for all $\tau_k \in c_i$, $\tau_k \in \mathscr{S}_{i-1}$, and (b) for all $d_i@n_i = (m_i, r_i, t_i, c_i, m'_i)$, either $m_i$ is a base message from an external source, or there exists $d_j@n_j = (m_j, r_j, t_j, c_j, m'_j)$ that precedes $d_i@n_i$ and $m_i \in m'_j$. We say that $\mathscr{E}'$ is a *subtrace* of $\mathscr{E}$ (written as $\mathscr{E}' \subseteq \mathscr{E}$) if $\mathscr{E}'$ consists of a subset of the events in $\mathscr{E}$ in the same order. In particular, we write $\mathscr{E}|n$ to denote the subtrace that consists of all the events on node $n$ in $\mathscr{E}$. We say that $\mathscr{E}'$ and $\mathscr{E}$ are equivalent (written as $\mathscr{E}' \sim \mathscr{E}$) if, for all $n$, $\mathscr{E}'|n = \mathscr{E}|n$.

To properly define minimality, we use the concept of a *reduction*: given negative provenance $G(e, \mathscr{E})$, if there exist vertices $v_1, v_2 \in V(G)$, where the time interval of $v_1$ and $v_2$ ($t(v_1)$ and $t(v_2)$ respectively) are adjacent, and $v_1$ and $v_2$ have the same dependencies, then $G$ can be reduced to $G'$ by combining $v_1$ and $v_2$ into $v \in V(G')$, where $t(v) = t(v_1) \cup t(v_2)$. Given two negative provenance $G(e, \mathscr{E})$ and $G'(e, \mathscr{E})$, we say $G'$ is *simpler* than $G$ (written as $G' < G$), if any of the following three hold: (1) $G'$ is a subgraph of $G$; (2) $G'$ is reduced from $G$ (by combining $v_1$ and $v_2$); or (3) there exists $G''$, such that $G' < G''$ and $G'' < G$.

Using these definitions, we can formally state the three properties from Section 3.3.1 as follows:

**Property (Soundness):** *Negative provenance $G(e, \mathscr{E})$ is sound iff (a) it is possible to extract a valid subtrace $\mathscr{E}_{sub} \subseteq \mathscr{E}'$, such that $\mathscr{E}' \sim \mathscr{E}$ and (b) for all vertices in $G(e, \mathscr{E})$, their corresponding predicates hold in $\mathscr{E}$.*

**Property (Completeness):** *Negative provenance $G(e, \mathscr{E})$ is complete iff no trace $\mathscr{E}'$ exists such that a) $\mathscr{E}'$ assumes the same external inputs as $G(e, \mathscr{E})$, and b) $e$ exists in $\mathscr{E}'$.*

**Property (Minimality):** *Negative provenance $G(e, \mathscr{E})$ is minimal, if no $G' < G$ is sound and complete.*

We have proven that our provenance graph has all three properties. The proofs are available in Appendix A.1.

## 3.4 Enhancing readability

So far, we have explained how to generate a "raw" provenance graph. This representation is correct and complete, but it is also extremely detailed: for instance, simple and common events, such as message exchanges between nodes, are represented with many different vertices. This "clutter" can make the provenance difficult to read. Next, we describe a post-processing technique that can often simplify the provenance considerably, by pruning unhelpful branches, and by summarizing common patterns into higher-level vertices.

### 3.4.1 Pruning unhelpful branches

**Logical inconsistencies:** Some explanations contain logical inconsistencies: for instance, the absence of a tuple $\tau_1$ with parameter space $S_1$ might be explained by the absence of a tuple $\tau_2$ with parameter space $S_2 \subseteq S_1$. If we can recognize such inconsistencies early, there is no need to continue generating the provenance until a set of base tuples is reached – the precondition is clearly unsatisfiable. Thus, we can safely truncate the corresponding branch of the provenance tree.

**Failed assertions:** Some branches explain the absence of events that the programmer has already ruled out. For instance, if a branch contains a vertex NEXIST($[t_1, t_2]$,N,P(5)) and it is known that P can only contain values between 0 and 4, the subtree below this vertex is redundant and can be removed. We use programmer-specified assertions to recognize situations of this type. The assertions do not have to be provenance-specific – they can be the ones that a good programmer would write anyway.

**Branch coalescing:** A naïve execution of the algorithm in Figure 3.2 would result in a provenance *tree*, but this tree would contain many duplicate vertices because many events have more than one effect. To avoid redundancy, we combine redundant vertices whenever possible, which turns the provenance tree into a DAG. If two vertices have overlapping time intervals but are otherwise identical, we use the union of the two intervals. (Note that a smart implementation of PARTITION could take the multiplicity of shared subtrees into account.)

**Application-specific invariants:** Some explanations may be irrelevant for the particular SDN that is being debugged. For instance, certain data – such as constants, topology information, or state from a configuration file – changes rarely or never, so the absence of changes, or the presence of a specific value, do not usually need to be explained. One simple way to identify constant tables is by the absence of derivation rules in which the table appears at the head. Optionally, the programmer can use a special keyword to designate additional tables as constant.

### 3.4.2 Different levels of detail

Another way to make negative provenance graphs more useful for the human investigator is to display the provenance at different levels of detail. For instance, if a message fails to appear at node $N_1$ but could only have originated at node $N_2$ several hops away, the basic provenance tree would show, for each node on the path from

$N_1$ to $N_2$, that the message was not sent from there, because it failed to appear there, because it was not received from the next-hop node, etc. We can improve readability by summarizing these (thematically related) vertices into a single *super-vertex*. When the graph is first shown to the human investigator, we include as many super-vertices as possible, but the human investigator has the option to expand each super-vertex into the corresponding fine-grained vertices if necessary.[1]

We have identified three situations where this summarization can be applied. The first is a chain of transient events that originates at one node and terminates at another, as in the above example; we replace such chains by a single super-vertex. The second is the (common) sequence NEXIST$([t_1,t_2],N,\tau)$ $\leftarrow$ NAPPEAR$([t_1,t_2],N,\tau)$ $\leftarrow$ NDERIVE$([t_1,t_2],N,\tau)$, which basically says that a tuple was never derived; we replace this with a single ABSENCE$([t_1,t_2],N,\tau)$ super-vertex; its positive counterpart EXISTENCE$([t_1,t_2],N,\tau)$ is used to replace a positive sequence. The third situation is a derivation that depends on a small set of triggers – e.g., flow entries can only be generated when a packet $p$ is forwarded to the controller $C$. In this case, the basic provenance will contain a long series of NAPPEAR$([t_i,t_{i+1}],C,p)$ vertices that explain the *common* case where the trigger packet *p does not* exist; we replace these with a single super-vertex ONLY-EXIST$(\{t_1,t_2,\ldots\}$ *in* $[t_{\text{start}},t_{\text{end}}],C,p)$ that initially focuses attention on the *rare* cases where the trigger *does* exist.

## 3.5 The Y! system

In this section, we describe the design of Y! (for "Why not?"), a system for capturing, storing, and querying both positive and negative provenance.

[1]More generally, visualization and interactive exploration are useful strategies for working with large provenance graphs [87].

### 3.5.1 Overview

Like any debugger, Y! is meant to be used in conjunction with some other application that the user wishes to diagnose; we refer to this as the *target application*. Y! consists of four main components: The *provenance extractor* (Section 3.5.2) monitors the target application and extracts relevant events, such as state changes or message transmissions. These events are passed to the *provenance storage* (Section 3.5.3), which appends them to an event log and also maintains a pair of indices to enable efficient lookups of negative events. When the user issues a provenance query, the *query processor* uses the stored information to construct the relevant subtree of the provenance graph, simplifies the subtree using the heuristics from Section 3.4, and then sends the result to the *frontend*, so that the user can view, and interactively explore, the provenance. We now explain some key components in more detail.

### 3.5.2 Provenance extractor

Recall from Section 3.3 that the input to the graph construction algorithm is a sequence of entries $(\pm\tau, N, t, r, c)$, which indicate that the $c$.th derivation of tuple $\tau$ appeared or disappeared on node $N$ at time $t$, and that the reason was $r$, i.e., a derivation rule or an incoming message. The purpose of the provenance extractor is to capture this information from the target application. This functionality is needed for all provenance systems (not just for negative provenance), and it should be possible to use any of the several approaches that have been described in the literature. For instance, the target application can be annotated with calls to a special library whenever a relevant event occurs [97], the runtime that executes the target application (e.g., an NDlog engine or a virtual machine) can report the relevant events [147], or a special proxy can reconstruct the events from the sequence of messages that each node sends and receives [145]. Note that the latter two approaches can be applied even to legacy software and unmodified binaries.

### 3.5.3 Provenance storage

The provenance storage records the extracted events in an append-only log and makes this log available to the query processor. A key challenge is efficiency: with positive provenance, it is possible to annotate each event with pointers to the events that directly caused it, and, since there is a fairly direct correspondence between events and positive vertices in the provenance graph, these pointers can then be used to quickly navigate the graph. With negative provenance, however, it is frequently necessary to evaluate range queries over the time domain ("Did tuple $\tau$ ever exist during interval $[\tau_1, \tau_2]$"). Moreover, our PARTITION heuristic requires range queries over other domains, e.g., particular subspaces of a given table ("Are there any `X(a,b,c)` tuples on this node with $5 \leq b \leq 20$?") to decide which of several possible explanations might be the simplest. If Y! evaluated such range queries by scanning the relevant part of the log, performance would suffer greatly.

Instead, Y! uses R-trees [55] to efficiently access the log. R-trees are tree data structures for indexing multi-dimensional data; briefly, the key idea is to group nearby objects and to represent each group by its minimum bounding rectangle at the next-higher level of the tree. Their key advantage in our setting is that they can efficiently support multidimensional range queries.

On each node, Y! maintains two different R-trees for each table on that node. The first, the *current tree*, contains the tuples that currently exist in the table; when tuples appear or disappear, they are also added or removed from the current tree. The second, the *historical tree*, contains the tuples that have existed in the past. State tuples are added to the historical tree when they are removed from the current tree; event tuples, which appear only for an instant, are added to the historical tree.

The reason for having two separate trees is efficiency. It is known that the performance of R-trees degrades when elements are frequently inserted and removed because the bounding rectangles will no longer be optimal and will increasingly overlap. By separating the historical tuples (where deletions can no longer happen) from the current tuples, we can obtain a more compact tree for the former and confine fragmentation to the latter, whose tree is much smaller. As an additional benefit, since tuples are appended to the historical tree in timestamp order, splits in that tree will typically occur along the time dimension; this creates a kind of "time index" that works very well for our queries.

### 3.5.4 Pruning the historical tree

Since the historical tree is append-only, it would eventually consume all available storage. To avoid this, Y! can reclaim storage by deleting the oldest tuples from the tree. For instance, Y! can maintain a cut-off time $T_{cut}$; whenever the tree exceeds a certain pre-defined size limit, Y! can slowly advance the cut-off time and keep removing any tuples that existed before that time until enough space has been freed. To enable the user to distinguish between tuples that were absent at runtime and tuples that have been deleted from the tree, the graph construction algorithm can, whenever it accesses information beyond $T_{cut}$, annotate the corresponding vertex as potentially incomplete.

### 3.5.5 Limitations

Like other provenance systems, Y!'s explanations are limited by the information that is available in the provenance graph. For instance, Y! could trace a misconfiguration to the relevant setting, but not to the person who changed the setting (unless that information were added to the provenance graph). Y! also has no notion of a program's *intended* semantics: for instance, if a program has a concurrency bug that causes a negative event, a query for that event will yield a detailed explanation of how the given program produced that event. Only the operator can determine that the program was supposed to do something different.

## 3.6 Case Studies

In this section, we describe how we have applied Y! to two application domains: software-defined networks (SDN) and BGP routing. We chose these domains partly because they yield interesting debugging challenges, and partly because they do not already involve declarative code (applying Y! to NDlog applications is straightforward!). We illustrate two different implementation strategies: automatically extracting declarative rules from existing code (for SDN) and writing a declarative description of an existing implementation (for BGP). We report results from several specific debugging scenarios in Section 3.7.

### 3.6.1 SDN debugging

Our first case study is SDN debugging: as others [57] have pointed out, better debugging support for SDNs is urgently needed. This scenario is challenging for Y! because SDNs can have almost arbitrary control programs, and because these programs are typically written in non-declarative languages. Provenance can be extracted directly from imperative programs [97], but switching to a different programming model would require some adjustments to our provenance graph. Hence, we use automated transformation to extract declarative rules from existing SDN programs.

**Language: Pyretic** We chose to focus on the Pyretic language [96]. We begin by briefly reviewing some key features of Pyretic that are relevant here. For details, please see [96].

Pyretic programs can define a mix of *static* policies, which are immutable, and *dynamic* policies, which can change at runtime based on system events. Figure 3.3 shows a summary of the relevant syntax. A static policy consists of *actions*, e.g., for forwarding packets to a specific port (`fwd(port)`), and *predicates* that restrict these actions to certain types of packets, e.g., to packets with certain header values (`match(h=v)`). Two policies `a` and `b` can be combined through *parallel composition*

**Primitive actions:**

```
A  ::= drop | passthrough | fwd(port) | flood |
       push(h=v) | pop(h) | move(h1=h2)
```

**Predicates:**

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

**Query policies:**

```
Q  ::= packets(limit,[h]) | counts(every,[h])
```

**Policies:**

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

Figure 3.3: Static Pyretic syntax (from [96]).

(`a|b`), meaning that `a` and `b` should be applied to separate copies of each packet, and/or through *sequential composition* (`a>>b`), meaning that `a` should be applied to incoming first, and `b` should then be applied to any packet(s) that `a` may produce. For instance, `match(inport=1)>>(fwd(2)|fwd(3))` says that packets that arrive on port 1 should be forwarded to both ports 2 and 3.

Dynamic policies are based on *queries*. A query describes packets or statistics of interest – for instance, packets for which no policy has been defined yet. When a query returns new data, a callback function is invoked that can modify the policy. Figure 3.4 (taken from [96]) shows a simple self-learning switch that queries for packets with unknown MAC addresses; when a packet with a new source MAC $m$ is observed on port $p$, the policy is updated to forward future packets with destination MAC $m$ to port $p$.

Pyretic has other features besides these, and providing comprehensive support for them is beyond the scope of our case study. Here, our goal is to support an interesting subset, to demonstrate that our approach is feasible.

**Translation to NDlog:** Our Pyretic frontend transforms all static policies into a "normal form" that consists of groups of parallel "atoms" (with a sequence of matches and a single action) that are arranged sequentially. This form easily translates to OpenFlow wildcard entries: we can give the highest priority to the atoms in the first

```
def learn(self):
  def update(pkt):
    self.P = if_(match(dstmac=pkt['srcmac']),
    switch=pkt['switch']),
      fwd(pkt['inport']), self.P)
    q = packets(1,['srcmac','switch'])
    q.when(update)
    self.P = flood | q
def main():
  return dynamic(learn)()
```

Figure 3.4: Self-learning switch in Pyretic (from [96]).

group, and assign further priorities to the following groups in descending order. To match Pyretic's behavior, we do not install the wildcard entries in the switch directly, but rather keep them as base tuples in a special `MacroRule` table in the controller. A second stage then matches incoming packets from the switches against this table, and generates the corresponding microflow entries (without wildcards), which are then sent to the switch.

For each query policy, the frontend creates a separate table and a rule that sends incoming packets to this table if they match the query. The trigger is evaluated using NDlog aggregations; for instance, waiting for a certain number of packets is implemented with NDlog's `count<>` operator.

Our frontend supports dynamic policies that append new logic in response to external events. These are essentially translated to a single NDlog rule that is triggered by the relevant external event (specified as a query policy) and computes and installs the new entry. The self-learning switch from Figure 3.4 is an example of such a policy; Figure 3.5 shows the rule that it is translated to. The rule directly corresponds to the `if-then` part in Figure 3.4, which forwards packets to newly observed MAC addresses to the correct port, and otherwise (in the `else` branch) falls back on the existing policy. With such translation, it is easy to see the provenance of a dynamic policy change: it is simply the packet that triggered the change.

```
MacroRule(@C,sw,inPort0,dstMAC0,act,Prio0) :-
  UpdateEvent(@C,sw,srcMac), HighestP(@C,Prio),
  PktIn(@sw,inPort1,srcMAC,dstMAC1), inPort0=*,
  dstMAC0=srcMAC, act=fwd(inPort1), Prio0=Prio1+10
```

Figure 3.5: NDlog translation of the self-learning switch.

### 3.6.2 BGP debugging

Our second case study focuses on BGP. There is a rich literature on BGP root-cause analysis, and a variety of complex real-world problems have been documented. Here, we focus exclusively on the question whether Y! can be used to diagnose BGP problems with negative symptoms, and we ignore many other interesting questions, e.g., about incentives and incremental deployment. (Briefly, we believe that the required infrastructure and the privacy implications would be roughly comparable to those of [124]; in a partial deployment, some queries would return partial answers that are truncated at the first vertex from a network outside the deployment.)

To apply Y!, we follow the approach from [145] and write a simple declarative program that describes how the BGP control plane makes routing decisions. Our implementation is based on an NDlog encoding of a general path vector protocol provided by the authors of [128]; since this code was generic and contained no specific policies, we extended it by adding simple routing policies that respect the Gao-Rexford guidelines and import/export filters that implements the valley-free constraint. This yielded essentially a declarative specification of an ISP's routing behavior. With this, we could capture BGP message traces from unmodified routers, as described in [145], and infer the provenance of the routing decisions by replaying the messages to our program.

## 3.7 Evaluation

In this section, we report results from our experimental evaluation of Y! in the context of SDNs and BGP. Our experiments are designed to answer two high-level questions: 1) is negative provenance useful for debugging realistic problems? and 2) what is the cost for maintaining and querying negative provenance?

We ran our experiments on a Dell OptiPlex 9020 workstation, which has a 8-core 3.40 GHz Intel i7-4770 CPU with 16 GB of RAM. The OS was Ubuntu 13.04, and the kernel version was 3.8.0.

### 3.7.1 Prototype implementation

For our experiments, we built a prototype of Y! based on the RapidNet declarative networking engine [84]. We instrumented RapidNet to capture provenance for the NDlog programs it executes, and we added provenance storage based on the R-tree implementation from [56]. To experiment with SDNs, we set up networks in Mininet [78]. Since NDlog is not among the supported controllers, we wrote a simple proxy for Trema [127] that translates controller messages to NDlog tuples and vice versa. To capture the provenance of packets flowing through the network, we set up port mirroring on the virtual switches and used `libpcap` to record packet traces on the mirrored ports. Since Y!'s provenance graphs only use the packet headers, we capture only the first 96 bytes of header and its timestamp.

To demonstrate that our approach does not substantially affect throughput and latency, we also built a special Trema extension that can capture the provenance directly, without involving RapidNet. This extension was used for some of experiments in Section 3.7.5, as noted there. Other than that, we focused more on functionality and complexity than on optimizing performance; others have already shown that provenance can be captured at scale [82], and the information Y! records is not substantially different from theirs – Y! merely uses it in a different way.

| | | |
|---|---|---|
| Q1 | SDN1 | NAPPEAR($[t_1,t_2]$, packet($@D$, PROTO=HTTP)) |
| Q2 | SDN2 | NAPPEAR($[t_1,t_2]$, packet($@D$, PROTO=ICMP)) |
| Q3 | SDN3 | NAPPEAR($[t_1,t_2]$, packet($@D$, PROTO=SQL)) |
| Q4 | SDN2 | APPEAR ($t_3$, packet($@D$, PROTO=ICMP)) |
| Q5 | SDN3 | APPEAR ($t_3$, packet($@D$, PROTO=SQL)) |
| Q6 | BGP1 | NAPPEAR($[t_1,t_2]$, bestroute($@AS2$, TO=AS7)) |
| Q7 | BGP2 | NAPPEAR($[t_1,t_2]$, packet($@AS7$, SRC=AS2)) |
| Q8 | BGP3 | NAPPEAR($[t_1,t_2]$, bestroute($@AS2$, TO=AS7)) |
| Q9 | BGP4 | NAPPEAR($[t_1,t_2]$, bestroute($@AS2$, TO=AS7)) |

Table 3.2: Queries we used in our experiments.

### 3.7.2 Usability: SDN debugging

For our SDN experiments, we used Mininet to create the following three representative SDN scenarios:

- **SDN1: Broken flow entry.** A server receives no requests because an overly general flow entry redirects them to a different server (Section 3.2.2).
- **SDN2: MAC spoofing.** A host, which is connected to the self-learning switch from Figure 3.4, receives no responses to its DNS lookups because another machine has spoofed its MAC address.
- **SDN3: Incorrect ACL.** A firewall, intended to allow Internet users to access a web server $W$ and internal users a database $D$, is misconfigured: Internet users can access only $D$, and internal users only $W$.

Each scenario consists of four hosts and three switches. For all three scenarios, we use Pyretic programs that have been translated to NDlog rules (Section 3.6.1) and are executed on RapidNet; however, we verified that each problem also occurs with the original Pyretic runtime. Note that, in all three scenarios, positive provenance cannot be used to diagnose the problem because there is no state whose provenance could be queried.

The first three queries we ask are the natural ones in these scenarios: in SDN1, we ask why the web server is not receiving any requests (Q1); in SDN2, we ask

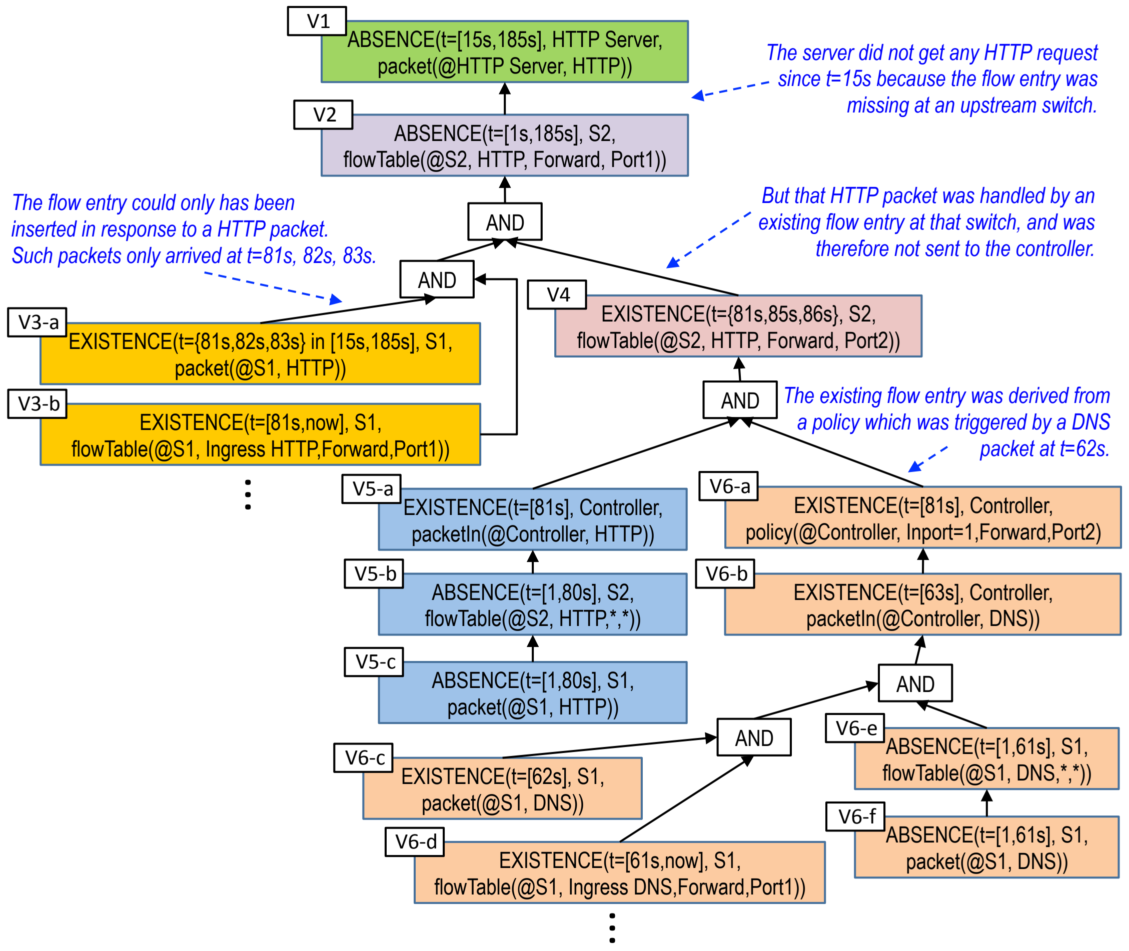

V1
ABSENCE(t=[15s,185s], HTTP Server, packet(@HTTP Server, HTTP))
The server did not get any HTTP request since t=15s because the flow entry was missing at an upstream switch.
V2
ABSENCE(t=[1s,185s], S2, flowTable(@S2, HTTP, Forward, Port1))
AND
The flow entry could only has been inserted in response to a HTTP packet. Such packets only arrived at t=81s, 82s, 83s.
But that HTTP packet was handled by an existing flow entry at that switch, and was therefore not sent to the controller.
AND
V3-a
EXISTENCE(t={81s,82s,83s} in [15s,185s], S1, packet(@S1, HTTP))
V4
EXISTENCE(t={81s,85s,86s}, S2, flowTable(@S2, HTTP, Forward, Port2))
V3-b
EXISTENCE(t=[81s,now], S1, flowTable(@S1, Ingress HTTP,Forward,Port1))
AND
The existing flow entry was derived from a policy which was triggered by a DNS packet at t=62s.
V5-a
EXISTENCE(t=[81s], Controller, packetIn(@Controller, HTTP))
V6-a
EXISTENCE(t=[81s], Controller, policy(@Controller, Inport=1,Forward,Port2))
V5-b
ABSENCE(t=[1,80s], S2, flowTable(@S2, HTTP,*,*))
V6-b
EXISTENCE(t=[63s], Controller, packetIn(@Controller, DNS))
V5-c
ABSENCE(t=[1,80s], S1, packet(@S1, HTTP))
AND
AND
V6-e
ABSENCE(t=[1,61s], S1, flowTable(@S1, DNS,*,*))
V6-c
EXISTENCE(t=[62s], S1, packet(@S1, DNS))
V6-f
ABSENCE(t=[1,61s], S1, packet(@S1, DNS))
V6-d
EXISTENCE(t=[61s,now], S1, flowTable(@S1, Ingress DNS,Forward,Port1))

Figure 3.6: Answer to Q1, as returned by Y!.

why there are no responses to the DNS lookups (Q2); and in SDN3, we ask why the internal users cannot get responses from the database (Q3). To exercise Y!'s support for positive provenance, we also ask two positive queries: why a host in SDN2 *did* receive a certain ICMP packet (Q4), and why the internal database is receiving connections from the Internet (Q5). To get a sense of how useful negative provenance would be for debugging realistic problems in SDNs, we ran diagnostic queries in our three scenarios and examined the resulting provenance. The first five rows in Table 3.2 show the queries we used. The full responses are in Appendix A.2; here, we focus on Q1 from scenario SDN1, which asks why HTTP requests are no longer appearing at the web server.

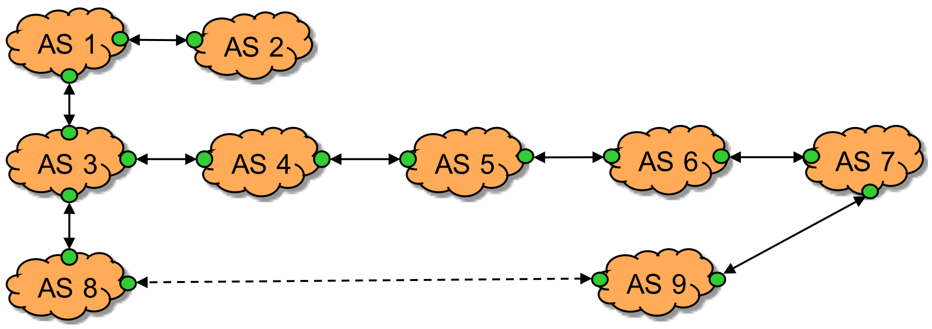


Figure 3.7: Topology for the BGP1 scenario.

Figure 3.6 shows the provenance generated by Y! for Q1. The explanation reads as follows: HTTP requests did not arrive at the HTTP server (V1) because there was no suitable flow entry at the switch (V2). Such an entry could only have been installed if a HTTP packet had arrived (V3a+b) and caused a table miss, but the latter did not happen because there already was an entry – the low-priority entry (V4) – that was forwarding HTTP packets to a different port (V5a-c), and that entry had been installed in response to an earlier DNS packet (V6a-f). We believe that "backtraces" of this kind would be useful in debugging complex problems.

### 3.7.3 Usability: BGP debugging

For our BGP experiments, we picked four real BGP failure scenarios from our survey (Section 3.2.3):

- **BGP1: Off-path change.** In the topology from Figure 3.7, AS 2 initially has a route to AS 7 via AS 1,3,4,5,6, but loses that route when a new link is added between AS 8 and AS 9 (neither of which is on the path). This is a variant of a scenario from [124].
- **BGP2: Black hole.** A router advertises a spurious /32 route to a certain host, creating a "black hole" and preventing that host from responding to queries.
- **BGP3: Link failure.** An ISP temporarily loses connectivity, due to a link failure at one of its upstream ASes.

- **BGP4: Bogon list.** A network cannot reach a number of local and federal government sites from its newly acquired IP prefix because that prefix was on the bogon list earlier.

We set up small BGP topologies, using between 4 and 18 simulated routers, to recreate each scenario. In each scenario, we then asked one query: why AS 2 in scenario BGP1 has no route to AS 7 (Q6), why a host in scenario BGP2 cannot reach the black-holed host (Q7), why the ISP in scenario BGP3 cannot reach a certain AS (Q8), and why the network in scenario BGP4 cannot connect to a particular site (Q9). Table 3.2 shows the specific queries. As expected, Y! generated the correct response in all four scenarios; here, we focus on one specific query (Q6/BGP1) due to lack of space. The other results (available in Appendix A.2) are qualitatively similar.

Figure 3.8 shows the provenance generated by Y! for query Q6. The explanation reads as follows: AS 2 has no route to AS 7 (V1-a) because its previous route expired (V1-b) and it has not received any new advertisements from its provider AS 1 (V1-c). This is because AS 1 itself has no suitable route: its peer AS 3 stopped advertising routes to AS 7 (V2a-c) because AS 3 only advertises customer routes to AS 7 due to the valley-free constraint (V3-a). AS 3 previously had a customer route but it disappeared (V3-b). Although AS 3 continues to receive the customer route from AS 4 (V3-c), the peer route through AS 8 (V4) is preferred because it has a shorter AS path (V3-d). The provenance of the peer route could be further explored by following the graph beyond V4.

### 3.7.4 Complexity

Recall from Section 3.4 that Y! uses a number of heuristics to simplify the provenance before it is shown to the user. To quantify how well these heuristics work, we reran the queries in Table 3.2 with different subsets of the heuristics disabled, and we measured the size of the corresponding provenance graphs.

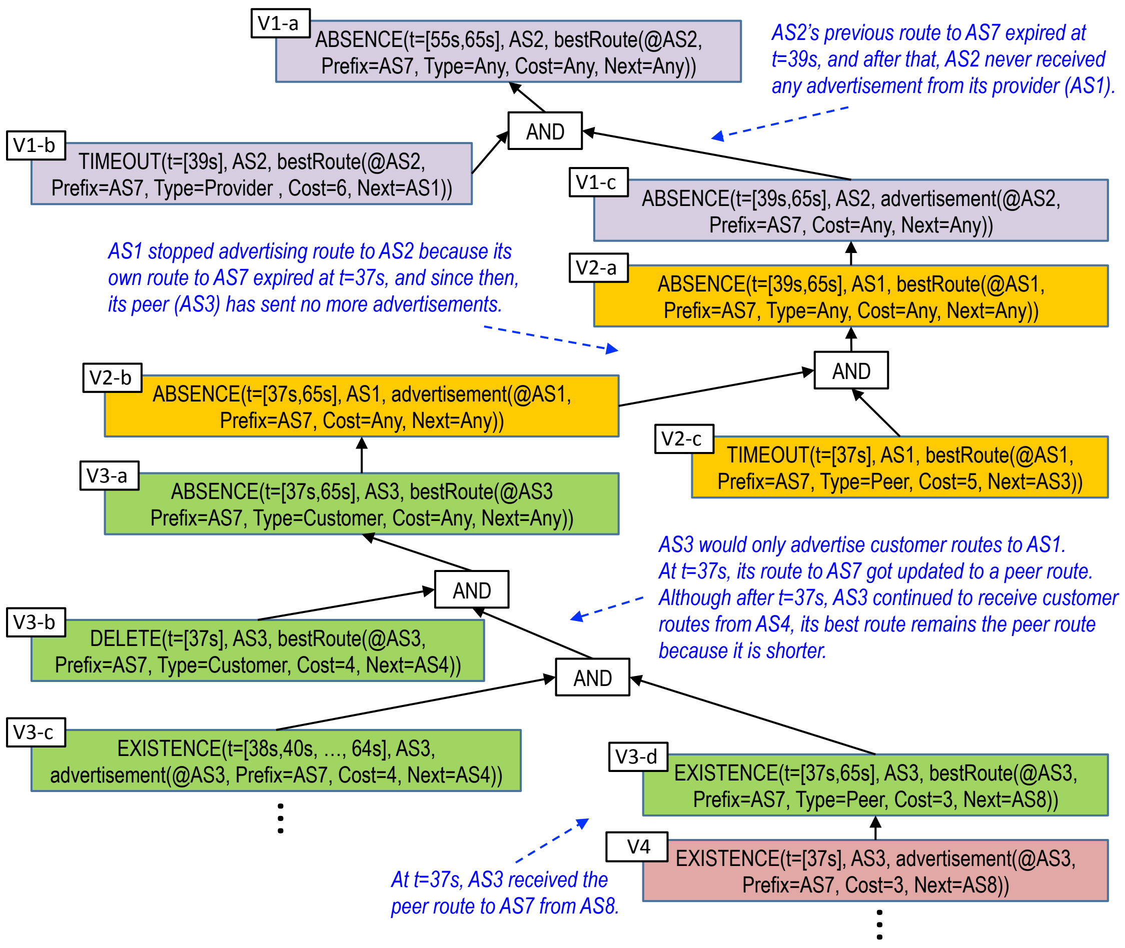

V1-a
ABSENCE(t=[55s,65s], AS2, bestRoute(@AS2, Prefix=AS7, Type=Any, Cost=Any, Next=Any))
AS2's previous route to AS7 expired at t=39s, and after that, AS2 never received any advertisement from its provider (AS1).
AND
V1-b
TIMEOUT(t=[39s], AS2, bestRoute(@AS2, Prefix=AS7, Type=Provider , Cost=6, Next=AS1))
V1-c
ABSENCE(t=[39s,65s], AS2, advertisement(@AS2, Prefix=AS7, Cost=Any, Next=Any))
AS1 stopped advertising route to AS2 because its own route to AS7 expired at t=37s, and since then, its peer (AS3) has sent no more advertisements.
V2-a
ABSENCE(t=[39s,65s], AS1, bestRoute(@AS1, Prefix=AS7, Type=Any, Cost=Any, Next=Any))
V2-b
ABSENCE(t=[37s,65s], AS1, advertisement(@AS1, Prefix=AS7, Cost=Any, Next=Any))
AND
V2-c
TIMEOUT(t=[37s], AS1, bestRoute(@AS1, Prefix=AS7, Type=Peer, Cost=5, Next=AS3))
V3-a
ABSENCE(t=[37s,65s], AS3, bestRoute(@AS3 Prefix=AS7, Type=Customer, Cost=Any, Next=Any))
AS3 would only advertise customer routes to AS1. At t=37s, its route to AS7 got updated to a peer route. Although after t=37s, AS3 continued to receive customer routes from AS4, its best route remains the peer route because it is shorter.
AND
V3-b
DELETE(t=[37s], AS3, bestRoute(@AS3, Prefix=AS7, Type=Customer, Cost=4, Next=AS4))
AND
V3-c
EXISTENCE(t=[38s,40s, ..., 64s], AS3, advertisement(@AS3, Prefix=AS7, Cost=4, Next=AS4))
V3-d
EXISTENCE(t=[37s,65s], AS3, bestRoute(@AS3, Prefix=AS7, Type=Peer, Cost=3, Next=AS8))
V4
EXISTENCE(t=[37s], AS3, advertisement(@AS3, Prefix=AS7, Cost=3, Next=AS8))
At t=37s, AS3 received the peer route to AS7 from AS8.

Figure 3.8: Answer to Q6, as returned by Y!.

Figure 3.9 shows our results. Without heuristics, the provenance contained between 55 and 386 vertices, which would be difficult for a human user to interpret. The pruning heuristics from Section 3.4.1 generally remove about half the vertices, but the size of the provenance remains substantial. However, the super-vertices from Section 3.4.2 are able to shrink the provenance considerably, to between 4 and 24 vertices, which should be much easier to interpret.

To explain where the large reduction comes from, we show the raw provenance tree (without the heuristics) for Q1 in Figure 3.10. The structure of this tree is typical of the ones we have generated: a "skeleton" of long causal chains, which typically correspond to messages and events propagating across several nodes, and a

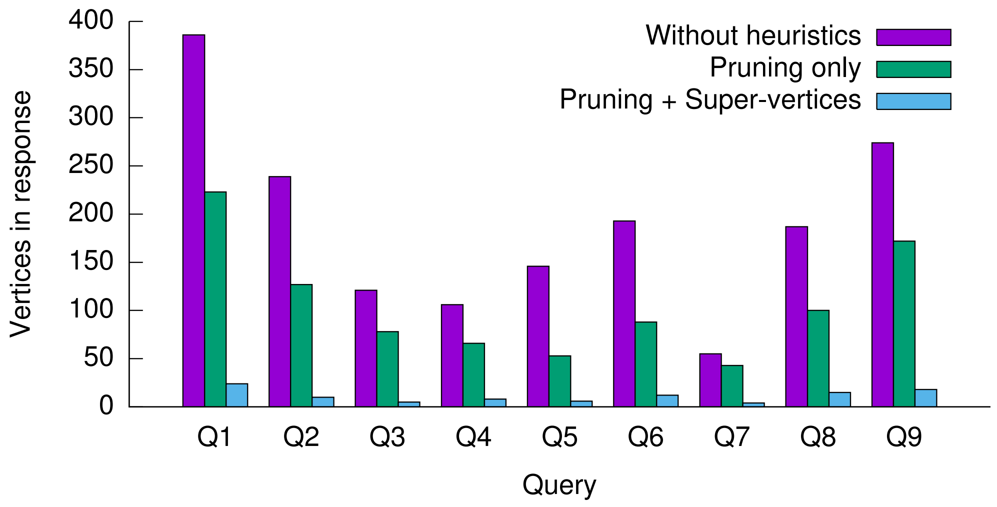


Figure 3.9: Size of the provenance with some or all heuristics disabled.

large number of small branches. The pruning heuristics remove most of the smaller branches, while the super-vertices "collapse" the long chains in the skeleton. In combination, this yields the much-simplified tree from Figure 3.6.

### 3.7.5 Run-time overhead

**Disk storage:** Y! maintains two data structures on disk: the packet traces and the historical R-tree. The size of the former depends on the number of captured packets; each packet consumes 120 bytes of storage. To estimate the size of the latter, we ran a program that randomly inserted and removed `flowEntry` tuples, and we measured the number of bytes per update. We found that, for trees with $10^3$ to $10^6$ updates, each update consumed about 450 byte of storage on average.

These numbers allow us to estimate the storage requirements in a production network. We assume that there are 400 switches that each handle 45 packets per second, and that the SDN controller generates 1,200 flow entries per second. Under these assumptions, a commodity hard disk with 1TB capacity could easily hold the provenance for the most recent 36 hours. If necessary, the storage cost could easily be reduced further, e.g., by compressing the data, by storing only a subset of the

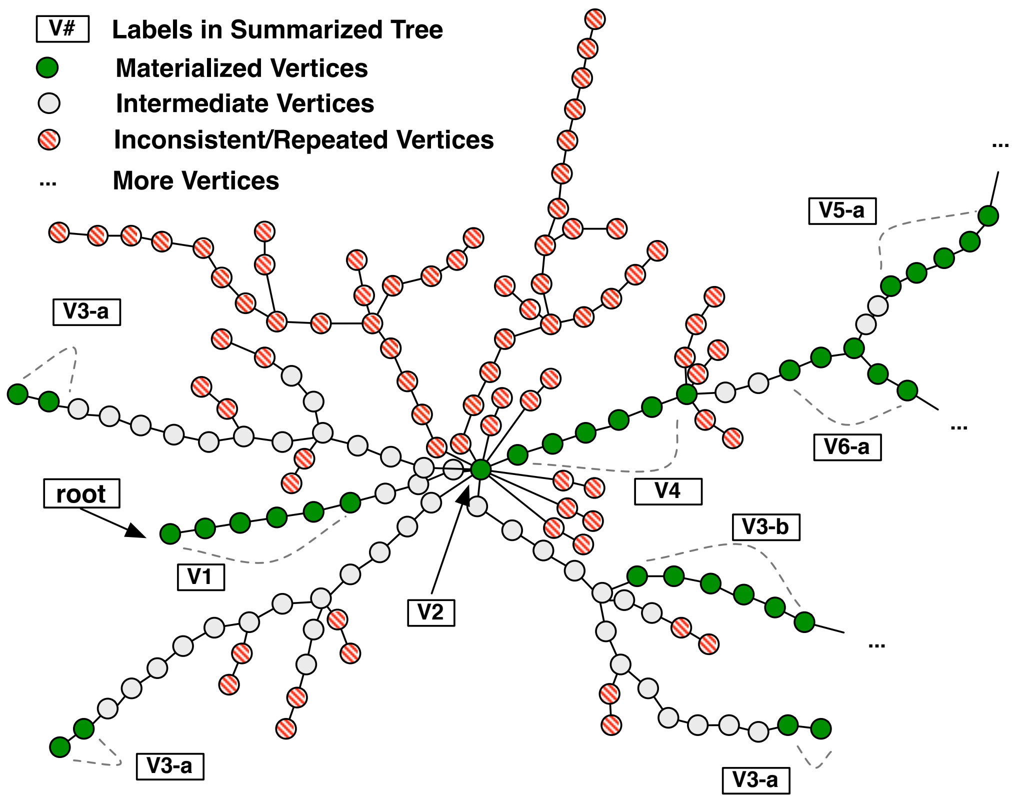


Figure 3.10: Raw provenance for query Q1 before post-processing.

header fields, and/or by removing redundant copies of the headers in each flow.

**Latency and throughput:** Maintaining provenance requires some additional processing on the SDN controller, which increases the latency of responses and decreases throughput. We first measured this effect in our prototype by using `Cbench` to send streams of PacketIn messages, which is a current standard in evaluating OpenFlow controllers [35]. We found that the 95th percentile latency increased by 29%, from 48.38 ms to 62.63 ms, when Y! was enabled; throughput dropped by 14%, from 56.0 to 48.4 requests per second.

However, these results are difficult to generalize because RapidNet's performance as an SDN controller is not competitive with state-of-the-art controllers, even without Y!. We therefore repeated the experiment with our Y! extension for native Trema (Section 3.7.1); here, adding Y! increased the average latency by only 1.6%, to 33 microseconds, and decreased the average throughput by 8.9%, to 100,540 Pack-

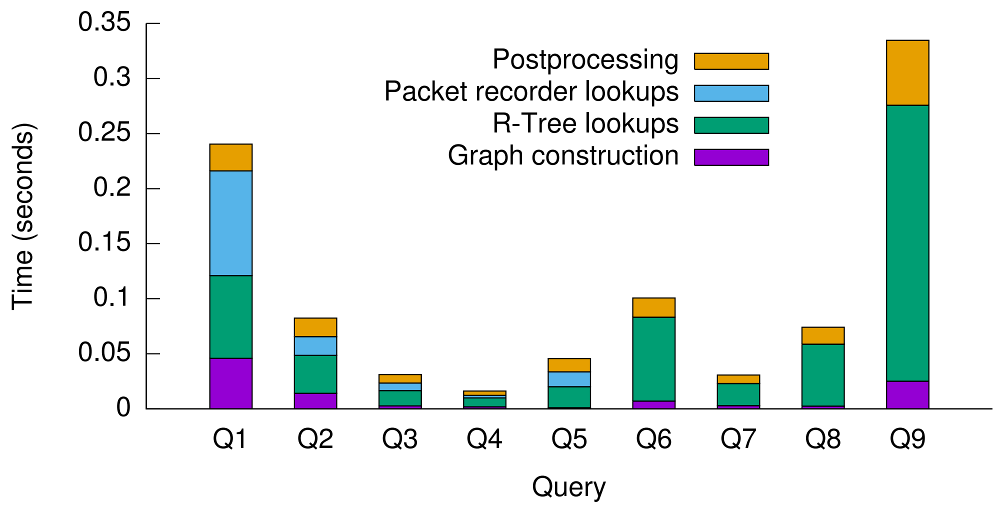


Figure 3.11: Turnaround time for the queries in Table 3.2.

etIn messages per second. We note that this comparison is slightly unfair because we manually instrumented a specific Trema program to work with our extension, whereas the RapidNet prototype can work with any program. However, adding instrumentation to Trema programs is not difficult and could be automated. More generally, our results suggest that capturing provenance is not inherently expensive, and that an optimized RapidNet could potentially do a lot better.

### 3.7.6 Query processing speed

When the user issues a provenance query, Y! must recursively construct the response using the process from Section 3.3.4 and then post-process it using the heuristics from Section 3.4. Since debugging is an interactive process, a quick response is important. To see whether Y! can meet this requirement, we measured the turnaround time for the queries in Table 3.2, as well as the fraction of time consumed by Y!'s major components.

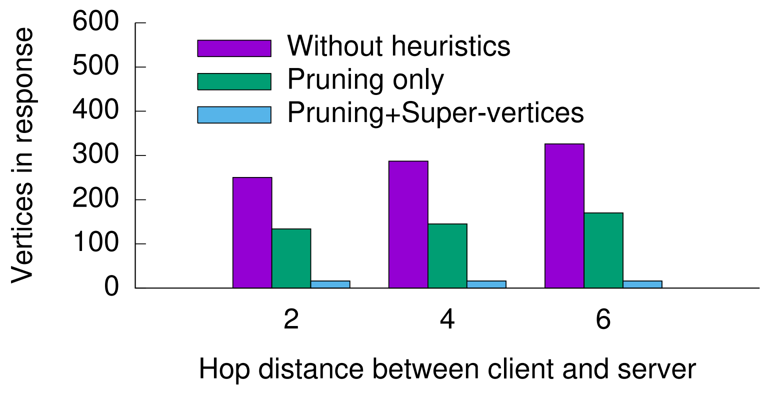
Figure 3.11 shows our results. We make two high-level observations. First, the turnaround time is dominated by R-tree and packet recorder lookups. This is expected because the graph construction algorithm itself is not very complex. Second, although the queries vary in complexity and thus their turnaround times are difficult to compare, we observe that none of them took more than one second; the most expensive query was Q9, which took 0.33 seconds to complete.

### 3.7.7 Scalability

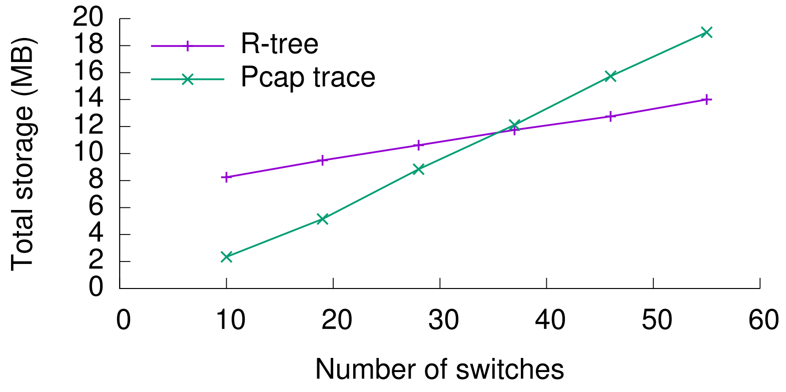
We do not yet have experience with Y!, or negative provenance, in a large-scale deployment. However, we have done a number of experiments to get an initial impression of its scalability.

**Complexity:** In our first experiment, we tested whether the complexity of the provenance increases with the number of possible traffic sources. We simulated a four-layer fat-tree topology with 15 switches, and we placed the client and the server on different leaves, to vary the hop distance between them from 2 to 6. Our results for running the learning-switch query (Q1) are shown in Figure 3.12(a) (the bars are analogous to Figure 3.9). As expected, the size of the raw provenance for Q1 grew substantially – from 250 to 386 vertices – because 1) there number of possible sources for the missing traffic increased, because each additional hop brings additional branches on the backtrace path and 2) each additional hop required extra vertices to be represented in the provenance. But the first effect was mitigated by our pruning heuristics, since the extra sources were inconsistent with the network state, and the second effect was addressed by the summarization, which merged the vertices along the propagation path into a single super-vertex. Once these heuristics had been applied, the size of the provenance was 16 vertices, independent of the number of hops.

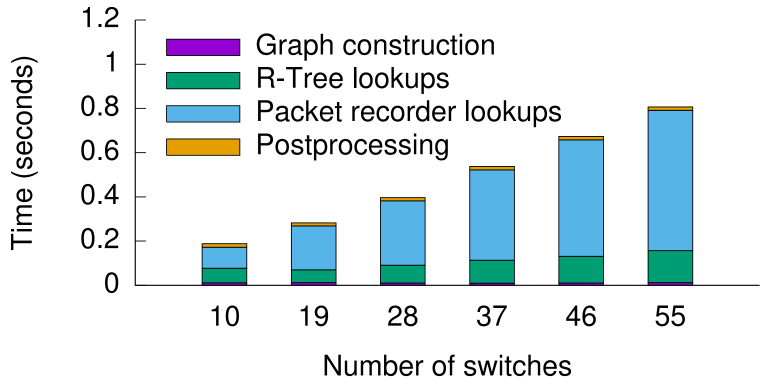
**Storage:** In our second experiment, we simulated three-layer fat-tree topologies of different sizes (i.e., with different node degrees); each edge switch was connected to a fixed number of active hosts that were constantly sending HTTP requests to the server. Figure 3.12(b) shows how Y!'s storage requirements grew with the number of switches in the network. As expected, the size of both the pcap trace and the R-tree



(a) Size of query results



(b) Use of storage space



(c) Query turnaround time

Figure 3.12: Scalability results for the fat-tree topology.

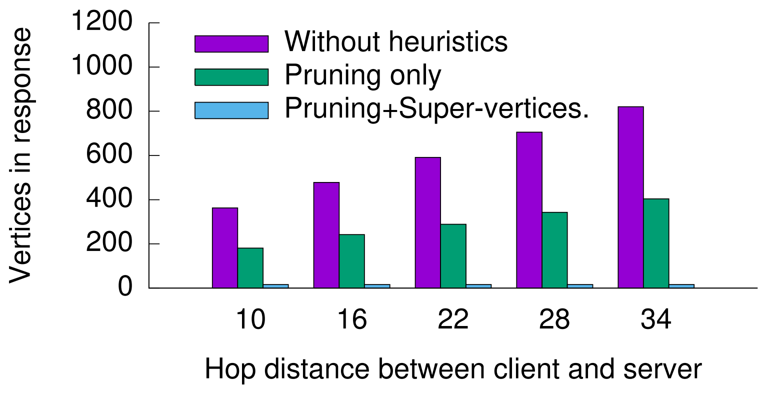
was roughly proportional to the size of the network; this is expected because a) each new switch added a fixed number of hosts, and b) the depth of the tree, and thus the hop count between the server and its clients, remained constant. Generally, the storage requirement depends on the rate at which events of interest (packet transmissions, routing changes, etc.) are captured, as well as on the time for which these records are retained.

**Query speed:** Our third experiment is analogous to the second, except that we issued a query at the end and measured its turn-around time. Figure 3.12(c) shows our results. The dominant cost was the time it took to find packets in the pcap trace; the R-tree lookups were much faster, and the time needed to construct and post-process the graph was so small that it is difficult to see in the figure. Overall, the lookup time was below one second even for the largest network we tried.

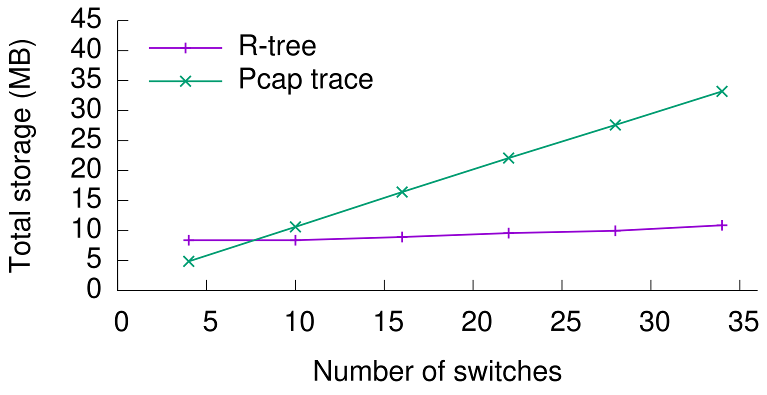
So far, we have discussed the result on fat-tree topologies, where the network diameter is bounded by tree height. To get a sense of how Y! scales in networks with even larger diameters, we focused on a linear topology. In each experiment, we changed the number of hops between the HTTP server and client. The other settings remain the same. For example, the number of packets sent through the network is stable.

**Complexity:** We tested whether the complexity of the provenance (number of vertices) increases with the hop count between the server and the client. Our results are shown in Figure 3.13(a). As expected, the size of the raw provenance for Q1 grew substantially from 363 to 820 vertices. However once all the heuristics had been applied, the size of the provenance was 16 vertices, independent of the hop count. The reason is similar to what we described in Section 3.7.7.

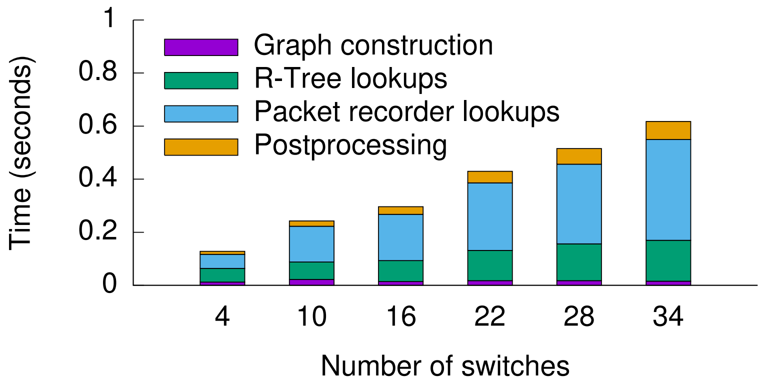
**Storage:** Figure 3.13(b) shows how Y!'s storage requirements grew with the number of switches in the network. As expected, the size of both the pcap trace and the R-tree was roughly proportional to the size of the network. The pcap trace grows because when network grows, each packet will traverse more switches, and each switch will



(a) Size of query results



(b) Use of storage space



(c) Query turnaround time

Figure 3.13: Scalability results for the linear topology.

append the event to its trace. The R-tree grows because handling packets at each additional switch will generate state changes, which are recorded in R-trees.

**Query speed:** We issued a query Q2 at the end and measured its turn-around time. Figure 3.13(c) shows our results: the query speed scales linearly with the number of switches. The dominant cost was the time it took to find packets in the `pcap` trace; the R-tree lookups were much faster, and the time needed to construct and post-process the graph was so small that it is difficult to see in the figure. This is expected because the `pcap` trace query is not optimized. As we mentioned in Section 3.7.7, an additional time index should reduce the time significantly.

In all our experiments, the complexity of explanation is reduced considerably by Y!'s post-processing heuristics, which reduced the number of vertices by more than an order of magnitude. Y!'s main run-time cost is the storage it needs to maintain a history of the Y!'s past states.

**Possible optimizations:** Since our implementation has not been optimized, some of the costs could grow quickly in a large-scale deployment. For instance, in a data center with 400 switches that handle 1 Gbps of traffic each, our simple approach of recording `pcap` traces at each switch would consume approximately 30 GB of storage *per second* for the date center, or about 75 MB for each switch. Packet recorder lookups, which compromise a major portion of query latency, in such a large trace would be limited by disk read throughput, and could take minutes. However, we note that there are several ways to reduce these costs; for instance, techniques from the database literature – e.g., a simple time index – could be used to speed up the lookups, and the storage cost could be reduced by applying filters.

### 3.7.8 Summary

Our results show that Y! – and, more generally, negative provenance – can be a useful tool for diagnosing problems in networks: the provenance of the issues we looked at was compact and readable, and Y! was able to find it in less than a second in each case. Our results also show that the readability is aided considerably by Y!'s post-processing heuristics, which reduced the number of vertices by more than an order of magnitude. Y!'s main run-time cost is the storage it needs to maintain a history of the system's past states, but a commodity hard-drive should be more than sufficient to keep this history for more than a day.

## 3.8 Related Work

We have discussed the literature on provenance and network debugging in Chapter 2. Next, we briefly expand on papers that have considered negative provenance.

In the database literature, Huang et al. [62] and Tiresias [94] focus on instance-based explanations for missing answers, that is, how to obtain the missing answers by making modifications to the value of base instances (tuples); Why-Not [21] and ConQueR [126] provide query-based explanations for SQL queries, which reveal over-constrained conditions in the queries and suggest modifications to them. Note that databases typically consider multiple query plans to efficiently execute a SQL query, whereas, in our setting, the programs are always executed using one particular "query plan". Thus, finding negative provenance for general databases is more challenging because it needs to account for all possible query plans. For example, Why-Not [21] requires as input a specific query plan to find culprit conditions; Tiresias [94] is agnostic of query plans but can exhibit substantial query turnaround.

To the best of our knowledge, Y! is the first to adapt negative provenance to distributed environments and networks. Networks do perform careful planning to efficiently and faithfully execute specifications from operators. For example, NDlog [83] exploits the commutativity of join and selection when executing rules. However, the search space of implementing common operations in networks is usually more restrictive compared to that in databases. For example, OpenFlow [91] processes packets using a pipeline of flow entries, P4 [15] compiles programs to match+action stages in series or in parallel, and NDlog [83] uses a sequence of relational operators to implement rules (Section 2.1). This limits the space of possible explanations and makes it easier for Y! to construct negative provenance efficiently and concisely.

## 3.9 Conclusion

In this chapter, we have argued that debuggers for distributed systems should not only be able to explain why an unexpected event *did* occur, but also why an expected event *did not* occur. We have shown how this can be accomplished with the concept of *negative provenance*, which so far has received relatively little attention. We have defined a formal model of negative provenance, we have presented an algorithm generating such provenance, and we have introduced Y!, a practical system that can maintain both positive and negative provenance in a distributed system and answer queries about it. Our evaluation in the context of software-defined networks and BGP suggests that negative provenance can be a useful tool for diagnosing complex problems in distributed systems.

# 4
# Meta Provenance

## 4.1 Introduction

Debugging networks is notoriously hard. The advent of software-defined networking (SDN) has added a new dimension to the problem: networks can now be controlled by programs, and, like all other programs, these programs can have bugs.

There is a substantial literature on network debugging and root cause analysis (Section 2.3). However, in practice, diagnosing the problem is only the first step. Once the root cause of a problem is known, the operator must find an effective fix that not only solves the problem at hand, but also avoids creating *new* problems elsewhere in the network. Given the complexity of modern controller programs and configuration files, finding a good fix can be as challenging as – or perhaps even more challenging than – diagnostics, and it often requires considerable expertise. Current tools offer far less help with this second step than with the first.

In this chapter, we present a step towards automated bug fixing in SDN applications. Ideally, we would like to provide a "Fix it!" button that automatically finds and fixes the root cause of an observed problem. However, removing the human operator from the loop entirely seems risky, since an automated tool cannot know the operator's intent. Therefore we opt for a slightly less ambitious goal, which is to provide the operator with a list of suggested repairs.

Our approach is to leverage and enhance the concept of network provenance (Section 2.3.2). *So far this work has considered provenance only in terms of packets and configuration data – the SDN controller program was assumed to be immutable.* This is sufficient for diagnosis, but not for repair: we must also be able to infer which parts of the controller program were responsible for an observed event, and how the event might be affected by changes to that program.

In this chapter, we take the next step and extend network provenance to *both* programs *and* data. At a high level, we accomplish this with a combination of two ideas. First, we treat programs as just another kind of data; this allows us to reason about the provenance of data not only in terms of the data it was computed from, but also in terms of the parts of the program it was computed *with*. Second, we use counterfactual reasoning to enable a form of negative provenance (Chapter 3), so that operators can ask why some condition did *not* hold (Example: "Why didn't any DNS requests arrive at the DNS server?"). This is a natural way to phrase a diagnostic query, and the resulting meta provenance is, in essence, a tree of changes (to the program and/or to configuration data) that could make the condition true.

Our approach presents three key challenges. First, there are infinitely many possible repairs to a given program (including, e.g., a complete rewrite), and not all of them will make the condition hold. To address this challenge, we show how to find suitable repairs efficiently using properties of the provenance itself. Second, even if we consider only suitable changes, there are still infinitely many possibilities. We leverage the fact that most bugs affect only a small part of the program, and that programmers tend to make certain errors more often than others [70, 106]. This allows us to rank the possible changes according to plausibility, and to explore only

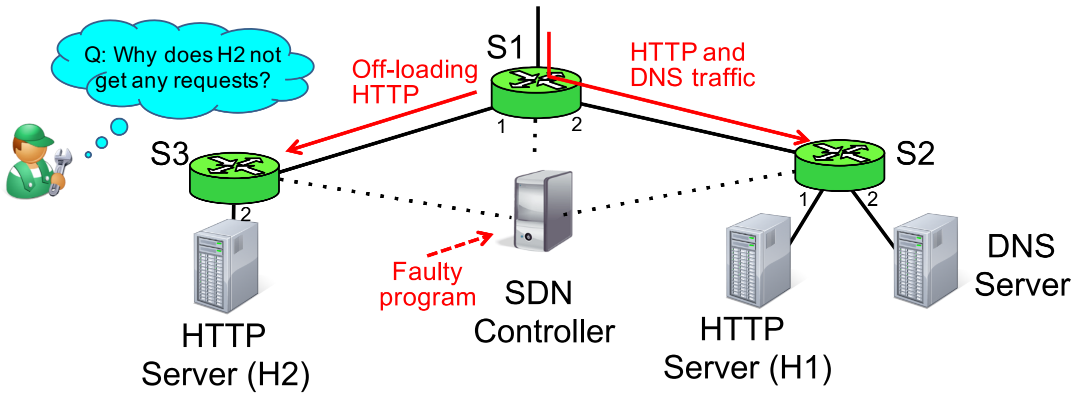


Figure 4.1: Example scenario. The primary web server (H1) is too busy, so the network offloads some traffic to a backup server (H2). The offloaded requests are never received because of a bug in the controller program.

the most plausible ones. Finally, even a small change that fixes the problem at hand might still cause problems elsewhere in the network. To avoid such fixes, we backtest them using historical information that was collected in the network. In combination, this approach enables us to produce a list of suggested repairs that 1) are small and plausible, 2) fix the problem at hand, and 3) are unlikely to affect unrelated parts of the network.

We present a concrete algorithm that can generate meta provenance for arbitrary controller programs, as well as a prototype system that can collect the necessary data in SDNs and suggest repairs. We have applied our approach to three different controller languages, and we report results from several case studies; our results show that our system can generate high-quality repairs for realistic bugs, typically in less than one minute.

## 4.2 Overview

We illustrate the problem with a simple scenario (Figure 4.1). A network operator manages an SDN that connects two web servers and a DNS server to the Internet. To balance the load, incoming web requests are forwarded to different servers based

```
r1 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), WebLoadBalancer(@C,Hdr,Prt), Swi == 1.
r2 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 1, Hdr == 53, Prt := 2.
r3 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 1, Hdr != 53, Prt := -1.
r4 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 1, Hdr != 80, Prt := -1.
r5 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 2, Hdr == 80, Prt := 1.
r6 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 2, Hdr == 53, Prt := 2.
r7 FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 2, Hdr == 80, Prt := 2.
```

Figure 4.2: Part of an SDN controller program written in NDlog: Switch S1 load-balances HTTP requests across S2 and S3 (rule `r1`), forwards DNS requests to S3 (rule `r2`); and drops all other traffic (rules `r3`–`r4`). S2 and S3 forward the traffic to the correct server based on the destination port (rules `r5`–`r7`). The bug from Section 4.2.2 is underlined.

on their source IP. At some point, the operator notices that web server H2 is not receiving any requests from the Internet.

Our goal is to build a debugger that accepts a simple specification of the observed problem (such as "H2 is not receiving any traffic on TCP port 80") and returns a) a detailed causal explanation of the problem, and b) a ranked list of suggested fixes. We consider a suggested fix to be useful if it a) fixes the specified problem and b) has few or no side-effects on the rest of the network.

### 4.2.1 Classical provenance

Since our approach involves tracking causal dependencies, we will explain it using a declarative language, specifically *network datalog (NDlog)* [83] (briefly discussed in Section 2.1), which makes these dependencies obvious. However, these dependencies are fundamental, and they exist in all the other languages that are used to program SDNs. To demonstrate this, we have applied our approach to three different languages, of which only one is declarative; for details, please see Section 4.5.9.

In NDlog, it is easy to see why a given tuple exists: if the tuple was derived using some rule `r` (e.g., `A(@X,5)`), then it must be the case that all the predicates in `r` were true (e.g., `B(@X,10)`), and all the constraints in `r` were satisfied (e.g., `10=2*5`.). This concept can be applied recursively (e.g., to explain the existence of `B(@X,10)`) until a set of base tuples is reached that cannot be explained further (e.g., configuration data or packets at border routers). The result is as a *provenance tree*, in which each vertex represents a tuple and edges represent direct causality; the root tuple is the one that is being explained, and the base tuples are the leaves. Using negative provenance (Chapter 3), we can also explain why a tuple *does not* exist, by reasoning counterfactually about how the tuple *could* have been derived.

### 4.2.2 Case study: Faulty program

We now return to the scenario in Figure 4.1. One possible reason for this situation is that the operator has made a copy-and-paste error when writing the program. Figure 4.2 shows part of the (buggy) controller program: when the operator added the second web server H2, she had to update the rules for switch S3 to forward HTTP requests to H2. Perhaps she saw that rule `r5`, which is used for sending HTTP requests from S2 to H1, seemed to do something similar, so she copied it to another rule `r7` and changed the forwarding port, but forgot to change the condition `Swi==2` to check for S3 instead of S2.

When the operator notices that no requests are arriving at H2, she can use a provenance-based debugger to get a causal explanation. Provenance trees are more useful than large packet traces or the system-wide configuration files because they only contain information that is causally related to the observed problem. But the operator is still largely on her own when interpreting the provenance information and fixing the bug.

### 4.2.3 Meta provenance

Classical provenance is inherently unable to generate fixes because it reasons about the provenance of data that was generated *by a given program*. To find a fix, we also need the ability to reason about program changes.

We propose to add this capability, in essence, *by treating the program as just another kind of data*. Thus, the provenance of a tuple that was derived via a certain rule `r` does not only consist of the tuples that triggered `r`, but also of the syntactic components of `r` itself. For instance, when generating the provenance that explains why, in the scenario from Figure 4.1, no HTTP requests are arriving at H2, we eventually reach a point where we must explain the absence of a flow table entry in switch S3 that would send HTTP packets to port #2 on that switch. At this point, we can observe that rule `r7` would *almost* have generated such a flow entry, were it not for the predicate `Swi==2`, which did not hold. We can then, analogous to negative provenance, use counterfactual reasoning to determine that the rule *would* have the desired behavior if the constant were 3 instead of 2. Thus, the fact that the constant in the predicate is 2 and not 3 should become part of the missing flow entry's meta provenance.

### 4.2.4 Challenges

An obvious challenge with this approach is that there are infinitely many possible changes to a given program: constants, predicates, and entire rules can be changed, added, or deleted. However, only a tiny subset of these changes is actually relevant. Observe that, at any point in the provenance tree, we know exactly what we need to explain – e.g., the absence of a particular flow entry for HTTP traffic. Thus, we need not consider changes to the destination port in the header (`Hdr`) in `r7` (because that predicate is already true) or to unrelated rules that do not generate flow entries.

Of course, the number of relevant changes, and thus the size of any meta provenance graph, is still infinite. This does mean that we can never fully draw or materialize it – but there is also no need for that. Studies have shown that "real" bugs are often small [106], such as off-by-one errors or missing predicates. Thus, it seems useful to define a cost metric for changes (perhaps based on the number of syntactic elements they touch), and to explore only the "cheapest" changes.

Third, it is not always obvious what to change in order to achieve a desired effect. For instance, when changing `Swi==2` in the above example, why did we change the constant to 3 and not, say, 4? Fortunately, we can use existing tools, such as SMT solvers, that can enumerate possibilities quickly for the more difficult cases.

```
program ← rule | rule program
rule    ← id func ":-" funcs "," sels "," assigns "."
id      ← (0-9a-zA-Z)+
funcs   ← func | func func
func    ← table "(" location "," arg "," arg ")"
table   ← (a-zA-Z)+
assigns ← assign | assign assigns
assign  ← arg ":=" expr
sels    ← sel "," sel
sel     ← expr opr expr
opr     ← == | < | > | !=
expr    ← integer | arg
```

Figure 4.3: $\mu$Dlog grammar.

Finally, even if a change fixes the problem at hand, we cannot be sure that it will not cause new problems elsewhere. Such side-effects are difficult to capture in the meta provenance itself, but we show that they can be estimated in another way, namely by backtesting changes with historical information from the network.

## 4.3 Meta Provenance

In this section, we show how to derive a simple meta provenance graph for both positive and negative events. We begin with a basic provenance graph for declarative programs, and then extend it to obtain meta provenance.

For ease of exposition, we explain our approach using a toy language, which we call $\mu$Dlog. In essence, $\mu$Dlog is a heavily simplified variant of NDlog: all tables have exactly two columns; all rules have one or two predicates and exactly two selection predicates, all selection predicates must use one of four operators (<, >, `!=`, ==), and there are no data types other than integers. The grammar of this simple language is shown in Figure 4.3. The controller program from our running example (in Figure 4.2) happens to already be a valid $\mu$Dlog program.

```
h1 Tuple(@C,Tab,Val1,Val2) :- Base(@C,Tab,Val1,Val2).
h2 Tuple(@L,Tab,Val1,Val2) :-
      HeadFunc(@C,Rul,Tab,Loc,Arg1,Arg2), HeadVal(@C,Rul,JID,Loc,L), Val == True,
      HeadVal(@C,Rul,JID1,Arg1,Val1), HeadVal(@C,Rul,JID2,Arg2,Val2), Sel(@C,Rul,JID,SID,Val),
      Val' == True, Sel(@C,Rul,JID,SID',Val'), True == f_match(JID1,JID), True == f_match(JID2,JID),
      SID != SID'.
p1 TuplePred(@C,Rul,Tab,Arg1,Arg2,Val1,Val2) :-
      Tuple(@C,Tab,Val1,Val2), PredFunc(@C,Rul,Tab,Arg1,Arg2).
p2 PredFuncCount(@C,Rul,Count<N>) :- PredFunc(@C,Rul,Tab,Arg1,Arg2).
j1 Join4(@C,Rul,JID,Arg1,Arg2,Arg3,Arg4,Val1,Val2,Val3,Val4) :-
      TuplePred(@C,Rul,Tab,Arg1,Arg2,Val1,Val2), TuplePred(@C,Rul,Tab',Arg3,Arg4,Val3,Val4),
      PredFuncCount(@C,Rul,N), N==2, Tab != Tab', JID := f_unique().
j2 Join2(@C,Rul,JID,Arg1,Arg2,Val1,Val2) :-
      TuplePred(@C,Rul,Tab,Arg1,Arg2,Val1,Val2), PredFuncCount(@C,Rul,N),
      N == 1, JID := f_unique().
e1 Expr(@C,Rul,JID,ID,Val) :- Const(@C,Rul,ID,Val), JID := *.
e2 Expr(@C,Rul,JID,Arg1,Val1) :- Join2(@C,Rul,JID,Arg1,Arg2,Val1,Val2).
e3-e7 // analogous to e2 for Arg2/Val2 (Join2) and Arg1..4/Val1..4 (Join4)
a1 HeadVal(@C,Rul,JID,Arg,Val) :- Assign(@C,Rul,Arg,ID), Expr(@C,Rul,JID,ID,Val).
s1 Sel(@C,Rul,JID,SID,Val) :- Oper(@C,Rul,SID,ID',ID'',Opr), Expr(@C,Rul,JID',ID',Val'),
      Expr(@C,Rul,JID'',ID'',Val''), True == f_match(JID',JID''), JID := f_join(JID',JID''),
      Val := (Val' Opr Val''), ID' != ID''.
```

Figure 4.4: Meta rules for $\mu$Dlog.

### 4.3.1 The basic provenance graph

Recall from Section 4.2.1 that provenance can be represented as a DAG in which the vertices are events and the edges indicate direct causal relationships. Since NDlog's declarative syntax directly encodes dependencies, we can define relatively simple provenance graphs for it. For convenience, we adopt a graph from negative provenance (Chapter 3), which contains the following *positive* vertexes:

- EXIST($[t_1,t_2],N,\tau$): Tuple $\tau$ existed on node $N$ from time $t_1$ to $t_2$;
- INSERT($t,N,\tau$): Base tuple $\tau$ was inserted on node $N$ at time $t$;
- DELETE($t,N,\tau$): Base tuple $\tau$ was deleted on node $N$ at time $t$;
- DERIVE($t,N,\tau$): Derived tuple $\tau$ acquired support on $N$ at time $t$;
- UNDERIVE($t,N,\tau$): Derived tuple $\tau$ lost support on $N$ at time $t$;
- APPEAR($t,N,\tau$): Tuple $\tau$ appeared on node $N$ at time $t$;
- DISAPPEAR($t,N,\tau$): Tuple $\tau$ disappeared on node $N$ at time $t$;
- SEND($t,N\rightarrow N',\pm\tau$): $\pm\tau$ was sent by node $N$ to/from $N'$ at $t$; and
- RECEIVE($t,N\leftarrow N',\pm\tau$): $\pm\tau$ was received by node $N$ to/from $N'$ at $t$.

Conceptually, the system builds the provenance graph incrementally at runtime: whenever a new base tuple is inserted, the system adds an INSERT vertex, and whenever a rule is triggered and generates a new derived tuple, the system adds a DERIVE vertex. The APPEAR and EXIST vertexes are generated whenever a tuple is added to the database (after an insertion or derivation), and the interval in the EXIST vertex is updated once the tuple is deleted again. The rules for DELETE, UNDERIVE, and DISAPPEAR are analogous. The SEND and RECEIVE vertexes are used when a rule on one node has a tuple $\tau$ on another node as a precondition; in this case, the system sends a message from the latter to the former whenever $\tau$ appears ($+\tau$) or disappears ($-\tau$), and the two vertexes are generated when this message is sent and received, respectively. Notice that – at least conceptually – vertexes are never deleted; thus, the operator can inspect the provenance of past events.

The system inserts an edge $(v_1, v_2)$ between two vertexes $v_1$ and $v_2$ whenever the event represented by $v_1$ is a direct cause of the event represented by $v_2$. Derivations are caused by the appearance (if local) or reception (if remote) of the tuple that satisfies the last precondition of the corresponding rule, as well as by the existence of any other tuples that appear in preconditions; appearances are caused by derivations or insertions, message transmissions by appearances, and message arrivals by message transmissions. The rules for underivations and disappearances are analogous. Base tuple insertions are external events that have no cause within the system.

So far, we have described only the vertexes for positive provenance. The full graph also supports *negative* events (Chapter 3) by introducing a negative "twin" for each vertex. For instance, the counterpart to APPEAR is NAPPEAR, which represents the fact that a certain tuple *failed* to appear. For a more detailed discussion of negative provenance, please see (Chapter 3).

### 4.3.2 The meta provenance graph

The above provenance graph can only represent causality between data. We now extend the graph to track provenance of programs by introducing two elements: *meta tuples*, which represent the syntactic elements of the program itself (such as conditions and predicates) and *meta rules*, which describe the operational semantics of the language. For clarity, we describe the meta model for $\mu$Dlog here; our meta model for the full NDlog language is more complex but follows the same approach.

**Meta tuples:** We distinguish between two kinds of meta tuples: program-based tuples and runtime-based tuples. Program-based tuples are the syntactic elements that are visible to the programmer: rule heads (`HeadFunc`), predicates (`PredFunc`), assignments (`Assign`), constants (`Const`), and operators (`Oper`). Runtime-based tuples describe data structures inside the NDlog runtime: base tuple insertions (`Base`), tuples (`Tuple`), satisfied predicates (`TuplePred`), evaluated expressions (`Expr`), joins (`Join`), selections (`Sel`) and values in rule heads (`HeadVal`). Although concrete implementations may maintain additional data structures (e.g., for optimizations), these tuples are sufficient to describe the operational semantics.

**Meta rules:** Figure 4.4 shows the full set of meta rules for $\mu$Dlog. Notice that these rules are written in NDlog, not in $\mu$Dlog itself. We explain each meta rule below.

Tuples can exist for two reasons: they can be inserted as base tuples (`h1`) or derived via rules (`h2`). Recall that, in $\mu$Dlog's simplified syntax, each rule joins at most two tables and has exactly two selection predicates to select tuples from these tables. A rule "fires" and produces a tuple `T(a,b)` iff there is an assignment of values to $a$, and $b$ that satisfies both predicates. (Notice that the selection predicates are distinguished by a unique selection ID, or `SID`.) We will return to this rule again shortly.

The next four meta rules compute the actual joins. First, whenever a (syntactic) tuple appears as in a rule definition, each concrete tuple that exists at runtime generates one variable assignment for that tuple (p1). For instance, if a rule r contains Foo(A,B), where A and B are variables, and at runtime there is a concrete tuple Foo(5,7), meta rule p1 would generate a TuplePred(@C,r,Foo,A,B,5,7) meta tuple to indicate that 5 and 7 are assignments for A and B.

Depending on the number of tuples in the rule body (calcuated in rule p2), meta rule j1 or j2 will be triggered: When it contains two tuples from different tables, meta rule j1 computes a Join4 tuple for each pair of tuples from these tables. Note that this is a full cross-product, from which another meta rule (s1) will then select the tuples that match the selection predicates in the rule. For this purpose, each tuple in the join is given a unique join ID (JID), so that the values of the selection predicates can later be matched up with the correct tuples. If a rule contains only a tuple from one table, we compute a Join2 tuple instead (j2).

The next seven meta rules evaluate expressions. Expressions can appear in two different places – in a rule head and in a selection predicate – but since the evaluation logic is the same, we use a single set of meta rules for both cases. Values can come from integer constants (e1) or from any element of a Join2 or Join4 meta tuple (e2–e7). Notice that most of these values are specific to the join on which they were evaluated, so each Expr tuple contains a specific JID; the only exception are the constants, which are valid for all joins. To capture this, we use a special JID wildcard (*), and we test for JID equality using a special function f_match(JID1,JID2) that returns true iff JID1==JID2 or if either of them is *.

The last two meta rules handle assignments (a1) and selections (s1). An assignment simply sets a variable in a rule head to the value of an expression. The s1 rule determines, for each selection predicate in a rule (identified by SID) and for each join state (identified by JID) whether the check succeeds or fails. Function f_join(JID1, JID2) is introduced to handle JID wildcard: it returns JID1 if JID2 is *, or JID2 otherwise. The result is recorded in a Sel meta tuple, which is used in h2 to decide whether a head tuple is derived.

$\mu$Dlog requires only 13 meta tuples and 15 meta rules; the full meta model for NDlog contains 23 meta tuples and 23 meta rules. We omit the details here; they are included in Appendix B.1.

### 4.3.3 Meta provenance forests

So far, we have essentially transformed the original NDlog program into a new "meta program". In principle, we could now generate meta provenance graphs by applying a normal provenance graph generation algorithm on the meta program – e.g., the one from negative provenance (Chapter 3). However, this is not sufficient for our purposes. Because there are cases where the same effect can be achieved in multiple ways. For instance, suppose that we are explaining the absence of an `X` tuple, and that there are two different rules, `r1` and `r2`, that could derive `X`. If our goal was to *explain why* `X` was absent, we would need to include explanations for both `r1`'s and `r2`'s failure to fire. However, our goal is instead to *make `X` appear*, which can be achieved by causing *either* `r1` *or* `r2` to fire. If we included both in the provenance tree, we would generate only repairs that cause both rules to fire, which is unnecessary and sometimes even impossible.

Our solution is to replace the meta provenance tree with a meta provenance *forest*. Whenever our algorithm encounters a situation with $k$ possible choices that are each individually sufficient for repair, it replaces the current tree with $k$ copies of itself and continues to explore only one choice in each tree.

### 4.3.4 From explanations to repairs

The above problem occurs in the context of disjunctions; next, we consider its "twin", which occurs in the context of conjunctions. Sometimes, the meta provenance must explain why a rule with multiple preconditions did *not* derive a certain tuple. For diagnostic purposes, the absence of one missing precondition is already sufficient to explain the absence of the tuple. However, meta provenance is intended for repair, i.e., it must allow us to find a way to make the missing tuple appear. Thus, it is not enough to find a way to make a single precondition true, or even ways to make each precondition true individually. What we need is a way to satisfy *all* the preconditions *at the same time*!

For concreteness, consider the following simple example, which involves a meta rule `A(x,y):-B(x), C(x,y),x+y>1,x>0`. Suppose that the operator would like to find out why there is no `A(x,y)` with `y==2`. In this case, it would be sufficient to show that there is no `C(x,y)` with `y==2` and `x>0`; cross-predicate constraints, such as `x+y>1`, can be ignored. However, if we want to actually make a suitable `A(x,y)` appear, we need to *jointly* consider the absence of both `B(x)` and `C(x,y)`, and ensure that all branches of the provenance tree respect the cross-predicate constraints. In other words, we cannot explore the two branches separately; we must make sure that their contents "match".

To accomplish this, our algorithm automatically generates a constraint pool for each tree. It encodes the attributes of tuples as variables, and it formulates constraints over these variables. For instance, given the missing tuple $A_0$, we add two variables $A_0$`.x` and $A_0$`.y`. To initialize the constraint pool, the root of the meta provenance graph must satisfy the operator's requirement: $A_0$`.y == 2`. While expanding any missing tuple, the algorithm adds constraints as necessary for a successful derivation. In this example, three constraints are needed: first, the predicates must join together, i.e., $B_0$`.x ==` $C_0$`.x`. Second, the predicates must satisfy the constraints, i.e., $B_0$`.x>0` and $C_0$`.x+`$C_0$`.y>1`. Third, the predicates must derive the head, i.e., $A_0$`.x==`$C_0$`.x` and $A_0$`.y==`$C_0$`.y`. In addition, tuples must satisfy primary key constraints. For instance, suppose deriving `B(x)` requires $D_0$`(9,1)` while deriving `C(x,y)` requires $D_1$`(9,2)`. If `x` is the only primary key of table `D(x,y)`, $D_0$`(9,1)` and $D_1$`(9,2)` cannot co-exist at the same time. Therefore, the explanation is inconsistent for generating repairs. To address such cases, we encode additional constraints: `D.x ==` $D_0$`.x implies D.y == 1` and `D.x ==` $D_1$`.x implies D.y == 2`.

### 4.3.5 Generating meta provenance

In general, meta provenance forests may consist of infinitely many trees, each with infinitely many vertexes. Thus, we cannot hope to materialize the entire forest. Instead, we adopt a variant of the approach from negative provenane (Chapter 3) and use a step-by-step procedure that constructs the trees incrementally. We define a function QUERY($v$) that, when called on a vertex $v$ from any (partial) tree in the meta provenance forest, returns the immediate children of $v$ and/or "forks" the tree as described above. By calling this function repeatedly on the leaves of the trees, we can explore the trees incrementally.

The two key differences to negative provenance are the procedures for expanding NAPPEAR and NDERIVE vertices: the former must now "fork" the tree when there are multiple children that are each individually sufficient to make the missing tuple appear (Section 4.3.3), and the latter must now explore a join across *all* preconditions of a missing derivation, while collecting any relevant constraints (Section 4.3.4).

To explore an infinite forest with finite memory, our algorithm maintains a set of partial trees. Initially, this set contains a single "tree" that consists of just one vertex – the vertex that describes the symptom that the operator has observed. Then, in each step, the algorithm picks one of the partial trees, randomly picks a vertex within that tree that does not have any children yet, and then invokes QUERY on this vertex to find the children, which are then added to that tree. As discussed before, this step can cause the tree to fork, adding multiple copies to the set that differ only in the newly added children. Another possible outcome is that the chosen partial tree is completed, which yields a repair candidate.

Each tree – completed or partial – is associated with a *cost*, which intuitively represents the implausibility of the repair that the tree represents. (Lower-cost trees are more plausible.) Initially, the cost is zero. Whenever a base tuple is added that represents a program change, we increase the total cost of the corresponding tree by the cost of that change. In each step, our algorithm picks the partial tree with the

```
function GenerateRepairCandidates($P$)
    $R \leftarrow \emptyset$, $\tau_r \leftarrow$ RootTuple($P$)
    if MissingTuple($\tau_r$) then
        $C \leftarrow$ ConstraintPool($P$)
        $A \leftarrow$ SatAssignment($C$)
        for ($\tau_i$) $\in$ BaseTuples($P$)
            if MissingTuple($\tau_i$) then
                $R \leftarrow R \cup$ ChangeTuple($\tau_i$,$A$)
    else if ExistingTuple($\tau_r$) then
        for ($T_i$) $\in$ BaseTupleCombinations($P$)
            $R_{ci} \leftarrow \emptyset$, $R_{di} \leftarrow \emptyset$
            $C_i \leftarrow$ SymbolicPropagate($P$,$T_i$)
            $A_i \leftarrow$ UnsatAssignment($C_i$)
            for ($\tau_i$) $\in T_i$
                $R_{ci} \leftarrow R_{ci} \cup$ ChangeTuple($\tau_i$,$A_i$)
                $R_{di} \leftarrow R_{di} \cup$ DeleteTuple($\tau_i$)
            $R \leftarrow R \cup R_{ci} \cup R_{di}$
    return $R$
```

Figure 4.5: Algorithm for extracting repair candidates from the meta provenance graph. For a description of the helper functions, please see Appendix B.2.

lowest cost; if there are multiple trees with the same cost, our algorithm picks the one with the smallest number of unexpanded vertexes. Repair candidates are output only once there are no trees with a lower cost. Thus, repair candidates are found in cost order, and the first one is optimal with respect to the chosen cost metric; if the algorithm runs long enough, it eventually finds a working repair. (For a more detailed discussion, please see Appendix B.3.) In practice, the algorithm runs until some reasonable cut-off cost is reached, or until the operator's patience runs out.

The question remains how to assign costs to program changes. We assign a low cost to common errors (such as changing a constant by one or changing a == to a !=) and a high cost to unlikely errors (such as writing an entirely new rule, or defining a new table). Thus, we can prioritize the search of fixes to software bugs that are more commonly observed in actual programming, and thus increase the chances that a working fix will be found.

### 4.3.6 Limitations

The above approach is likely to find simple problems, such as incorrect constraints or copy-and-paste errors, but it is not likely to discover fundamental flaws in the program logic that require repairs in many different places and/or several new rules. However, software engineering studies have consistently shown that simple errors, such as copy-and-paste bugs, are very common: simple typos already account for 9.4—9.8% of all semantic bugs [81], and 70–90% of bugs can be fixed by changing only existing syntactic elements [106]. Because of this, we believe that an approach that can automatically fix "low-cost" bugs can still be useful in practice.

Our approach focuses exclusively on incorrect computations; there are classes of bugs, such as concurrency bugs or performance bugs, that it cannot repair. We speculate that such bugs can be found with a richer meta model, but this is beyond the scope of the present chapter.

## 4.4 Generating repair candidates

As discussed in Section 4.3.5, our algorithm explores the meta provenance forest in cost order, adding vertexes one by one by invoking QUERY on a leaf of an existing partial tree. Thus, the algorithm slowly generates more and more trees; at the same time, some existing trees are eventually completed because none of their leaves can be further expanded (i.e., QUERY returns $\emptyset$ on them). Once a tree is completed, we invoke the algorithm in Figure 4.5 to extract a candidate repair.

The algorithm has two cases: one for trees that have an existing tuple at the root (e.g., a packet that reached a host it should not have reached), and one for trees that have a missing tuple at the root (e.g., a packet failed to reach its destination). We discuss each in turn. Furthermore, we note that the ensuing analysis is performed on the *meta* program, which is independent from the language that the *original* program is written in.

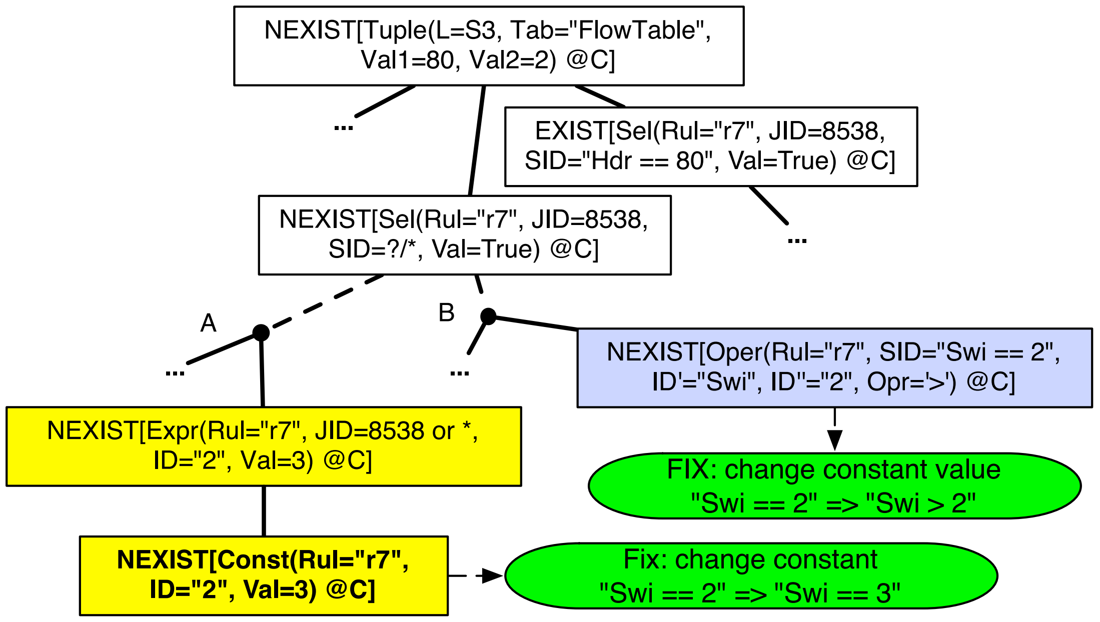

NEXIST[Tuple(L=S3, Tab="FlowTable", Val1=80, Val2=2) @C]
...
EXIST[Sel(Rul="r7", JID=8538, SID="Hdr == 80", Val=True) @C]
...
NEXIST[Sel(Rul="r7", JID=8538, SID=?/*, Val=True) @C]
A
B
...
...
NEXIST[Oper(Rul="r7", SID="Swi == 2", ID'="Swi", ID"="2", Opr='>') @C]
NEXIST[Expr(Rul="r7", JID=8538 or *, ID="2", Val=3) @C]
FIX: change constant value "Swi == 2" => "Swi > 2"
NEXIST[Const(Rul="r7", ID="2", Val=3) @C]
Fix: change constant "Swi == 2" => "Swi == 3"

Figure 4.6: Meta provenance of a missing flow entry. It consists of two trees (white + yellow, white + blue), each of which can generate a repair candidate.

### 4.4.1 Handling negative symptoms

If the root of the tree is a missing tuple, its leaves will contain either missing tuples or missing meta tuples, which can be then created by inserting the corresponding tuples or program elements. However, some of these tuples may still contain variables – for instance, the tree might indicate that an `A(x)` tuple is missing, but without a concrete value for `x`. Hence, the algorithm first looks for a satisfying assignment of the tree's constraint pool (Section 4.3.4). If such an assignment is found, it will supply concrete values for all remaining variables; if not, the tree cannot produce a working repair and is discarded.

As an example, Figure 4.6 shows part of the meta provenance of a missing event. It contains two meta provenance trees, which have some vertices in common (colored white), but do not share other vertices (colored yellow and blue). The constraint pool includes `Const0.Val = 3`, `Const0.Rul = r7`, and `Const0.ID = 2`. That is, the repair requires the existence of a constant of value `3` in rule `r7`. Therefore, we can change value of the original constant (identified by identical primary keys `Rul` and `ID`) to `3`.

### 4.4.2 Handling positive symptoms

Meta provenance can also help with debugging scenarios with positive symptoms. Figure 4.7 shows the meta provenance graph of a tuple that exists, but should not exist. We can make this tuple disappear by deleting (or changing in the proper way) any of the base tuples or meta tuples on which the derivation is based.

However, neither base tuples nor meta tuples are always safe to change. In the case of meta tuples, we must ensure that the change does not violate the syntax of the underlying language (in this case, $\mu$Dlog). For instance, it would be safe to delete a `PredFunc` tuple to remove a predicate, but it may not be safe to delete a `Const` meta tuple, since this might result in an incomplete expression, such as `Swi >`.

In the case of changes to base tuples, the problem is to find changes that a) will make the current derivation disappear, and that b) will *not* cause an alternate derivation of the same tuple via different meta rules. To handle the first problem, we do not directly replace elements of a tuple with a different value. Rather, we initially replace the elements with symbolic constants and then re-execute the derivation of meta rules symbolically while collecting constraints over the symbolic constants that must hold for the derivation to happen. Finally, we can negate these constraints and use a constraint solver to find a satisfying assignment for the negation. If successful, this will yield concrete values we can substitute for the symbolic constant that will make the derivation disappear.

For concreteness, we consider the green repair in Figure 4.7. We initially replace `Const('r1',1,1)` with `Const('r1',1,Z)` and then reexecute the derivation to collect constraints – in this case, `1==Z`. Since `Z=2` does not satisfy the constraints, we can make the tuple at the top disappear by changing Z to 2 (which corresponds to changing `Swi==1` to `Swi==2` in the program).

This leaves the second problem from above: even if we make a change that disables one particular derivation of an undesired tuple, that very change could enable some other derivation that causes the undesired tuple to reappear. For instance,

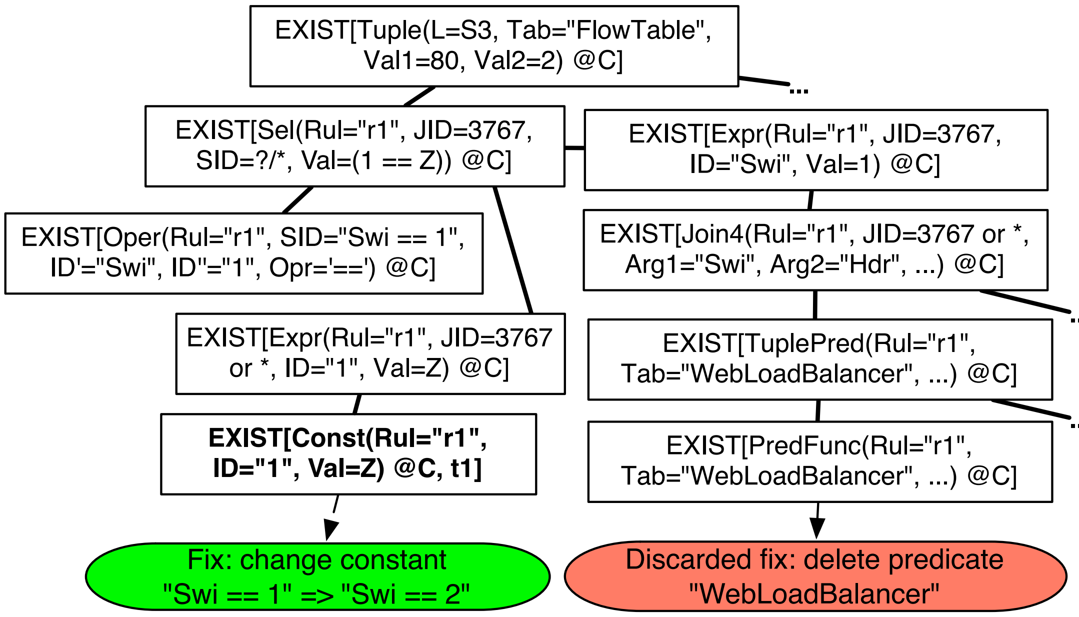

EXIST[Tuple(L=S3, Tab="FlowTable", Val1=80, Val2=2) @C]
...
EXIST[Sel(Rul="r1", JID=3767, SID=?/*, Val=(1 == Z)) @C]
EXIST[Expr(Rul="r1", JID=3767, ID="Swi", Val=1) @C]
EXIST[Oper(Rul="r1", SID="Swi == 1", ID'="Swi", ID"="1", Opr='==') @C]
EXIST[Join4(Rul="r1", JID=3767 or *, Arg1="Swi", Arg2="Hdr", ...) @C]
...
EXIST[Expr(Rul="r1", JID=3767 or *, ID="1", Val=Z) @C]
EXIST[TuplePred(Rul="r1", Tab="WebLoadBalancer", ...) @C]
...
EXIST[Const(Rul="r1", ID="1", Val=Z) @C, t1]
EXIST[PredFunc(Rul="r1", Tab="WebLoadBalancer", ...) @C]
Fix: change constant "Swi == 1" => "Swi == 2"
Discarded fix: delete predicate "WebLoadBalancer"

Figure 4.7: Meta provenance of a harmful flow entry. All repairs (e.g., green and red) prevent this derivation, but the red one rederives the tuple via other meta rules.

suppose we delete the tuple `PredFunc('r1','WebLoadBalancer', ...)`, which corresponds to deleting the `WebLoadBalancer` predicate from the $\mu$Dlog rule `r1` (shaded red in Figure 4.7). This deletion will cause the `Join4` tuple to disappear, and it will change the value of `PredFuncCount` from 2 to 1. As a result, the derivation through meta rule `j1` will duly disappear; however, this will instead trigger meta rule `j2`, which leads to another derivation of the same flow entry.

Solving this for arbitrary programs is equivalent to solving the halting problem, which is NP-hard. However, we do not need a perfect solution because this case is rare, and because we can either use heuristics to track certain rederivations or we can easily eliminate the corresponding repair candidates during backtesting.

### 4.4.3 Backtesting a single repair candidate

Although the generated repairs will (usually) solve the problem immediately at hand, by making the desired tuple appear or the undesired tuple disappear, each repair can also have a broader effect on the network as a whole. For instance, if the problem is

```
r7(v1) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 3, Hdr == 80, Prt := 2.
```

```
r7(v2) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi > 2, Hdr == 80, Prt := 2.
```

```
r7(v3) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi != 2, Hdr == 80, Prt := 2.
```

(a)

```
r6(v1,v2,v3) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 2, Hdr == 53, Prt := 2.
r7(v1,v2,v3) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi == 3, Hdr == 80, Prt := 2.
r7(v2,v3) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi > 3, Hdr == 80, Prt := 2.
r7(v3) FlowTable(@Swi,Hdr,Prt) :- PacketIn(@C,Swi,Hdr), Swi < 2, Hdr == 80, Prt := 2.
```

(b)

Figure 4.8: (a) Three repair candidates, all of which can generate forwarding flow entries for switch S2 by fixing r7 in the original program in Figure 4.2; other parts of the program are unchanged. (b) Backtesting program that evaluates all three repair candidates simultaneously while running shared computations only once.

that a switch forwarded a packet to the wrong host, one possible "repair" is to disable the rule that generates flow entries for that switch. However, this would also prevent *all other* packets from being forwarded, which is probably too restrictive.

To mitigate this, we adopt the maxim of "primum non nocere" [54] and assess the global impact of a repair candidate before suggesting it. Specifically, we backtest the repair candidates in simulation, using historical information from the network. We can approximate past control-plane states from the diagnostic information we already record for the provenance; to generate a plausible workload, we can use a Netflow trace or a sample of packets. We then collect some key statistics, such as the number of packets delivered to each host. Since the problems we are aiming to repair are typically subtle (total network failures are comparatively easy to diagnose!), they should affect only a small fraction of the traffic. Hence, a "good" candidate repair should have little or no impact on metrics that are not related to the problem.

In essence, the metrics play the role of the test suite that is commonly used in the wider literature on automated program fixing. While the simple metric from above should serve as a good starting point, operators could easily add metrics of their own, e.g., to encode performance goals (load balancing, link utilization) or security restrictions (traffic from X should never reach Y). However, recall that, in contrast to much of the earlier work on program fixing, we do *not* rely on this "test suite" to *find* candidate repairs (we use the meta provenance for that); the metrics simply serve as a sanity check to weed out repairs with serious side effects. That a repair passed the backtesting stage is not a guarantee that no side effects will occur.

As an additional benefit, the metrics can be used to rank the repairs, and to give preference to the candidates that have the smallest impact on the overall network.

### 4.4.4 Backtesting multiple repair candidates

It is important for the backtesting to be fast: the less time it takes, the more candidate repairs we can afford to consider. Fortunately, we can leverage another concept from the database literature to speed up this process considerably. Recall that each backtest simulates the behavior of the network with the repaired program. Thus, we are effectively running many very similar "queries" (the repaired programs, which differ only in the fixes that were applied) over the same "database" (the historical network data), where we expect significant overlaps among the query computations. This is a classical instance of *multi-query optimization*, for which powerful solutions are available in the literature [88, 45].

Multi-query optimization exploits the fact that almost all computation is shared by almost all repair candidates, and thus has to be performed only once. We accomplish this by transforming the original program into a *backtesting program* as follows. First, we associate each tuple with a set of tags, we extend all relations to have a new field for storing the tags, and we update all the rules such that the tag of the head is the intersection of the tags in the body. Then, for each repair candidate, we create a new tag and add copies of all the rules the repair candidate modifies, but we restrict them to this particular tag. Finally, we add rules that evaluate the metrics from Section 4.4.3, separately for each tag.

The effect is that data flows through the program as usual, but, at each point where a repair candidate has modified something, the flow forks off a subflow that has the tag of that particular candidate. Thus, the later in the program the modification occurs, the fewer computations have to be duplicated for that candidate. Overall, the backtesting program correctly computes the metrics for each candidate, but runs considerably faster than computing each of the metrics round after round.

As an example, Figure 4.8(a) shows three repair candidates (`v1`, `v2`, and `v3`) for the buggy program in Figure 4.2. Each of them alters the rule `r7` in a different way: `v1` changes a constant, `v2` and `v3` change an operator. (Other rules are unchanged.)

In some cases, it is possible to determine, through static analysis, that rules with different tags produce overlapping output. For instance, in the above example, the three repairs all modify the same predicate, and some of the predicates are implied by others; thus, the output for switch 3 is the same for all three tags, and the output for switches above 3 is the same for tags `v2` and `v3`. By coalescing the corresponding rules, we can further reduce the computation cost. Finding *all* opportunities for coalescing would be difficult, but recall that this is merely an optimization: even if we find none at all, the program will still be correct, albeit somewhat slower.

## 4.5 Evaluation

In this section, we report results from our experimental evaluation, which aim to answer five high-level questions: 1) Can meta provenance generate reasonable repair candidates? 2) What is the runtime overhead of meta provenance? 3) How fast can we process diagnostic queries? 4) Does meta provenance scale well with the network size? And 5) how well does meta provenance work across different SDN frameworks?

### 4.5.1 Prototype implementation

We have built a prototype based on declarative and imperative SDN environments as well as Mininet [78]. It generates and further backtests repair candidates, such that the operator can inspect the suggested repairs and decide whether and which to apply. Our prototype consists of around 30,000 lines of code, including the following three main components.

**Controllers:** We validate meta provenance using three types of SDN environments. The first is a declarative controller based on RapidNet [84]; it includes a proxy that interposes between the RapidNet engine and the Mininet network and that translates NDlog tuples into OpenFlow messages and vice versa. The other two are existing environments: the Trema framework [127] and the Pyretic language [96]. (Notice that neither of the latter two is declarative: Trema is based on Ruby, an imperative language, and Pyretic is an imperative domain-specific language for SDNs that is embedded in Python.)

At runtime, the controller and the network each record relevant control-plane messages and packets to a log, which can be used to answer diagnostic queries later. The information we require from runtime is not substantially different from existing provenance systems [82, 134, 147, 24], which have shown that provenance can be captured at scale and for SDNs.

**Tuple generators:** For each of the above languages, we have built a meta tuple generator that automatically generates meta tuples from the controller program and from the log. The program-based meta tuples (e.g., constants, operators, edges) only need to be generated once for each program; the log-based meta tuples (e.g., messages, constraints, expressions) are generated by replaying the logged control-plane messages through automatically-instrumented controller programs.

**Tree constructor:** This component constructs meta provenance trees from the meta tuples upon a query. As we discussed in Section 4.3.4, this requires checking the consistency of repair candidates. Our constructor has an interface to the Z3 solver [31] for this purpose. However, since many of the constraint sets we generate are trivial, we have built our own "mini-solver" that can quickly solve the trivial instances on its own; the nontrivial ones are handed over to Z3. The mini-solver also serves as an optimizer for handling cross-table meta tuple joins. Using a naïve nested loop join that considers all combinations of different meta tuples would be inefficient; instead, we solve simple constraints (e.g., equivalence, ranges) first. This allows us to filter the meta tuples before joining them, and use more efficient join paradigms, such as hash joins. Our cost metric is based on a study of common bug fix patters (Pan et al. [106]).

### 4.5.2 Experimental setup

To obtain a representative experimental environment, we set up the Stanford campus network from ATPG [138] in Mininet [78], with 16 Operational Zone and backbone routers. Moreover, we augmented the topology with edge networks, each of which is connected to the main network by at least one core router; we also set up 1 to 15 end hosts per edge network. In most experiments, the core network is either proactively configured using forwarding entries from the Stanford campus network; the edge networks run a mix of reactive and proactive applications. In Section 4.5.8, we include an experiment where the controller reactively installs core routing policies. Overall, our smallest topology across all scenarios consisted of 19 routers and 259 hosts, and our largest topology consisted of 169 routers and 549 hosts. In addition, we created realistic background traffic using two traffic traces obtained in a similar campus network setting [14]; 1 to 16 of the end hosts replayed the traces continuously during the course of our experiments. Moreover, we generated a mix of ICMP `ping` traffic and HTTP traffic on the remaining hosts. Overall, 4.6–309.4 million packets were sent through the network. We ran experiments on a Dell OptiPlex 9020 workstation, which has a 8-core 3.40 GHz Intel i7-4770 CPU with 16 GB of RAM and a 128 GB OCZ Vector SSD. The OS was Ubuntu 13.10, and the kernel version was 3.8.0.

### 4.5.3 Usability: Diagnosing SDNs

A natural first question to ask is whether meta provenance can repair real problems. To avoid distorting our results by picking our own toy problems to debug, we have chosen four diagnostic scenarios from four different networking chapters that have

| | **Query description** | **Result** |
|---|---|---|
| **Q1** | H20 is not receiving HTTP requests from H2 | 9/2 |
| **Q2** | H17 is not receiving DNS queries from H1 | 12/3 |
| **Q3** | H20 is not receiving HTTP requests from H1 | 11/3 |
| **Q4** | First HTTP packet from H2 to H20 is not received | 13/3 |
| **Q5** | H2's MAC address is not learned by the controller | 9/3 |

Table 4.1: The diagnostic queries, the number of repair candidates generated by meta provenance, and the number of remaining candidates after backtesting.

appeared at CoNEXT [138, 33], NSDI [18], and HotSDN [13], plus one common class of bugs from an OSDI chapter [80]. We focused on scenarios where the root cause of the problem was a bug in the controller program. We recreated each scenario in the lab, based on its published description. The five scenarios were:

- **Q1: Copy-and-paste error [80].** A server received no requests because the operator made a copy-and-paste error when modifying the controller program. The scenario is analogous to the one in Figure 4.1, but with larger topology and more realistic traffic.

- **Q2: Forwarding error [138].** A server could not receive queries from certain clients because the operator made a error when specifying the action of the forwarding rule.

- **Q3: Uncoordinated policy update [33].** A firewall controller app configured white-list rules for web servers. A load-balancing controller app updated the policy on an ingress point, without coordinating with the firewall app; this caused some traffic to shift, and then to be blocked by the firewall.

- **Q4: Forgotten packets [18].** A controller app correctly installed flow entries in response to new flows; however, it forgot to instruct the switches to forward the first incoming packet in each flow.

- **Q5: Incorrect MAC learning [13].** A MAC learning app should have matched packets based on their source IP, incoming port, and destination

IP; however, the program only matched on the latter two fields. As a result, some switches never learned about the existence of certain hosts.

To get a sense of how useful meta provenance would be for repairing the problems, we ran diagnostic queries in our five scenarios as shown in Table 4.1, and examined the generated candidate repairs. In each of the scenarios, we bounded the cost and asked the repair generator to produce all repair candidates. Table 4.2 shows the repair candidates returned for Q1; the others are included in Appendix B.4.

Our backtesting confirmed that each of the proposed candidates was effective, in the sense that it caused the backup web server to receive at least some HTTP traffic. This phase also weeded out the candidates that caused problems for the rest of the network. To quantify the side effects, we replayed historical packets in the original network and in each repaired network. We then computed the traffic distribution at end hosts for each of these networks. We used the Two-Sample Kolmogorov-Smirnov test with significance level 0.05 to compare the distributions before and after each repair. A repair candidate was rejected if it significantly distorted the original traffic distribution; the statistics and the decisions are shown in Table 4.2. For instance, repair candidate G deleted `Swi==2` and `Dpt==53` in rule `r6`. This causes the controller to generate a flow entry that forwards HTTP requests at S3; however, the modified `r6` *also* causes HTTP requests to be forwarded to the DNS server.

After backtesting, the remaining candidates are presented to the operator in complexity order, i.e., the simplest candidate is shown first. In this example, the second candidate on the list (B) is also the one that most human operators would intuitively have chosen – it fixes the copy-and-paste bug by changing the switch ID in the faulty predicate from `Swi==2` to `Swi==3`.

Table 4.1 summarizes the quality of repairs our prototype generated for all scenarios for the RapidNet controller. Each scenario resulted in two or three repair suggestions. In the first stage, meta provenance produced between 9 and 13 repair candidates for each query, for a total of 54 repair candidates. Note that these num-

|   | **Repair candidate (Accepted?)** | **KS-test** |
|---|---|---|
| A | Manually installing a flow entry (✓) | 0.00007 |
| B | Changing `Swi==2` in `r7` to `Swi==3` (✓) | 0.00007 |
| C | Changing `Swi==2` in `r7` to `Swi!=2` (✗) | 0.00865 |
| D | Changing `Swi==2` in `r7` to `Swi>=2` (✗) | 0.00826 |
| E | Changing `Swi==2` in `r7` to `Swi>2` (✗) | 0.00826 |
| F | Deleting `Swi==2` in `r7` (✗) | 0.00867 |
| G | Deleting `Swi==2` and `Dpt==53` in `r6` (✗) | 0.05287 |
| H | Deleting `Swi==2` and `Dpt==80` in `r7` (✗) | 0.00999 |
| I | Changing `Swi==2 and Act=output-1 in r5 to Swi==3` and `Act=output-2` (✗) | 0.05286 |

Table 4.2: Candidate repairs generated by meta provenance for Q1, which are then filtered by a KS-test.

bers do not count expensive repair candidates that were discarded by the ranking heuristic (Section 4.3.5). The backtesting stage then confirmed that 48 of these candidates were effective, i.e., they fixed the problem at hand (e.g., the repair caused the server to receive at least a few packets). However, 34 of the effective candidates caused nontrivial side effects, and thus were discarded.

We note that the final set of candidates included a few non-intuitive repairs – for instance, one candidate fixed the problem in Q1 by manually installing a new flow entry. However, these repairs were nevertheless effective and had few side effects, so they should suffice as an initial fix. If desired, a human operator could always refactor the program later on.

### 4.5.4 Runtime overhead

**Latency and throughput:** To measure the latency and throughput overhead incurred by maintaining meta provenance, we used a standard approach of stress-testing OpenFlow controllers [36] which involves streaming incoming packets through the Trema controller using `Cbench`. Latency is defined as the time taken to process each packet within the controller. We observe that provenance maintenance resulted in a latency increase of 4.2% to 54ms, and a throughput reduction of 9.8% to 45,423 packets per second.

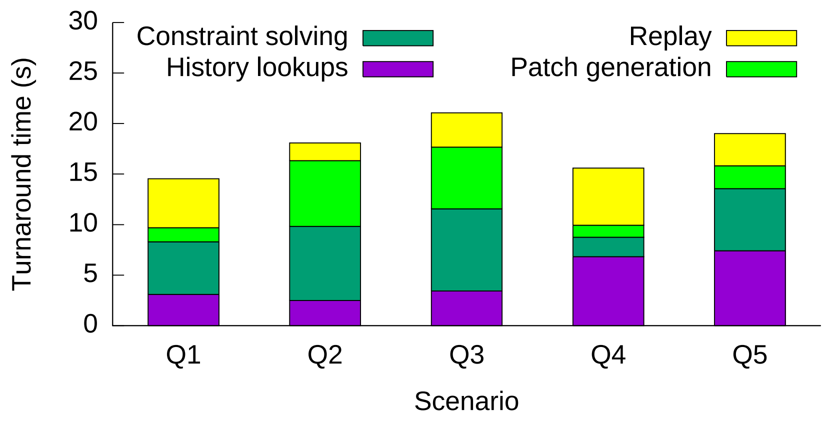


Figure 4.9: Time to generate the repairs for each of the scenarios in Section 4.5.3.

**Disk storage:** To evaluate the storage overhead, we streamed the two traffic traces obtained from [14] through our SDN scenario in Q1. For each packet in the trace, we recorded a 120-byte log entry that contains the packet header and the timestamp. The logging rates for the two traces are 20.2 MB/s and 11.4 MB/s per switch, respectively, which are only a fraction of the sequential write rate of commodity SSDs. Note that this data need not be kept forever: most diagnostic queries are about problems that currently exist or have appeared recently. Thus, is should be sufficient to store the most recent entries, perhaps an hour's worth.

### 4.5.5 Time to generate repairs

Diagnostic queries does not always demand a real-time response; however, operators would presumably prefer a quick turnaround. Figure 4.9 shows the turnaround time for constructing the meta provenance data structure and for generating repair candidates, including a breakdown by category. In general, scenarios with more complex control-plane state (Q1, Q4, and Q5) required more time to query the time index and to look up historical data; the latter can involve loop-joining multiple meta tables, particularly for the more complicated meta rules with over ten predicates. Other scenarios (Q2 and Q3) forked larger meta-provenance forests and thus spent more time on generating repairs and on solving constraints. However, we observe

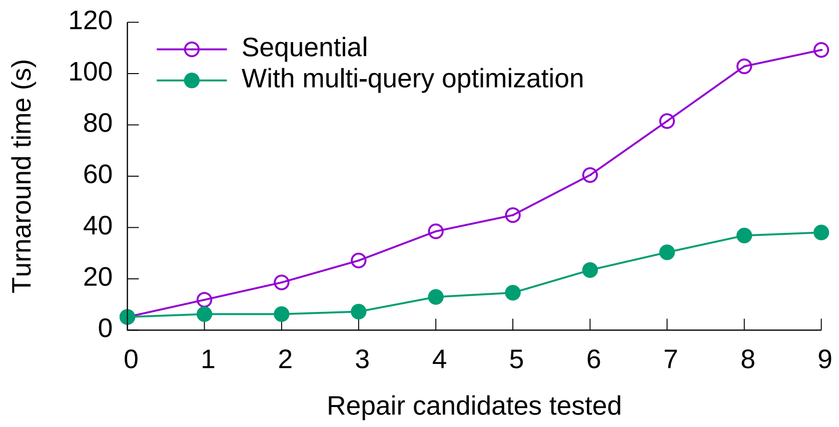


Figure 4.10: The times needed to *jointly* backtest the first $k$ repairs from Q1.

that, even when run on a single machine, the entire process took less than 25 seconds in all scenarios, which does not seem unreasonable. This time could be further reduced by parallelization, since different machines could work on different parts of the meta-provenance forest in parallel.

### 4.5.6 Backtesting speed

Next, we evaluate the backtesting speed using the repair candidates listed in Table 4.2. For each candidate, we sampled packet traces at the network ingresses from the log, and replayed them for backtesting. The top line in Figure 4.10 shows the time needed to backtest all the candidates sequentially; testing all nine of them took about two minutes, which already seems reasonably fast. However, the less time backtesting takes, the more repair candidates we can afford to consider. The lower line in Figure 4.10 shows the time needed to *jointly* backtest the first $k$ candidates using the multi-query optimization technique from Section 4.4.4, which merges the candidates into a single "backtesting program". With this, testing all nine candidates took about 40 seconds. This large speedup is expected because the repairs are small and fairly similar (since they are all intended to fix the same problem); hence, there is a substantial amount of overlap between the individual backtests, which the multi-query technique can then eliminate.

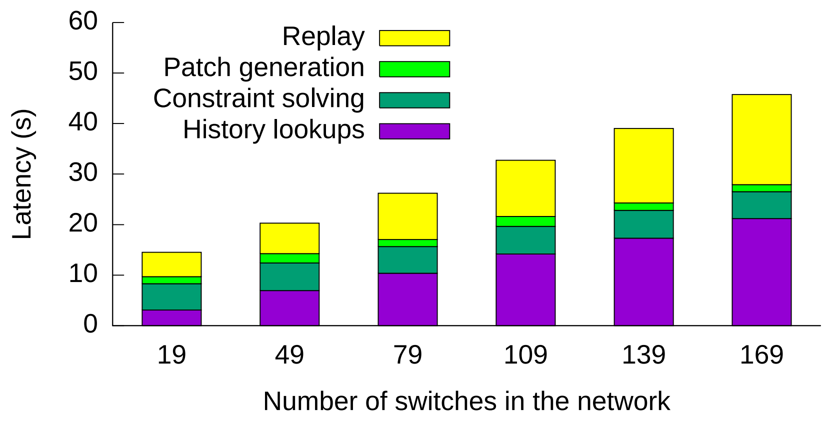


Figure 4.11: Scalability of repair generation phase with network size for Q1.

### 4.5.7 Scalability: Network size

To evaluate the scalability of meta provenance with regard to the network size, we tested the turnaround time of query Q1 on larger networks which contained up to 169 routers and 549 hosts. We obtained these networks by adding more routers and hosts to the basic Stanford campus network. Moreover, we increased the number of hosts that replay traffic traces [14] to up to 16. We generated synthetic traffic on the remaining hosts, and used higher traffic rates in larger networks to emulate more hosts. As we can see from Figure 4.11, the turnaround time increased linearly with the network size, but it was within 50 seconds for all cases. As the breakdown shows, the increase mainly comes from the latency increase of the historical lookups and of the replay. This is because the additional nodes and traffic caused the size of the controller state to increase. This in turn resulted in a longer time to search through the controller state, and to replay the messages. Repair generation and constraint solving time only see minor increases. This is expected because the meta provenance forest is generated from only relevant parts of the log, the size of which is relatively stable when the affected flows are given.

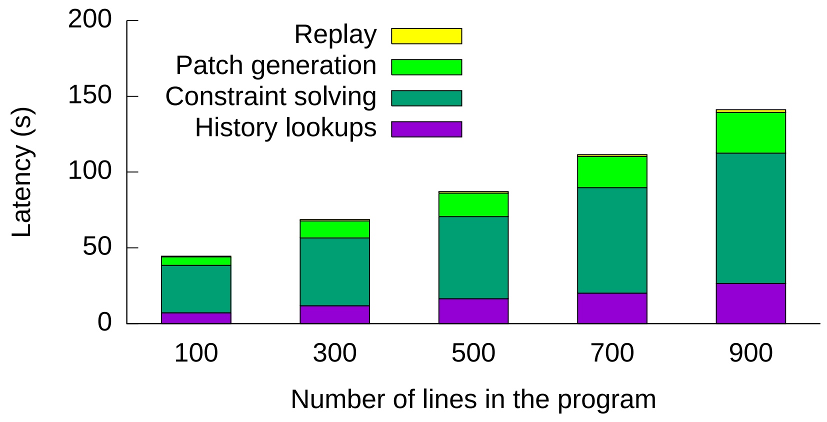


Figure 4.12: Scalability of repair generation phase with program size for Q1.

### 4.5.8 Scalability: Program size

The controller programs from the scenarios in Section 4.5.3 have between 19 and 120 lines of code. To evaluate the scalability of meta provenance with regard to program size, we tested the engine on larger programs which contained from 100 to 900 lines of code. We obtained these programs by augmenting the Trema controller program in Q1 with various numbers of policies of an operational zone switch in the Stanford campus network. As we can see from Figure 4.12, the turnaround time increases linearly with the program size. The increase is uniform in all components: first, we need to replay more controller traffic because more policies are enforced reactively; second, history lookups took longer because the size of controller state is larger; last, the time spent on constraint solving and patch generation increases because larger programs contain more syntactic elements to consider for repair and therefore incur larger provenance forests. However, despite the increase in program size, the number of repairs are stable across the experiment. Although the additional code causes the provenance forest to grow initially, most of the trees quickly become too costly and are not explored further. Instead, meta provenance focuses on relevant parts of the program.

| | Q1 | Q2 | Q3 | Q4 | Q5 |
|---|---|---|---|---|---|
| **Trema (Ruby)** | 7/2 | 10/2 | 11/2 | 10/2 | 14/3 |
| **Pyretic (DSL + Python)** | 4/2 | 11/2 | 9/2 | - | 14/3 |

Table 4.3: Results for Trema and Pyretic. For each scenario from Section 4.5.3, we show how many repair candidates are generated, and how many passed backtesting.

### 4.5.9 Applicability to other languages

To see how well meta provenance works for languages other than NDlog, we developed meta models for Trema [127] and Pyretic [96]. This required only a moderate effort (16 person-hours). Our Trema model contains 42 meta rules and 32 meta tuples; it covers basic control flow (e.g., functional calls, conditional jumps) and data flow semantics (e.g., constants, expressions, variables, and objects) of Ruby. The Pyretic model contains 53 meta rules and 41 meta tuples; it describes a set of imperative features of Python, similar to that of Ruby. It also encodes the Pyretic NetCore syntax (from Figure 4 in [96]). We omit the full models here; they are included in Appendix B.1. Developing such a model is a one-time investment – once rules for a new language are available, they can be applied to any program in that language.

To verify that these models generate effective fixes, we recreated the scenarios in Section 4.5.3 for Trema and Pyretic. We could not reproduce Q4 in Pyretic because the Pyretic abstraction and its runtime already prevents such problems from happening. Table 4.3 shows our results. Overall, the number of repairs that were generated and passed backtesting are relatively stable across the different languages. For Q1, we found fewer repair candidates for Pyretic than for RapidNet and Trema; this is because an implementation of the same logic in different languages can provide different "degrees of freedom" for possible repairs. (For instance, an equality check `Swi==2` in RapidNet would be $match(switch = 2)$ in Pyretic; a fix that changes the operator to $>$ is possible in the former but disallowed in the latter because of the syntax of *match*.) In all cases, meta provenance produced at least one repair that passed the backtesting phase.

## 4.6 Related Work

We have discussed related work in Chapter 2. In this section, we expland on automated tools for repairing programs.

The software engineering community has used genetic programming [79], symbolic execution [102], and program synthesis [20] to fix programs; they usually rely on a test suite or a formal specification to find fixes and sometimes propose only specific kinds of fixes. In the systems community, ClearView [109] mines invariants in programs, correlates violations with failures, and generates fixes at runtime; ConfDiagnoser [140] compares correct and undesired executions to find suspicious predicates in the program; and Sidiroglou et al. [119] runs attack vectors on instrumented applications and then generates fixes automatically. In databases, ConQueR [126] can refine a SQL query to make certain tuples appear in, or disappear from, the output; however, it is restricted to SPJA queries and cannot handle general controller programs. These systems primarily rely on heuristics, whereas our proposed approach uses provenance to track causality and can thus pinpoint specific root causes more precisely.

In the networking domain specifically, the closest solutions are NetGen [114] and Hojjat et at. [60], which synthesize changes to an existing network to satisfy a desired property or to remove incorrect configurations, which are specified as regular expressions or Horn clauses. While these tools can generate optimal changes, e.g., the smallest number of next-hop routing changes, they are designed for repairing the data plane, i.e., a snapshot of the network configuration at a particular time; our approach repairs control programs and considers dynamic network configuration changes triggered by network traffic.

## 4.7 Conclusion

Network diagnostics is almost a routine for today's operators. However, most debuggers can only find bugs, but not suggest a fix. In this chapter, we have taken a step towards better tool support for network repair, using a novel data structure that we call meta provenance. Like classic provenance, meta provenance tracks causality; but it goes beyond data causality and treats the program as just another kind of data. Thus, it can be used to reason about program changes that prevent undesirable events or create desirable events. While meta provenance falls short of our (slightly idealistic) goal of an automatic "Fix it!" button for SDNs, we believe that it does represent a step in the right direction. As our case studies show, meta provenance can generate high-quality repairs for realistic network problems in one minute, with no help from the human operator.

# 5 Temporal Provenance

## 5.1 Introduction

Debugging networked systems is already difficult for functional problems, such as requests that are processed incorrectly, and this has given rise to a rich literature on sophisticated debugging tools. Diagnosing timing-related problems, such as requests that incur a high delay, adds another layer of complexity: delays are often nondeterministic and can arise from subtle interactions between different components.

However, in practice, locating a bottleneck is only the first step. The operator must then find the *causes* of the bottleneck in order to fix the problem. Existing tools offer far less help with this step. For instance, consider the following scenario: a misconfigured machine is sending a large number of RPCs to a storage backend, which becomes overloaded and delays requests from other clients. When the operator receives complaints from one of the clients about the delayed requests, she can inspect the trace tree or the provenance and identify the bottleneck (in this case, the storage backend). However, neither of these data structures explains *why* the bottleneck exists – in fact, the actual root cause (in this case, the misconfigured machine) would not even appear in either of them!

The reason why existing approaches fall short in this scenario is that they focus exclusively on *functional causality* – they explain why a given computation had some particular result. This kind of explanation looks only at the direct inputs of the computation: for instance, if we want to explain the *existence* of a cup of coffee, we can focus on the source of the coffee beans, the source of the cup, and the barista's actions. In contrast, *temporal causality* may also involve other, seemingly unrelated computations: for instance, the reason *why it took too so long* to get the cup of coffee might be the many customers that were waiting in front of us. At the same time, some functional dependencies may turn out to be irrelevant when explaining delays: for instance, even though the coffee beans were needed to make the coffee, they may not have contributed the delay because they were already available in the store.

The above example illustrates that reasoning about temporal causality requires a very different process than reasoning about functional causality. This is not a superficial difference: as we will show, temporal causality requires some additional information (about event ordering) that existing tracing systems do not capture. Thus, although systems like Dapper or DTaP do record timestamps and thus may appear to be capable of reasoning about time, they are in fact limited to functional causality and use the timestamps merely as an annotation.

In this chapter, we propose a way to reason about temporal causality, and we show how it can be integrated with an existing diagnostic technique – specifically, network provenance. The result is a technique we call *temporal provenance* that can reason about *both* functional *and* temporal causality. We present a concrete algorithm that can generate temporal provenance for distributed systems, and we describe Zeno, a prototype debugger that implements this algorithm. We have applied Zeno to four scenarios with high delay that are based on real incident reports from Google Cloud Engine. Our evaluation shows that, in each case, the resulting temporal provenance

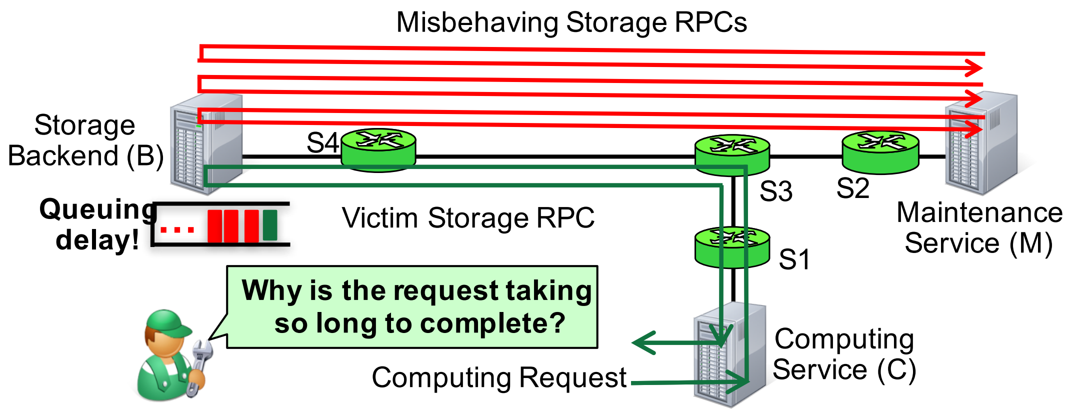


Figure 5.1: Scenario: The misconfigured maintenance service is overloading the storage backend and is causing requests from the computing service to be delayed.

clearly identifies the root cause of the delay; we also show that the runtime overhead is comparable to that of existing tools, such as Zipkin framework [148], which is based on Google Dapper [120]. In summary, our contributions are:

- The concept of temporal provenance (Section 5.2);
- an algorithm that generates temporal provenance (Section 5.4);
- a postprocessing technique that improves the readability of timing provenance graphs (Section 5.5);
- Zeno, a debugger that generates temporal provenance (Section 5.6); and
- an experimental evaluation of Zeno (Section 5.7).

In the following two sections, we begin with an overview of timing diagnostics and its key challenges.

## 5.2 Overview

Figure 5.1 illustrates the example scenario we have already sketched above. An operator manages a small network that connects a maintenance service M, a computing service C, and a storage backend B. Both M and C communicate with the backend using RPCs. A job on M is misconfigured and is sending an excessive number of RPCs (red) to the storage backend. This is causing queuing at the backend, which is delaying RPCs from the computing service (green). The operator notices the delays on C, but is unaware of the misconfiguration on M.

We refer to this situation as a *timing fault*: the RPCs from C are being handled correctly, but not quickly enough. A particularly challenging aspect of this scenario is that the root cause of the delays that C's requests are experiencing (the misconfiguration on M) is not on the path from C to B; we call this an *off-path root cause*.

Timing faults are quite common in practice. To illustrate this, we surveyed incidents disclosed by Google Cloud Platform [50], which occur across a variety of different cloud services and directly impact cloud tenants. We examined 95 incidents that happened from Jan 2014 until May 2016; all incident reports describe both the symptom and the root cause. We find that more than a third (34.7%) of these incidents were timing faults.

### 5.2.1 Prior work: Trace trees

Today, a common way to diagnose such a situation is to track the execution of the request and to identify the bottleneck – that is, components that are contributing unusual delays. For instance, a distributed tracing system would produce a "trace tree" [120]. Figure 5.2 shows an example tree for one of the delayed responses from the computing service C in Figure 5.1. The yellow bars represent basic units of work, which are usually referred to as *spans*, and the up/down arrows indicate causal relationships between a span and its parent span. A span is also associated with a simple log of timestamped records that encode events within the span.

Trace trees are helpful because they show the steps that were involved in executing the request: the computation was started at $t_0$ and issued an RPC to the storage backend at $t_1$; the backend received the RPC at $t_2$, started processing it at $t_3$, and sent a response at $t_4$, which the client received at $t_5$; the computation ended at $t_6$. This

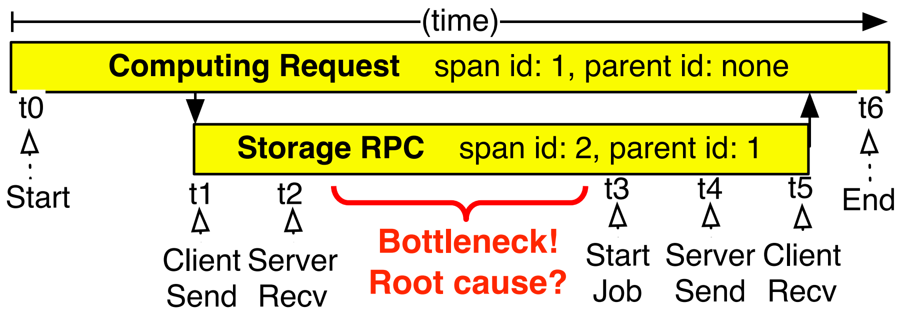


Figure 5.2: A trace tree for the delayed computing requests in Figure 5.1. B received the storage RPC at $t_2$ but only started processing it at $t_3$, after a long queuing delay.

data structure helps the operator to find abnormal delays: for instance, the operator will notice that the RPC waited unusually long ($t_2 \ldots t_3$) before it was processed by the storage backend.

However, the operator also must understand what *caused* the unusual delay, and trace trees offer far less help with this step. In our scenario, the root cause – the misbehaving maintenance service – never even appears in any span! The reason is that *trace trees include only the spans that are on the execution path* of the request that is being examined. In practice, off-path causes are very common: when we further investigated the 33 timing faults in our survey from above, we found that, in over 60% of the cases, the real problem was not on the execution path of the original request, so it would not have appeared in the corresponding trace tree.

### 5.2.2 Prior work: Provenance

Another approach that has been explored recently [134, 24, 133, 146, 147, 145] is to use *provenance* [17] as a diagnostic tool. Provenance is a way to obtain causal explanations of an event; a provenance system maintains, for each (recent) event in the network, a bit of metadata that keeps track of the event's direct causes. Thus, when the operator requests an explanation for some event of interest (say, the arrival of a packet), the system can produce a recursive explanation that links the event to a set of root causes (such as the original transmission of the packet and the relevant rout-

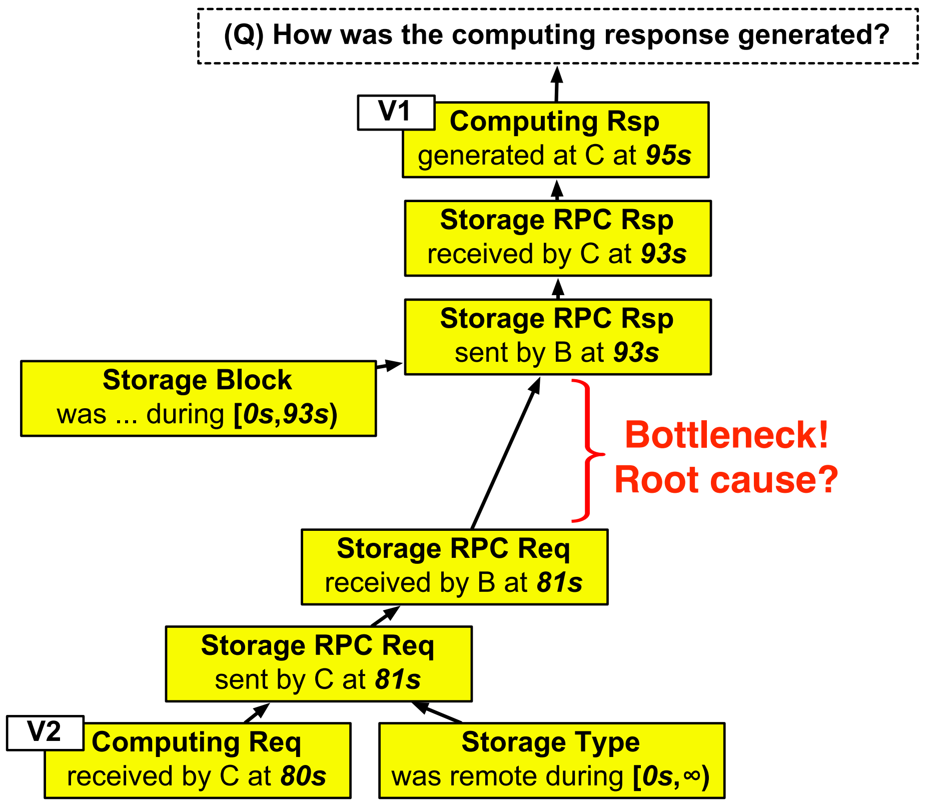

(Q) How was the computing response generated?
V1
Computing Rsp generated at C at 95s
Storage RPC Rsp received by C at 93s
Storage RPC Rsp sent by B at 93s
Storage Block was ... during [0s,93s)
Bottleneck! Root cause?
Storage RPC Req received by B at 81s
Storage RPC Req sent by C at 81s
V2
Computing Req received by C at 80s
Storage Type was remote during [0s,∞)

Figure 5.3: Time-aware provenance, as in DTaP [146], for the example scenario from Figure 5.1.

ing state). The explanation can be represented as a DAG, whose vertexes represent events and whose edges represent direct causality.

Figure 5.3 shows the tree that a provenance system like DTaP [146] would generate for our example scenario. (We picked DTaP because it proposed a "time-aware" variant of provenance, which already considers a notion of time.) This data structure is considerably more detailed than a trace tree; for instance, it not only shows the path from the original request (V2) to the final response (V1), but also the data and the configuration state that were involved in processing the request along the way. However, the actual root cause from the scenario (the misconfigured maintenance service) is *still* absent from the data structure. The reason is that DTaP's provenance is "time-aware" only in the sense that it can remember the provenance of past system states. It does annotate each event with a timestamp, as shown in the figure, but it

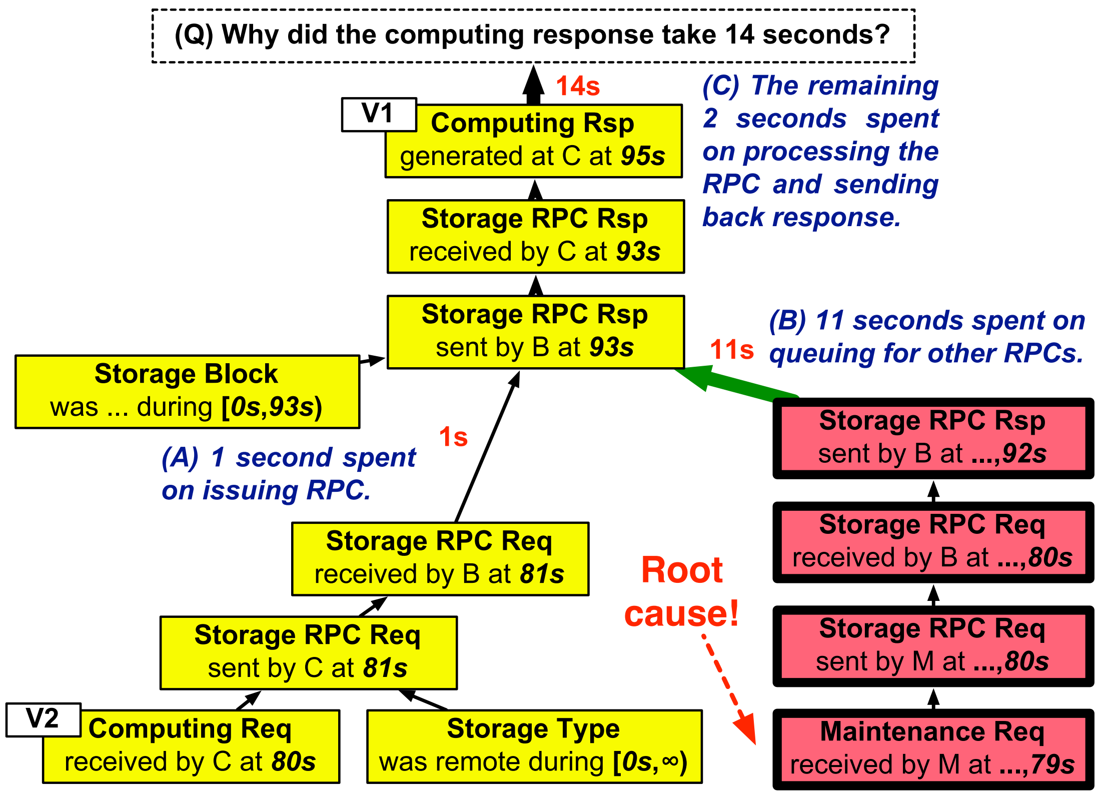

(Q) Why did the computing response take 14 seconds?
14s
V1
Computing Rsp generated at C at 95s
(C) The remaining 2 seconds spent on processing the RPC and sending back response.
Storage RPC Rsp received by C at 93s
Storage RPC Rsp sent by B at 93s
(B) 11 seconds spent on queuing for other RPCs.
11s
Storage Block was ... during [0s,93s)
1s
(A) 1 second spent on issuing RPC.
Storage RPC Rsp sent by B at ...,92s
Storage RPC Req received by B at ...,80s
Storage RPC Req received by B at 81s
Root cause!
Storage RPC Req sent by C at 81s
Storage RPC Req sent by M at ...,80s
V2
Computing Req received by C at 80s
Storage Type was remote during [0s,∞)
Maintenance Req received by M at ...,79s

Figure 5.4: Temporal provenance, as proposed in this chapter, for the example scenario from Figure 5.1.

does not reason about temporal causality. Thus, it actually does not offer very much extra help compared to trace trees: like the latter, it can be used to *find* bottlenecks, such as the high response times in the backend, but it is not able to *explain* them.

## 5.2.3 Our approach

We propose to solve this problem with a combination of three insights. The first is that temporal causality critically depends on a type of information that existing tracing techniques tend not to capture: the *sequence* in which the system has processed requests, *whether the requests are related or not*. By looking only at functional dependencies, these techniques simply consider each request in isolation, and thus cannot make the connection between the slow storage RPC and the requests from the maintenance service that are delaying it. With provenance, we can fix this by including a second kind of edge $e_1 \rightarrow e_2$ that connects each event $e_1$ to the event $e_2$ that was processed on the same node and immediately after $e_1$. We refer to these edges as *sequencing edges* (Section 5.4.1).

Our next insight is a connection between temporal reasoning and the critical path analysis from scheduling theory. When scheduling a set of parallel tasks with dependencies, the critical path is the dependency chain with the longest accumulative execution time, and it determines the overall completion time. We extend this concept to our more fine-grained setting, and we turn it into a method that recursively allocates delay to the various branches of the provenance tree (Section 5.4.3). The result is a data structure we call *temporal provenance*.

Our third insight has to do with readability. At first glance, temporal provenance is considerably more complex than classical provenance because it considers not only functionally related events, but also events that may have contributed only delay (of which there can be many). However, in practice, many of these events do not actually contribute any delay, e.g., because they are not on the critical path; the ones that do are often structurally similar, such as the maintenance requests in our example scenario, and can thus be aggregated. Thus, it is usually possible to extract a compact representation that can be easily understood by the human operator (Section 5.5).

Figure 5.4 shows the temporal provenance for a random computing request in our example scenario. Starting at the root, the provenance forks into two branches; the thin branch shows that one second was spent on issuing the RPC itself (A); and the thick branch shows that the majority of the delay (11 seconds) was caused by RPCs from the maintenance service (B). This tree has all the properties we motivated earlier: it provides a quantitative explanation of the delay, and it includes the actual root cause (the maintenance service), even though it does not appear on the path the request has taken.

## 5.3 Background

Since temporal provenance is a generalization of network provenance, we begin with a brief description of the latter, and refer interested readers to [145] for more detail.

### 5.3.1 System model

If our goal was classical data provenance, the description of NDlog in Section 2.1 would already be sufficient. However, since we are particularly interested in *timing*, we need to also consider some details of the NDlog runtime. The runtime execution consists of insertions and deletions of tuples, which we refer to as *updates*. For instance, in a network that is modeled in NDlog, the updates could include packet arrivals or configuration changes. The tuple can then trigger additional updates, which may in turn cause more updates, etc., until the state stabilizes again.

One important question that arises is what should happen when an update triggers more than one update – specifically, in what order these updates should be processed. Here, we will assume that the execution follows *pipelined semi-naïve evaluation* (PSN) [83], which is a good fit for networks and services with FIFO queues. In PSN, each node maintains an *update queue* in which it buffers all updates whose effects have not yet been applied to the tables. The node picks an update from the queue according to FIFO policy, applies it to the tables, and then computes whether any additional updates have been triggered. If so, local updates are added to the local queue; remote updates are sent to the remote node.

### 5.3.2 Classical provenance

In order to be able to answer provenance queries, a system must collect some additional metadata at runtime. Conceptually, this can be done by maintaining a large DAG, the *provenance graph*, that contains a vertex for every event that has occurred in the system, and in which there is an edge $(a, b)$ between two vertexes if event $a$ was a direct cause of event $b$. (A practical implementation would typically not maintain this graph explicitly, and instead collect only enough information to reconstruct a recent subgraph when necessary; however, we will use this concept for now because it is easier to explain.) If the system later receives a provenance query PROV($e$) for some event $e$, it can find the answer by locating the vertex that corresponds to $e$ and then projecting out the subgraph that is rooted at $e$. This subgraph will be the *provenance* of $e$.

For concreteness, we will use a provenance graph with six types of vertexes, which is loosely based on [146]:

- INS($[t_s,t_e]$,$N$,$\tau$): Base tuple $\tau$ was inserted on node $N$ during $[t_s,t_e]$;
- DEL($[t_s,t_e]$,$N$,$\tau$): Base tuple $\tau$ was deleted on node $N$ during $[t_s,t_e]$;
- DRV($[t_s,t_e]$,$N$,$\tau$): Derived tuple $\tau$ acquired support on $N$ during $[t_s,t_e]$;
- UDRV($[t_s,t_e]$,$N$,$\tau$): Derived tuple $\tau$ lost support on $N$ during $[t_s,t_e]$;
- SND($[t_s,t_e]$,$N \rightarrow N'$,$\pm\tau$): $\pm\tau$ was sent by $N$ to (from) $N'$ during $[t_s,t_e]$; and
- RCV($[t_s,t_e]$,$N \leftarrow N'$, $\pm\tau$): $\pm\tau$ was received by $N$ to (from) $N'$ during $[t_s,t_e]$.

Note that each vertex is annotated with the node on which it occurred, as well as with a time interval that indicates when the node processed that event. For instance, when a switch makes a forwarding decision for a packet, it derives a new tuple that specifies the next hop, and the time $[t_s,t_e]$ that was spent on this decision is indicated in the corresponding DRV vertex. This will be useful (but not yet sufficient) for temporal provenance later on.

The edges between the vertexes represent their causal relationships. A SND vertex has an edge from an INS or a DRV that produced the tuple that is being sent; a RCV has an edge from the SND vertex for the received message; and a DRV vertex for a rule `A:-B,C,D` has an edge from each precondition (B, C, and D) that leads to the vertex that produced the corresponding tuple. An INS vertex corresponds to the insertion of a base tuple, which cannot be explained further; thus, it has no incoming edges. The edges for the negative "twins" of these vertexes – UDRV and DEL – are analogous to their positive counterparts.

### 5.3.3 Desirable properties

Before we go on, we briefly reflect on some key properties of this existing notion of provenance. First, provenance is *recursive*: the provenance of an event *e* includes, as subgraphs, the provenances of all the events that contributed to *e*. This is useful to an operator because she can start at the root and "drill down" into the explanation until she identifies a root cause. Second, there is a *single data structure* – the provenance graph – that can be maintained at runtime, without knowing a priori what kinds of queries will be asked later on.

Besides this, there are some key properties (formulated, e.g., in [146]) that we would expect the provenance to have. These include validity (the provenance describes a correct execution of the system's program), soundness (the provenance respects happens-before relationships), completeness (the provenance fully explains the event of interest), and minimality (the provenance contains no more vertexes than necessary). These properties are clearly important for the provenance to "make sense", so we should expect a generalization to preserve them.

## 5.4 Temporal provenance

In this section, we generalize the basic provenance model from Section 5.3 to reason about the timing of events.

### 5.4.1 Sequencing edges

The provenance model we have introduced so far would produce provenance that looks like the tree in Figure 5.3: it would explain why the event at the top occurred, but it would not explain why the event occurred *at that particular time*. The fact that the vertexes are annotated with timestamps, as in prior work [146], does not change that fact: the operator would be able to see, for instance, that the storage service took a long time to respond to a request, but the underlying reason (the requests from another node that were queued in front of it) is not shown; in fact, it does not even appear in the graph!

To rectify this, we need to capture some additional information – namely the *sequence* in which events were processed by a given node. Thus, we introduce a second type of edge that we call *sequencing edge*. A sequencing edge ($v_1$,$v_2$) exists between two vertexes *a* and *b* iff either a) the corresponding events happened on the same node, and *a* was the event that immediately preceded *b*, or b) *a* is an SND vertex and *b* is the corresponding RCV vertex. We refer to the first type of edge as a *local* sequencing edge, and to the second type as a *remote* sequencing edge. In the illustrations, we will render the sequencing edges with green, dotted lines to distinguish them from the causal edges that are already part of classical provenance. Notice that these edges essentially capture the well-known happens-before relation [77].

Although causal edges and sequencing edges often coincide, they are in fact orthogonal. For instance, consider the scenario in Figure 5.5(a). Here, a node X has two rules, `B:-A` and `C:-A`; in the concrete execution (shown on the timeline), A is inserted at time 0, which triggers both rules, but B is derived first, and then C. In the provenance graph (shown at the bottom), INS(A) is connected to DRV(B) by both a causal and a sequencing edge, since the two events happened back-to-back and B's derivation was directly caused by A's insertion. But DRV(B) is connected to DRV(C) only by a sequencing edge, since the former did precede the latter but was not a direct cause; in contrast, INS(A) is connected to DRV(C) only by a causal edge, since A's insertion did cause C's derivation, but the latter was delayed by another event.

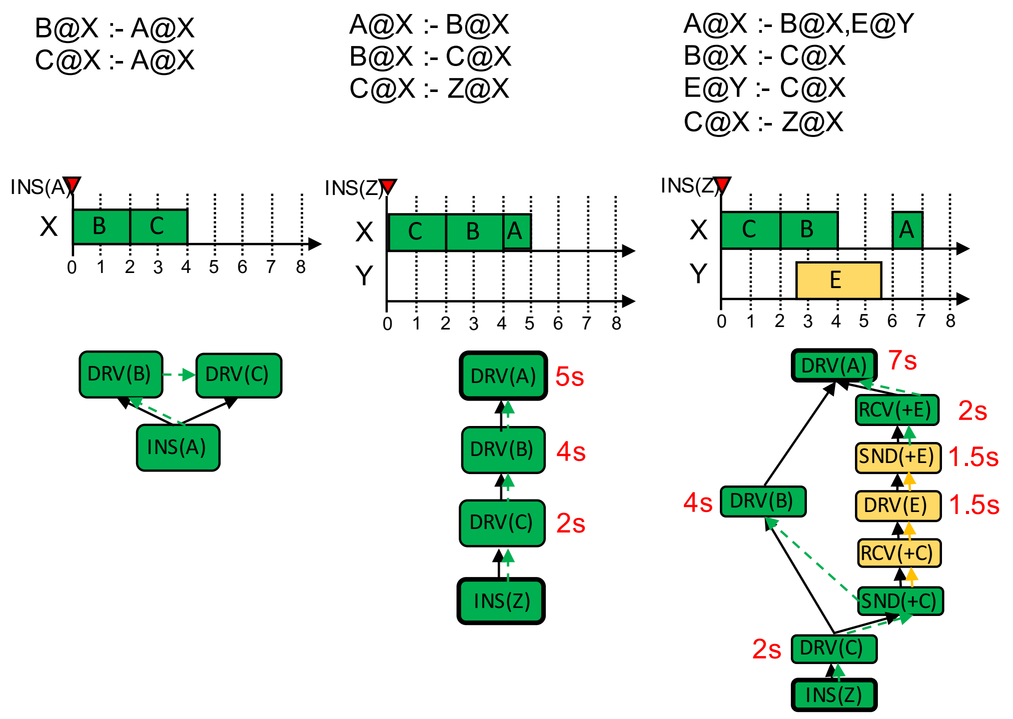
### 5.4.2 Queries

Next, we turn to the question what a query for temporal provenance should look like, and what it should return. Unlike a classical provenance query QUERY(e), which aims to explain a specific event $e$, a temporal provenance query aims to explain a *delay* between *a pair of* events $e_1$ and $e_2$. For instance, in the scenario from Figure 5.1, the operator wanted to know why his request had taken so long to complete, which is, in essence, a question about the delay between the request itself ($e_1$) and the resulting response ($e_2$). Hence, the primitive we provide is a query T-QUERY($e_1$,$e_2$), which asks about delay between $e_1$ and $e_2$. Our only requirement is that the events are causally related – i.e., that there is a causal path from $e_1$ to $e_2$.

As a first approximation, we can answer T-QUERY($e_1$,$e_2$) as follows. We first query the classical provenance $P :=$ QUERY($e_2$). Due to the above requirement, $P$ will include a vertex for $e_1$. We then identify all pairs of vertexes ($v_1$,$v_2$) in $P$ that are connected by a causal edge but not by a sequencing edge. We note that, a) in each such pair, $v_2$ must have been delayed by some other intervening event, and b) $v_1$ is nevertheless connected to $v_2$ via a multi-hop path along the sequencing edges. (The reason is simply that $v_1$ was one of $v_2$'s causes and must therefore have happened before it.) Thus, we can augment the causal provenance by adding these sequencing paths, as well as the provenance of any events along such a path. The resulting provenance $P'$ contains all the events that have somehow contributed to the delay between $e_1$ and $e_2$. We can then return $P'$ as the answer to T-QUERY($e_1$,$e_2$).

### 5.4.3 Delay annotations

As defined so far, the temporal provenance still lacks a way for the operator to tell *how much* each subtree has contributed to the overall delay. As discussed in Section 5.3.3, this is important for usability: the operator should have a way to "drill down" into the graph to look for the most important causes of delay. To facilitate



(a) (b) (c)

Figure 5.5: Example scenarios (a)-(c), with NDlog rules at the top, the timing of a concrete execution in the middle, and the resulting temporal provenance at the bottom. The query is T-QUERY(INS(Z), DRV(A)) in all scenarios; the start and end vertexes are marked in bold. Vertex names have been shortened and some fields have been omitted for clarity.

this, we additionally annotate the vertexes with the delay that they (and the subtrees below them) have contributed.

Computing these annotations is surprisingly nontrivial and involves some interesting design decisions. Our algorithm is shown in Figure 5.6; we explain it below in several refinements, using the simple examples in Figures 5.5(b)–(c) and in Figures 5.7(d)–(f). The examples are shown in the same format as in Figure 5.5(a): each shows a set of simple NDlog rules, the timing of events during the actual execution, and the resulting temporal provenance, with the delay annotations in red. The query is always the same: T-QUERY(INS(Z),DRV(A)); that is, we want to explain the delay between the insertion of Z and the derivation of A. One difference to Figure 5.5(a) is that some of the examples require two nodes, X and Y. To make the connections

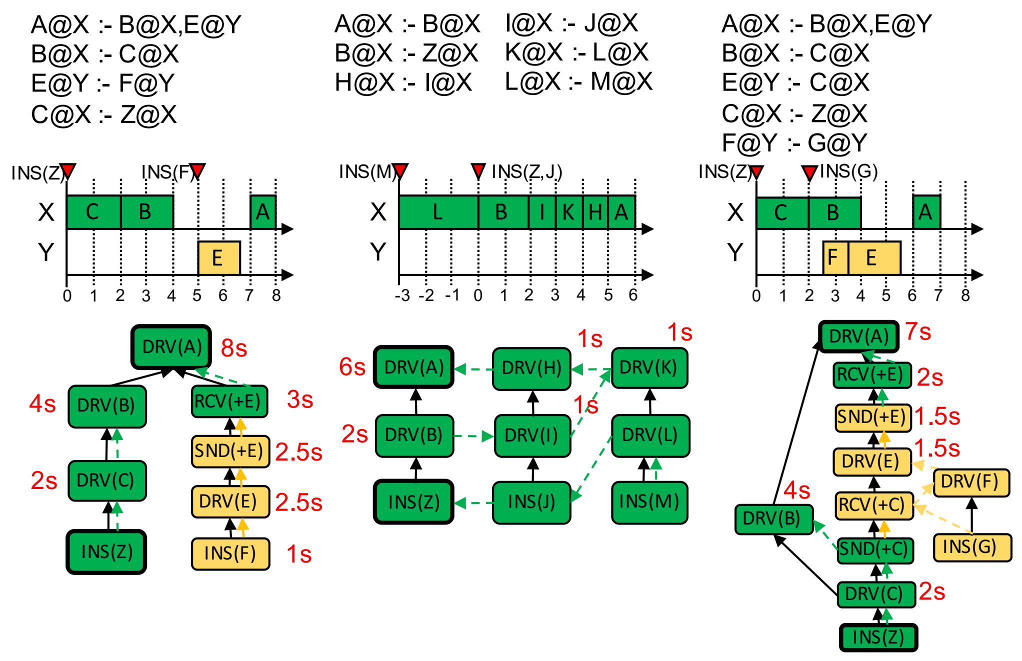
```
// the subtree rooted at $v$ is responsible for the delay during $[t_s, t_e]$
function ANNOTATE($v$, $[t_s, t_e]$)
    ASSERT($t_e == t_{end}(v)$)
    if $[t_s, t_e] = \emptyset$ then
        RETURN

    // weight $v$ by the amount of delay it contributes
    SET-WEIGHT($v$, $t_e - t_s$)

    // recursive calls for functional children in order of appearance
    C $\leftarrow$ FUNCTIONAL-CHILDREN(v)
    T $\leftarrow t_s$
    while $C \neq \emptyset$ do
        $v' \leftarrow c \in C$ WITH MIN $t_{end}(c)$
        C $\leftarrow C \setminus \{v'\}$
        if $t_{end}(v') \geq T$ then
            ANNOTATE($v'$, $[T, t_{end}(v')]$)
            $T \leftarrow t_{end}(v')$

    // recursive calls for sequencing children
    s $\leftarrow$ SEQUENCING-CHILD(v)
    E $\leftarrow t_{start}(v)$
    while $T < E$ do
        ANNOTATE(s, $[\text{MAX}(T, t_{start}(s)), E]$)
        $E \leftarrow t_{start}(s)$
        $s \leftarrow$ SEQUENCING-CHILD(s)
```

Figure 5.6: Algorithm for computing delay annotations (explained in Sections 5.4.3–5.4.5).

more visible, we show the vertexes that belong to Y in orange, and the ones that belong to X in green, as in Figure 5.5(a). If a vertex did not contribute to the delay at all, we omit its annotation.

Our algorithm computes the delay annotations recursively. A function ANNOTATE is called on the root of the provenance; the function then invokes itself on (some of) the children to compute the annotations on the subgraphs. As a first approximation, this works as follows:

**Rule #1: Annotate the top vertex with the overall delay $T$, and then recursively annotate each (causal) child with $T - t$, where $t$ is the parent's execution time.**



(d) (e) (f)

Figure 5.7: Example scenarios (d)-(f), with NDlog rules at the top, the timing of a concrete execution in the middle, and the resulting temporal provenance at the bottom. The query is T-QUERY(INS(Z), DRV(A)) in all scenarios; the start and end vertexes are marked in bold. Vertex names have been shortened and some fields have been omitted for clarity.

In our algorithm, this corresponds to line 8, which sets the weight for the current vertex, and the recursive call in line 17; lines 4–5 contain the base case, where the delay is zero.

### 5.4.4 Handling multiple preconditions

This approach works well for linear provenance, such as the one in Figure 5.5(b): deriving A from Z took 5s because it took 1s to compute A itself, and 4s to derive A's precondition, B; deriving B from Z took 4s because 2s were spent on B itself and another 2s on C. However, it does *not* work well for rules with multiple preconditions. Consider the scenario in Figure 5.5(c): A now has two preconditions, B and E, so the question is how much of the overall delay should be attributed to each.

Two answers immediately suggest themselves: 1) since B finished its derivation after 4s, we can attribute 4s to B and the remaining 2s to E, which finished later, or 2) we can attribute the entire 6s to E, because it was the last to finish. The latter is somewhat similar to the choice made in critical path analysis [29, 136]; however, the theorems in Section 5.4.6 actually require the former: if we find a way to speed up $E$ (or cause $F$ to be inserted sooner), this can only reduce the end-to-end delay by 3s. Any further reductions would have to come from speeding up $B$. This leads to the following refinement:

**Refinement #2: Like rule #1, except that the remaining delay is allocated among the preconditions in the order in which they were satisfied.**

This refinement is implemented in lines 13–15 and 18 of our algorithm, which iterate through the preconditions in the order of their appearance (that is, local derivation or arrival from another node) and allocate to each the interval between its own appearance and the appearance of its predecessor.

Notice that this approach deviates from critical-path analysis in an interesting way. Consider the scenario in Figure 5.7(d): here, the provenance has two "branches", one connected to the insertion of Z and the other to the insertion of F, but there is no causal path from Z to F. (We call such a branch an *off-path branch*, and any other branch an *on-path branch*.) This raises the question whether any delay should be apportioned to off-path branches, and if so, how much. Critical path analysis has no guidance to offer for this case because it only considers tasks that are transitively connected to the start task.

At first glance, it may seem that F's branch should not get any delay at all; for instance, F could be a configuration entry that is causally unrelated to Z and thus did not obviously contribute to a delay from Z to A. However, notice that all the "on-path" derivations (in Z's branch) finished at $t = 4s$, but A's derivation was nevertheless delayed until $t = 7s$ because E was not yet available. Thus, it seems appropriate that the remaining 3s should be apportioned to the other branch.

### 5.4.5 Handling sequencing delays

The one question that remains is what to do if there is further delay after the last precondition is satisfied. This occurs in the scenario in Figure 5.7(e): although B is derived immediately after Z is inserted at $t = 0$, A's derivation is delayed by another 3s due to some causally unrelated derivations (I, K, and H). Here, the sequencing edges come into play: we can attribute the remaining delay to the predecessor along the *local* sequencing edge (here, DRV(H), which will subtract its own computation time and pass on any remaining delay to its own predecessor, etc. This brings us to the final rule:

**Final rule: Like #2, except that, if any delay remains after the last precondition, that delay is attributed to the predecessors along the local sequencing edge.**

This is implemented in lines 20–26.

So far, we have focused on the rule for DRV vertexes, which is the most complex one. SND vertexes are easy because they only have one (causal) child; RCV vertexes are even easier because they cannot be delayed by sequencing at all; and INS vertexes are trivial because they have no causal children.

### 5.4.6 Properties

Our generalization preserves all of the four key properties of provenance from [146]: the result of T-QUERY($e_1$,$e_2$) is valid, sound, complete, and minimal. Formal statements of these properties, as well as the corresponding proofs, is in Appendix C.1.

Temporal provenance also satisfies almost all of the requirements we have described in Section 5.3.3: it is recursive – that is, it allows the operator to "zoom in" on the root cause of a problem by following a path from the root – and it is independent of a particular query, which means that the provenance can be recorded at runtime without knowing what queries will asked later on. Our only sacrifice is that the delay annotations are query-specific and must be computed during query processing; however, as we will show in Section 5.7.4, this takes only a few milliseconds, so the cost should not be significant.

We have also formally modeled the properties of the delay annotations that our algorithm creates. We again omit the details for lack of space (they are in the appendix), but we briefly summarize our key findings. Our model first carefully defines what it means for a derivation $\tau : -c_1, c_2, \ldots$ to be directly "delayed" by one of its preconditions, and then recursively extends this definition to transitive delays (that is, one of the $c_i$ was itself delayed by one of its own preconditions, etc.). Our first theorem (Section C.1.5) states that each vertex is labeled with the amount of (direct or transitive) delay that is contributed by the subtree that is rooted at that vertex. Our second theorem (Section C.1.6) essentially states that, if there is a vertex $v$ in a temporal provenance tree that is annotated with $T$ and the sum of the annotations on its children and immediate predecessors is $S < T$, then it is possible to construct another valid (but hypothetical) execution in which $v$'s execution time is reduced by $(T - S)$ and in which the derivation finishes $(T - S)$ units of time earlier. Thus, the annotations really do correspond to the "potential for speedup" that we intuitively associate with the concept of delay.

## 5.5 Improving readability

As defined in the previous section, temporal provenance is already useful for diagnostics because it can explain the reasons for a delay between two events. However, the provenance can be somewhat verbose and hard to read for a human operator. There are two reasons for this. The first is that the temporal provenance for $[e_1, e_2]$ contains the entire classical provenance of $e_2$ as a subgraph, even though some of the functional causes did not actually contribute to the delay and are thus irrelevant if the delay is the only thing the operator is interested in. The second reason is that sequencing delay is often the result of many similar events (such as other packets queued in front of the packet of interest) that each contribute a relatively small amount of delay. Including the full provenance of each such event results in a somewhat cluttered graph that makes it difficult to see the “big picture”.

To make the graph more usable for operators, we perform two post-processing steps, which we describe next.

### 5.5.1 Pruning zero-delay subgraphs

Our first step hides any vertexes that are annotated with zero (or not annotated at all) by the ANNOTATE function. The only exception is that we keep vertexes that are connected via a causal path (i.e., a path with only causal edges) to a vertex that is annotated with a positive delay. For instance, in the samples from Figure 5.5 and Figure 5.7, the original INS(z) vertex – which is the starting point of the interval – would be preserved, even though the insertion itself did not contribute any delay.

To illustrate the effect of this step we consider the example in Figure 5.7(f), which is almost identical to the one in Figure 5.5(c), except that an additional, unrelated derivation (F) occurred before the derivation of E. Here, the INS(G) and the DRV(F) would be hidden because they do not contribute to the overall delay.

### 5.5.2 Provenance aggregation

Our second post-processing step aggregates subgraphs that are structurally similar [115]. This helps with cases where there are many events that each contribute only a very small amount of delay. For instance, in our scenario from Figure 5.1, the delay is caused by a large number of RPCs from the maintenance service that are queued in front of the RPC whose delay is being explained. The “raw” temporal provenance contains a subtree for each such RPC. During postprocessing, these nearly identical subtrees would be aggregated into a single subtree whose weight is the sum of the weights of the individual trees, as shown in Figure 5.4.

There are two key challenges with this approach. The first is to decompose the temporal provenance into smaller subgraphs that can potentially be aggregated. At first glance, there are exponentially many decompositions, so the problem seems intractable. However, we observe that (1) aggregation is most likely to be possible for sequencing delays, which are often due to similar kinds of events (network packets, RPCs) that have a similar provenance; and that (2) the corresponding subtrees can easily be identified in the temporal provenance because they are laterally connected to the functional provenance through a chain of sequencing edges. Thus, we can extract candidates simply by following such lateral sequencing edges and by taking the subgraphs below any vertexes we encounter.

The second challenge is that the candidate subgraphs are often similar but rarely identical. Thus, we need to define an equivalence relation that controls which vertexes can be considered "similar" and are thus safe to aggregate. We use a simple heuristic that considers two vertexes to be similar for aggregation purposes if they share a tuple name and have been derived on the same node. To aggregate two subgraphs, we start at their root vertexes; if the vertexes are similar, we merge them, annotate them with the sum of their individual annotations, and recursively attempt to merge pairs of their children. If the vertexes are not similar, we stop aggregation at that point and connect the two remaining subgraphs directly to the last vertex that was successfully aggregated.

The aggregation procedure is commutative and associative; thus, rather than attempting aggregation for all pairs of subgraphs, we can simply try to aggregate each new subgraph with all existing aggregates. In our experience, the $O(N^2)$ complexity is not a problem in practice because $N$ is often relatively small and/or most of the subgraphs are similar, so there are very few aggregates.

## 5.6 The Zeno debugger

We have built a debugger called Zeno that can generate temporal provenance using the algorithm described above. Zeno can be configured to run with RapidNet [84], a declarative networking engine, as well as Zipkin [148], a distributed tracing framework. Our Zeno prototype is written in 16,960 lines of code in C++, and it includes three main components.

**Runtime:** To demonstrate that Zeno can work with different languages and platforms, we built two different front-ends. The first is integrated with RapidNet [84] and enables Zeno to generate temporal provenance for NDlog programs. The second is integrated with the Zipkin [148] framework – a cross-language and cross-framework distributed tracing library that is based on Google Dapper [120] and can run a network of microservices written in Node.js [103] (JavaScript) and Pyramid [110] (Python). Both front-ends share the same back-end for reasoning about temporal provenance. In our evaluation, we use the first front-end for SDN applications and the second for native Zipkin services.

**Provenance recorder:** At runtime, our debugger must record enough metadata to be able to answer provenance queries later on. Previous work [147, 82, 134] has already shown that provenance can be captured at scale; this is typically done either (1) by explicitly recording all events and their direct causes, or (2) by recording only nondeterministic inputs at runtime, and by later replaying the execution with additional instrumentation to extract events and causes if and when a query is actually asked [146]. The Zipkin front-end follows the first approach, because Zipkin already has well-defined interfaces to capture both base events and intermediate events (such as RPC invocations and completions) in a distributed system, which yields a complete trace tree for each request. Therefore, Zeno merely adds a post-processing engine that converts the trace trees to functional provenance and that infers the sequencing edges from the recorded sequence of events across all trace trees. The NDlog front-end uses the second approach and is based on the record and replay engine from DTaP [146].

Obtaining sequencing edges is not always straightforward, especially at the network switches. Fortunately, we can leverage the in-band network telemetry (INT) capability [72] in emerging switches [15] to obtain sequencing edges [101].

INT-capable switches can stamp into a packet's header the queue depths and the ingress/egress timestamps at each hop, which is exactly what we need in order to obtain sequencing edges. We have implemented an extension in our prototype to approximate this capability. Although we have implemented INT in software, an INT-capable switch can perform these operations in hardware, at line speed.

**Query processor:** The third and final component accepts queries T-QUERY($e_1$,$e_2$) from the operator, as defined in Section 5.4.2, and it answers each query by first executing the algorithm from Section 5.4 to generate the raw temporal provenance and then applying the post-processing steps from Section 5.5 to improve readability. The resulting graph is then displayed to the operator.

## 5.7 Evaluation

In this section, we report results from an experimental evaluation of our debugger. We have designed our experiments to answer four high-level questions: 1) Is temporal provenance useful for debugging realistic timing faults? 2) What is the cost for maintaining temporal provenance? 3) How fast can our debugger process diagnostic queries? And 4) Does temporal provenance scale well with the network size?

We ran our experiments on a Dell OptiPlex 7010 workstation, which has a 4-core 3.20 GHz Intel i5-3470 CPU with 16 GB of RAM. The OS was Ubuntu 13.10, and the kernel version was 3.11.0.

### 5.7.1 Diagnostic scenarios

For our experiments, we reproduced the following four representative scenarios that we sampled from incidents reported by Google Cloud Engine [50]:

- **R1, Z1: Misbehaving maintenance task** [47]**.** Clients of the Compute Engine API experienced delays of up to two minutes because a scheduled maintenance task caused queuing within the compute service. This is the scenario from Section 5.2.

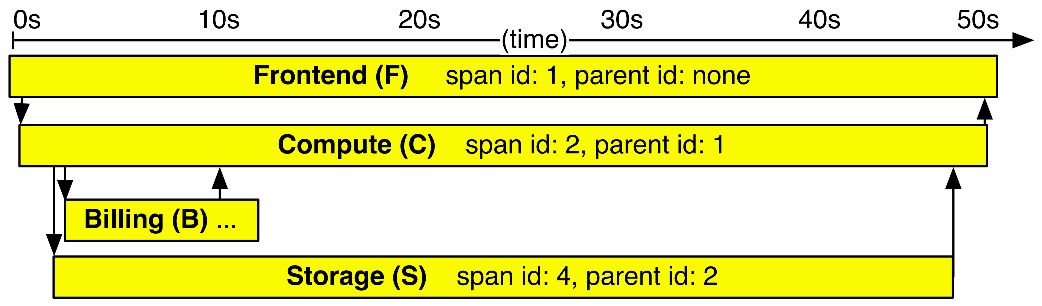

0s
10s
20s
30s
40s
50s
(time)
Frontend (F) span id: 1, parent id: none
Compute (C) span id: 2, parent id: 1
Billing (B) ...
Storage (S) span id: 4, parent id: 2

Figure 5.8: Zipkin trace tree for scenario Z1. The tree clearly shows that the RPC to the storage service is the bottleneck, but the actual root cause (the RPCs from the maintenance service) is off-path and thus is absent.

- **R2, Z2: Elevated API latency** [46]**.** A failure caused the URL Fetch API infrastructure to migrate to a remote site. This increased the latency, which in turn caused clients to retry, worsening the latency. Latencies remained high for more than 3 hours.
- **R3: Slow deployments after release** [48]**.** A new release of App Engine caused the underlying pub/sub infrastructure to send an update to each existing instance. This additional load slowed down the delivery of deployment messages; thus, the creation of new instances remained slow for more than an hour.
- **R4: Network traffic changes** [49]**.** Rapid changes in external traffic patterns caused the networking infrastructure to reconfigure itself repeatedly, which created a substantial queue of modifications. Since the network registration of new instances had to wait on events in this queue, the deployment of new instances was slowed down for 90 minutes.

We reproduced all scenarios in RapidNet (R1–R4) and the first two additionally in the microservice environment (Z1–Z2). The first two scenarios are relatively small; the NDlog versions use four switches, one controller, and three servers, while the microservice versions used 8 and 6 servers, respectively. (We do not model switches in the microservice scenarios.) For the last two scenarios, we used 4 switches and one controller but a larger number of servers: 115 for R3, and 95 for R4.

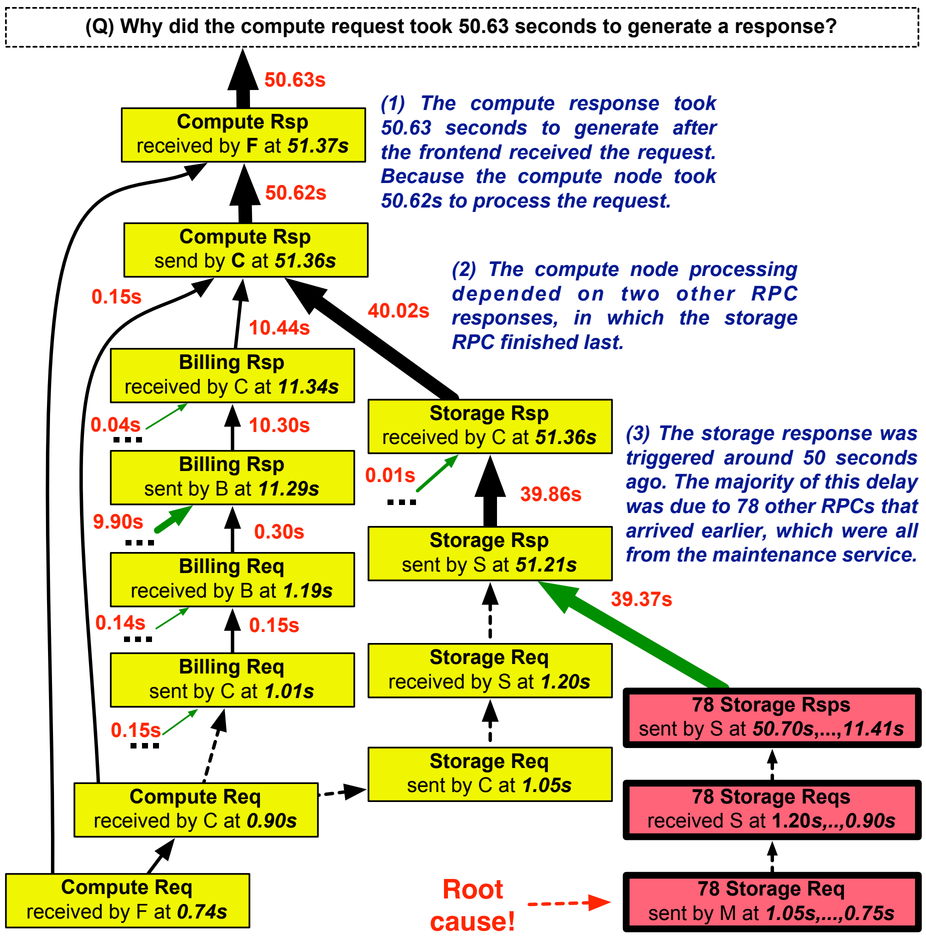

(Q) Why did the compute request took 50.63 seconds to generate a response?
50.63s
Compute Rsp received by F at 51.37s
(1) The compute response took 50.63 seconds to generate after the frontend received the request. Because the compute node took 50.62s to process the request.
50.62s
Compute Rsp send by C at 51.36s
0.15s
10.44s
40.02s
(2) The compute node processing depended on two other RPC responses, in which the storage RPC finished last.
Billing Rsp received by C at 11.34s
0.04s
10.30s
Storage Rsp received by C at 51.36s
(3) The storage response was triggered around 50 seconds ago. The majority of this delay was due to 78 other RPCs that arrived earlier, which were all from the maintenance service.
Billing Rsp sent by B at 11.29s
0.01s
39.86s
9.90s
0.30s
Storage Rsp sent by S at 51.21s
Billing Req received by B at 1.19s
39.37s
0.14s
0.15s
Billing Req sent by C at 1.01s
Storage Req received by S at 1.20s
78 Storage Rsps sent by S at 50.70s,...,11.41s
0.15s
Compute Req received by C at 0.90s
Storage Req sent by C at 1.05s
78 Storage Reqs received S at 1.20s,...,0.90s
Compute Req received by F at 0.74s
Root cause!
78 Storage Req sent by M at 1.05s,...,0.75s

Figure 5.9: Temporal provenance for scenario Z1. In contrast to the trace tree in Figure 5.8, the off-path root cause (the requests from the maintenance service) does appear and can easily be found by starting at the root and by following the vertexes with the highest delay.

### 5.7.2 Identifying off-path causes

A key motivator for this work is the fact that off-path root causes for a delay are often not even visible with an existing debugger. To test whether Zeno addresses this, we generated trace trees (using Zipkin) and classic provenance (using DTaP), and compared their ability to identify off-path root causes with temporal provenance.

Figure 5.8 shows a Zipkin trace tree for Z1. A human operator can clearly see that the API call to the frontend took 50 seconds, and that the compute RPC and the storage RPC both took almost as long. The latter may seem suspicious, but the trace

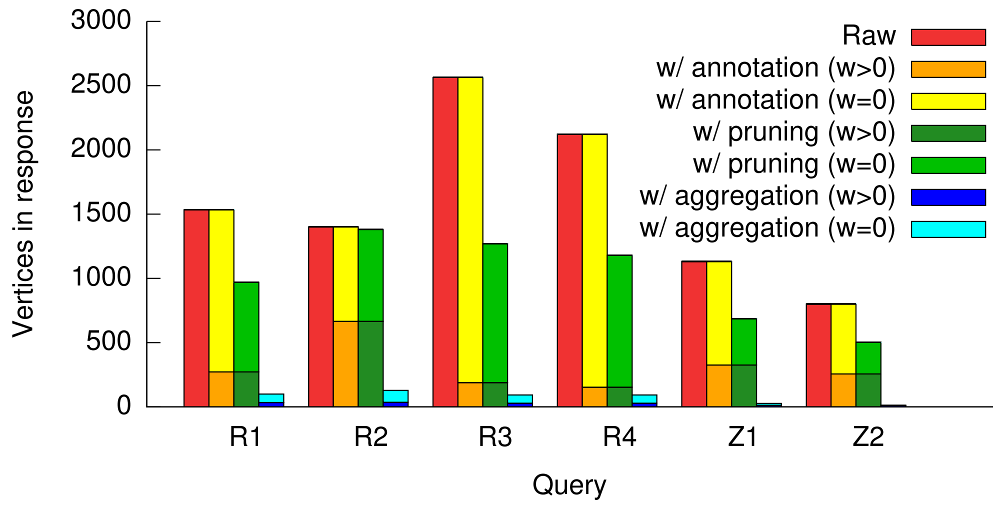


Figure 5.10: Size of the temporal provenance for all scenarios, with different combinations of postprocessing steps.

tree contains no further explanation. Similarly, the classic provenance tree for Z1, which are essentially the yellow vertexes in Figure 5.9, offers a more comprehensive explanation compared to the trace tree; however, it still misses the actual off-path root cause. This problem happened in all scenarios with Zipkin and DTaP.

Figure 5.9 shows the temporal provenance for the Z1. Again, the provenance shows clearly that the API call took 50 seconds (1), and that this was caused by a delayed response from the storage service (2); however, unlike the trace tree or classic provenance, the temporal provenance *also* shows that most of this delay was caused by requests from the maintenance service M (3). This cause can be found simply by starting at the root and following the chain of vertexes that are annotated with the highest delay. This information would make it much easier for the operator to find the root cause.

### 5.7.3 Complexity

Including the actual root cause is not enough for the provenance to be useful; it also has to be simple enough for the operator to make sense of it. Recall from Sec-

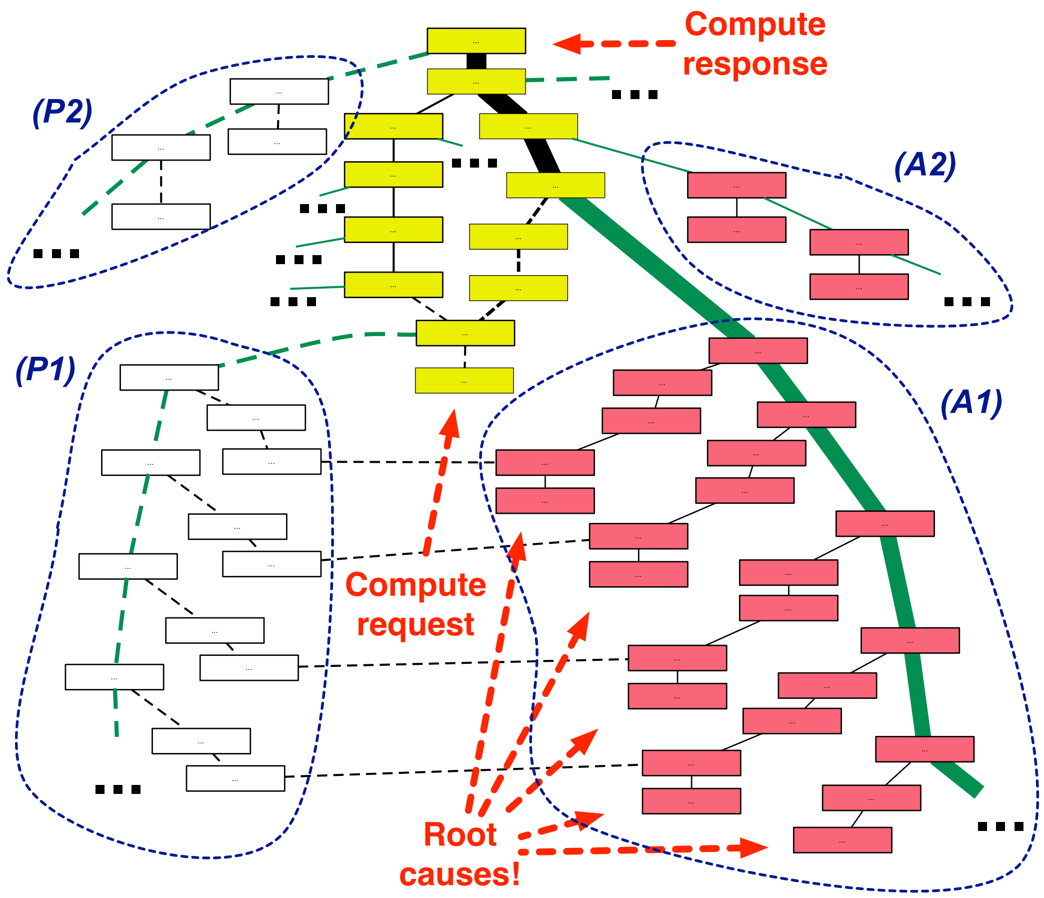


Figure 5.11: Sketch of the *raw* temporal provenance for scenario Z1. The post-processing steps from Section 5.5 would reduce this to the provenance shown in Figure 5.9.

tion 5.5 that, before showing the provenance graph to administrators, our debugger (1) prunes vertexes that do not contribute to the overall delay, and (2) aggregates subgraphs that are structurally similar. To quantify how well these techniques work, and whether they do indeed lead to a readable explanation, we re-ran the diagnostic queries in Section 5.7.1 with different subsets of these steps disabled, and we measured the size of the corresponding provenance graphs.

Figure 5.10 shows our results. The leftmost bars show the size of the raw temporal provenance, which ranged from 800 to 2,564 vertexes. A graph of this size would be far too complex for most operators to interpret. However, not all of these vertexes actually contribute to the overall delay. The second set of bars shows the number of vertexes that the algorithm from Section 5.4.3 would annotate with a nonzero delay ($w > 0$) and a zero delay ($w = 0$), respectively. These results show that only 7.2–47.5% of all vertexes actually contributed any delays. However, the subgraphs with nonzero delays nevertheless remain too large for an operator to read effectively.

Our first postprocessing step (Section 5.5.1) prunes vertexes and subtrees that are annotated with zero delay and also do not make a causal contribution. As the third set of bars shows, this reduces the size of the graph considerably, by more than 30% in all scenarios except R2. In R2, both extended network transmission delays and heavy server loads contribute delays and therefore few events are actually pruned.

The second and final postprocessing step (Section 5.5.2) coalesces structurally similar subtrees and aggregates their delays. As the rightmost set of bars shows, this is extremely effective and shrinks the graph to between 13 and 129 vertexes; the number of vertexes that actually contribute delay is between 11 and 35. (Recall that vertexes with a causal contribution are preserved even if they do not contribute delay.) A provenance graph of this size should be relatively easy to interpret.

To explain where the large reduction comes from, we sketch the raw provenance tree – without postprocessing – for scenario Z1 in Figure 5.11. The structure of this tree is typical for the ones we have generated. First, there is a relatively small "backbone" (shown in yellow) that functionally explains the result and roughly corresponds to classical provenance. Second, there is a large number of small branches (shown in red) along long sequencing chains (shown in green) that describe the sources of any delay; these are collapsed into a much smaller number of branches, or even a single branch. Third, there are further branches (shown in white) that are connected via sequencing edges but do *not* contribute any delay; these are pruned entirely. The last two categories typically contain the vast majority of vertexes, and our postprocessing steps can shrink them very effectively, which in this case yields the much simpler tree from Figure 5.9.

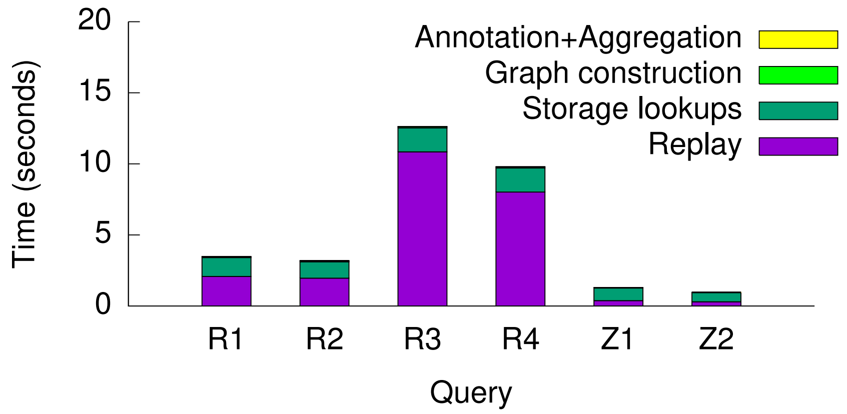


Figure 5.12: Turnaround for all queries in Section 5.7.1.

### 5.7.4 Runtime overhead

Next, we quantified the overhead of collecting the necessary metadata for temporal provenance at runtime. Our experiments focused on RapidNet instead of Zipkin. This is because the latter is closely based on Dapper, which is proven to introduce low overhead in production systems [120]; temporal provenance simply uses the data Zipkin collects, but does not modify its collection system. We ran a fixed workload of 1,000 requests in all scenarios, and we measured the overall latency and the storage needed to maintain provenance. Maintaining classical provenance alone resulted in a latency increase of 0.3–1.2% and a storage overhead of 145–168 bytes per *input event*. Maintaining temporal provenance causes an additional latency increase 0.4–1.5% overall and a storage overhead of 49 bytes per *event*. Notice that, for temporal provenance, it is not enough to merely record input events, since this would not necessarily reproduce the timing of events or their sequence. To limit the additional overhead, Zeno can remember temporal information only for recent executions. This is sufficient as older executions would have not contributed to recent delays (Section 5.4.3).

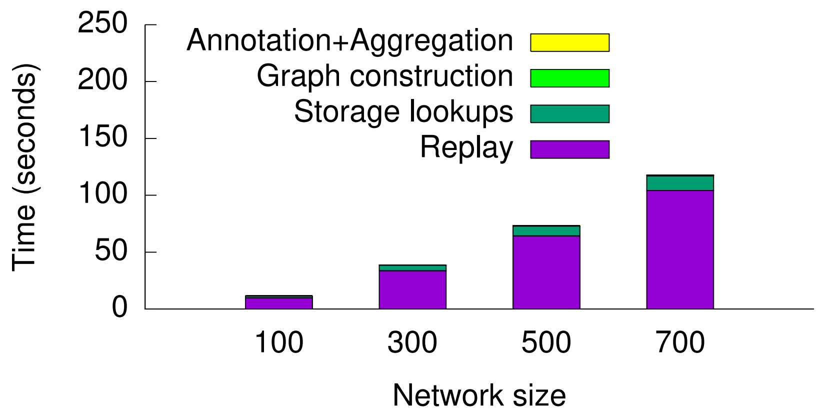


Figure 5.13: Scalability of turnaround time for R3.

### 5.7.5 Query processing speed

When the operator queries the temporal provenance, our debugger must execute the algorithm from Section 5.4 and apply the postprocessing steps from Section 5.5. Since debugging is an interactive process, a quick response is important. To see whether our debugger can meet this requirement, we measured the turnaround time for the queries in Section 5.7.1, as well as the fraction of time consumed by each of the major components.

Figure 5.12 shows our results. We make two high-level observations. First, for scenarios where provenance is captured using deterministic replay (R1–R4), the turnaround time is dominated by the replay and by the storage lookups that would be needed even for classical provenance. This is expected because neither our annotation algorithm nor the postprocessing steps are computationally expensive. Second, although the queries vary in complexity and thus their turnaround times are difficult to compare, we observe that none of them took more than 13 seconds, which should be fast enough for interactive use. This delay is incurred only once per query; the operator can then explore the resulting temporal provenance without further delays.

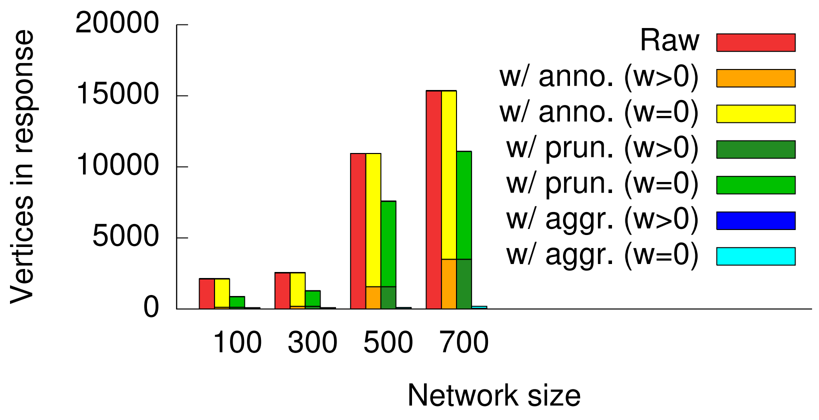


Figure 5.14: Scalability of provenance size for R3.

### 5.7.6 Scalability

To evaluate the scalability of temporal provenance with regard to the network size, we tested the turnaround time and provenance size of query R3 on larger networks which different number of servers (up to 700).

**Turnaround time:** As we can see from the left part of Figure 5.13, the turnaround time increased linearly with the network size, but it was within 120 seconds for all cases. As the breakdown shows, the increase mainly comes from the latency increase of the historical lookups and of the replay. This is because the additional nodes and traffic caused the size of the logs to increase. This in turn resulted in a longer time to replay the logs, and to search through the events.

**Complexity:** The right part of Figure 5.14 shows how the size of the provenance grows with the network size. As expected, the size of the raw provenance grew linearly to the network size – by 7x from 2,123 to 15,348 vertexes – because traffic from additional servers caused additional delays, which required extra vertexes to be represented in the provenance. With the annotation and aggregation heuristics applied, the number of vertexes that actually contribute delay still grew, because more hops were congested due to busier networks. However, the increase – 1.6x, from 28 to 43 – is much less than 7x, which suggests that the heuristic scales well.

## 5.8 Related Work

We have covered the literature on provenance and diagnosis of systems in Chapter 2. As discussed in Section 5.2.2, none of the provenance systems we are aware of reason about temporal causality, which is essential for diagnosing problems such as the one in Figure 5.1. This is true even for DTaP [146] and its predecessor TAP [144], which are "time-aware" only in a limited sense that they remember the provenance of past events. Furthermore, to the best of our knowledge, all prior diagnostic systems that reason about the causes of delays *infer* causality. Our approach is the first to explicitly record temporal *causality* (in the form of sequencing edges) and thus also the first to offer precise reasoning about the causes of timing behavior. Next, we briefly discuss research on temporal behaviors from other communities.

**Timing faults:** Our approach is potentially useful for diagnosing timing faults in real-time systems, where tasks have "hard" or "soft" deadlines [25]. Researchers have proposed solutions to control program timing, but they either require specialized hardware [34] or incur significant runtime overhead [23]. Worst-case execution time (WCET) analysis [132] can predict program timing, but it only gives an upper bound and does not reason about the causes of any delays.

**Queuing theory:** Queuing theory [73, 74, 16] has been used to model, analyze, and optimize the worst-case or average timing performance of distributed systems during the design process. This approach, however, relies on a certain distribution or arrival patterns of the input workloads, which does not always hold in practice, and it does not automatically identify the causes of a performance violation. In contrast, temporal provenance can help diagnosing problems that arise at runtime, without any assumption on the input model.

## 5.9 Conclusion

In this chapter, we have proposed temporal provenance, an approach to performance diagnosis. Temporal provenance produces a comprehensive explanation by tracking *temporal causality*: causes that affect not the result of a computation but its timing. This makes it possible to find the root causes of performance problems, even when they do not appear on the path that a delayed request has taken. We have presented a concrete algorithm for tracking temporal provenance, as well as a prototype debugger Zeno that implements this algorithm. Our experimental evaluation with Zeno shows that temporal provenance can provide helpful explanations for temporal behavior, and that it can be generated at a reasonable cost.

# 6
# Conclusion

In this dissertation, the goal is to extend provenance to handle diagnostics problems that require deeper investigations: explaining the absence of events, repairing controller programs, and handling timing-related faults. Overall, the above three chapters have shown extensions to network provenance address the proposed problems. In retrospect, three steps are proven to be crucial in extending provenance to handle diagnostics problems that require deeper investigations.

First, deeper investigations require additional collections of runtime information. This is not unexpected as a provenance-based debugger can only be as good as the causality that it collects. Sequencing edges enable temporal provenance to capture temporal causality. Meta models expose to meta provenance the causality between inputs/outputs and syntactic elements of programs. Tracking such additional information incurs runtime costs. Fortunately, the results in the above chapters suggest that the additional runtime overhead is minimal and sometimes can be eliminated when the information can be reconstructed via deterministic replay.

Second, deeper investigations often require more sophisticated reasoning. This can happen due to updates to the provenance model. For instance, negative provenance introduces negative vertexes. To construct such vertexes, it relies on counterfactual reasoning, which is substantially different from the procedures that build positive vertexes. Furthermore, materializing the provenance graph is often merely the first step in the diagnosis pipeline. Customized postprocessing is necessary to diagnose the problem. For example, in order to determine how much delay each vertex contributes, temporal provenance relies on delay annotations.

Last but not least, improving the readability of provenance is imperative to enabling deeper investigations. Provenance captures causality and thus can weed out irrelevant factors in diagnosis. This is already sufficient for some diagnostic tasks. However, as discussed above, deeper investigations often require analysis of information from multiple dimensions and reasoning of greater sophistication. The provenance graphs are inevitably larger. For example, the temporal provenance graph of a single RPC can involve vertexes that represent hundreds, if not thousands, of other RPCs that contributed delays. Such "raw" provenance graphs, unless careful pruned or summarized, would not be useful for human investigators.

# 7
# Future work

Next, we discuss the limitations of the three extensions and possible future work.

First, capturing fine-grain causalities of all events in the system can be expensive. In practice, techniques such as gradually expiring old provenance or reconstructing provenance via replay have addressed this limitation to some extent. However, these techniques are not always applicable. For instance, in order to construct temporal provenance via replay for general distributed systems, the replay engine must reproduce the temporal characteristics of the original execution precisely. A potential future direction is to leverage techniques such as timing-deterministic replay [23, 34] to achieve this. Another interesting approach, which is commonly used by distributed tracing systems to reduce runtime overhead [120, 29], is sampling: enabling diagnosis for only a fraction of all requests. A possible direction forward is to adapt sampling techniques to build light-weight provenance tracking. For example, one can selectively track provenance for a set of "targeted" requests such that only these requests themselves and other causally related requests (as well as configurations) are recorded. This will reduce the runtime overhead by tracking less requests and meanwhile preserve the ability to diagnose “targeted” requests.

Second, most provenance-based debuggers, including those proposed in this dissertation, can only explain the faulty behavior of a single tuple (i.e., one misrouted packet or one corrupted configuration entry). This is sufficient in many practical cases, as we have shown in the above chapters. Nonetheless, it maybe preferable for future debuggers to have the ability to explain a collection of faulty tuples, for two reasons: (a) Operators sometimes observe multiple faulty tuples in an outage (e.g., all RPCs between 4pm and 5pm are slow). It is infeasible for the operator to diagnose all faulty tuples one by one, and it is not always clear how the operator can choose which specific tuple to diagnose first. (b) By correlating the provenance of multiple faulty tuples in the same incident, the debugger maybe able to pinpoint the root cause more precisely. For example, there maybe thousands of slow RPCs, the explanation for each of which seems random. However, the debugger may find a common culprit which contributed significant delays in almost all instances.

Third, existing provenance systems assume that the needed provenance data is always complete. This is reasonable for many scenarios, such as data center networks. However, there are cases where the provenance data maybe incomplete: (a) blackbox components cannot report provenance due to privacy restrictions or legacy software; (b) the provenance tracking system itself maybe unreliable (e.g., when a network accident partitions the tracking component and the storage component). Some efforts have addressed this limitation to some extent. For example, SNP uses external annotations to extract provenance from Quagga [145], but this does not address all cases. One future direction maybe developing debuggers that can tolerate incomplete provenance data. Probabilistic databases can answer queries even when parts of the database are unknown [30]. Perhaps a future provenance-based debugger can generate a (small) number of possible explanations such that each is consistent with the incomplete input data. This may still be helpful as it reduces the number of explanations (and the number of root causes) that the operator has to consider.

# A
# Negative Provenance

We have sketched the properties of negative provenance – soundness, completeness, and minimality – in Section 3.3.7. The full formal model, including the definitions and proofs of the three properties, is in Appendix A.1. In Section 3.7.2 and Section 3.7.3, we only provided the responses of Y! for scenarios Q1 and Q6. The full responses for all remaining scenarios are in Appendix A.2.

## A.1 Formal Model

### A.1.1 Background: Execution Traces

To set stage for the discussion, we introduce some necessary definitions of system execution. The execution of an NDlog program can be characterized by the sequence of events that take place; we refer to this sequence as an *execution trace*. More formally, an execution trace is defined as follows.

**Definition 1. (Event)**: *An event $d@n = (m, r, t, c, m')$ represents that rule $r$ was triggered by message $m$ and generated a set of (local or remote) messages $m'$ at time $t$, given the precondition $c$. The precondition $c$ is a set of tuples that existed on $n$ at time $t$.*

**Definition 2. (Trace)**: *A trace $\mathscr{E}$ of a system execution is an ordered sequence of events from an initial state $\mathscr{S}_0$, $\mathscr{S}_0 \xrightarrow{d_1@n_1} \mathscr{S}_1 \xrightarrow{d_2@n_2} \dots \xrightarrow{d_k@n_k} \mathscr{S}_k$.*

**Definition 3. (Subtrace):** *We say that $\mathscr{E}'$ is a subtrace of $\mathscr{E}$ (written as $\mathscr{E}' \subseteq \mathscr{E}$) if $\mathscr{E}'$ consists of a subset of the events in $\mathscr{E}$ in the same order. In particular, we write $\mathscr{E}|n$ to denote the subtrace that consists of all the events on node $n$ in $\mathscr{E}$.*

**Definition 4. (Equivalence):** *We say that two traces $\mathscr{E}'$ and $\mathscr{E}$ are equivalent (written as $\mathscr{E}' \sim \mathscr{E}$), if, for all $n$, $\mathscr{E}'|n = \mathscr{E}|n$.*

### A.1.2 Properties

Given the negative provenance of $G(e, \mathscr{E})$ of the absence of an event $e$ in execution trace $\mathscr{E}$, we formally define the following properties.

**Definition 5. (Validity)**: *We say a trace $\mathscr{E}$ is valid, if (a) for all $\tau_k \in c_i$, $\tau_k \in S_{i-1}$, and (b) for all $d_i@n_i = (m_i, r_i, t_i, c_i, m'_i)$, either $m_i$ is a base message from an external source, or there exists $d_j@n_j = (m_j, r_j, t_j, c_j, m'_j)$ that proceeds $d_i@n_i$ and $m_i \in m'_j$.*

**Property (Soundness):** *Negative provenance $G(e, \mathscr{E})$ is sound iff (a) it is possible to extract a valid subtrace $\mathscr{E}_{sub} \subseteq \mathscr{E}'$, such that $\mathscr{E}' \sim \mathscr{E}$, and (b) for all vertices in $G(e, \mathscr{E})$, their corresponding predicates hold in $\mathscr{E}$.*

**Property (Completeness):** *Negative provenance $G(e, \mathscr{E})$ is complete iff there exists no trace $\mathscr{E}'$ such that a) $\mathscr{E}'$ assumes same external inputs as $G(e, \mathscr{E})$, that is, $(\Delta\tau@n, -, t, -, -) \in \mathscr{E}$ iff $\textsc{insert}/\textsc{delete}(\tau, n, t) \in V(G)$, and b) $e$ exists in the $\mathscr{E}'$.*

**Definition 6. (Reduction):** *Given negative provenance $G(e, \mathscr{E})$, if there exists $v_1, v_2 \in V(G)$, where the time interval of $v_1$ and $v_2$ ($t(v_1)$ and $t(v_2)$ respectively) are adjacent, and $v_1$ and $v_2$ have the same dependencies, then $G$ can be reduced to $G'$ by combining $v_1$ and $v_2$ into $v \in V(G')$, where $t(v) = t(v_1) \cup t(v_2)$.*

**Algorithm 1** Extracting traces from provenance

```
1: proc $ExtractTrace(G = (V,E))$
2: // calculate the out-degree of all vertices in $G$
3: for $\forall\ e = (v,v') \in E$ do $degree(v)$++
4: // generate the subtrace based on topological sort
5: $NodeToProcess = V$
6: while $NodeToProcess \neq \phi$ do
7:     select $v \in NodeToProcess : degree(v) = 0$, and $\nexists v'$ that is located on the same node and has a larger timestamp
8:     $NodeToProcess.remove(v)$
9:     if $typeof(v) =$ DERIVE or UNDERIVE then
10:        $find\ e = (v,v') \in E,\ output \leftarrow v'$
11:        for $\forall\ e = (v',v) \in E$ do
12:            if $typeof(v') =$ INSERT or DELETE then
13:                $trigger = v'$ // $v'$ is the triggering event
14:            else
15:                $condition.add(v')$ // $v'$ is a condition
16:        $ruleName \leftarrow v.ruleName,\ time \leftarrow v.time$
17:        $event \leftarrow (trigger, ruleName, time, condition, output)$
18:        $trace.push\_front(event)$
19:    else if $typeof(v) =$ RECEIVE then
20:        $find\ e = (v',v) \in E$ // $v'$ must be a SEND vertex
21:        $event \leftarrow (v', -, v.time, 1, v)$
22:        $trace.push\_font(event)$
23:    $find\ e = (v',v) \in E,\ degree(v') \leftarrow degree(v') - 1$
24: return $trace$
```

**Definition 7. (Simplicity):** *Given two negative provenance $G(e,\mathscr{E})$ and $G'(e,\mathscr{E})$, we say $G'$ is simpler than $G$ (written as $G' < G$), if any of the following three hold:*

*(1) $G'$ is a subgraph of $G$;*

*(2) $G'$ is reduced from $G$ (by combining $v_1$ and $v_2$);*

*(3) There exists $G''$, such that $G' < G''$ and $G'' < G$.*

**Property (Minimality):** *Negative provenance $G(e,\mathscr{E})$ is minimal, if no $G' < G$ is sound and complete.*

## A.1.3 Proofs

**Definition 8. (Trace Extraction):** *Given a negative provenance $G(e,\mathscr{E})$, the trace is extracted by running Algorithm 1 based on topological sort.*

Algorithm 1 converts *positive* vertices in the provenance graph to events and thus uses a topological ordering to assemble the events into a trace. In particular, Line 9-19 implements the construction of one individual event, where the information of a rule evaluation (such as triggering event, conditions, and action) is extracted from the corresponding vertices in $G(e, \mathscr{E})$

**Lemma 1.** *For any vertex $v \in V(G)$, v's corresponding predicate $p_v$ holds in $\mathscr{E}$.*

**Proof.** By construction (see Figure 3.2), any positive vertex $v \in V(G)$ is generated only if the corresponding event entry exists in the log (which is deterministically determined by $\mathscr{E}$. Therefore, predicate $p_v$ naturally holds in $\mathscr{E}$. Similarly, the predicate $p_w$ (for negative vertex $w$) also hold in $\mathscr{E}$. □

**Lemma 2.** *The extracted trace $\mathscr{E}_{sub}$ is a valid trace.*

**Proof.** We need to show that in the generated trace $\mathscr{S}_0 \xrightarrow{d_1@n_1} \mathscr{S}_1 \xrightarrow{d_2@n_2} \dots \xrightarrow{d_k@n_k} \mathscr{S}_k$, (a) for all $\tau \in c_i$, $\tau \in S_{i-1}$, and (b) for all $d_i@n_i = (m_i, r_i, t_i, c_i, m'_i)$, either $m_i$ is a base message from an external source, or there exists $d_j@n_j = (m_j, r_j, t_j, c_j, m'_j)$ that precedes $d_i@n_i$ and $m_i \in m'_j$.

It has been shown in [146] that, for a positive event $e'$ in $\mathscr{E}$, the extraction algorithm yields a valid trace for the provenance of $e'$ (which is the subgraph $G'$ of $G$ that is rooted by $e'$). Note that executing Algorithm 1 on $G$ can be considered as combining traces generated from multiple such $G'$s, $\{d_{i,1}@n_{i,1}\}, \{d_{i,2}@n_{i,2}\}, \dots, \{d_{i,k}@n_{i,k}\}$, where each trace $\{d_{i,p}@n_{i,p}\}$ is valid.

**Condition a.** We know that, for any event $d_{i,p}@n_{i,p}$, the conditions for the event $c_{i,p}$ hold (i.e., the corresponding tuples exist when the rule is triggered). It can be shown that, in the combined trace, any event $d@n$ that contributes to $c_{i,p}$ precedes $d_{i,p}@n_{i,p}$; this is because $d@n$ has to be a (transitive) children of $d_{i,p}@n_{i,p}$ in $G$, and, therefore, should also be in trace $p$. We can then conclude that, in the combined trace $d_i@n_i$, the conditions for the event $c_i$ hold when the event is triggered (i.e., condition b holds).

**Condition b.** Because any trace $\{d_{i,p}@n_{i,p}\}$ is valid, we know that, for any event $d_{i,p}@n_{i,p} = (m_{i,p}, r_{i,p}, t_{i,p}, c_{i,p}, m'_{i,p})$, the trigger event $m_{i,p}$ was indeed generated by a preceding event $d_{j,p}@n_{j,p}$. This also holds in the combined trace (i.e., condition a holds), as the topological relationship between $d_{i,p}@n_{i,p}$ and $d_{j,p}@n_{j,p}$ remains. □

**Lemma 3.** *The extracted trace $\mathscr{E}_{sub} \subseteq \mathscr{E}'$, where $\mathscr{E}' \sim \mathscr{E}$.*

**Proof.** We need to show that a) all the events in $\mathscr{E}_{sub}$ also appear in $\mathscr{E}$ (and thus in any $\mathscr{E}' \sim \mathscr{E}$), and b) the local event ordering pertains on each node.

**Condition a.** An event $d_i@n_i$ is generated and included in $\mathscr{E}_{sub}$ for each DERIVE (or UNDERIVE) vertex (and its direct parent and children) in $G(\Delta\tau, \mathscr{E})$. On the other hand, by construction (Figure 3.2), each DERIVE (or UNDERIVE) vertex $v$ corresponds to an event $d_j@n_j$, and it is not difficult to see that $d_i@n_i$ is identical to $d_j@n_j$. Therefore, all the events in $\mathscr{E}_{sub}$ do appear in $\mathscr{E}$ as well.

**Condition b.** According to Algorithm 1 (specifically, Line 7), $d_1@n$ precedes $d_2@n$, iff $d_2@n$ has a larger timestamp than $d_1@n$ (that is, $d_1@n$ precedes $d_2@n$ in the actually execution $\mathscr{E}$. Note that events on different nodes may be reordered, but this is captured by the equivalence ($\sim$) relation. □

**Theorem 4.** *Negative provenance $G(\Delta\tau, \mathscr{E})$ is sound.*

**Proof.** Directly from Lemma 1, 2, 3. □

**Theorem 5.** *Negative provenance $G(\Delta\tau, \mathscr{E})$ is complete.*

**Proof.** We prove the completeness property by induction on the depth of the provenance graph.

**Base case.** In the base case, the negative provenance $G(e, \mathscr{E})$ is a single-vertex graph (i.e., NINSERT($[t1, t2], N, \tau$) or NDELETE($[t1, t2], N, \tau$)). In this case, $e$ is an update of a base tuple $\tau$, and it is obvious that there exists no trace $\mathscr{E}'$ that has the same external input and contains $\Delta\tau$.

**Induction case.** Suppose there exists $\mathscr{E}'$, such that $\mathscr{E}'$ assumes the same external input as $G(e,\mathscr{E})$, and $e$ exists in $\mathscr{E}'$. We perform a case analysis by considering the type of the root vertex of $G(e,\mathscr{E})$.

- NEXIST($[t1,t2],N,\tau$). Suppose $\tau$ existed (partly) between $[t1',t2']$ in $\mathscr{E}'$, where $[t1,t2]\cap[t1',t2'] = [t3,t4] \neq \phi$. By construction (in Figure 3.2), NAPPEAR($[t0,t2],N,\tau$) is in $G$, where $t0 \leq t1$. We can conclude, based on induction hypothesis, that there is no execution that assumes same external input as $G$ but has $\tau$ appeared between $[t0,t2]$. This contradicts the existence of $\tau$ in $\mathscr{E}'$.

- NAPPEAR($[t1,t2],N,\tau$). Suppose $\tau$ appeared at $t \in [t1,t2]$ in $\mathscr{E}'$. $\tau$ cannot be a base tuple, otherwise, $G$ and $\mathscr{E}'$ would have conflicting external inputs. If $\tau$ is a locally derived tuple, $\tau$ was derived by some rule execution in $\mathscr{E}'$, however, for any rule $r$ that might derive $\tau$, NDERIVE($[t1,t2],N,\tau,r$) is in $G$; a contradiction is reached by applying the induction hypothesis. If $\tau$ is received from communication, message $+\tau$ was received at time $t$ in $\mathscr{E}'$, whereas NRECEIVE($[t1,t2],N,+\tau$) is in $G$; a contradiction is reached by applying the induction hypothesis.

- NDERIVE($[t1,t2],N,\tau,r$). Suppose $\tau$ was derived at $t \in [t1,t2]$ by rule $r$ in $\mathscr{E}'$. It must be the case that all the condition held at time $t$ (that is, their corresponding tuples $c_1 ... c_k$ existed at $t$). On the other hand, for each rule (including rule $r$) that might derive $\tau$, the graph construction algorithm considers the parameter space in which $\tau$ would not be derived, and includes a set of negative events that cover the whole parameter space.

  Since $G$ contains the cover set, there exists $c_i$ such that NEXIST($[t1',t2'],N,c_i$) exists in $G$, where $t \in [t1',t2']$. Based on induction hypothesis, there is no execution that assumes same external input as $G$ but has $c_i$ existed time $t$. This contradicts the rule evaluation that derived $\tau$ at time $t$ in $\mathscr{E}'$.

- NSEND($[t1,t2],N,\Delta\tau$). Suppose $\Delta\tau$ was sent at $t \in [t1,t2]$ in $\mathscr{E}'$. It must be the case that $\tau$ appeared (or disappeared) at time $t$ in $\mathscr{E}'$. However, by construction, NAPPEAR($[t1,t2],N,\tau$) (or NDISAPPEAR($[t1,t2],N,\tau$)) is in $G$, which, based on induction hypothesis, indicates that there is no execution that assumes same external input as $G$ but has $\tau$ appeared (or disappeared) between $[t1,t2]$. This contradicts the appearance (or disappearance) of $\tau$ in $\mathscr{E}'$.

- NRECEIVE($[t1,t2],N,\Delta\tau$). Suppose message $\Delta\tau$ was received at time $t \in [t1,t2]$ in $\mathscr{E}'$. It must be the case that $\Delta\tau$ was sent by some node $M$ at time $t_{send}$, and arrived $N$ at time $t_{recv} \in [t1,t2]$. On the other hand, the graph construction algorithm considers all nodes (including $M$) that could have sent $\Delta\tau$ to $N$. Specifically, for node $M$, NSEND($[s,t],M,\Delta\tau$) is constructed for any time interval $[s,t]$ in which $M$ did not send $\Delta\tau$ in $E$.

  If $t_{send} \in [s,t]$, a contradiction is reached by applying the induction hypothesis. Otherwise ($M$ did send $\Delta\tau$ at $t_{send}$ in $E$), NARRIVE($[t1,t2],M \to N,t_{send},\Delta\tau$) is in $G$. Based on the induction hypothesis, there exists no execution that assumes same external input as $G$ but has $\Delta\tau$ received at $N$ between $[t1,t2]$.

- NARRIVE($[t1,t2],N_1 \to N_2,t0,\Delta\tau$). Since execution trace $\mathscr{E}'$ agrees with $G$ on all external inputs (including how the network affects the message delivery), they must agree on the delivery time $t$ of message event $(\Delta\tau@N_1,-,t,1,\Delta\tau@N_2)$. Therefore, message $\Delta\tau$ could not arrive within $[t1,t2]$ in $\mathscr{E}'$ but not in $\mathscr{E}$.

The case analysis of vertices NDISAPPEAR and NUNDERIVE is analogous to the those of NAPPEAR and NDERIVE respectively; the case analysis for positive vertices have been shown in [146]. □

**Theorem 6.** *Negative provenance $G(e,\mathscr{E})$ is minimal.*

**Proof.** By construction, $G(e,\mathscr{E})$ should not be further reducible; this is because the graph construction algorithm (described in Section 3.3.4) considers semantically identical vertices with adjacent or overlapping interval, and coalesces these vertices (i.e., apply reduction on $G$). We next prove, by induction on the depth of $G(e,\mathscr{E})$, that there is no strict subgraph of $G(e,\mathscr{E})$ that is sound and complete.

**Base case.** In the base case, the negative provenance $G(e,\mathscr{E})$ is a single-vertex graph (i.e., NINSERT($[t1,t2],N,\tau$) or NDELETE($[t1,t2],N,\tau$)). In this case, a strict subgraph of $G$ (an empty graph) is not complete.

**Induction case.** Assume the root vertex $v$ of $G(e,\mathscr{E})$ has children vertices $v_1,v_2,...,v_k$. We write $G_{v_i}$ to denote the subgraphs rooted by children vertex $v_i$. Based on the induction hypothesis, any strict subgraph of $G_{v_i}$ is either not sound or not complete. Therefore, any sound and complete provenance that contains $v_i$ must contain the complete $G_{v_i}$. We perform a case analysis by considering the type of $v$.

- NEXIST($[t1,t2],N,\tau$). NEXIST has children vertices NAPPEAR and (potentially) DISAPPEAR (if $\tau$ existed previously and disappeared at time $t < t1$). NAPPEAR($[t,t2],N,\tau$) is essential for the completeness of $G$; otherwise, an execution $\mathscr{E}'$ in which $\tau$ appeared in $[t,t2]$ would have $\tau$ existed during this time interval (that is, the subgraph without NAPPEAR is not complete). The DISAPPEAR vertex is also essential; this is because $\tau$ could have already existed before $t1$, in which case, $\tau$ would still exist in time interval $[t1,t2]$ though it did not appear again.

- NAPPEAR($[t1,t2],N,\tau$). If $\tau$ is a base tuple, NAPPEAR has child vertex NINSERT, which is essential for the completeness of $G$; otherwise, execution $\mathscr{E}'$ can have $\tau$ inserted, and therefore appear, at $t \in [t1,t2]$. Similarly, tuples that are locally derived or received from network, we can also show NAPPEAR's child vertex NDERIVE (or NRECEIVE) are essential for the completeness.

- NDERIVE($[t1,t2],N,\tau$). NDERIVE has a set of NEXIST as its children vertices. These NEXIST vertices consist of a cover set $S$ for the parameter space in which

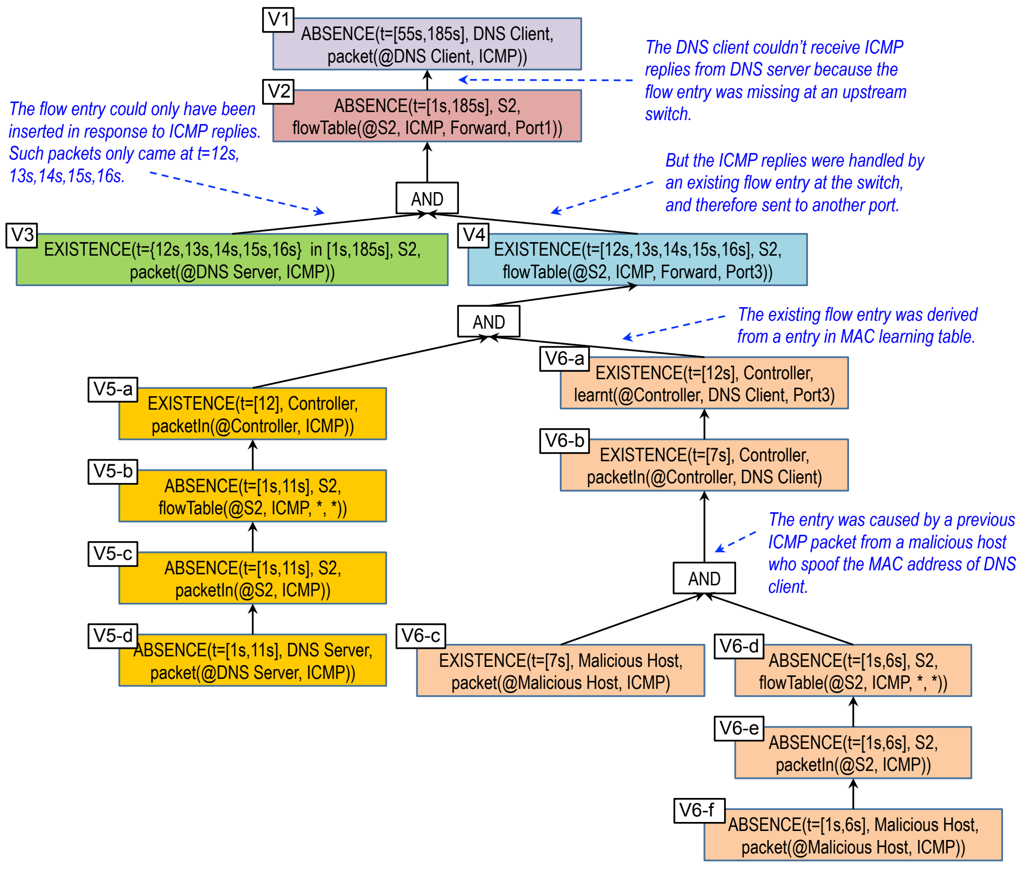

V1
ABSENCE(t=[55s,185s], DNS Client, packet(@DNS Client, ICMP))
The DNS client couldn't receive ICMP replies from DNS server because the flow entry was missing at an upstream switch.
V2
ABSENCE(t=[1s,185s], S2, flowTable(@S2, ICMP, Forward, Port1))
The flow entry could only have been inserted in response to ICMP replies. Such packets only came at t=12s, 13s,14s,15s,16s.
AND
But the ICMP replies were handled by an existing flow entry at the switch, and therefore sent to another port.
V3
EXISTENCE(t={12s,13s,14s,15s,16s} in [1s,185s], S2, packet(@DNS Server, ICMP))
V4
EXISTENCE(t=[12s,13s,14s,15s,16s], S2, flowTable(@S2, ICMP, Forward, Port3))
AND
The existing flow entry was derived from a entry in MAC learning table.
V6-a
EXISTENCE(t=[12s], Controller, learnt(@Controller, DNS Client, Port3)
V5-a
EXISTENCE(t=[12], Controller, packetIn(@Controller, ICMP))
V6-b
EXISTENCE(t=[7s], Controller, packetIn(@Controller, DNS Client)
V5-b
ABSENCE(t=[1s,11s], S2, flowTable(@S2, ICMP, *, *))
The entry was caused by a previous ICMP packet from a malicious host who spoof the MAC address of DNS client.
V5-c
ABSENCE(t=[1s,11s], S2, packetIn(@S2, ICMP))
AND
V5-d
ABSENCE(t=[1s,11s], DNS Server, packet(@DNS Server, ICMP))
V6-c
EXISTENCE(t=[7s], Malicious Host, packet(@Malicious Host, ICMP)
V6-d
ABSENCE(t=[1s,6s], S2, flowTable(@S2, ICMP, *, *))
V6-e
ABSENCE(t=[1s,6s], S2, packetIn(@S2, ICMP))
V6-f
ABSENCE(t=[1s,6s], Malicious Host, packet(@Malicious Host, ICMP))

Figure A.1: Answer to Q2, as returned by Y!

$\tau$ would not be derived. Recall from Section 3.3.5 that the PARTITION algorithm that generates $S$ ensures the minimality of $S$, that is, any strict subset of $S$ is not a cover set. Therefore, each of the NEXIST vertices is essential for completeness.

- NSEND$([t1,t2],N,+\tau)$. NSEND has children vertices NAPPEAR and (potentially) EXIST (if $\tau$ existed partly in $[t1,t2]$ and disappeared at time $t \in [t1,t2]$). NAPPEAR$([t,t2],N,\tau)$ is essential for the completeness of $G$; otherwise, an execution $\mathscr{E}'$ in which $\tau$ appeared in $[t,t2]$ would have sent $+\tau$ during this time interval. The EXIST$([t1,t],N,\tau)$ vertex is also essential; this is because there exists execution $E'$ in which $+\tau$ was derived in $[t1,t]$, which could have resulted in a state change, and, therefore, a message sent from $N$ (it didn't, in $\mathscr{E}$, only because $\tau$ was already existed).

- NRECEIVE($[t1,t2],N,\Delta\tau$). NRECEIVE has children vertices NSEND and NARRIVE, both of which are essential for completeness; this is because there exists execution $E'$ in which some node $N'$ has sent $\Delta\tau$ that arrived at node $N$ within $[t1,t2]$.

- NARRIVE($[t1,t2],N_1 \rightarrow N_2,t0,\Delta\tau$). NARRIVE has children vertices SEND and DELAY, both of which are necessary for the completeness of $G$; otherwise, there exists an execution $\mathscr{E}'$ in which $\Delta\tau$ was never sent by $N_1$ or $\Delta\tau$ has been delivered to $N_2$ between $[t1,t2]$.

The case analysis of vertices NDISAPPEAR and NUNDERIVE is analogous to the those of NAPPEAR and NDERIVE respectively. $\square$

## A.2 Responses

**Q2.** Q2 from scenario SDN2 asks why a host cannot receive ICMP relies from the DNS server. Figure A.1 shows the provenance generated by Y!. The explanation is: ICMP replies did not return to the DNS client (V1) because a suitable flow entry was missing at an upstream switch (V2). The flow entry could only have been triggered by a ICMP reply packet, which did arrive several times (V3). But these replies were handled by an existing flow entry which forwarded the replies to another port (V4). This flow entry was derived from the MAC learning table (V6a) when packetIn arrived (V5a-d). The false MAC learning table was there because a malicious host with spoofed MAC address sent an ICMP packet earlier (V6c).

**Q3.** Q3 from scenario SDN3 asks why an internal user cannot access the database. As shown in Figure A.2, Y! explains the problem (V1) by showing that an existing flow entry (V3a-e) had been dropping internal database requests (V2).

**Q4.** Q4 from scenario SDN2 focuses on the situation when a false MAC learning table entry caused a host to receive ICMP relies without issuing ICMP requests.

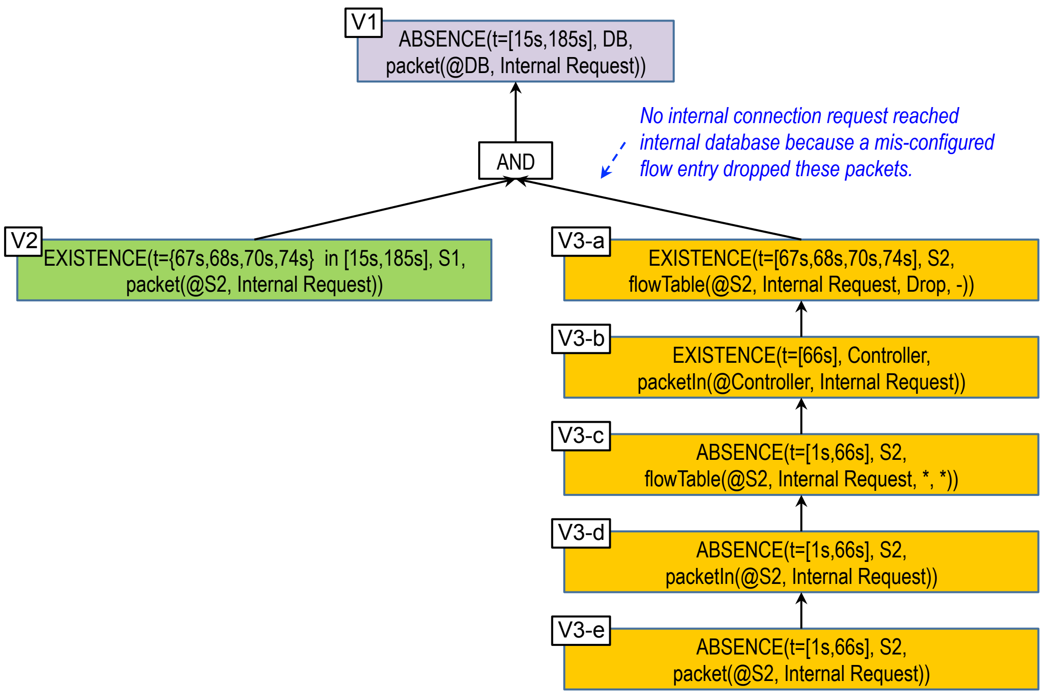

V1
ABSENCE(t=[15s,185s], DB, packet(@DB, Internal Request))
AND
No internal connection request reached internal database because a mis-configured flow entry dropped these packets.
V2
EXISTENCE(t={67s,68s,70s,74s} in [15s,185s], S1, packet(@S2, Internal Request))
V3-a
EXISTENCE(t=[67s,68s,70s,74s], S2, flowTable(@S2, Internal Request, Drop, -))
V3-b
EXISTENCE(t=[66s], Controller, packetIn(@Controller, Internal Request))
V3-c
ABSENCE(t=[1s,66s], S2, flowTable(@S2, Internal Request, *, *))
V3-d
ABSENCE(t=[1s,66s], S2, packetIn(@S2, Internal Request))
V3-e
ABSENCE(t=[1s,66s], S2, packet(@S2, Internal Request))

Figure A.2: Answer to Q3, as returned by Y!.

Figure A.3 depicts how this had happened: The false MAC-port binding (V1) was learned from a previous packetIn (V2), which in turn was triggered because the host send a packet with spoofed MAC address (V3), and there had been no matching flow entry for this packet (V4a-d).

**Q5.** Q5 from scenario SND3 asks why Internet requests made it to a internal database. The problem is explained in Figure A.4: An Internet request were seen at the internal database (V1) because such a request did arrive (V2), and there was a misconfigured flow entry (V3a-e) that allowed this packet to pass.

**Q7.** Q7 from scenario BGP2 asks why a host cannot reach the black-holed host. The problem is explained in Figure A.5: When the packet was sent (V2), it was handled by an existing route at AS 1 (V3). That route was advertised by AS 4 (V4), who had no connectivity to the destination of the packet.

**Q8.** Q8 from scenario BGP3 asks why why the ISP cannot reach a certain AS. As shown in Figure A.6, the route timeout at AS 4 (V1-b) and since then the provider

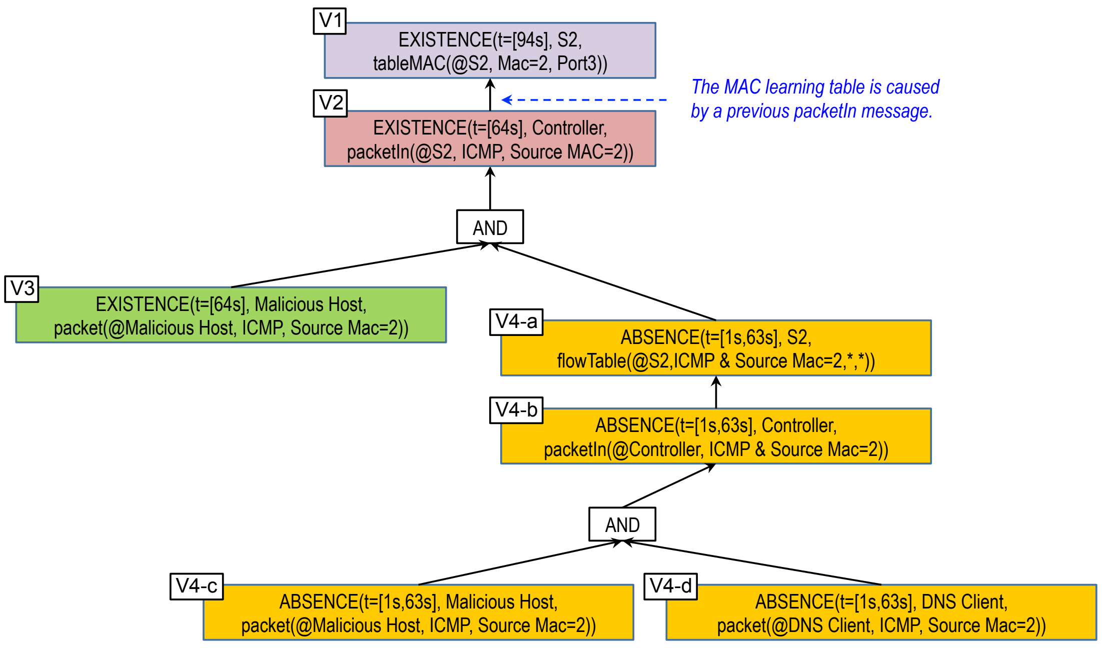

V1
EXISTENCE(t=[94s], S2, tableMAC(@S2, Mac=2, Port3))
The MAC learning table is caused by a previous packetIn message.
V2
EXISTENCE(t=[64s], Controller, packetIn(@S2, ICMP, Source MAC=2))
AND
V3
EXISTENCE(t=[64s], Malicious Host, packet(@Malicious Host, ICMP, Source Mac=2))
V4-a
ABSENCE(t=[1s,63s], S2, flowTable(@S2,ICMP & Source Mac=2,*,*))
V4-b
ABSENCE(t=[1s,63s], Controller, packetIn(@Controller, ICMP & Source Mac=2))
AND
V4-c
ABSENCE(t=[1s,63s], Malicious Host, packet(@Malicious Host, ICMP, Source Mac=2))
V4-d
ABSENCE(t=[1s,63s], DNS Client, packet(@DNS Client, ICMP, Source Mac=2))

Figure A.3: Answer to Q4, as returned by Y!.

AS 1 never sent any advertisements (V1-c). AS 1 did not sent advertisements because it lost its own route (V2-b) and its peer AS 2 stopped advertising routes (V2-c). AS 2 stopped advertising because AS 2 itself lost its route (V3-b). And the route was never recovered because its customer AS 5 never sent the route again (V3-c). AS 5 never sent advertisements again because the route at AS 5 timeout (V4-b) and AS 5 never receive any advertisements since then (V4-c). AS 6 could have sent advertisements to AS 5. It did not because its link to AS 7, where the route could have come from, was down since t=38s (V7).

**Q9.** Q9 from scenario BGP4 asks why the network cannot connect to a particular site. Figure A.7 shows the provance for Q9. The explanation goes as follows: the route timeout at AS 4 (V1-b) and since then the provider AS 1 never sent any advertisements (V1-c). AS 1 did not sent advertisements because it lost its own route (V2-b) and its peer AS 2 stopped advertising routes (V2-c). AS 2 stopped advertising because AS 2 itself lost its route (V3-b). And the route was never recovered because its customer AS 5 never sent the route again (V3-c). AS 5 never sent advertisements again because the route at AS 5 timeout (V4-b) and AS 5 never receive any advertisements since then (V4-c). AS 6 could have sent advertisements to AS 5, but it did not. Because although it kept receiving advertisements from AS 7 (V6-c), there was an updated entry in the BOGON list which caused AS 6 to ignore the advertisements (V6-d).

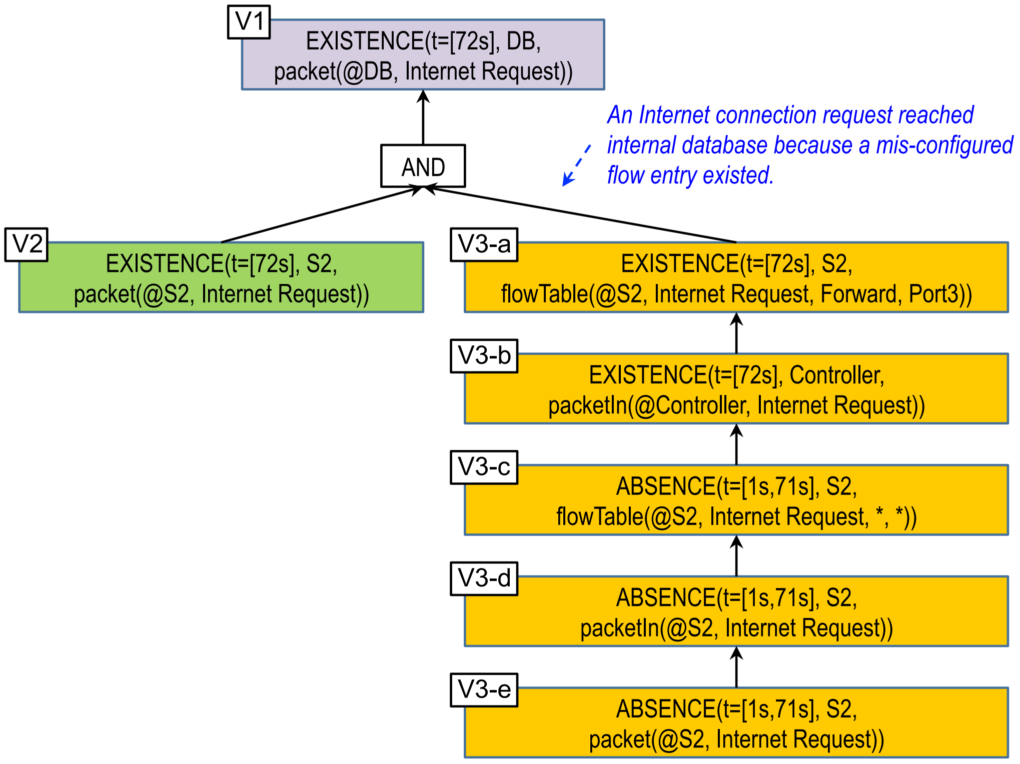

V1
EXISTENCE(t=[72s], DB, packet(@DB, Internet Request))
AND
An Internet connection request reached internal database because a mis-configured flow entry existed.
V2
EXISTENCE(t=[72s], S2, packet(@S2, Internet Request))
V3-a
EXISTENCE(t=[72s], S2, flowTable(@S2, Internet Request, Forward, Port3))
V3-b
EXISTENCE(t=[72s], Controller, packetIn(@Controller, Internet Request))
V3-c
ABSENCE(t=[1s,71s], S2, flowTable(@S2, Internet Request, *, *))
V3-d
ABSENCE(t=[1s,71s], S2, packetIn(@S2, Internet Request))
V3-e
ABSENCE(t=[1s,71s], S2, packet(@S2, Internet Request))

Figure A.4: Answer to Q5, as returned by Y!.

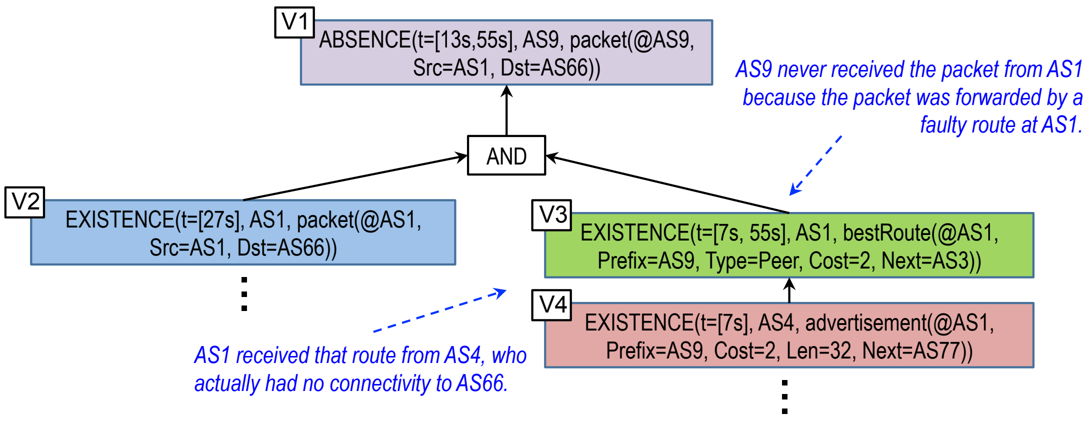

V1
ABSENCE(t=[13s,55s], AS9, packet(@AS9, Src=AS1, Dst=AS66))
AND
AS9 never received the packet from AS1 because the packet was forwarded by a faulty route at AS1.
V2
EXISTENCE(t=[27s], AS1, packet(@AS1, Src=AS1, Dst=AS66))
V3
EXISTENCE(t=[7s, 55s], AS1, bestRoute(@AS1, Prefix=AS9, Type=Peer, Cost=2, Next=AS3))
V4
EXISTENCE(t=[7s], AS4, advertisement(@AS1, Prefix=AS9, Cost=2, Len=32, Next=AS77))
AS1 received that route from AS4, who actually had no connectivity to AS66.

Figure A.5: Answer to Q7, as returned by Y!.

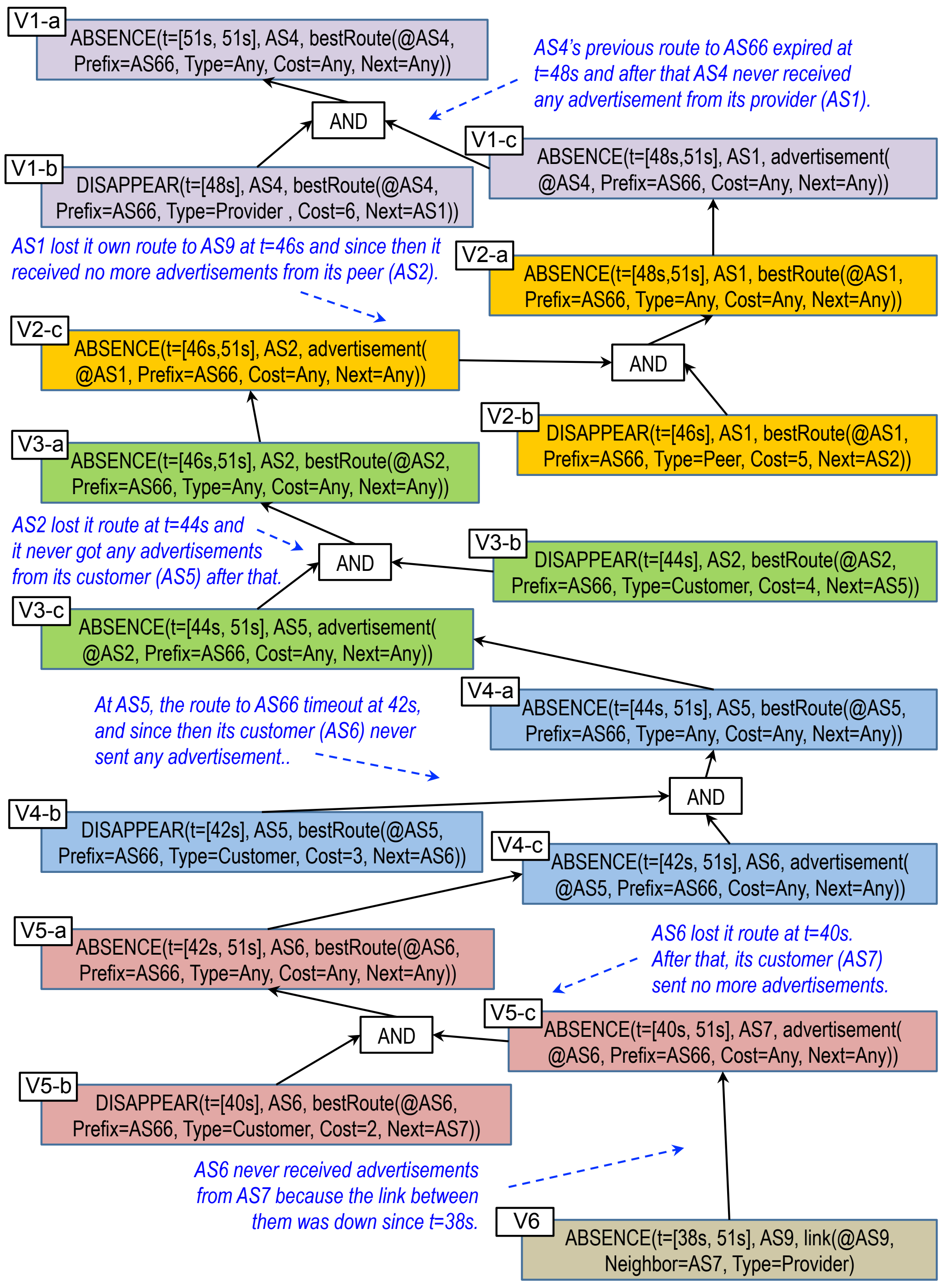
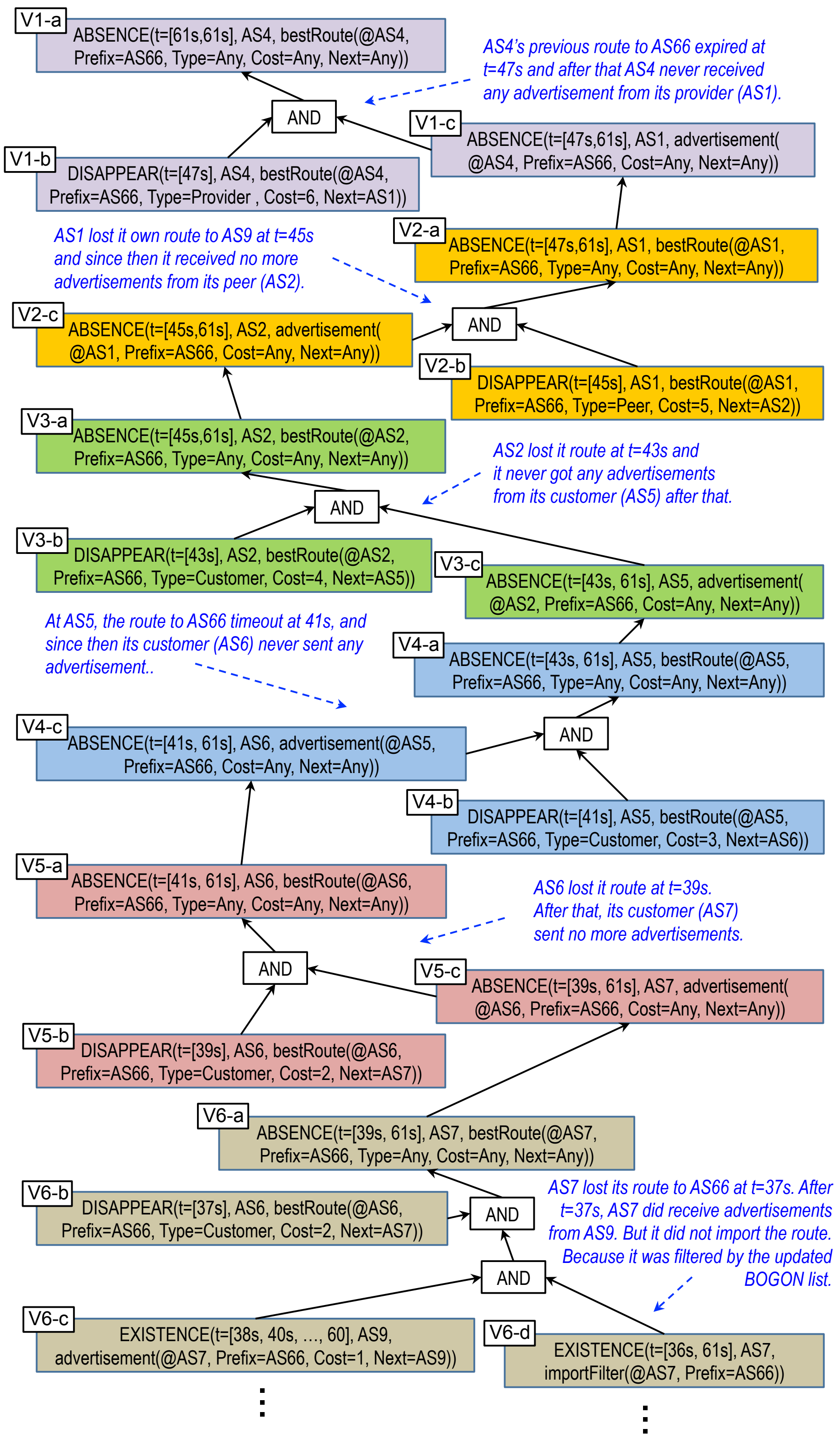

V1-a
ABSENCE(t=[51s, 51s], AS4, bestRoute(@AS4, Prefix=AS66, Type=Any, Cost=Any, Next=Any))
AS4's previous route to AS66 expired at t=48s and after that AS4 never received any advertisement from its provider (AS1).
AND
V1-c
ABSENCE(t=[48s,51s], AS1, advertisement( @AS4, Prefix=AS66, Cost=Any, Next=Any))
V1-b
DISAPPEAR(t=[48s], AS4, bestRoute(@AS4, Prefix=AS66, Type=Provider , Cost=6, Next=AS1))
AS1 lost it own route to AS9 at t=46s and since then it received no more advertisements from its peer (AS2).
V2-a
ABSENCE(t=[48s,51s], AS1, bestRoute(@AS1, Prefix=AS66, Type=Any, Cost=Any, Next=Any))
V2-c
ABSENCE(t=[46s,51s], AS2, advertisement( @AS1, Prefix=AS66, Cost=Any, Next=Any))
AND
V2-b
DISAPPEAR(t=[46s], AS1, bestRoute(@AS1, Prefix=AS66, Type=Peer, Cost=5, Next=AS2))
V3-a
ABSENCE(t=[46s,51s], AS2, bestRoute(@AS2, Prefix=AS66, Type=Any, Cost=Any, Next=Any))
AS2 lost it route at t=44s and it never got any advertisements from its customer (AS5) after that.
AND
V3-b
DISAPPEAR(t=[44s], AS2, bestRoute(@AS2, Prefix=AS66, Type=Customer, Cost=4, Next=AS5))
V3-c
ABSENCE(t=[44s, 51s], AS5, advertisement( @AS2, Prefix=AS66, Cost=Any, Next=Any))
V4-a
ABSENCE(t=[44s, 51s], AS5, bestRoute(@AS5, Prefix=AS66, Type=Any, Cost=Any, Next=Any))
At AS5, the route to AS66 timeout at 42s, and since then its customer (AS6) never sent any advertisement..
AND
V4-b
DISAPPEAR(t=[42s], AS5, bestRoute(@AS5, Prefix=AS66, Type=Customer, Cost=3, Next=AS6))
V4-c
ABSENCE(t=[42s, 51s], AS6, advertisement( @AS5, Prefix=AS66, Cost=Any, Next=Any))
V5-a
ABSENCE(t=[42s, 51s], AS6, bestRoute(@AS6, Prefix=AS66, Type=Any, Cost=Any, Next=Any))
AS6 lost it route at t=40s. After that, its customer (AS7) sent no more advertisements.
V5-c
ABSENCE(t=[40s, 51s], AS7, advertisement( @AS6, Prefix=AS66, Cost=Any, Next=Any))
AND
V5-b
DISAPPEAR(t=[40s], AS6, bestRoute(@AS6, Prefix=AS66, Type=Customer, Cost=2, Next=AS7))
AS6 never received advertisements from AS7 because the link between them was down since t=38s.
V6
ABSENCE(t=[38s, 51s], AS9, link(@AS9, Neighbor=AS7, Type=Provider)

Figure A.6: Answer to Q8, as returned by Y!.



V1-a
ABSENCE(t=[61s,61s], AS4, bestRoute(@AS4, Prefix=AS66, Type=Any, Cost=Any, Next=Any))
AS4's previous route to AS66 expired at t=47s and after that AS4 never received any advertisement from its provider (AS1).
AND
V1-c
ABSENCE(t=[47s,61s], AS1, advertisement( @AS4, Prefix=AS66, Cost=Any, Next=Any))
V1-b
DISAPPEAR(t=[47s], AS4, bestRoute(@AS4, Prefix=AS66, Type=Provider , Cost=6, Next=AS1))
AS1 lost it own route to AS9 at t=45s and since then it received no more advertisements from its peer (AS2).
V2-a
ABSENCE(t=[47s,61s], AS1, bestRoute(@AS1, Prefix=AS66, Type=Any, Cost=Any, Next=Any))
V2-c
ABSENCE(t=[45s,61s], AS2, advertisement( @AS1, Prefix=AS66, Cost=Any, Next=Any))
AND
V2-b
DISAPPEAR(t=[45s], AS1, bestRoute(@AS1, Prefix=AS66, Type=Peer, Cost=5, Next=AS2))
V3-a
ABSENCE(t=[45s,61s], AS2, bestRoute(@AS2, Prefix=AS66, Type=Any, Cost=Any, Next=Any))
AS2 lost it route at t=43s and it never got any advertisements from its customer (AS5) after that.
AND
V3-b
DISAPPEAR(t=[43s], AS2, bestRoute(@AS2, Prefix=AS66, Type=Customer, Cost=4, Next=AS5))
V3-c
ABSENCE(t=[43s, 61s], AS5, advertisement( @AS2, Prefix=AS66, Cost=Any, Next=Any))
At AS5, the route to AS66 timeout at 41s, and since then its customer (AS6) never sent any advertisement..
V4-a
ABSENCE(t=[43s, 61s], AS5, bestRoute(@AS5, Prefix=AS66, Type=Any, Cost=Any, Next=Any))
V4-c
ABSENCE(t=[41s, 61s], AS6, advertisement(@AS5, Prefix=AS66, Cost=Any, Next=Any))
AND
V4-b
DISAPPEAR(t=[41s], AS5, bestRoute(@AS5, Prefix=AS66, Type=Customer, Cost=3, Next=AS6))
V5-a
ABSENCE(t=[41s, 61s], AS6, bestRoute(@AS6, Prefix=AS66, Type=Any, Cost=Any, Next=Any))
AS6 lost it route at t=39s. After that, its customer (AS7) sent no more advertisements.
AND
V5-c
ABSENCE(t=[39s, 61s], AS7, advertisement( @AS6, Prefix=AS66, Cost=Any, Next=Any))
V5-b
DISAPPEAR(t=[39s], AS6, bestRoute(@AS6, Prefix=AS66, Type=Customer, Cost=2, Next=AS7))
V6-a
ABSENCE(t=[39s, 61s], AS7, bestRoute(@AS7, Prefix=AS66, Type=Any, Cost=Any, Next=Any))
V6-b
DISAPPEAR(t=[37s], AS6, bestRoute(@AS6, Prefix=AS66, Type=Customer, Cost=2, Next=AS7))
AND
AS7 lost its route to AS66 at t=37s. After t=37s, AS7 did receive advertisements from AS9. But it did not import the route. Because it was filtered by the updated BOGON list.
AND
V6-c
EXISTENCE(t=[38s, 40s, …, 60], AS9, advertisement(@AS7, Prefix=AS66, Cost=1, Next=AS9))
V6-d
EXISTENCE(t=[36s, 61s], AS7, importFilter(@AS7, Prefix=AS66))
⋮
⋮

Figure A.7: Answer to Q9, as returned by Y!.

# B Meta Provenance

Appendix B.1 presents full meta models for all languages to which we applied meta provenance. We describe all helpful functions in Appendix B.2. Resuming our discussion in Section 4.3.5, we expand on the properties of the generated repairs in Appendix B.3. In Section 4.5.3, we only provided the repair candidates for Q1. Appendix B.4 includes results for all remaining scenarios.

## B.1 Meta models

In this section, we present meta models for NDlog [83], Trema [127], and Pyretic [96]. The meta models are all written in NDlog. Unlike in the heavily-simplified $\mu$Dlog, instances of the same syntactic element can have different arities in real-world languages. For instance, considering a NDlog rule `r` with three selection predicates. The meta rule `h2` from Figure 4.4 can only encode rules with exactly two selection predicates and a different meta rule is required to encode `r`. Writing all possible meta rules is repetitive. For compactness, we present them using template rules. For

| **Procedure** | **Template rule.** $\rightarrow$ **Concrete rule with arity two (k = 2).** |
|---|---|
| Replace `(k)` with `k`. | `A(@X):=B(@X,Z),Z==(k).` $\rightarrow$ `A(@X):=B(@X,Z),Z==2.` |
| Replace `Z[k]` with `Z1`, ..., `Zk`. | `A(@X):=B(k)(@X,Z[k]).` $\rightarrow$ `A(@X):=B2(@X,Z1,Z2).` |
| Replace `B(@X,Z{k})` with `B(@X,Z1)`, ..., `B(@X,Zk).` | `A(@X):=B(@X,Z{k}).` $\rightarrow$ `A(@X):=B(@X,Z1),B(@X,Z2).` |
| Replace `B(@X,Z{k},Z{k'})` with ..., `B(@X,Zi,Zj)`, ... where $i, j \in 1..k, i < j$ | `A(@X):=B(@X,Z{k}),Z{k}>Z{k'}.` $\rightarrow$ `A(@X):=B(@X,Z1,Z2),Z1>Z2.` |
| Replace `B(@X,Z{k},Z{k''})` with ..., `B(@X,Zi,Zj)`, ... where $i, j \in 1..k, i \neq j$ | `A(@X):=B(@X,Z{k}),Z{k}>Z{k''}.` $\rightarrow$ `A(@X):=B(@X,Z1,Z2),Z1>Z2.` |

Table B.1: Procedures for deriving concrete rules from template rules. Each template rule has one or more arity specifiers (e.g., `k`). Each template rule expands into a set of concrete rules with different arities (e.g., `k=1` until `k=99`). Rules with multiple arity specifiers are expanded recursively.

instance, the meta model uses `Base(k)(@C,Tab,Vals[k])` to represent a base table with `k` columns. Table B.1 summarizes the procedures for expanding template rules into concrete rules.

### B.1.1 NDlog

Figure B.1 and Figure B.2 show the full set of meta rules for NDlog. We briefly explain each meta rule below.

Two types of tuples exist in the NDlog runtime: `State` represents materialized states, `Message` represents transient messages. Tuples can exist for two reasons: they can be inserted as base tuples (`h1`–`h2`) or derived via rules (`h3`–`h4`). A rule "fires" and derives a tuple when a) there is an assignment of values in the head and b) all constraint predicates are satisfied (`h5`–`h7`).

The next seven meta rules compute joins. First, whenever a (syntactic) tuple appears as in a rule definition, each concrete tuple that exists at runtime generates one variable assignment for that tuple (`p1`–`p2`). Depending on the number of tuples in the rule body (calculated in rules `j1`–`j2`), a different meta rule is triggered to compute a cross-product of all values in a combination of tuple predicates (`j3`–`j5`).

```
/* Tuple derivation */
h1 Message(k)(@C,Tab,Key,TID,Vals[k],Typs[k]) :- Base(k)(@C,Tab,Vals[k]),
     Schema(k)(@C,Tab,Keys[k],Timeout,Typs[k]), Timeout == 0, Key := f_primary(Keys,Vals),
     TID := f_unique().
h2 State(k)(@C,Tab,Key,Vals[k],Typs[k]) :- Base(k)(@C,Tab,Vals[k]),
     Schema(k)(@C,Tab,Keys[k],Timeout,Typs[k]), Timeout == 1, Key := f_primary(Keys,Vals).
h3 Message(k)(@C,Tab,Key,TID,Vals[k],Typs[k]) :-
     Head(k)(@C,Rul,Tab,Key,Vals[k],Timeout,Typs[k]), Timeout == 0, TID := f_unique().
h4 State(k)(@C,Tab,Key,Vals[k],Typs[k]) :-
     Head(k)(@C,Rul,Tab,Key,Vals[k],Timeout,Typs[k]), Timeout == 1.
h5 Head(k)(@C,Rul,Tab,Key,Val[k],Timeout,Typ[k]) :-
     HeadMeta(k)(@C,Rul,Tab,Arg[k]), Schema(k)(@C,Tab,Key[k],Timeout,Typ[k]),
     Key := f_primary(Key[k],Val[k]), HeadValue(@C,Rul,JID,Arg{k},Val{k},Typ{k}),
     Arg{k} != Arg{k''}, ConstraintMatch(@C,Rul,JID).
h6 ConstraintMatch(@C,Rul,JID) :- ConstraintCount(@C,Rul,N), N == 0.
h7 ConstraintMatch(@C,Rul,JID) :-
     ConstraintCount(@C,Rul,N), Constraint(@C,Rul,JID,CID{k},Val{k}), N == (k), Val{k} == 1,
     CID{k} > CID{k'}.

/* Predicate derivation */
p1 MessagePredicate(k)(@C,Rul,Tab,Key,TID,Args[k],Vals[k],Typs[k]) :-
     Message(k)(@C,Tab,Key,TID,Vals[k],Typs[k]), PredicateMeta(k)(@C,Rul,Tab,Args[k]).
p2 StatePredicate(k)(@C,Rul,Tab,Key,Args[k],Vals[k],Typs[k]) :-
     State(k)(@C,Tab,Key,Vals[k],Typs[k]), PredicateMeta(k)(@C,Rul,Tab,Args[k]).

/* Joining predicates */
j1 MessagePredicateCount(@C,Rul,a_count<Tab>) :- PredicateMeta(k)(@C,Rul,Tab,Args[k]).
j2 StatePredicateCount(@C,Rul,a_count<Tab>) :- PredicateMeta(k)(@C,Rul,Tab,Args[k]).
j3 Join($\sum_{i=1}^{n} p_i$)(@C,Rul,JID,TID,Args''{n}_[$p_i$],Vals''{n}_[$p_i$],Typs''{n}_[$p_i$]) :- JID := f_unique(),
     StatePredicate($p_i$)(@C,Rul,Tab''{n},Key''{n},Args''{n}_[$p_i$],Vals''{n}_[$p_i$],Typs''{n}_[$p_i$]),
     StatePredicateCount(@C,Rul,N'), MessagePredicateCount(@C,Rul,N), N' == (n), N == 0,
     TID := f_unique().
j4 Join($\sum_{i=1}^{m} k_i$)(@C,Rul,JID,TID,Args'{m}_[$k_i$],Vals'{m}_[$k_i$],Typs'{m}_[$k_i$]) :- JID := f_unique(),
     MessagePredicate($k_i$)(@C,Rul,Tab'{m},Key'{m},TID,Args'{m}_[$k_i$],Vals'{m}_[$k_i$],Typs'{m}_[$k_i$]){m},
     MessagePredicateCount(@C,Rul,N), N == (m), StatePredicateCount(@C,Rul,N'), N' == 0.
j5 Join($\sum_{i=1}^{n} p_i + \sum_{i=1}^{m} k_i$)(@C,Rul,JID,TID,Args'{m}_[$k_i$],Args''{n}_[$p_i$],Vals'{m}_[$k_i$],Vals''{n}_[$p_i$],
     Typs'{m}_[$k_i$],Typs''{n}_[$p_i$]) :- MessagePredicate($k_i$)(@C,Rul,Tab'{m},Key'{m},TID,Args'{m}_[$k_i$],
     Vals'{m}_[$k_i$],Typs'{m}_[$k_i$]){m}, MessagePredicateCount(@C,Rul,N), StatePredicate($p_i$)(@C,Rul,
     Tab''{n},Key''{n},Args''{n}_[$p_i$],Vals''{n}_[$p_i$],Typs''{n}_[$p_i$]), StatePredicateCount(@C,Rul,N'),
     N == (m), N' == (n), JID := f_unique().

/* Expression */
e1 Expression(@C,Rul,JID,Arg(q),Val(q),Typ(q)) :- Join(k)(@C,Rul,JID,TID,Arg[k],Val[k],Typ[k]).
e2 Expression(@C,Rul,JID,ID,Val,Typ) :- Constant(@C,Rul,ID,Val), JID := *.
e3 Expression(@C,Rul,JID,ID''',Val,Typ) :-
     Operator(@C,Rul,ID''',Opr), LeftEdge(@C,Rul,ID',ID'''), RightEdge(@C,Rul,ID'',ID'''),
     Expression(@C,Rul,JID,ID',Val',Typ'), Expression(@C,Rul,JID,ID'',Val'',Typ''),
     Val := Val' Opr Val'', ID' != ID'', f_match(Typ',Typ''), Typ := f_type(Typ',Typ'').

/* Assignments */
a1 HeadValue(@C,Rul,JID,Arg,Val,Typ) :-
     Assignment(@C,Rul,Arg,ID,Typ), Expression(@C,Rul,JID,ID,Val,Typ'), f_match(Typ,Typ').

/* Constraint */
c1 ConstraintCount(@C,Rul,a_count<ID>) :- IsConstraint(@C,Rul,ID).
c2 Constraint(@C,Rul,JID,ID,Val) :-
     Expression(@C,Rul,JID,ID,Val,Typ), IsConstraint(@C,Rul,ID), Typ == Bool.
```

Figure B.1: Meta rules for NDlog [83] (part 1).

```
/* Aggregation */
g1 Match(@C,Rul,TID,JID) :- Join(k)(@C,Rul,JID,TID,Args[k],Vals[k],Typs[k]), ConstraintCount(@C,Rul,N),
      IsAggWrap(@C,Rul), Constraint(@C,Rul,JID,ID{p},Val{p}), Val{p} == 1, ID{k} > ID{k'}, N == (k).
g2 Count(@C,Rul,TID,a_count<JID>) :- Match(@C,Rul,TID,JID).
g3 MessagePredicate(k)(@C,Rul,Tab,Key,TID',Args[k],Arg,Vals[k],N) :- Count(@C,Rul,TID,N),
      MessagePredicate(k)(@C,Rul,Tab,Key,TID,Args[k],Vals[k]), Arg := 'count', TID' := f_unique().
```

Figure B.2: Meta rules for NDlog [83] (part 2).

The next six meta rules evaluate expressions. Expressions can appear in two different places – in a rule head and in a constraint predicate – but since the evaluation logic is the same, we use a single set of meta rules for both cases. Expressions can come from integer constants (`e1`), composition of sub-expressions (`e2`), or any value of a `Join` meta tuple (`e3`). Values in the head tuple are assigned from expressions (`a1`). Constraint predicates are defined over boolean expressions (`c1`–`c2`).

The final three meta rules describe an aggregation syntax in NDlog called "AggWrap", which counts the number of derivations triggered by a message via itself. Note that, each message is associated with an unique trigger ID, or `TID` (generated in `h1` and `h3`). A derivation is triggered when all constraints match (`g1`). The aggregation rule counts all such derivations caused by a unique message (`g2`–`g3`).

### B.1.2 Trema

Figures B.3–B.5 show the full set of meta rules for Trema [127]. Unlike NDlog, Trema is based on Ruby and mostly imperative. To capture the imperative control flow of Ruby, the meta model remembers which lines have been executed using a table called `ExecLine(@P,B,SID,Ln)`. For example, the tuple `ExecLine(@P,F,97,10)` says that function `F` executed its 10th line during its 97th invocation. To model the imperative data flow of Ruby, the meta model maintains a table called `Value(@P,Adr,B,SID,Ln,Arg,Val,Typ)`, which tracks which variables are visible at each line and what their values are. For instance, the tuple `Value(@P,0x8A,F,99,10,a,2,int)` says that an `int` variable `a` had a value of `2` when function executed its 10th line during its 99th invocation. Next, we briefly explain each meta rule below.

```
/* Processing PacketIn */
// Entering the "packet_in" handler
pi1 ExecLine(@P,B,SID,Ln) :- packetIn(k)(@C,Swi,SID,Fld[k],Val[k],Typ[k]),
      EntryLine(@P,B,Ln), FuncCall(k)(@P,B',Ln,B,EID[k]), B == packet_in.
// Creating the "packet" object
pi2 Expression(@P,B,SID,Ln,EID,Adr,Val,Typ) :-
     packetIn(k)(@C,Swi,SID,Fld[k],Val[k],Typ[k]), EntryLine(@P,B,Ln), FuncCall(k)(@P,B',Ln,B,EID[k]),
      B == packet_in, EID := EID2, Adr := f_new(), Val := f_new(), Typ := Class.
// Creating attributes of the "packet" object
pi3 Value(@P,Adr',B,SID,Ln,Arg',Val',Typ'), ClassMap(@P,Val,Attr,Adr') :-
     packetIn(k)(@C,Swi,SID,Fld[k],Val[k],Typ[k]), EntryLine(@P,B,Ln), FuncCall(k)(@P,B',Ln,B,EID[k]),
      Adr' := f_new(), Arg := Fld(k), Attr := Fld(k), Val' := Val(k), Typ' := Typ(k),
      B == packet_in, Expression(@P,B,SID,Ln,EID,Adr,Val,Typ).
// Creating the "switch" variable
pi4 Expression(@P,B,SID,Ln,EID,Adr,Val,Typ) :-
     packetIn(k)(@C,Swi,SID,Fld[k],Val[k],Typ[k]), EntryLine(@P,B,Ln), FuncCall(k)(@P,B',Ln,B,EID[k]),
      B == packet_in, EID := EID1, Adr := f_new(), Val := Swi, Typ := Fixnum.

/* Installing flow entries */
// Installing a "micro" flow entry
fe1 flowEntryMicro(@P,Swi,SID,Prt) :-
      FuncExec(k)(@P,B,SID,Ln,EID[k],B',Ln',Arg'[k]), B' == send_flow_mod_add,
      Expression(@P,B,SID,Ln,EID1,Adr1,Val1,Typ1), Typ1 == Fixnum, Swi := Val1,
      Expression(@P,B,SID,Ln,EID2,Adr2,Val2,Typ2), Typ2 == Fixnum, Prt := Val2.
fe2 flowEntry(k)(@P,Swi,SID,Fld[k],Val[k],Typ[k],Prt) :- flowEntryMicro(@P,Swi,SID,Prt),
      packetIn(k)(@P,Swi,SID,Fld[k],Val[k],Typ[k]).
// Installing a "macro" flow entry
fe3 flowEntry(k)(@P,Swi,SID,Fld[k],Val[k],Typ[k],Prt) :-
      FuncExec(k)(@P,B,SID,Ln,EID[k],B',Ln',Arg[k]), B' == send_flow_mod_wildcard,
      Expression(@P,B,SID,Ln,EID{k},Adr{k},Val{k},Typ{k}), Prt := Val9.
// Sending a PacketOut message to handle the cached packet on the switch
fe4 packetOutMicro(@P,Swi,SID,Prt) :-
      FuncExec(k)(@P,B,SID,Ln,EID[k],B',Ln',Arg'[k]), B' == send_packet_out,
      Expression(@P,B,SID,Ln,EID1,Adr1,Val1,Typ1), Typ1 == Fixnum, Swi := Val1,
      Expression(@P,B,SID,Ln,EID2,Adr2,Val2,Typ2), Typ2 == Fixnum, Prt := Val2.
fe5 packetOut(k)(@P,Swi,SID,Fld[k],Val[k],Typ[k],Prt) :- packetOutMicro(@P,Swi,SID,Prt),
      packetIn(k)(@P,Swi,SID,Fld[k],Val[k],Typ[k]).

/* If clauses */
// Executing a if clause when its predicate is satisfied
cj1 ExecLine(@P,B,SID,Tln) :-
    IfClause(@P,B,Ln,EID,Tln,Fln), Expression(@P,B,SID,Ln,EID,Adr,Val,Typ), Val == 1, Typ := TrueClass.
cj2 Value(@P,Adr',B,SID,Tln,Arg',Val',Typ') :- Value(@P,Adr',B,SID,Ln,Arg',Val',Typ'),
    IfClause(@P,B,Ln,EID,Tln,Fln), Expression(@P,B,SID,Ln,EID,Adr,Val,Typ), Val == 1, Typ := TrueClass.
// Skipping a if clause when its predicate is not satisfied
cj3 ExecLine(@P,B,SID,Fln) :-
    IfClause(@P,B,Ln,EID,Tln,Fln), Expression(@P,B,SID,Ln,EID,Adr,Val,Typ), Val == 0, Typ := FalseClass.
cj4 Value(@P,Adr',B,SID,Fln,Arg',Val',Typ') :- Value(@P,Adr',B,SID,Ln,Arg',Val',Typ'),
    IfClause(@P,B,Ln,EID,Tln,Fln), Expression(@P,B,SID,Ln,EID,Adr,Val,Typ), Val == 0, Typ := FalseClass.
```

Figure B.3: Meta rules for Trema [127] (part 1).

```
/* Expression */
// A constant derives an expression
e1 Expression(@P,B,SID,Ln,EID,Adr,Val,Typ) :-
      ExecLine(@P,B,SID,Ln), Constant(@P,B,Ln,CID,Val,Typ), f_hash(EID), Adr := f_new().
// A local variables derives an expression
e2 Expression(@P,B,SID,Ln,EID,Adr,Val,Typ) :-
      ExecLine(@P,B,SID,Ln), VarName(@P,B,Ln,Arg), Value(@P,Adr,B,SID,Ln,Arg,Val,Typ), EID := Arg,
      f_arg(Arg,0).
// An object attribute derives an expressions
e3 Expression(@P,B,SID,Ln,Arg,Adr',Val',Typ') :- Expression(@P,B,SID,Ln,EID,Adr,Val,Typ),
      AttributeOf(@P,B,Ln,EID,Attr), f_concat(Arg,EID,.,Attr), Value(@P,Adr',B,SID,Ln,Arg,Val',Typ'),
      Typ == Class, ClassMap(@P,Val,Attr,Adr'), f_arg(EID,0), f_arg(Attr,0).
// Compositions of sub-expressions
e4 Expression(@P,B,SID,Ln,EID,Adr,Val,Typ) :-
      Operator(@P,B,Ln,Opr,EID1,EID2), Expression(@P,B,SID,Ln,EID1,Adr1,Val1,Typ1),
      Expression(@P,B,SID,Ln,EID2,Adr2,Val2,Typ2), EID1 != EID2, Val := Val1 Opr Val2,
      f_hash(EID), Adr := f_new(), f_match(Typ1, Typ2), Typ := f_type(Typ1, Typ2).
// Checking or retrieving a hash table entry derive expressions
e5 HashTableCount(@P,B,SID,Ln,Arg,Key,a_count<Val>) :-
      HashTableCheck(@P,B,SID,Ln,Arg,KID), Expression(@P,B,SID,Ln,KID,Adr',Val',Typ'),
      HashTableEntry(@P,B,SID,Ln,Arg,Key,Val,Typ), Key == Val'.
e6 Expression(@P,B,SID,Ln,EID,Adr,Val,Typ) :- HashTableCount(@P,B,SID,Ln,Arg,Key,N),
      N > 0, f_concat(EID,Key,in,Arg), Adr := f_new(), Val := True, Typ := TrueClass.
e7 Expression(@P,B,SID,Ln,EID,Adr,Val,Typ) :- HashTableCount(@P,B,SID,Ln,Arg,Key,N),
      N == 0, f_concat(EID,Key,in,Arg), Adr := f_new(), Val := False, Typ := FalseClass.
e8 Expression(@P,B,SID,Ln,EID,Adr,Val,Typ) :-
      HashTableGet(@P,B,Ln,Arg,KID), Expression(@P,B,SID,Ln,KID,Adr',Val',Typ'),
      HashTableEntry(@P,B,SID,Ln,Arg,Key,Val,Typ), Key == Val',
      Adr = f_new(), f_concat(EID,Arg,#,Key).

/* Function call */
// Triggering a function call
fc1 FuncExec(k)(@P,B,SID,Ln,EID[k],B',Ln',Arg'[k]) :-
      ExecLine(@P,B,SID,Ln), FuncDecl(k)(@P,B',Arg'[k],Ln'), FuncCall(k)(@P,B,Ln,B',EID[k]).
// Copying arguments to the callee
fc2 Value(@P,Adr',B',SID,Ln',Arg',Val,Typ) :- FuncExec(k)(@P,B,SID,Ln,EID[k],B',Ln',Arg'[k]),
      Expression(@P,B,SID,Ln,EID,Adr,Val,Typ), EID == EID(k), Adr' := f_new(), Arg' := Arg'(k).
// Copying attributes of object arguments to the callee
fc3 Value(@P,Adr'',B',SID,Ln',Arg'',Val'',Typ'') :- FuncExec(k)(@P,B,SID,Ln,EID[k],B',Ln',Arg'[k]),
      Expression(@P,B,SID,Ln,EID,Adr,Val,Typ), EID == EID(k), ClassMap(@P,Val,Attr,Adr''),
      Typ == Class, f_concat(Arg'',Arg'(k),.,Attr), Value(@P,Adr'',B,SID,Ln,Arg'',Val'',Typ'').
// Executing the function body
fc4 ExecLine(@P,B',SID,Ln') :- FuncExec(k)(@P,B,SID,Ln,EID[k],B',Ln',Arg'[k]).

/* Function return */
// Triggering a function return
fr1 FuncRet(@P,B',SID,Ln',EID') :- ExecLine(@P,B',SID,Ln'), Return(@P,B',Ln',EID').
// Copying the return value to the caller
fr2 Expression(@P,B,SID,Ln,EID,Adr,Val,Typ) :-
      FuncRet(@P,B',SID,Ln',EID'), FuncExec(k)(@P,B,SID,Ln,EID[k],B',Ln',Arg'[k]), Adr := f_new(),
      Expression(@P,B',SID,Ln',EID',Adr',Val',Typ'), Ln' > 0, EID := EID', Val := Val', Typ := Typ'.
// Returning to the caller
fr3 ExecLine(@P,B,SID,Ln'') :- FuncRet(@P,B',SID,Ln',EID'),
      FuncExec(k)(@P,B,SID,Ln,EID[k],B',Ln',Arg'[k]), NextLine(@P,B,Ln,Ln''), Ln != Ln''.
```

Figure B.4: Meta rules for Trema [127] (part 2).

```
/* Objects */
// Calling the constructor
of1 FuncCallObject(@P,B,SID,Ln,B',Ln',EID[k],Adr,Val,Typ,Ini) :-
      ExecLine(@P,B,SID,Ln), ObjectNew(@P,B,Ln,Typ,B',Ln',EID[k]), Adr := f_new(), Val := f_new(),
      Ini := True. // Allocating attributes
of2 Value(@P,Adr',B',SID,Ln',Arg',Val',Typ'), ClassMap(@P,Val,Attr,Adr') :-
      FuncCallObject(@P,B,SID,Ln,B',Ln',EID[k],Adr,Val,Typ,Ini), ObjectDecl(@P,Typ,Attrs,Typs),
      Adr' := f_new(), Arg' := Attr, Val' := Null, Typ' := Typs(k), Attr := Attrs(k), Ini == True.
// Allocating the object itself
of3 Value(@P,Adr',B',SID,Ln',Arg',Val',Typ'), ClassMap(@P,Val,Attr,Adr') :-
      FuncCallObject(@P,B,SID,Ln,B',Ln',EID[k],Adr,Val,Typ,Ini), Adr' := f_new(), Ln' := 1,
      Arg' := 'self', Val' := Adr, Typ' := Ref, Ini == True.
// Calling a member function
of4 FuncCallObject(@P,B,SID,Ln,B',Ln',EID[k],Adr,Val,Typ,Ini) :-
      Expression(@P,B,SID,Ln,EID,Adr,Val,Typ), FunctionOf(@P,B,Ln,EID,B',Ln',EID[k]), Typ == Ref,
      Ini := False.
// Copying attributes to a member function call and starting its execution
of5 Value(@P,Adr',B',SID,Ln',Arg',Val',Typ') :-
      FuncCallObject(@P,B,SID,Ln,B',Ln',EID[k],Adr,Val,Typ,Ini), ObjectDecl(@P,Typ,Attrs,Typs),
      Value(@P,Adr,B,SID,Ln,Arg,Val,Typ), ClassMap(@P,Val,Attr,Adr), Adr' := Adr, Arg' := Attr,
      Val' := Val, Typ' := Typ, Attr == Attrs(k), Ini == False.
of6 FuncCall(k)(@P,B,Ln,B',EID[k]) :- FuncCallObject(@P,B,SID,Ln,B',Ln',EID[k],Adr,Val,Typ,Ini).

/* Assignment */
// Assigning value to a variable
a1 Value(@P,Adr,B,SID,Ln',Arg,Val',Typ), ExecLine(@P,B,SID,Ln') :-
      Expression(@P,B,SID,Ln,Arg,Adr,Val,Typ), f_arg(Arg,1), Assignment(@P,B,Ln,Arg,EID),
      Expression(@P,B,SID,Ln,EID,Adr',Val',Typ), NextLine(@P,B,Ln,Ln'), Ln != Ln'.
// Propagating unchanged values
a2 AssignmentCount(@P,B,Ln,Arg,a_count<EID>) :- Assignment(@P,B,Ln,Arg,EID).
a3 NoAssignment(@P,B,Ln,Arg) :- AssignmentCount(@P,B,Ln,Arg,N), N == 0.
a4 Value(@P,Adr,B,SID,Ln',Arg,Val,Typ) :- Value(@P,Adr,B,SID,Ln,Arg,Val,Typ),
      NoAssignment(@P,B,Ln,Arg), NextLine(@P,B,Ln,Ln'), Ln != Ln', f_arg(Arg,1).

/* Hash table */
// Modifying a hash table entry
ht1 HashTableEntry(@P,B,SID,Ln,Arg,Key,Val,Typ), ExecLine(@P,B,SID,Ln') :-
      HashTableSet(@P,B,Ln,Arg,KID,VID), Expression(@P,B,SID,Ln,KID,Adr',Val',Typ'),
      Expression(@P,B,SID,Ln,VID,Adr'',Val'',Typ''), Key := Val', Val := Val'',Typ := Typ'',
      NextLine(@P,B,Ln,Ln'), Ln != Ln'.
// Propagating unchanged values
ht2 HashTableCount(@P,B,Ln,Arg,a_count<KID>) :- HashTableSet(@P,B,Ln,Arg,KID,VID).
ht3 NoHashTableSet(@P,B,SID,Ln,Arg,Key) :- HashTableCount(@P,B,Ln,Arg,N), N == 0.
ht4 HashTableEntry(@P,B,SID,Ln',Arg,Key,Val,Typ) :- HashTableEntry(@P,B,SID,Ln,Arg,Key,Val,Typ),
      NoHashTableSet(@P,B,Ln,Arg,Key), NextLine(@P,B,Ln,Ln'').
```

Figure B.5: Meta rules for Trema [127] (part 3).

The first nine meta rules describe inputs and outputs of a Trema program. The program reactively installs flow entries. When a switch does not know how to handle a flow, it caches and encapsulates its first packet in a `PacketIn` message to the controller. A Trema controller invokes a handler function called `packet_in` with two arguments `switch` and `packet` (`pi1`–`pi4`). The former identifies the switch which sent the `PacketIn`. The latter stores header fields of the encapsulated packet. A network operator specifies the policy by implementing the `packet_in` function. The function install flow entries by calling the built-in `send_flow_mod_add` function. Depending on the provided arguments, Trema may install either a "micro" flow entry that exactly match one flow or a "macro" flow entry that can control multiple flows using wildcard matches (`fe1`–`fe3`). In addition, the programmer can instruct the switch to handle the cached packet by calling `send_packet_out` (`fe4`–`fe5`).

The next four meta rules evaluate `if` clauses. An `if` clause performs two actions when its predicate is satisfied: a) it updates `ExecLine` to execute the first line in the `if` body, and b) it propagates variables in the current context to the `if` body by changing the line number of `Value` tuples (`cj1`–`cj2`). Otherwise, the clause skips the `if` body and executes the following line (`cj3`–`cj4`).

The next seven meta rules evaluate expressions. Expressions can come from constants (`e1`), local variables (`e2`), object attributes (`e3`), compositions of subexpressions (`e4`), checking the existence of a hash table entry (`e5`–`e7`), or retrieving a hash table entry (`e8`).

The next seven meta rules execute functions. When a function is called (`fc1`), the program: a) copies the arguments in order for them to become visible to the callee (`fc2`); b) copies attributes of object arguments (`fc3`); and c) updates `ExecLine` to execute the first line in the function body (`fc4`). When the function returns (`fr1`), the returned value (if any) is copied to the caller and the following line in the caller executes (`fr2`–`fr3`).

The next six meta rules describe objects. When the program creates an object, it allocates the object and calls the constructor for the class (`of1`–`of3`). When the object calls its member function, the program copies attributes of the object to the callee and executes the first line of the function body (`of4`–`of6`).

The final nine meta rules assign values. The program can assign values to a variable (`a1`), or to a hash table entry (`ht1`). Depending on whether the variable has been updated, its new or old value propagates to the next line to execute (`a2`–`a4`). Hash table entries propagate in a similar fashion (`ht2`–`ht4`).

## B.1.3 Pyretic

Pyretic [96] is a domain-specific language for programming software-defined networks. It is embedded in Python. Its meta model describes a subset of the imperative features of Python, similar to that of Ruby. The meta rules in Figures B.6–B.8 are analogous to rules in the Trema model, therefore we do not discuss them in detail. We focus on the NetCore syntax [96], which are encoded by rules in Figure B.9.

NetCore supports several types of static policies: a primitive action forwards, drops, floods, or modifies all incoming traffic; a restricted policy `P[C]` applies the policy `C` to a flow if it satisfies the predicate `P`; a sequential policy `C1 >> C2` first process the flow using `C1`, directs the immediate result through `C2`, and return the final output; a parallel policy `C1 | C2` applies `C1` and `C2` simultaneously to a flow and returns the union of the outputs. Note that programmers can recursively compose policies from sub-policies to specify complex packet processing pipelines.

The model represents policy with the table `Policy(@P,B,Typ,Ln,Sub[],Act[])`. Each policy has a type of `Typ`, resides in the function `B`, and has a unique ID `Ln`. A policy may be composed from sub-policies, whose unique IDs are `Sub[]`. After a policy is executed, all its sub- policies are invoked. In addition, `Policy` maintains the IDs of all primitive actions from which it is recursively composed (`Act[]`). (We will come back to this shortly when discussing sequential policies.)

```
/* Processing PacketIn */
// Entering the "packet_in" handler
pi1 ExecLine(@P,B,SID,Ln) :-
      packetIn(k)(@C,Swi,SID,Fld[k],Val[k],Typ[k]), EntryLine(@P,B,Ln), B == packet_in.
// Creating the "packet" object
pi2 Expression(@P,B,SID,Ln,EID,Adr,Val,Typ) :-
      packetIn(k)(@C,Swi,SID,Fld[k],Val[k],Typ[k]), EntryLine(@P,B,Ln), B == packet_in, EID := packet,
      Adr := f_new(), Val := f_new(), Typ := Class.
// Creating attributes of the "packet" object
pi3 Value(@P,Adr',B,SID,Ln,Arg',Val',Typ'), ClassMap(@P,Val,Attr,Adr') :-
     packetIn(k)(@C,Swi,SID,Fld[k],Val[k],Typ[k]), EntryLine(@P,B,Ln), Adr' := f_new(), Arg := Fld[k],
      Attr := Fld(k), Val' := Val(k), Typ' := Typ(k), B == packet_in,
      Expression(@P,B,SID,Ln,EID,Adr,Val,Typ).

/* Installing flow entries */
// Installing a "micro" flow entry
fe1 flowEntryMicro(@P,SID,Prt) :-
      ExecLine(@P,B,SID,Ln'), Policy(@P,B,Typ,Ln,Sub[i],Act[j]), Value(@P,Adr,B,SID,Ln,Arg,Val,Typ'),
      Ln < 0, Arg == packet.@action, Prt := Val, Typ == Prim, Typ' == Fixnum..
fe2 flowEntry(k)(@P,Swi,SID,Fld[k],Val[k],Typ[k],Prt) :- flowEntryMicro(@P,SID,Prt),
      packetIn(k)(@P,Swi,SID,Fld[k],Val[k],Typ[k]).
// Installing a "macro" flow entry
fe3 flowEntry(k)(@P,Swi,SID,Fld[k],Val[k],Typ[k],Prt) :-
      FuncExec(k)(@P,B,SID,Ln,EID[k],B',Ln',Arg[k]), B' == wildcard_helper,
      Expression(@P,B,SID,Ln,EID{k},Adr{k},Val{k},Typ{k}), Swi := Val1, Prt := Val9.
// Sending a PacketOut message to handle the cached packet on the switch
fe4 packetOutMicro(@P,SID,Prt) :-
      ExecLine(@P,B,SID,Ln'), Policy(@P,B,Typ,Ln,Sub[i],Act[j]), Value(@P,Adr,B,SID,Ln,Arg,Val,Typ'),
      Ln < 0, Arg == packet.@action, Prt := Val, Typ == Prim, Typ' == Fixnum..
fe5 packetOutMicro(@P,SID,Prt) :-
      FuncExec(k)(@P,B,SID,Ln,EID[k],B',Ln',Arg[k]), B' == wildcard_helper,
      Expression(@P,B,SID,Ln,EID{k},Adr{k},Val{k},Typ{k}), Prt := Val9.
fe6 packetOut(k)(@P,Swi,SID,Fld[k],Val[k],Typ[k],Prt) :- packetOutMicro(@P,SID,Prt),
      packetIn(k)(@P,Swi,SID,Fld[k],Val[k],Typ[k]).

/* If clauses */
// Executing a if clause when its predicate is satisfied
cj1 ExecLine(@P,B,SID,Tln) :-
      IfClause(@P,B,Ln,EID,Tln,Fln), Expression(@P,B,SID,Ln,EID,Adr,Val,Typ), Val == 1,
      Typ := TrueClass.
cj2 Value(@P,Adr',B,SID,Tln,Arg',Val',Typ') :-
      Value(@P,Adr',B,SID,Ln,Arg',Val',Typ'), IfClause(@P,B,Ln,EID,Tln,Fln), Val == 1,
      Expression(@P,B,SID,Ln,EID,Adr,Val,Typ), Typ := TrueClass.
// Skipping a if clause when its predicate is not satisfied
cj3 ExecLine(@P,B,SID,Fln) :-
      IfClause(@P,B,Ln,EID,Tln,Fln), Expression(@P,B,SID,Ln,EID,Adr,Val,Typ), Val == 0,
      Typ := FalseClass.
cj4 Value(@P,Adr',B,SID,Fln,Arg',Val',Typ') :-
      Value(@P,Adr',B,SID,Ln,Arg',Val',Typ'), IfClause(@P,B,Ln,EID,Tln,Fln), Val == 0,
      Expression(@P,B,SID,Ln,EID,Adr,Val,Typ), Typ := FalseClass.
```

Figure B.6: Meta rules for Pyretic [96] (part 1).

```
/* Expression */
// A constant derives an expression
e1 Expression(@P,B,SID,Ln,EID,Adr,Val,Typ) :-
      ExecLine(@P,B,SID,Ln), Constant(@P,B,Ln,CID,Val,Typ), f_hash(EID), Adr := f_new().
// A local variables derives an expression
e2 Expression(@P,B,SID,Ln,EID,Adr,Val,Typ) :-
      ExecLine(@P,B,SID,Ln), VarName(@P,B,Ln,Arg), Value(@P,Adr,B,SID,Ln,Arg,Val,Typ), EID := Arg,
      f_arg(Arg,0).
// An object attribute derives an expressions
e3 Expression(@P,B,SID,Ln,Arg,Adr',Val',Typ') :- Expression(@P,B,SID,Ln,EID,Adr,Val,Typ),
      AttributeOf(@P,B,Ln,EID,Attr), f_concat(Arg,EID,.,Attr), Value(@P,Adr',B,SID,Ln,Arg,Val',Typ'),
      Typ == Class, ClassMap(@P,Val,Attr,Adr'), f_arg(EID,0), f_arg(Attr,0).
// Compositions of sub-expressions
e4 Expression(@P,B,SID,Ln,EID,Adr,Val,Typ) :-
      Operator(@P,B,Ln,Opr,EID1,EID2), Expression(@P,B,SID,Ln,EID1,Adr1,Val1,Typ1),
      Expression(@P,B,SID,Ln,EID2,Adr2,Val2,Typ2), EID1 != EID2, Val := Val1 Opr Val2, f_hash(EID),
      Adr := f_new(), f_match(Typ1, Typ2), Typ := f_type(Typ1, Typ2).
// Checking or retrieving a hash table entry derive expressions
e5 HashTableCount(@P,B,SID,Ln,Arg,Key,a_count<Val>) :-
      HashTableCheck(@P,B,SID,Ln,Arg,KID), Expression(@P,B,SID,Ln,KID,Adr',Val',Typ'),
      HashTableEntry(@P,B,SID,Ln,Arg,Key,Val,Typ), Key == Val'.
e6 Expression(@P,B,SID,Ln,EID,Adr,Val,Typ) :- HashTableCount(@P,B,SID,Ln,Arg,Key,N),
      N > 0, f_concat(EID,Key,in,Arg), Adr := f_new(), Val := True, Typ := TrueClass.
e7 Expression(@P,B,SID,Ln,EID,Adr,Val,Typ) :- HashTableCount(@P,B,SID,Ln,Arg,Key,N),
      N == 0, f_concat(EID,Key,in,Arg), Adr := f_new(), Val := False, Typ := FalseClass.
e8 Expression(@P,B,SID,Ln,EID,Adr,Val,Typ) :-
      HashTableGet(@P,B,Ln,Arg,KID), Expression(@P,B,SID,Ln,KID,Adr',Val',Typ'),
      HashTableEntry(@P,B,SID,Ln,Arg,Key,Val,Typ), Key == Val', Adr = f_new(),
      f_concat(EID,Arg,#,Key).

/* Function call */
// Triggering a function call
fc1 FuncExec(k)(@P,B,SID,Ln,EID[k],B',Ln',Arg'[k]) :-
      ExecLine(@P,B,SID,Ln), FuncDecl(k)(@P,B',Arg'[k],Ln'), FuncCall(k)(@P,B,Ln,B',EID[k]).
// Copying arguments to the callee
fc2 Value(@P,Adr',B',SID,Ln',Arg',Val,Typ) :- FuncExec(k)(@P,B,SID,Ln,EID[k],B',Ln',Arg'[k]),
      Expression(@P,B,SID,Ln,EID,Adr,Val,Typ), EID == EID(k), Adr' := f_new(), Arg' := Arg'(k).
// Copying attributes of object arguments to the callee
fc3 Value(@P,Adr'',B',SID,Ln',Arg'',Val'',Typ'') :- FuncExec(k)(@P,B,SID,Ln,EID[k],B',Ln',Arg'[k]),
      Expression(@P,B,SID,Ln,EID,Adr,Val,Typ), EID == EID(k), ClassMap(@P,Val,Attr,Adr''),
      Typ == Class, f_concat(Arg'',Arg'(k),.,Attr), Value(@P,Adr'',B,SID,Ln,Arg'',Val'',Typ'').
// Executing the function body
fc4 ExecLine(@P,B',SID,Ln') :- FuncExec(k)(@P,B,SID,Ln,EID[k],B',Ln',Arg'[k]).

/* Function return */
// Triggering a function return
fr1 FuncRet(@P,B',SID,Ln',EID') :- ExecLine(@P,B',SID,Ln'), Return(@P,B',Ln',EID').
// Copying the return value to the caller
fr2 Expression(@P,B,SID,Ln,EID,Adr,Val,Typ) :-
      FuncRet(@P,B',SID,Ln',EID'), FuncExec(k)(@P,B,SID,Ln,EID[k],B',Ln',Arg'[k]), Adr := f_new(),
      Expression(@P,B',SID,Ln',EID',Adr',Val',Typ'), Ln' > 0, EID := EID', Val := Val', Typ := Typ'.
// Returning to the caller
fr3 ExecLine(@P,B,SID,Ln'') :- FuncRet(@P,B',SID,Ln',EID'),
      FuncExec(k)(@P,B,SID,Ln,EID[k],B',Ln',Arg'[k]), NextLine(@P,B,Ln,Ln''), Ln != Ln''.
```

Figure B.7: Meta rules for Pyretic [96] (part 2).

```
/* Objects */
// Calling the constructor
of1 FuncCallObject(@P,B,SID,Ln,B',Ln',EID[k],Adr,Val,Typ,Ini) :-
      ExecLine(@P,B,SID,Ln), ObjectNew(@P,B,Ln,Typ,B',Ln',EID[k]), Adr := f_new(), Val := f_new(),
      Ini := True.
// Allocating attributes
of2 Value(@P,Adr',B',SID,Ln',Arg',Val',Typ'), ClassMap(@P,Val,Attr,Adr') :-
      FuncCallObject(@P,B,SID,Ln,B',Ln',EID[k],Adr,Val,Typ,Ini), ObjectDecl(@P,Typ,Attrs,Typs),
      Adr' := f_new(), Arg' := Attr, Val' := Null, Typ' := Typs(k), Attr := Attrs(k), Ini == True.
// Allocating the object itself
of3 Value(@P,Adr',B',SID,Ln',Arg',Val',Typ'), ClassMap(@P,Val,Attr,Adr') :-
      FuncCallObject(@P,B,SID,Ln,B',Ln',EID[k],Adr,Val,Typ,Ini), Adr' := f_new(), Ln' := 1,
      Arg' := 'self', Val' := Adr, Typ' := Ref, Ini == True.
// Calling a member function
of4 FuncCallObject(@P,B,SID,Ln,B',Ln',EID[k],Adr,Val,Typ,Ini) :-
      Expression(@P,B,SID,Ln,EID,Adr,Val,Typ), FunctionOf(@P,B,Ln,EID,B',Ln',EID[k]), Typ == Ref,
      Ini := False.
// Copying attributes to a member function call and starting its execution
of5 Value(@P,Adr',B',SID,Ln',Arg',Val',Typ') :-
      FuncCallObject(@P,B,SID,Ln,B',Ln',EID[k],Adr,Val,Typ,Ini), ObjectDecl(@P,Typ,Attrs,Typs),
      Value(@P,Adr,B,SID,Ln,Arg,Val,Typ), ClassMap(@P,Val,Attr,Adr), Adr' := Adr, Arg' := Attr,
      Val' := Val, Typ' := Typ, Attr == Attrs(k), Ini == False.
of6 FuncCall(k)(@P,B,Ln,B',EID[k]) :- FuncCallObject(@P,B,SID,Ln,B',Ln',EID[k],Adr,Val,Typ,Ini).

/* Assignment */
// Assigning value to a variable
a1 Value(@P,Adr,B,SID,Ln',Arg,Val',Typ), ExecLine(@P,B,SID,Ln') :-
      Expression(@P,B,SID,Ln,Arg,Adr,Val,Typ), f_arg(Arg,1), Assignment(@P,B,Ln,Arg,EID),
      Expression(@P,B,SID,Ln,EID,Adr',Val',Typ), NextLine(@P,B,Ln,Ln'), Ln != Ln'.
// Propagating unchanged values
a2 AssignmentCount(@P,B,Ln,Arg,a_count<EID>) :- Assignment(@P,B,Ln,Arg,EID).
a3 NoAssignment(@P,B,Ln,Arg) :- AssignmentCount(@P,B,Ln,Arg,N), N == 0.
a4 Value(@P,Adr,B,SID,Ln',Arg,Val,Typ) :- Value(@P,Adr,B,SID,Ln,Arg,Val,Typ),
      NoAssignment(@P,B,Ln,Arg), NextLine(@P,B,Ln,Ln'), Ln != Ln', f_arg(Arg,1).

/* Hash table */
// Modifying a hash table entry
ht1 HashTableEntry(@P,B,SID,Ln,Arg,Key,Val,Typ), ExecLine(@P,B,SID,Ln') :-
      HashTableSet(@P,B,Ln,Arg,KID,VID), Expression(@P,B,SID,Ln,KID,Adr',Val',Typ'),
      Expression(@P,B,SID,Ln,VID,Adr'',Val'',Typ''), Key := Val', Val := Val'',Typ := Typ'',
      NextLine(@P,B,Ln,Ln'), Ln != Ln'.
// Propagating unchanged values
ht2 HashTableCount(@P,B,Ln,Arg,a_count<KID>) :- HashTableSet(@P,B,Ln,Arg,KID,VID).
ht3 NoHashTableSet(@P,B,SID,Ln,Arg,Key) :- HashTableCount(@P,B,Ln,Arg,N), N == 0.
ht4 HashTableEntry(@P,B,SID,Ln',Arg,Key,Val,Typ) :- HashTableEntry(@P,B,SID,Ln,Arg,Key,Val,Typ),
      NoHashTableSet(@P,B,Ln,Arg,Key), NextLine(@P,B,Ln,Ln'').
```

Figure B.8: Meta rules for Pyretic [96] (part 3).

```
/* Primitive actions */
a1 Value(@P,Adr',B,SID,Ln',Arg,Val',Typ') :- ExecLine(@P,B,SID,Ln), Ln' := Sub(i),
      Policy(@P,B,Typ,Ln,Sub[i],Act[j]), Expression(@P,B,SID,Ln,EID,Adr',Val',Typ'), Typ == Prim,
      ConstantAction(@P,B,Ln,EID), Arg == packet.@action.
a2 Value(@P,Adr',B,SID,Ln',Arg,Val',Typ') :- ExecLine(@P,B,SID,Ln), Ln' := Sub(i),
      Policy(@P,B,Typ,Ln,Sub[i],Act[j]), Expression(@P,B,SID,Ln,EID,Adr',Val',Typ'), Typ == Prim,
      ModifyAction(@P,B,Fld,Ln,EID), Value(@P,Adr,B,SID,Ln,Arg,Val,Typ'), f_concat(Arg,packet,.,Fld).
a3 ExecLine(@P,B,SID,Sub) :- ExecLine(@P,B,SID,Ln), Policy(@P,B,Typ,Ln,Sub,Act), Typ == Prim.
a4 Value(@P,Adr,B,SID,Ln',Arg,Val,Typ) :- Value(@P,Adr,B,SID,Ln,Arg,Val,Typ),
      NoAssignment(@P,B,Ln,Arg), Policy(@P,B,Typ,Ln,Sub[i],Act[j]), Ln' := Sub(i).

/* Restricted policies */
r1 PredicateValue(@P,B,SID,Ln,Val) :-
      FieldPredicate(@P,B,Ln,Fld,VID), Value(@P,Adr',B,SID,Ln,Arg,Val',Typ'),
      f_concat(Arg,packet,.,Fld), Expression(@P,B,SID,Ln,VID,Adr'',Val'',Typ''), f_match(Typ', Typ''),
      Val := Val' == Val''.
r2 PredicateValue(@P,B,SID,Ln,Val) :- ExecLine(@P,B,SID,Ln), ConstantPredicate(@P,B,Ln,Typ,Val).
r3 ExecLine(@P,B,SID,Ln') :- ExecLine(@P,B,SID,Ln), Ln' := Sub(i),
      Policy(@P,B,Typ,Ln,Sub[i],Act[j]), PredicateValue(@P,B,SID,Ln,Val), Val == true, Typ == Rest.

/* Parallel composition */
p1 Policy(@P,B,Typ,Ln,Sub[2],Act'[k'],Act''[k'']) :- Parallel(@P,B,Ln,Ln',Ln''), Typ := Para,
      Policy(@P,B,Typ',Ln',Sub'[k'],Act'[k']), Policy(@P,B,Typ'',Ln'',Sub''[k''],Act''[k'']),
      Sub[1] := Ln', Sub[2] := Ln''.
p2 ExecLine(@P,B,SID,Ln') :-
      ExecLine(@P,B,SID,Ln), Policy(@P,B,Typ,Ln,Sub[i],Act[j]), Ln' := Sub(i), Typ == Para.

/* Sequential composition */
s1 Policy(k)(@P,B,Typ,Ln',Sub'[k'],Act''[k'']) :- Sequential(@P,B,Ln',Ln''), Typ := Sequ,
      Policy(@P,B,Typ',Ln',Sub'[k'],Act'[k']), Policy(@P,B,Typ'',Ln'',Sub''[k''],Act''[k'']).
s2 Policy(@P,B,Typ,Ln,Sub,Ln'') :- Sequential(@P,B,Ln',Ln''),
     Policy(@P,B,Typ,Ln,Sub,Act), Typ == Prim, Policy(@P,B,Typ',Ln',Sub'[k'],Act'[k']), Ln == Act'(k').
```

Figure B.9: Meta rules for Pyretic [96] (part 4).

Consider a policy `P1` that forwards all HTTP traffic from the host `1.2.3.4` towards two ports: `match(dstport=80)[match(srcip=1.2.3.4) [fwd(1)|fwd(2)]]`. Note that `P1` is a restricted policy composed from a predicate that matches HTTP traffic and a sub-policy `P2`. `P2` is in turn composed from a predicate that matches the traffic source to `1.2.3.4` and a sequential composition of two primitive actions `fwd(1)` and `fwd(2)` (denoted as `P3` and `P4`). `P1` will generate a meta tuple `Policy(@P,B,Rest,P1,[P2],[P3,P4])`. Next, we explain each meta rule below.

The first four rules evaluate primitive actions. Depending on the associated action, the policy can change the output port or modify a header field of incoming flows (`a1`–`a2`). The flow is then forwarded to sub-policies or compiles to flow entries on the switch if no sub-policies exist (`a3`–`a4`).

The next three rules evaluate restricted policies. Each restricted policy filters incoming traffic with a predicate. A predicate can restrict the value of a header field or apply a constant filter such as "no packets" (`r1`–`r2`). Note that a restricted policy does not redirect or modify any incoming flow itself. Instead, it applies sub-policies on flows that satisfy the predicate (`r3`).

The final four rules describe compositions. A parallel policy derives form two sub-policies and a parallel operator (`p1`). For instance, consider `P3 | P4` from the previous example. The parallel policy `P'` is `Policy(@P,B,Para,P',[P3,P4],[P3,P4])`. Note that when `P'` is invoked, `P3` and `P4` will execute in parallel (`p2`). A sequential operator chains two sub-policies into a sequential policy (`s1`). Instead of generating flow entries after evaluating the first policy, the program applies the second policy to the processed flow (`s1`). Note that in order to support this, the meta model maintains all primitive actions in each policy (`Act[]`). For example, suppose the programmer writes `P3 >> P4`. Before the composition, `P3` will compile a flow entry that forwards packets to port `1` (`P3=Policy(@P,B,Prim,P3,-1,P3)`). The sequential operator causes `P4` to become a sub-policy of `P3`, such that `P4` will process the output of `P3` (`P3=Policy(@P,B,Prim,P3,P4,P3)`).

| **Function** | **Input** | **Description** |
|---|---|---|
| BaseTupleCombs | A provenance graph $P$. | Return all combinations of base events in $P$ such that the total costs of each combination is less than a cutoff. The returned combinations are sorted in cost order. |
| BaseTuples | A provenance graph $P$. | Return all base events in $P$. |
| ChangeTuple | A set of assignments $A$. A base event $\tau$. | Change $\tau$ to $\tau'$ by replacing all variables in $\tau$ with their concrete values in $A$. If the syntax remains valid after the change, return $\tau \rightarrow \tau'$. Return $\emptyset$ otherwise. |
| ConstraintPool | A provenance graph $P$. | Return the set of constraints associated with $P$. |
| DeleteTuple | A base event $\tau$. | If the syntax remains valid after deleting $\tau$, return $-\tau$. Return $\emptyset$ otherwise. |
| ExistingTuple | An event $\tau$. | Return true iff the event type of $\tau$ and is positive. |
| ForkWithRule | A provenance graph $P$. An event $\tau$ in $P$. A derivation rule $r$ that can derive $\tau$. | Expand event $\tau$ in $P$. Fork a provenance graph for each possible derivation. Return all forked provenance graphs. More detailed explanation in Appendix B.2. |
| MissingTuple | An event $\tau$. | Return true iff the event type of $\tau$ and is negative. |
| RootTuple | A provenance graph $P$. | Return the root of $P$. |
| Rules | An event $\tau$. | All meta rules that can derive $\tau$. |
| SatAssignment | A set of constraints $C$. | Solve $C$. Return a valid assignment of all variables. |
| SymbolicPropagate | A positive provenance graph $P$. A combination of base events $T_i$. | Change each base event in $T_i$ symbolically. Propagate the changes bottom-up in $P$. Collect all constraints that is required for a) the derivations in $P$ to hold and b) the root of $P$ to satisfy the operator's query. Return the set of constraints collected. More detailed explanation in Appendix B.2. |
| UnsatAssignment | A set of constraints $C$. | Solve $!C$. Return a valid assignment of all variables. |

Table B.2: A description of helper functions used in the algorithms from Figure 4.5 and Figure B.10.

## B.2 Helper functions

Table B.2 describes all helper functions used in the algorithms from Figure 4.5 and Figure B.10. We also briefly explain the more complicated helper functions below.

In the repair extraction algorithm from Figure 4.5, SymbolicPropagate changes a subset of base events in a provenance graph of $\tau$ and return all constraints such that the derivation of $\tau$ still holds. The procedure: a) replaces attributes in base events with symbolic variables, b) copies the symbolic variables to the head if the replaced attribute also appears in the head tuple, c) collects constraints such that the derivation still holds, and d) recursively perform this process bottom-up until reaching the root. For instance, suppose the algorithm must remove a faulty flow entry, which was generated via a $\mu$Dlog rule `r7` with a selection predicate `Swi == 2`. The repair extracting algorithm will attempt to change the constant `2` as one possible repair. At this point, SymbolicPropagate is invoked: a) the procedure replaces the constant with a symbolic value, such that `Const(@C,r7,0x9,2)` becomes `Const(@C,r7,0x9,V)`; b) the constant derived, via meta rule `e1`, an expression, which becomes `Expr(@C,r7,*,0x9,V)`; c) the expression derived, via meta rule `s1`, a selection predicate, which becomes `Sel(@C,r7,3,0x9,V == 2)`; d) the selection predicate derived the flow entry via meta rule `h2`, and in order for it to trigger, the constraint `(V == 2) == True` must hold; note that we collect this constraint from the `Val == True` predicate of the meta rule `h2`.

In the repair exploration algorithm from Figure B.10, ForkWithRule attempts to derive an event $\tau$ by expanding it to child vertexes. It accepts a derivation rule $r$ and fork a provenance graph for each possible way of expanding $\tau$ using $r$. For instance, suppose the algorithm needs to make a selection predicate `Swi == 2` hold in a $\mu$Dlog program. At this point, the algorithm invokes ForkWithRule to derive the corresponding `Sel` meta tuple using the meta rule `s1` (from Figure 4.4). A naïve way is to recursively derive all its predicates. However, an optimization can produce a repair with a smaller search space: the procedure checks whether a subset of preconditions already hold during the original execution and only attempts to derive the missing predicates. Note that meta provenance tends to make small changes to the program, therefore syntactic elements or states are likely to be preserved in an re-execution. In our example, the procedure checks when an operator and its right-hand expression existed but a matching left-hand expression was missing. At some point during the original execution, there existed a `==` operator and a `Expr` derived from the constant 2; as a result, we add two existing tuples `Oper` and `Expr` as children of `Sel`; however, the `Swi` expression did not have a value of 2; therefore, a missing child vertex `Expr` is added for later exploration. ForkWithRule enumerates all such cases and forks one provenance graph for each.

## B.3 Properties

In this section, we discuss properties of repairs generated using meta provenance for NDlog. Figure B.10 shows the algorithm. The operator specifies a symptom event $\tau$, and the algorithm returns a list of repairs $R$. When the symptom event $\tau$ is positive, repairs in $R$ make $\tau$ disappear: Lines 25 builds a positive meta provenance tree of $\tau$, using the graph construction algorithm from negative provenance (Chapter 3); Line 26 extracts repairs from the positive tree (using the function from Figure 4.5). When the symptom $\tau$ is a missing event, repairs in $R$ make $\tau$ appear. The algorithm stores negative meta provenance trees in a priority queue $Q$, where the top tree has the lowest cost. This allows the algorithm to explore and output repair candidates in cost order (Section 4.3.5): Lines 13-20 fork a partial tree, expand a vertex $v$ in the forked tree (increase its cost if $v$ is a program change), and push all expanded trees into $Q$; Lines 10-11 accept a completed tree from $Q$ and extract a repair candidate that makes $\tau$ appear.

The algorithm may keep expanding vertexes in a partial tree without ever making a program change. Such a tree stays at the top of $Q$. As a result, the algorithm cannot

```
function FindRepairs($\tau$)
    if MissingTuple($\tau$) then
        // use a priority queue $Q$ to store meta provenance trees
        $Q \leftarrow \tau$, $R \leftarrow \emptyset$
        while $Q \neq \emptyset$ do
            // check the lowest-cost meta provenance trees
            $P \leftarrow Q$.Top()
            $Q$.Remove($P$)
            if $P$.Todos$= \emptyset$ then
                // extract a repair from a completed tree and output
                $R \leftarrow R \cup$ GenerateRepairCandidates($P$)
            else
                // choose one vertex $\tau_e$ in the paritial tree
                $\tau_e \leftarrow P$.Todos.First()
                $P$.Todos.Remove($v$)
                // iterate all meta rules that can derive $\tau_e$
                for $\forall\ r \in$ Rules($\tau_e$) do
                    // fork the tree by expanding $\tau_e$ with meta rule $r$
                    for $\forall\ P_i \in$ ForkWithRule($P, \tau_e, r$) do
                        $Q$.Push($P_i$)
        // return repairs to make $\tau$ appear
        return $R$
    else if ExistingTuple($\tau$) then
        // query the positive meta provenance tree of $\tau$
        $P \leftarrow$ Query($\tau$)
        // extract repairs to make $\tau$ disappear
        return GenerateRepairCandidates($P$)
```

Figure B.10: Algorithm for exploring repair candidates in cost order. For a description of the helper functions, please see Table B.2.

extract repairs from complete (but more expensive) trees in $Q$. To handle this, we can add a (possibly very small) cost to expanding each vertex, even if it does not represent a program change. This ensures that any viable repair with a finite cost will be output eventually.

The prototype does not have the completeness property. Because it does not actually output the type of repairs mentioned in the ensuing proof. Such repairs are too restrictive and are unlikely to be useful in practice. The algorithm in Figure 4.5 can be easily extended to support this case.

**Property (Optimality)**: *Repair candidates are generated in cost order. Consider a list of repairs $R \leftarrow$* FindRepairs*($\tau$), $R[i].cost \leq R[j].cost$ when $i < j$.*

**Proof.** We consider two cases: a) the symptom $\tau$ is an existing event, and b) the symptom $\tau$ is a missing event.

**Case a.** Lines 24-27 are executed. Note that the algorithm generates a single meta provenance tree. The result is optimal as long as GenerateRepairCandidates returns a sorted list of repairs. This is guaranteed because BaseTupleCombinations returns base tuple combinations in cost order (Table B.2).

**Case b.** Lines 3-22 are executed. The result is optimal by construction: the algorithm only extracts a repair $R[i]$ when $R[i]$ is the top of $Q$; any remaining provenance tree $P_r$ in $Q$ has a cost that is no less than $R[i]$; a future repair $R[j]$ must be extracted from a complete $P_r$ or a tree expanded from a partial $P_r$; therefore, the cost of $R[j]$ is the cost of $P_r$ plus what additional program changes cost (if any); the sum is no less than the cost of $R[i]$. $\square$

**Property (Completeness)**: *Given a symptom $\tau$, the algorithm finds at least one working repair.*

**Proof.** We consider two cases: a) the symptom $\tau$ is an existing event, and b) the symptom $\tau$ is a missing event.

**Case a.** The algorithm can eventually look at all the rules that are being used to derive $\tau$ in the current execution, and add a condition that prevents $\tau$, and only $\tau$, from being derived without causing any other changes[1]. For example, to remove a positive event `Bar(@3,5)`, the rule `Bar(@A,B):-Foo(@A,B),A>2` becomes `Bar(@A,B):-Foo(@A,B),A>2,A!=3 || B!=5`.

There is a concern that a repair may disable one particular derivation of an undesired tuple while enabling an alternative derivation of the same tuple. For instance, deleting a `X(5)` tuple that is being used to derive $\tau$ could trigger some other rule, perhaps one with condition `!X(5)`, that now also derives a $\tau$. However, notice that the repair in the above argument only removes an derivation of $\tau$, and makes no other changes. If the absence of $\tau$ would trigger a cascade of rules that made it reappear, the system as a whole would oscillate, since that rule would effectively depend on a negation of its own rule head.

**Case b.** The algorithm will eventually add a rule that derives exactly $\tau$ without causing any other changes. For example, to derive a tuple `Bar(@A,B)`, one repair will add a rule `Bar(@A,B):-Foo(@X),X==1,A:=2,B:=3`, where `Foo(@1)` is an event that happened during that time. If no such event exist in the original execution, the algorithm will insert a base event.

Note that the algorithm can eventually generate the above repair. Suppose that `Bar` is a state and `Foo` is a message. To make `Bar(@2,3)` appear, the algorithm, at some point, adds the rule head `Bar(@A,B)` (using meta rules `h4-h5` from Figure B.1); while recursively expanding the `HeadValue` predicates of `h5`, assignments `A:=2` and `B:=3` are created (`a1` and `e2`); while recursively expanding the `ConstraintMatch` predicate of `h5`, the selection predicate `X==1` is created (`h6-h7`); finally, the algorithm adds the predicate `Foo(@X)` while expanding the `X` expression in `X==1` (`e1`, `j4`, and `p1`). □

## B.4 Scenarios

Table B.3 and Table B.4 show the repair candidates returned for Q2-Q5. In Q2, a forwarding policy was too restrictive and dropped packets from one host. All repair candidates are effective as they cause packets from the blocked client to go through, but candidates D-L are too general and misroute flows that were handled correctly by the original program. In Q3, the operator updated the load-balancer program and offloaded a few clients to a backup route, but a stale firewall policy on that route dropped all such requests. All repair candidates are effective as the fixed firewall always allow at least of some of the offloaded requests. Some candidates have undesirable side effects: the firewall becomes too permissive (C, F, G, I, J, K); all hosts connected to a certain port receive additional traffic (E, F); legitimate

|   | **Repair candidate (Accepted?)** | **KS-test** |
|---|---|---|
| A | Manually installing a flow entry (✓) | 0.00086 |
| B | Changing `Sip<6` in `r1` to `Sip<7` (✓) | 0.00086 |
| C | Changing `Sip<6` in `r1` to `Sip<=6` (✓) | 0.00086 |
| D | Changing `Prt:=16` in `r2` to `Prt:=17` (✗) | 0.00257 |
| E | Changing `Sip<6` in `r1` to `Sip<99` (✗) | 0.00257 |
| F | Changing `Sip<6` in `r1` to `Sip<16` (✗) | 0.00257 |
| G | Changing `Sip<6` in `r1` to `Sip<2009` (✗) | 0.00257 |
| H | Deleting `Sip<6` in `r1` (✗) | 0.00257 |
| I | Deleting `Sip<6` and `Ipt<16` in `r1` (✗) | 0.00257 |
| J | Changing `Sip<6` in `r1` to `Dpt<6` (✗) | 0.00257 |
| K | Changing `Sip<6` in `r1` to `Spt<6` (✗) | 0.00257 |
| L | Changing `Sip<6` in `r1` to `Tmt<6` (✗) | 0.00257 |

(a) Q2

|   | **Repair candidate (Accepted?)** | **KS-test** |
|---|---|---|
| A | Manually installing a flow entry (✓) | 0.00085 |
| B | Changing `Sip>3` in `r5` to `Sip>1` (✓) | 0.00085 |
| C | Changing `Sip>3` in `r5` to `Sip>0` (✗) | 0.00171 |
| D | Changing `Sip>3` in `r5` to `Sip>2` (✓) | 0.00085 |
| E | Deleting `Dip==1` and `Swi==2001` in `e1h1` (✗) | 0.01886 |
| F | Deleting `Sip>3` and `Swi==2003` in `r5` (✗) | 0.00213 |
| G | Deleting `Sip>3` in `r5` (✗) | 0.00171 |
| H | Changing `Sip>3` in `r5` to `Sip<3` (✗) | 0.00689 |
| I | Changing `Sip>3` in `r5` to `Swi>3` (✗) | 0.00171 |
| J | Changing `Sip>3` in `r5` to `Dip>3` (✗) | 0.00171 |
| K | Changing `Sip>3` in `r5` to `Dmc>3` (✗) | 0.00171 |

(b) Q3

Table B.3: Candidate repairs generated by meta provenance for scenarios Q2 and Q3, which are then filtered by a KS-test.

traffic are blocked (H). In Q4, the controller program failed to instruct a switch to forward the first packet in each incoming flow. All repair candidates cause the switch to forward the first packet, but some cause side effects such as significant increases of controller traffic (C, D, E, G) or additional flows at endhosts (D, G, H, I, K, M). In Q5, a switch never learned about the existence of certain hosts because the MAC learning program only match packets based on the incoming port and the destination IP. Some repair candidates did not fix the problem (B, C, D, E, F, H). Candidate I manually configures an entry in the learning table on the controller. Candidates A and G fix the problem by adding a matching field on source IP or source MAC, which are probably the repairs which an operator would have chosen.

| | **Repair candidate (Accepted?)** | **KS-test** |
|---|---|---|
| A | Manually sending a packetOut message (✓) | 0.02693 |
| B | Changing the head of `e2po` to `packetOut(...,Sip,Dip,Spt,Dpt,...)` (✕) | 0.34439 |
| C | Changing the head of `r5` to `packetOut(...,Dip,Sip,Spt,Dpt,...)` (✕) | 0.38886 |
| D | Changing the head of `e2` to `packetOut(...,Sip,Dip,Spt,Dpt,...)` (✕) | 0.38886 |
| E | Changing the head of `r5` to `packetOut(...,Dip,Sip,Dpt,Spt,...)` (✕) | 0.38886 |
| F | Changing the head of `e2po` to `packetOut(...,Sip,Dip,Dpt,Spt,...)` (✕) | 0.34439 |
| G | Changing the head of `e2` to `packetOut(...,Sip,Dip,Dpt,Spt,...)` (✕) | 0.38886 |
| H | Copying `e2` and replacing head with `packetOut(...,Sip,Dip,Dpt,Spt,...)` (✕) | 0.20223 |
| I | Copying `e2po` and replacing head with `packetOut(...,Sip,Dip,Dpt,Spt,...)` (✕) | 0.20223 |
| J | Copying `r5` and replacing head with `packetOut(...,Dip,Sip,Dpt,Spt,...)` (✓) | 0.02693 |
| K | Copying `e2` and replacing head with `packetOut(...,Sip,Dip,Spt,Dpt,...)` (✕) | 0.20223 |
| L | Copying `r5` and replacing head with `packetOut(...,Dip,Sip,Spt,Dpt,...)` (✓) | 0.02693 |
| M | Copying `e2po` and replacing head with `packetOut(...,Sip,Dip,Spt,Dpt,...)` (✕) | 0.20223 |

(a) Q4

| | **Repair candidate (Accepted?)** | **KS-test** |
|---|---|---|
| A | Changing `Sip':=*` in `f2` to `Sip':=Sip` (✓) | 0.00007 |
| B | Changing `Sip':=*` in `f2` to `Sip':=Dip` (✕) | 0.00009 |
| C | Changing `Dip':=*` in `f2` to `Dip':=Sip` (✕) | 0.00009 |
| D | Changing `Dmc':=Dmc` in `f2` to `Dmc':=Smc` (✕) | 0.00009 |
| E | Changing `Ipt':=Ipt` in `f2` to `Ipt':=Prt` (✕) | 0.00009 |
| F | Changing `Smc':=*` in `f2` to `Smc':=Dmc` (✕) | 0.00009 |
| G | Changing `Smc':=*` in `f2` to `Smc':=Smc` (✓) | 0.00007 |
| H | Changing `Smc':=*` in `f2` to `Smc':=Dmc''` (✕) | 0.00009 |
| I | Manually installing a learning table entry (✓) | 0.00007 |

(b) Q5

Table B.4: Candidate repairs generated by meta provenance for scenarios Q4 and Q5, which are then filtered by a KS-test.

# C
# Temporal Provenance

We discussed properties of temporal provenance in Section 5.4.6. We provide the formal statements of the properties and their proofs in Appendix C.1.

## C.1 Formal Model

Temporal provenance preserves all properties of classical provenance (validity, soundness, completeness, and minimality). We have obtained the corresponding proofs by extending the formal model from TAP/DTaP [143]. Although there are some parts of the proof from [143] that require few or no changes (e.g., because they only relate to functional provenance and not to sequencing), we present the full formal model here for completeness. Our extensions include the following:

- Temporal provenance has a different set of vertex types (Section 5.3.2) and contains sequencing edges (Section 5.4.1); consequently, temporal provenance graphs are constructed differently (Section C.1.2).

- The validity property, in addition to its prior requirements from TAP, requires that the temporal provenance include all the events necessary to reproduce the execution temporally (Section C.1.3).
- The proofs follow the same structure as in TAP, but are adjusted to handle the different graph structure and the stronger validity property of temporal provenance (Section C.1.4).

We have also formally modeled the properties of the delay annotations that our algorithm creates (and that were not part of [143]):

- Definitions of direct delay and transitive delay; and a theorem states that each vertex is labeled with the amount of delay that is contributed by the subtree that is rooted at that vertex (Section C.1.5).
- A theorem states that the annotations do correspond to the "potential for speedup" that we intuitively associate with the concept of delay (Section C.1.6).

## C.1.1 Background: Execution Traces

To set stage for the discussion, we introduce some necessary concepts for reasoning about the execution of the system in our temporal provenance model.

An *execution trace* of an NDlog program can be characterized by a sequence of *events* that take place in the system, starting from the initial system state. Each event on a node captures the execution of a particular rule $r$ that is triggered by a certain tuple $\tau$, under the existence of some other tuples on the node, and that results in a new tuple being derived or an existing tuple being underived (i.e., lost). We formally define them below.

**Definition 9. (Event)**: *An event $d@n$ is represented by $d@n = (\tau, r, t_s, t_e, c, \pm\tau')$, where*

- *$n$ is the node on which the event happened,*
- *$\tau$ is the tuple that triggers the event,*

- *$r$ is the derivation rule that is being triggered,*
- *$t_s$ is the time at which $r$ is triggered (called start timestamp),*
- *$t_e$ is the time at which $r$ finishes its execution (called end timestamp),*
- *$c$ is the set of tuples that are preconditions of the event, which must exist on $n$ at time $t_s$, and*
- *$\tau'$ is the tuple that is derived ($+$) or underived ($-$) as a result of the derivation.*

**Definition 10. (Trace)**: *A trace $\mathscr{E}$ is a sequence of events $\langle d_1@n_1, d_2@n_2, \ldots, d_k@n_k\rangle$ that reflects an execution of the system from the initial state $\mathscr{S}_0$, i.e.,*

$$\mathscr{S}_0 \xrightarrow{d_1@n_1} \mathscr{S}_1 \xrightarrow{d_2@n_2} \cdots \xrightarrow{d_k@n_k} \mathscr{S}_k.$$

To quantify the timing behaviors of the system, it is necessary to reason about the order among events. Ideally, we would like to have a total ordering among all events in all nodes in the system; however, due to the lack of fully synchronized clocks, this is difficult to achieve in distributed systems. To address this, we introduce the concept of trace *equivalence* that preserves the total ordering of events on each node, without imposing a total ordering among events across nodes. Intuitively, two traces $\mathscr{E}$ and $\mathscr{E}'$ are considered equivalent if the subsequence of events that every node observes in $\mathscr{E}$ is the same as that is observed in $\mathscr{E}'$.

**Definition 11. (Subtrace):** *$\mathscr{E}'$ is a subtrace of $\mathscr{E}$ (written as $\mathscr{E}' \subseteq \mathscr{E}$) iff $\mathscr{E}'$ is a subsequence of $\mathscr{E}$. We denote by $\mathscr{E}|n$ the subtrace of $\mathscr{E}$ that consists of all and only the events of $\mathscr{E}$ that take place on node $n$.*

**Definition 12. (Equivalence):** *Two traces $\mathscr{E}$ and $\mathscr{E}'$ are equivalent (written as $\mathscr{E} \sim \mathscr{E}'$) iff for all nodes $n$, $\mathscr{E}|n = \mathscr{E}'|n$.*

By definition, the equivalence relation is transitive: $\mathscr{E} \sim \mathscr{E}', \mathscr{E}' \sim \mathscr{E}'' \Rightarrow \mathscr{E} \sim \mathscr{E}''$.
**Example:** As an example, consider the following traces:

$$\mathscr{E}_1 = \langle d_1@n_1, d_2@n_2, d_3@n_1, d_4@n_2\rangle,$$

$$\mathscr{E}_2 = \langle d_1@n_1, d_2@n_2, d_4@n_2, d_3@n_1 \rangle,$$

$$\mathscr{E}_3 = \langle d_1@n_1, d_2@n_2, d_3@n_1 \rangle.$$

It is easy to observe that $\mathscr{E}_1$ and $\mathscr{E}_2$ are equivalent, since $\mathscr{E}_1|n_1 = \mathscr{E}_2|n_1 = \langle d_1@n_1, d_3@n_1\rangle$ and $\mathscr{E}_1|n_2 = \mathscr{E}_2|n_2 = \langle d_2@n_2, d_4@n_2\rangle$. In contrast, $\mathscr{E}_3$ is a subtrace of $\mathscr{E}_1$, but it is not equivalent to $\mathscr{E}_1$ (since $\mathscr{E}_3|n_2 \neq \mathscr{E}_1|n_2$).

## C.1.2 Graph construction

We now describe our algorithm for constructing the temporal provenance that explains the reasons for a delay between two events. As discussed in Section 5.4.6, temporal provenance is *recursive* – the temporal provenance for $[e', e]$ includes, as subgraphs, the temporal provenances of all events that contributed to both $e$ and the delay from $e'$ to $e$. Leveraging this property, we can construct the temporal provenance of a pair of events "on demand" using a top-down procedure, without the need to materialize the entire provenance graph.

Towards this, we first define a function RAWQUERY that, when called on a vertex $v$ in the temporal provenance graph, returns two sets of immediate children of $v$: the first consists of vertexes that are connected to $v$ via causal edges, and the second consists of vertexes that are connected to $v$ via sequencing edges. Given an execution trace $\mathscr{E}$ of the system, the temporal provenance for a diagnostic query T-QUERY($e'$,$e$) can be obtained by first constructing a vertex $v_e$ that describes $e$ (i.e., a DRV/UDRV/RCV vertex for $e$), and then calling RAWQUERY recursively on the vertexes starting from $v_e$ until reaching the leaf vertexes (lines 1-15); note that a vertex returned by a RAWQUERY call is connected to its parent vertex via either a causal edge and/or a sequencing edge, depending on the set(s) it belongs to (lines 11-14). The resulting graph, denoted by $G(e', e, \mathscr{E})$, includes all necessary events to explain both $e$ and the delay from $e'$ to $e$. However, as it requires delay annotation (Sections 5.4.3–5.4.5) to be useful for diagnostics, we refer to it as the "raw" temporal provenance of T-QUERY($e'$,$e$).

```
function CONSTRUCT-GRAPH($v_e$)
    $G \leftarrow \{v_e\}$ // the "raw" temporal provenance graph
    $NodeToProcess \leftarrow \{v_e\}$ // a queue of vertexes that need explanation
    while $NodeToProcess \neq \emptyset$ do
        $v \leftarrow NodeToProcess$.POP()
        $S, S' \leftarrow$ RAWQUERY($v$)
        for $v' \in \{S \cup S'\}$ do
            if $v' \notin G$ then
                $G \leftarrow G \cup v'$ // add vertexes
                $NodeToProcess$.PUSH($v'$)
        for $v' \in S$ do
            $G \leftarrow G \cup (v', v)_{causal}$ // add causal edges
        for $v' \in S'$ do
            $G \leftarrow G \cup (v', v)_{sequencing}$ // add sequencing edges
    RETURN $G$
function RAWQUERY(DRV($[t_s, t_e]$,N,$\tau$, $\tau$:- $\tau_1, \tau_2, \ldots, \tau_m$))
    S $\leftarrow \emptyset$
    $t_e^{max} \leftarrow 0$ // the last precondition was satisfied at $t_e^{max}$
    for $\tau_i \in \{\tau_1, \tau_2, \ldots, \tau_m\}$ do
        Find $d_i@N = (\tau', r, t'_s, t'_e, \{c_1, c_2, \ldots, c_k\}, \pm\tau_i) \in \mathscr{E}$: $t'_e \leq t_s$ and $t'_e$ is maximized
        $t_e^{max} \leftarrow$ MAX($t_e^{max}$, $t'_e$)
        if $r = r_{\mathsf{ins}}$ then
            S $\leftarrow$ S $\cup$ INS($[t'_s, t'_e]$,$N$,$\tau_i$)
        else if $r = r_{\mathsf{rcv}}$ then
            S $\leftarrow$ S $\cup$ RCV($[t'_s, t'_e]$,$N \leftarrow r.N$,$\pm\tau_i$)
        else
            S $\leftarrow$ S $\cup$ DRV($[t'_s, t'_e]$,$N$,$\tau_i$, $\tau_i$:-$\tau'$,$c_1, c_2, \ldots, c_k$)
    // include all preceding events that happened after the last
    // precondition was satified and before the derivation of $\tau$
    RETURN (S; PREV-VERTEX($[t_e^{max}, t_s]$,$N$))
function RAWQUERY(INS($[t_s, t_e]$,$N$,$\tau$))
    RETURN ($\emptyset$; $\emptyset$)
function RAWQUERY(SND($[t_s, t_e]$,$N \rightarrow N'$,$\pm\tau$))
    Find $d@N = (\tau', r, t'_s, t'_e, \{c_1, c_2, \ldots, c_k\}, \pm\tau) \in \mathscr{E}$: $t'_e \leq t_s$ and $r \neq r_{\mathsf{rcv}}$ and $t'_e$ is maximized.
    if $r = r_{\mathsf{ins/del}}$ then
        RETURN (INS/DEL($[t'_s, t'_e]$,$N$,$\tau$); PREV-VERTEX($[t'_e, t_s]$,$N$))
    else
        RETURN (DRV/UDRV($[t'_s, t'_e]$,$N$,$\tau$, $\tau$:- $\tau'$,$c_1, c_2, \ldots$); PREV-VERTEX($[t'_e, t_s]$,$N$))
```

Figure C.1: Algorithm for constructing temporal provenance graph for a given execution trace $\mathscr{E}$ (part 1). The trace $\mathscr{E}$ consists of events (Definition 9), which are recorded at runtime or reconstructed via replay. The function RAWQUERY($v$), when called on a vertex $v$, returns two sets of immediate children of $v$, which are connected to $v$ via causal edges and sequencing edges, respectively. It calls PREV-VERTEX($[t', t]$,$N$) as a subprocedure, which finds a chain of vertexes connected via sequencing edges that immediately precedes $v$ during $[t', t]$.

```
39: function RAWQUERY(RCV($[t_s,t_e]$,$N \leftarrow N'$,$\pm\tau$))
40:     Find $d@N'$=($\tau'$,$r$,$t'_s$,$t'_e$,$\pm\tau$) $\in \mathscr{E}$: $t'_e \le t_s$ and $r = r_{\mathsf{snd}}$ and $t'_e$ is maximized
41:     $v \leftarrow$ SND($[t'_s,t'_e]$,$N' \rightarrow N$,$\pm\tau$)
42:     // a remote sequencing edge exists from the SND vertex
43:     RETURN $(v;\ v)$
44: function PREV-VERTEX($[t',t]$,$N$)
45:     // add a sequencing edge from an immediate preceding event, if one exists
46:     if $t' < t$ and $\exists\ d@N$=($\tau'$,$r$,$t_s$,$t_e$,$\{c_1,c_2,...\}$,$\pm\tau$): $t_e = t$ then
47:         $v \leftarrow null$
48:         if $r = r_{\mathsf{snd}}$ then
49:             $v \leftarrow$ SND($[t_s,t_e]$,$N \rightarrow r.N$,$\pm\tau$)
50:         else if $r = r_{\mathsf{rcv}}$ then
51:             $v \leftarrow$ RCV($[t_s,t_e]$,$N \leftarrow r.N$,$\pm\tau$)
52:         else if $r = r_{\mathsf{ins/del}}$ then
53:             $v \leftarrow$ INS/DEL($[t_s,t_e]$,$N$,$\tau$)
54:         else
55:             $v \leftarrow$ DRV/UDRV($[t_s,t_e]$,$N$,$\tau$,$\tau$:-$\tau'$,$c_1$,$c_2$,...)
56:         $v'$ = PREV-VERTEX($[t',t_s]$,$N$)
57:         if $v' != null$ then
58:             // recusively add preceding events until the entire $[t',t]$ interval is explained
59:             $G \leftarrow G \cup (v',v)_{sequencing}$
60:         RETURN $v$
61:     RETURN $null$
62: function RAWQUERY(UDRV($[t_s,t_e]$,N,$\tau$, $\tau$:- $\tau_1,\tau_2,\ldots,\tau_m$))
63:     S $\leftarrow \emptyset$
64:     $t_e^{max} \leftarrow 0$ // the last precondition was satisfied at $t_e^{max}$
65:     for $\tau_i \in \{\tau_1,\tau_2,\ldots,\tau_m\}$ do
66:         Find $d_i@N$ = ($\tau'$, $r$, $t'_s$, $t'_e$, $\{c_1,c_2,\ldots,c_k\}$, $\pm\tau_i$) $\in \mathscr{E}$: $t'_e \le t_s$ and $t'_e$ is maximized
67:         $t_e^{max} \leftarrow$ MAX($t_e^{max}$, $t'_e$)
68:         if $r = r_{\mathsf{ins/del}}$ then
69:             S $\leftarrow$ S $\cup$ INS/DEL($[t'_s,t'_e]$,$N$,$\tau_i$)
70:         else if $r = r_{\mathsf{rcv}}$ then
71:             S $\leftarrow$ S $\cup$ RCV($[t'_s,t'_e]$,$N \leftarrow r.N$,$\pm\tau_i$)
72:         else
73:             S $\leftarrow$ S $\cup$ DRV/UDRV($[t'_s,t'_e]$,$N$,$\tau_i$, $\tau_i$:-$\tau'$,$c_1$,$c_2,\ldots,c_k$)
74:     RETURN (S; PREV-VERTEX($[t_e^{max},t_s]$,$N$))
75: function RAWQUERY(DEL($[t_s,t_e]$,$N$,$\tau$))
76:     RETURN $(\emptyset;\ \emptyset)$
```

Figure C.2: Algorithm for constructing temporal provenance graph for a given execution trace $\mathscr{E}$ (part 2). The trace $\mathscr{E}$ consists of events (Definition 9), which are recorded at runtime or reconstructed via replay. The function RAWQUERY($v$), when called on a vertex $v$, returns two sets of immediate children of $v$, which are connected to $v$ via causal edges and sequencing edges, respectively. It calls PREV-VERTEX($[t',t]$,$N$) as a subprocedure, which finds a chain of vertexes connected via sequencing edges that immediately precedes $v$ during $[t',t]$.

The RAWQUERY($v$) procedures rely on a helper function called PREV-VERTEX to find vertexes that are connected to $v$ via sequencing edges. For ease of exposition, we first explain the pseudo-code of PREV-VERTEX in Figures C.1-C.2: given an interval $[t',t]$ and a node $N$ (supplied by RAWQUERY calls), PREV-VERTEX finds the chain of preceding events that happened on $N$ during $[t',t]$; it first locates the last event $v$ whose execution ends at $t$ and constructs a corresponding vertex (lines 46-55); it then shortens the interval to until the starting timestamp of $v$ and recursively find prior events on $N$ (line 56); it stops until the interval is exhausted or if no event can be found (line 61); finally, it recursively connects this chain of events using sequencing edges and returns the last event in the chain to its caller (line 59-60). For example, consider the provenance graph from Figure C.3: a PREV-VERTEX($[2.5s,3.5s]$,$Y$) call will first find the DRV(F) event, which ends at exactly $t = 3.5s$; it constructs a vertex and shortens the interval to $[2.5s,2.5s]$, by excluding the execution time of DRV(F); this interval is passed into a recursive call – PREV-VERTEX($[2.5s,2.5s]$,$Y$) – that finds the event of and constructs a vertex for INS(G); the recursion then stops because the interval becomes empty (because INS(G) takes a positive amount of time); the two constructed vertexes are connected via sequencing edges and the last event in the chain – DRV(F) – is returned to the caller.

Figures C.1-C.2 show the pseudo-code for the function RAWQUERY($v$) depending on the type of $v$ (DRV, UDRV, SND, RCV, INS and DEL). Note that each DRV or UDRV vertex is also associated with the corresponding derivation rule. Next, we explained the pseudo-code of RAWQUERY($v$) for each vertex type in more detail. We use the provenance graph from Figure C.3 as an example:

- To explain a SND vertex, we find the most recent event in the original trace that produced (or deleted) the tuple that is being sent (line 34), construct an INS (or DEL) or a DRV (or UDRV) vertex for the found event, and add an incoming causal edge from the constructed vertex (lines 35-38); in addition, a SND vertex has an incoming sequencing edge from the chain of preceding events that

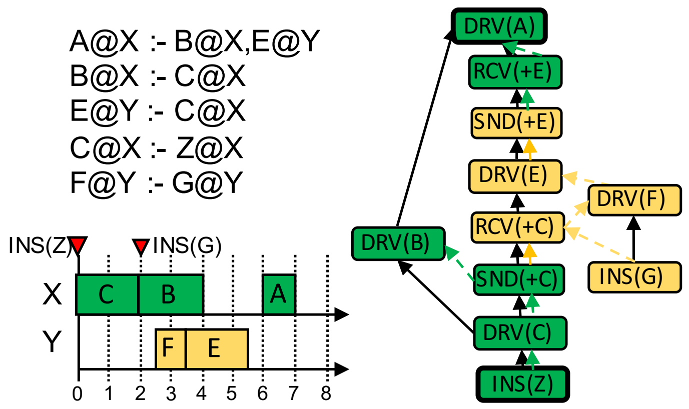


Figure C.3: An example scenario, with NDlog rules at the top left, the timing of a concrete execution in the bottom left, and the temporal provenance graph at the right. The query is T-QUERY(INS(Z), DRV(A)); the start and end vertexes are marked in bold. Vertex names are shortened and some fields are omitted for clarity.

happened after the message was produced or deleted (the PREV-VERTEX calls in lines 36 and line 38). For example, in the temporal provenance graph from Figure C.3, the SND(+C) vertex has a causal edge from the DRV(C) vertex, because DRV(C) functionally triggered SND(+C); in addition, a PREV-VERTEX($[2s, 2s]$,$X$) call adds a sequencing edge from DRV(C) to SND(+C), as the former event directly precedes the latter event.

- A RCV has an incoming causal edge and an incoming remote sequencing edge from the SND vertex for the received message (lines 39-43). This is the case for RCV(+E) and RCV(+C) in Figure C.3.
- A DRV vertex for a rule `A:-B,C,D` has an incoming causal edge for each precondition (B, C, and D) that leads to the vertex that produced the corresponding tuple (line 19); this can be an INS, a RCV, or another DRV (lines 22-27); in addition, a DRV vertex has an incoming sequencing edge from the chain of preceding events that happened after the last precondition was satisfied (the PREV-VERTEX call in line 30). For example, in the provenance from Figure C.3, DRV(F) has a causal edge from its (only) precondition INS(G); in addition, a PREV-VERTEX($[2.5s, 3.5s]$,$Y$) finds the preceding event RCV(+C) that occurred between INS(G) ended and DRV(F) started.

- An INS vertex corresponds to the insertion of a base tuple, which cannot be explained further; thus, it has no incoming edges (line 32). This is true for INS(Z) and INS(G) in Figure C.3.
- The edges for the negative "twins" of these vertexes – UDRV and DEL – are analogous to their positive counterparts.

### C.1.3 Properties

Given the "raw" temporal provenance $G(e', e, \mathscr{E})$ of a diagnostic query T-QUERY($e'$,$e$) in an execution trace $\mathscr{E}$, we say that $G(e', e, \mathscr{E})$ is correct if it is possible to extract a subtrace from G that has the properties of validity, soundness, completeness, and minimality. We first describe our algorithm for extracting such a subtrace, and then formally define these properties and their proofs.

**Definition 13. (Trace Extraction):** *Given a temporal provenance $G(e', e, \mathscr{E})$, the trace $A(e', e, \mathscr{E})$ is extracted by running Algorithm 2 based on topological sort.*

Algorithm 2 converts the vertexes in the provenance graph to events and then uses a topological ordering and timestamps to assemble the events into a trace. In particular, Line 12-29 implements the construction of one individual event, where the information of a rule evaluation (such as triggering event, conditions, and action) is extracted from vertexes in $G(e', e, \mathscr{E})$: a DRV/UDRV/SND vertex and their children; a pair of RCV and SND vertexes; or a INS/DEL vertex. In the algorithm, $\mathsf{type}(v)$, $\mathsf{tuple}(v)$, $\mathsf{rule}(v)$, $\mathsf{startTime}(v)$ and $\mathsf{endTime}(v)$ denote the vertex type, the tuple, the derivation rule, the start timestamp, and the end timestamp of the vertex $v$, respectively. For example, Algorithm 2 extracts the following trace from the provenance graph in Figure C.3: $\langle$INS(Z)@$X$, DRV(C)@$X$, SND(+C)@$X$, INS(G)@$Y$, DRV(B)@$X$, RCV(+C)@$Y$, DRV(F)@$Y$, DRV(E)@$Y$, SND(+E)@$Y$, RCV(+E)@$X$, DRV(A)@$X\rangle$.

We will show that the extracted trace $\mathscr{A}(e', e, \mathscr{E})$ obtained from Algorithm 2 satisfies the following four properties.

**Algorithm 2** Extracting traces from provenance

```
1: // this algorithm extracts the trace $\mathscr{A}(e', e, \mathscr{E})$ from the "raw" temporal provenance $G(e', e, \mathscr{E})$; for
   ease of explanation, we rewrite $G$ as $(V, E)$, where $V$ represents vertexes and $E$ represent edges
2: function EXTRACT-TRACE()($G = (V, E)$)
3:     // calculate the out-degree of every vertex in $G$
4:     for all $v \in V$ do $degree(v) \leftarrow 0$
5:     for all $e = (v, v') \in E$ do $degree(v)$++
6:     $trace \leftarrow \emptyset$
7:     $NodeToProcess \leftarrow V$
8:     while $NodeToProcess \neq \emptyset$ do
9:         // select the next event based on topological ordering and timestamps
10:        select $v \in NodeToProcess : degree(v) = 0$ and $\nexists v'$
               that is located on the same node and has a larger end timestamp
11:        $NodeToProcess$.REMOVE($v$)
12:        if type($v$) = DRV or UDRV or SND then
13:            $preconditions \leftarrow \emptyset$
14:            for $\forall$ $(v', v) \in E$ s.t. $(v', v)$ is a causal edge do
15:                $preconditions$.ADD(tuple($v'$)) // tuple($v'$) is a precondition
16:            // find the trigger from the preconditions
17:            $trigger \leftarrow \tau' \in preconditions$: (a) $\tau'$ is a message, or (b) $\tau'$ is a state and
                   $\nexists \tau'' \in preconditions$ that has a larger end timestamp
18:            $preconditions$.DELETE($trigger$)
19:            $event \leftarrow (trigger, \text{rule}(v), \text{startTime}(v), \text{endTime}(v), preconditions, \text{tuple}(v))$
20:            $trace.push\_front(event)$
21:        if type($v$) = RCV then
22:            $output \leftarrow tuple(v)$
23:            $trigger \leftarrow tuple(v')$: $(v', v) \in E$ s.t. type($v'$) = SND
24:            $event \leftarrow (trigger, \text{rule}(v), \text{startTime}(v), \text{endTime}(v), \emptyset, \text{tuple}(v))$
25:            $trace.push\_front(event)$
26:        if type($v$) = INS or DEL then
27:            $output \leftarrow tuple(v)$
28:            $event \leftarrow (\emptyset, \text{rule}(v), \text{startTime}(v), \text{endTime}(v), \emptyset, \text{tuple}(v))$
29:            $trace.push\_front(event)$
30:        for all $(v', v) \in E$, $degree(v') \leftarrow degree(v') - 1$
31:    return $trace$
```

**Definition 14. (Soundness)**: *A subtrace $\mathscr{A}$ extracted from $G(e',e,\mathscr{E})$ is sound if and only if it is a subtrace of some trace $\mathscr{E}'$ that is equivalent to $\mathscr{E}$, i.e., $\mathscr{A} \subseteq \mathscr{E}' \sim \mathscr{E}$.*

Intuitively, the soundness property means that $\mathscr{A}(e',e,\mathscr{E})$ must preserve all the happens-before relationships among events and the exact timestamps of events in the original execution trace obtained from running the NDlog program. Ideally, we would like $\mathscr{A}(e',e,\mathscr{E})$ to be a subtrace of $\mathscr{E}$, but without synchronized clocks, we cannot always order concurrent events on different nodes. However, for practical purposes $\mathscr{E}$ and $\mathscr{E}_0$ are indistinguishable: each node observes the same sequence of events in the same order and at the same times.

**Definition 15. (Completeness)**: *A subtrace $\mathscr{A}$ extracted from $G(e',e,\mathscr{E})$ is complete if and only if it ends with the event e and e happens at the same time as in $\mathscr{E}$.*

Intuitively, completeness means that $\mathscr{A}(e',e,\mathscr{E})$ must include all events necessary to reproduce $e$ both functionally and temporally. Note that the validity property already requires that any event that is needed for $e$ be included in $\mathscr{A}(e',e,\mathscr{E})$; hence, we can simply verify the completeness property of a valid trace by checking whether it ends with $e$.

**Definition 16. (Validity)**: *A subtrace $\mathscr{A}$ extracted from $G(e',e,\mathscr{E})$ is valid if and only if, given the initial state $S_0$, for every event $d_i@N_i = (\tau_i, r_i, t_i, t_i', c_i, \pm\tau_i') \in \mathscr{A}$:*

- *(a) there exists $d_j@N_j = (\tau_j, r_j, t_j, t_j', c_j, \pm\tau_j')$ that precedes $d_i@N_i$ in $\mathscr{A}$ such that $\tau_i = \tau_j'$;*
- *(b) for all $\tau_k \in c_i$, we have $\tau_k \in S_{i-1}$, where $\mathscr{S}_0 \xrightarrow{d_1@n_1} \mathscr{S}_1 \xrightarrow{d_2@n_2} \cdots \xrightarrow{d_{i-1}@n_{i-1}} \mathscr{S}_{i-1}$;*
- *(c) based on the conditions (a) and (b), consider the set of all events $P_i$ such that $d_k@N_k \in P_i$ generates $\tau_k \in (c_i \cup \tau_i)$; denote $d_j@N_j$ as the latest event in $P_i$; if $N_j = N_i$ and $t_j' < t_i$, there must exist a set of events $\{d_p^1@N_i, ..., d_p^n@N_i\} \in \mathscr{A}$ such that: $t_j' = t_p^1$; $t'^m_p = t_p^{m+1}, 1 \le m < n$; and $t'^n_p = t_i$.*

Intuitively, the validity property means that $\mathscr{A}(e',e,\mathscr{E})$ must correspond to a correct execution of the NDlog program both in terms of functionality and timing. Condition $(a)$ states that any event that triggers a rule evaluation must be generated before the rule is evaluated. Condition ($b$) states that the preconditions of the rule evaluation must hold at the time of the rule evaluation. Finally, condition ($c$) requires that the evaluation is "work-conserving": the node cannot be idle when it is ready to compute a derivation.

**Definition 17. (Minimality)**: *A subtrace $\mathscr{A}$ extracted from $G(e',e,\mathscr{E})$ is minimal iff there exists no trace $\mathscr{E}'$ such that: (a) there $\exists d_i@N_i$ where $d_i@N_i \in \mathscr{A}$ and $d_i@N_i \notin \mathscr{E}'$; (b) $\mathscr{E}'$ is valid, sound, and complete.*

Intuitively, minimality means that $\mathscr{A}(e',e,\mathscr{E})$ should not contain any events that are not necessary to reproduce $e$. If this property were omitted, $\mathscr{A}(e',e,\mathscr{E})$ could trivially output the complete trace $\mathscr{E}$.

### C.1.4 Proofs

**Lemma 7.** *For any execution $\mathscr{E}$, and a temporal provenance query* T-QUERY*($e'$,$e$), the provenance graph $G(e',e,\mathscr{E})$ is acyclic.*

**Proof.** We first show that if there exists a cycle in $G(e',e,\mathscr{E})$, the cycle cannot include two vertexes located on different nodes. Suppose there exists a cycle that contains two vertexes $v_1$ and $v_2$ located on $N_1$ and $N_2$ respectively. Then the cycle must contain a least one pair of SND and RCV vertexes in both the path from $v_1$ to $v_2$, and the path from $v_2$ and $v_1$. Each SND and RCV corresponds to a message communication which takes a positive amount of time. Therefore, traversing from $v_1$ along the cycle back to $v_1$ results in an absolute increment in the timestamp. This is a contradiction.

If all the vertexes in the cycle are located on the same node, then we can order the vertexes according to their associated timestamps (now all the timestamps are with respect to the same local clock). Such order corresponds to the precedence of events in the execution. As time always progresses forward, such cycle cannot exist in $G(e',e,\mathscr{E})$. $\square$

**Theorem 8.** *$A(e', e, \mathscr{E})$ is sound.*

**Proof.** We show that a) all the events in $\mathscr{A}(e', e, \mathscr{E})$ also appear in $\mathscr{E}$ at the same time (and thus in any $\mathscr{E}_0 \sim \mathscr{E}$), and b) the local event ordering pertains on each node, that is, for any two events $d_1@N_i$ and $d_2@N_i$ in $\mathscr{A}(e', e, \mathscr{E})$ that are located on the same node $N_i$, $d_1@N_i$ precedes $d_2@N_i$ in $\mathscr{A}(e', e, \mathscr{E})$ iff $d_1@N_i$ precedes $d_2@N_i$ in $\mathscr{E}$.

**Condition a**. We perform a case analysis by considering the type of the root vertex of $G(e', e, \mathscr{E})$:

- DRV. According to Algorithm 2 (lines 12-20), an event $d_i@N_i$ is generated and included in $\mathscr{A}(e', e, \mathscr{E})$ for each DRV vertex (and its children) in the provenance graph $G(e', e, \mathscr{E})$. However, by construction, each DRV vertex $v$ corresponds to an rule evaluation in $\mathscr{E}$. In our model, the rule evaluation is modeled as an event $d_j@N_j = (\tau_j, r_j, t_j, t'_j, \{c_j^1, ..., c_j^p\}, \pm\tau'_j)$, where $\tau_j$ is the trigger event, $r_j$ and $[t_j, t'_j]$ are the rule used in and the time interval of the rule evaluation, $c_j^k$ represents preconditions, and $\pm\tau'_j$ is the generated update. We need to show that $d_i@N_i$ is identical to $d_j@N_j$. This follows straightforwardly from the construction of $G(e', e, \mathscr{E})$: The RAWQUERY($v$) procedures generate a DRV vertex $v$ by: (a) find a derivation event $d_j@N_j$ from $\mathscr{E}$, (b) add incoming edges from the trigger event (a vertex for $\tau_j$), and (c) add incoming edges from the preconditions (vertexes for $\{c_j^1, ..., c_j^p\}$). Algorithm 2 reverses this process and generates event $d_i@N_i$ from these information, which is extracted from $d_j@N_j$, and therefore $d_i@N_i = d_j@N_j$.
- RCV/INS/DEL/UDRV/SND Following the same argument for the DRV case above, we can prove that condition (a) holds.

**Condition b**. According to Algorithm 2 (specifically, Line 10), $d_1@N_i$ precedes $d_2@N_i$ in $\mathscr{A}(e', e, \mathscr{E})$, iff $d_2@N_i$ has a larger timestamp than $d_1@N_i$. However, $d_2@N_i$ is assigned a larger timestamp iff $d_1@N_i$ precedes $d_2@N_i$ in the actual execution $\mathscr{E}$. Note that events on different nodes may be reordered in $\mathscr{A}(e', e, \mathscr{E})$, but this is captured by the equivalence ($\sim$) relation. $\square$

**Theorem 9.** *$A(e', e, \mathscr{E})$ is complete.*

**Proof.** We need to show that a) $\mathscr{A}(e', e, \mathscr{E})$ contains an event $d_i@N_i$ that generates $e$ at the same time as in $\mathscr{E}$, and b) $d_i@N_i$ is the last event in $\mathscr{A}(e', e, \mathscr{E})$.

**Condition a**. By construction, the vertex for $e$ has incoming edges from vertexes representing the triggering event $\tau$ and all preconditions $c_1, ..., c_p$ (if any). Algorithm 2 (specifically, Lines 12-25) construct an event $(\tau, r, t, t', c, \pm e)$, where r and $[t, t']$ are the rule name and time interval encoded in the vertex. Note that the tuple $\tau$ as well as the timestamps $t$ and $t'$ are exactly the ones that are extracted from $\mathscr{E}$ (Figures C.1-C.2).

**Condition b**. We have proved that some event $d_i@N_i$ that generates $e$ must exist in $\mathscr{A}(e', e, \mathscr{E})$, we next show that $d_i@N_i$ is the last event in $\mathscr{A}(e', e, \mathscr{E})$. The provenance graph $G(e', e, \mathscr{E})$ is rooted by a vertex that corresponds to $e$. Since all other vertexes in $G(e', e, \mathscr{E})$ have a directed path to the root vertex, the corresponding events must all be ordered before $d_i@N_i$, so $d_i@N_i$ must be the last event in the subtrace. $\square$

**Theorem 10.** *$A(e', e, \mathscr{E})$ is valid.*

**Proof.** Lemma 7 shows that any provenance graph $G(e', e, \mathscr{E})$ is acyclic, and thus $G(e', e, \mathscr{E})$ has a well-defined height: the length of the longest path from any leaf to $e$. We prove validity using structural induction on the height of the provenance graph $G(e', e, \mathscr{E})$.

**Base case**: The height of $G(e', e, \mathscr{E})$ is one. In this case, $e$ must be an insertion or deletion of a base tuple; $G(e', e, \mathscr{E})$ contains a single INS (or DEL) vertex that corresponds to the update of the base tuple. Therefore, $\mathscr{A}(e', e, \mathscr{E})$ consists of a single event and is trivially valid, because the event has neither a trigger nor any precondition (Algorithm 2 lines 26–29).

**Induction case**: Suppose the validity of the extracted trace $\mathscr{A}(e', e, \mathscr{E})$ holds for any provenance graph with height less than $k$ ($k \geq 1$). Consider a provenance graph $G(e', e, \mathscr{E})$ with height $k+1$. We perform a case analysis by considering the type of the root vertex of $G(e', e, \mathscr{E})$. For every event $d_i@N_i = (\tau_i, r_i, t_i, t_i', c_i, \pm\tau_i') \in \mathscr{A}(e', e, \mathscr{E})$, we prove that the three conditions in Definition 16 hold.

- DRV. We know that, by construction, the DRV vertex has an incoming edge from vertexes representing the triggering event $\tau$ and all preconditions $c_1, ..., c_p$. By the induction hypothesis, Algorithm 2 outputs a valid trace $d_1@N_1, ..., d_j@N_j$ for the subgraph for the trigger event $\tau$, where $d_j@N_j$ corresponds to the generation of $\tau$ (following the completeness property proved in Theorem 9). Because of the nature of Algorithm 2 (which is based on topological sort), $d_j@N_j$ must be ordered before $d_i@N_i$, which satisfies condition (a) in the definition of validity. For example, in the provenance graph from Figure C.3, the trigger event INS(G) must precede the derived event DRV(F) in the extracted trace, because a causal edge exists from the former to the latter. Similarly, valid traces are generated for the updates that support the preconditions $c_1, ..., c_p$, which satisfies conditions (b). Condition (c) holds by construction: the original execution trace $\mathscr{E}$ is valid and must include a set of events $\{d_p^1@N_i, ..., d_p^n@N_i\}$ that satisfies condition (c); the PREV-VERTEX call in Figures C.1-C.2 finds all these events because the call recursively find such events from $\mathscr{E}$ until the interval between the end of $d_j@N_j$ and the start of $d_i@N_i$ is fully exhausted (line 46); therefore, all events in $\{d_p^1@N_i, ..., d_p^n@N_i\}$ will be represented by vertexes in the temporal provenance; the extraction algorithm merely reverses this process and reconstructs each of $\{d_p^1@N_i, ..., d_p^n@N_i\}$, while preserving their ordering and timestamps (following the soundness property proved in Theorem 8). Therefore, the extracted trace $\mathscr{A}(e', e, \mathscr{E})$ is valid. For example, consider the DRV(F) event in the provenance from Figure C.3: there is a gap of $[2.5s, 3.5s]$ between when its last precondition INS(G) completed and when its own derivation started; in the original execution, node $Y$ must be busy during the gap, because it is work-conserving; in this case, $Y$ was busy with handling RCV(+C); while constructing the vertex for DRV(F), the RAWQUERY procedure calls PREV-VERTEX([$2.5s, 3.5s$],$Y$), which finds the RCV(+C) event from the original trace and added a vertex to $G$; Algorithm 2 extracts events from $G$ based on topological ordering, therefore, RCV(+C) will present in $\mathscr{A}$, after INS(G) and before DRV(F).

- RCV. We know that, by construction, the RCV vertex has an incoming edge from a SND vertex with the same tuple $\tau$. By the induction hypothesis, Algorithm 2 outputs a valid trace $d_1@N_1, \ldots, d_j@N_j$ for the subgraph rooted at the SND vertex, where $d_j@N_j$ corresponds to the generation of $\tau$ (following the completeness property proved in Theorem 9). Because of the nature of Algorithm 2 (which is based on topological sort), $d_j@N_j$ must be ordered before $d_i@N_i$, which satisfies condition (a) in the definition of validity. A SND vertex have no preconditions, consequently, conditions (b) holds trivially. $d_i@N_i$ and $d_j@N_j$ happened on different nodes, which satisfies condition (c) trivially. Therefore, the extracted trace $\mathscr{A}(e', e, \mathscr{E})$ is valid.
- UDRV/SND. Following the same argument for the DRV case above, we can prove that the extracted trace $\mathscr{A}(e', e, \mathscr{E})$ is valid.
- INS/DEL. This case cannot occur because INS and DEL have no preconditions, so the tree would have to have a height of one.

□

**Theorem 11.** *$A(e', e, \mathscr{E})$ is minimal.*

**Proof.** We prove the minimality property by induction on the syntactic structure of $\mathscr{A}(e', e, \mathscr{E})$: we show that an event $d_i@N_i \in \mathscr{A}(e', e, \mathscr{E})$ cannot be removed because it is necessary for some event $d_j@N_j$ appeared later in the trace. For presentation purposes, we suppose $\mathscr{A}(e', e, \mathscr{E}) = d_1@N_1, \ldots, d_m@N_m$.

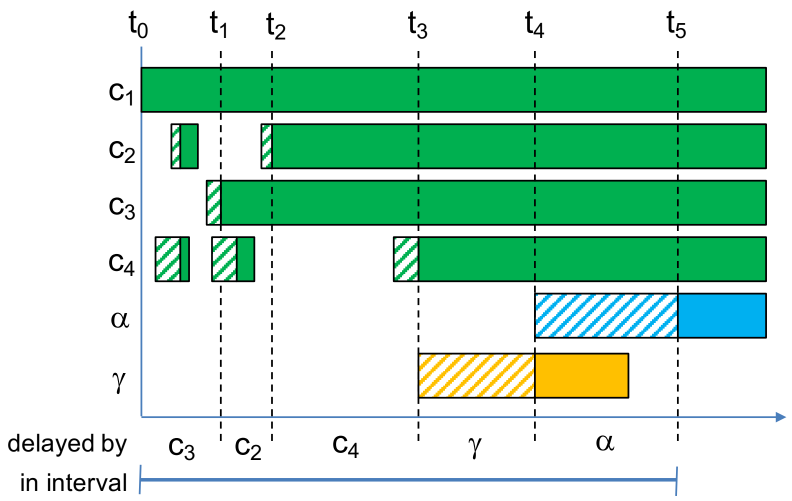


Figure C.4: Illustration for the definition of direct and transitive delay. Shaded boxes represent intervals where a tuple was being derived, and solid boxes represent intervals where the tuple existed. The derivation is $\alpha : -c_1, c_2, c_3, c_4$, and the interval in question is $[t_0, t_5]$; $\gamma$ is an unrelated tuple whose derivation just happened to be sequenced before that of $\alpha$.

**Base case.** According to the completeness property (Theorem 9), the last event $d_m@N_m$ in $\mathscr{A}(e', e, \mathscr{E})$ generates $e$. Therefore the base case trivially holds, as the removal of $d_m@N_m$ breaks the completeness property.

**Induction case.** Suppose the last $k$ events $d_{m-k+1}@N_{m-k+1}, \ldots, d_m@N_m (K >= 1)$ cannot be remove. We show that event $d_{m-k}@N_{m-k}$ cannot be removed as well: According to Algorithm 2, $d_{m-k}@N_{m-k}$ is constructed from a vertex $v$. $v$ must have an outgoing edge to some other vertex in $G(e', e, \mathscr{E})$. Otherwise, $v$ would not be included in $G(e', e, \mathscr{E})$ which is a subgraph rooted by $e$. Consider $u$ as the first vertex on the path from $v$ to the root of $G(e', e, \mathscr{E})$. According to Algorithm 2, an event $d_j@N_j$ is constructed from $u$ and its children (if any). Given the edge from $v$ to $u$, we know that $d_j@N_j$ depends on $d_{m-k}@N_{m-k}$, and that $d_{m-k}@N_{m-k}$ precedes $d_j@N_j$. By applying the induction hypothesis ($d_j@N_j$ cannot be removed from $\mathscr{A}(e', e, \mathscr{E})$), we can conclude that $d_{m-k}@N_{m-k}$ also cannot be removed. □

### C.1.5 Delay annotations

In this section, we show that each vertex is annotated with the delay that it contributed. We first define what it means for a derivation to be directly "delayed" by one of its preconditions (Definition 18), and then recursively extends this definition to transitive delays (Definition 19). We continue by discussing several properties of the annotations computed by the algorithm from Figure 5.6 (Definition 20, Lemmas 12-15). This allows us to further prove the first theorem which states that the algorithm from Figure 5.6 labels each vertex with the amount of delay that is contributed by the subtree that is rooted at that vertex (Theorem 16).

**Definition 18. (Direct delay):** *Consider a derivation $\alpha : -c_1, c_2, \ldots, c_k$ and an interval $[t_0, t_5]$, such that $\alpha$ begins its derivation at $t_4 < t_5$ and finishes it at time $t_5$. We say that a precondition $c_i$ directly delays the derivation of $\alpha$ during an interval $[t_x, t_y]$, $t_0 \leq t_x$, $t_y \leq t_4$, iff*

- *(a) $c_i$ became true at $t_y$ and remain true until $t_4$ (and was false before $t_y$); and*
- *(b) there either was some $c_j$, $i \neq j$, that delayed the derivation of $\alpha$ during some interval $[x, t_x)$; or there was no such $c_j$, and $t_x = t_0$.*

*For convenience, we say that $\alpha$ itself delays its own derivation during $[t_4, t_5]$ . Find the time $t_3 \leq t_4$ such that $t_3$ is the earliest time when all preconditions were true (and remained true until $t4$). If a tuple $\gamma$ resides on the same node as $\alpha$ and the derivation of $\gamma$ happened during $[t_x, t_y] \subseteq [t_3, t_4]$, we also say that $\gamma$ directly delays the derivation of $\alpha$.*

Figure C.4 contains a brief illustration. $c_3$ directly delays the derivation of $\alpha$ during the interval $[t_0, t_1]$, because: (a) $c_3$ became true at $t_1$ and remained true until $t_4$; (b) $t_0$ was the start of the interval in question (the second case of condition (b)). $c_2$ directly delays the derivation of $\alpha$ during the interval $[t_1, t_2]$, because: (a) $c_2$ became true at $t_2$ and remained true until $t_4$; (b) $c_3$ delayed the derivation of $\alpha$ during $[t_0, t_1]$ (the first case of condition (b)). Similarly, $c_4$ delays the derivation of $\alpha$ during the

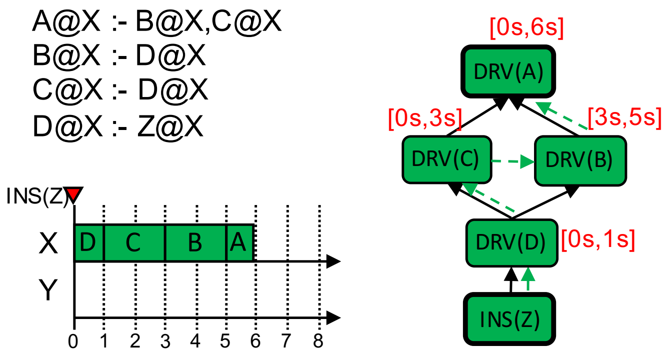


Figure C.5: An example scenario, with NDlog rules at the top left, the timing of a concrete execution in the bottom left, and the resulting temporal provenance at the right. The query is T-QUERY(INS(Z), DRV(A)); the start and end vertexes are marked in bold. Vertex names have been shortened and some fields have been omitted for clarity. Each vertex is annotated with its annotation interval (Definition 20).

interval $[t_2,t_3]$. $\gamma$ delays the derivation of $\alpha$ during the interval $[t_3,t_4]$ because, during that interval, all preconditions were true and $\gamma$ was derived on the same node as $\alpha$. Finally, $\alpha$ delays the derivation of itself during $[t_4,t_5]$. We can now expand this definition to other derivations:

**Definition 19. (Transitive delay):** *Consider two derivations $\alpha : -c_1,c_2,\ldots,c_k$ and $\beta : -d_1,d_2,\ldots,d_m$, and suppose $\beta$ (directly or transitively) delays the derivation of $\alpha$ during an interval $[t_0,t_3]$. Then we say that $d_i$ transitively delays the derivation of $\alpha$ during an interval $[t_1,t_2]$, $t_0 \le t_1$, $t_2 \le t_3$, iff $d_i$ directly delays the derivation of $\beta$ during the interval $[t_1,t_2]$.*

We can think of the definition of transitive delay as recursively partitioning the interval $[t_0,t_5]$ into smaller intervals that are each associated with some lower-level derivation that caused delay to the top-level derivation of $\alpha$.

**Definition 20. (Annotation interval):** *We associate a vertex $v$ in the temporal provenance graph $G$ with an annotation interval $I_v^\alpha = [t_s,t_e]$ for each call of the* ANNOTATE$(v,[t_s,t_e])$ *procedure in the algorithm from Figure 5.6.*

Figure C.5 shows how the algorithm in Figure 5.6 would have assigned annotation intervals to an example temporal provenance graph. Before presenting the main theorem, we discuss a few properties of annotation intervals.

**Lemma 12.** *In the algorithm in Figure 5.6, each invocation of the* ANNOTATE$(v, [t_s, t_e])$ *procedure assigns a set of annotation intervals* $\{I^{\alpha}_{v^i}\}$ *to vertexes* $\{v^i\}$ *such that* $\bigcap_i I^{\alpha}_{v^i} = \emptyset$.

**Proof.** This holds by construction. When $v$ has no child, $\{I^{\alpha}_{v^i}\} = \emptyset$ and the condition holds trivially. When $v$ has children: the first WHILE loop in the ANNOTATE procedure subdivides the interval between $t_s$ and the end timestamp of the last precondition into annotation intervals for functional children (in lines 10–18); the second WHILE loop subdivides the interval between the end timestamp of the last precondition and $t_s(v)$ into annotation intervals for sequencing vertexes (in lines 20–26); note that if a functional precondition $v'$ is also connected via a sequencing edge to $v$, it is only handled by the first while loop, because $T = t_{end}(v') = t_{start}(v) = E$ after the first while loop finishes and the second while loop will not execute; therefore, all the generated annotation intervals within an ANNOTATE call are non-overlapping. □

This lemma states that the annotation intervals generated by recursive calls within the same ANNOTATE invocation do not overlap. For example, in Figure C.5, the annotation intervals of the DRV(C) and DRV(B) vertexes are both assigned by a recursive call on the DRV(A) vertex and thus do not overlap.

**Lemma 13.** *An annotation interval* $I^{\alpha}_{v}$ *of vertex* $v$ *always ends at* $t_e(v)$, *where* $t_e(v)$ *is when the execution of* $v$ *finishes or the end timestamp of* $v$ *(Section 5.3.2).*

**Proof.** This holds by construction of the algorithm in Figure 5.6. In the first WHILE loop in the ANNOTATE procedure (in lines 10–18), the annotation interval associated with $v$ always ends with $t_e(v)$. In the second WHILE loop (in lines 20–26), the annotation interval of the current vertex is $E = t_s(s)$, which is the start timestamp of the previous vertex connected via a sequencing edge; $E$ is also the end timestamp of the current vertex, which follows from the construction of sequencing edges (PREV-VERTEX calls in the algorithm in Figures C.1-C.2). □

This lemma states that the annotation interval ends when the actual execution finishes. For example, this holds for all annotation intervals in Figure C.5.

**Lemma 14.** *Given a vertex v associated with an annotation interval $I_v^\alpha$, there exists a chain of ancestor vertexes $v \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow e$ ($\rightarrow$ represents an edge in G, and e is the root of G) such each $a_i$ (including e) is associated with an annotation interval $I_{a_i}^\alpha$ that satisfies $I_v^\alpha \subseteq I_{a_i}^\alpha$.*

**Proof.** This holds by construction of the algorithm in Figure 5.6. It follows from the recursive nature of ANNOTATE calls that the annotation interval of each vertex $v$ is a subinterval of one annotation interval of one of its parents: in the first while loop (in lines 10–18), the ANNOTATE is called with an interval of $[T, t_{end}(v')]$, $t_s \leq T$ and $t_{end}(v') \leq t_{end}(v)$ because $v'$ is a child of $v$; in the second while loop (in lines 20–26), the ANNOTATE is called with an interval of $[\textsc{max}(T, t_{start}(s)), E]$, $t_s \leq T \leq \textsc{max}(T, t_{start}(s))$ and $E \leq t_{start}(v) \leq t_e$ (Lemma 13). We can simply find the specified chain by following such parents recursively until reaching the root vertex $e$. As the annotation interval is initially $I_v^\alpha$ and is gradually extended as we climb the chain, $I_v^\alpha \subseteq I_{a_i}^\alpha$. □

For instance, consider the provenance from Figure C.5, suppose $[0s, 1s]$@DRV(D) represents that the DRV(D) vertex is associated with an annotation interval of $[0s, 1s]$ ; the ancestor chain of $[0s, 1s]$@DRV(D) would be $[0s, 1s]$@DRV(D)$\rightarrow[0s, 3s]$@DRV(C)$\rightarrow$ $[0s, 6s]$@DRV(A).

**Lemma 15.** *Each vertex v in G is associated with at most one annotation interval $I_v^\alpha$, that is, each vertex v is annotated at most once by the algorithm in Figure 5.6.*

**Proof.** We prove by contradiction. Without loss of generality, suppose a vertex $v$ is associated with two annotation intervals $I_v^\alpha$ and $I_v^{\alpha\prime}$. There must exist two corresponding ancestor chains (Lemma 14). We make two observation about the chains: (a) they cannot be identical, because an ancestor chain represents a unique stack of recursive ANNOTATE calls; by the nature of a single-rooted DAG, there cannot exist two stacks of recursive calls that visit the exact same sequence of vertexes; (b) the two chains must share a common suffix, this holds trivially because both of the chains end at the root of $G$. Based on these observations, we can represent the two ancestor chains as $v \rightarrow \ldots \rightarrow a_i \rightarrow a_j \rightarrow \ldots$ and $v \rightarrow \ldots \rightarrow a_i' \rightarrow a_j \rightarrow \ldots$, where $a_i \neq a_i'$. It follows from Lemma 14 that $I_v^\alpha \subseteq I_{a_i}^\alpha$ and $I_v^{\alpha\prime} \subseteq I_{a_i'}^\alpha$. It follows from Lemma 13 that $[t_e(v) - \varepsilon, t_e(v)] \subseteq I_v^\alpha$ and $[t_e(v) - \varepsilon, t_e(v)] \subseteq I_v^{\alpha\prime}$, where $\varepsilon$ is a small value. Therefore, $I_{a_i}^\alpha$ and $I_{a_i'}^\alpha$ overlap. This contradicts with Lemma 12, because $I_{a_i}^\alpha$ and $I_{a_i'}^\alpha$ and divided from $I_{a_j}^\alpha$ in the same ANNOTATE call and cannot overlap. □

These lemmas allow us to formulate our main claim:

**Theorem 16.** *Suppose $G(e', e, \mathscr{E})$ is the output of* T-QUERY$(e', e)$ *in some execution $\mathscr{E}$, and suppose a vertex $v$ in $G$ is annotated with a value $T$ by the algorithm in Figure 5.6. Then $T > 0$ iff $v$ directly or transitively delayed the derivation of $e$ during an interval* $[t_1, t_2] \subseteq [\text{START}(e'), \text{FINISH}(e)]$ *and $T = t_2 - t_1$, and $T = 0$ otherwise.*

**Proof.** We begin by observing that the algorithm in Figure 5.6 labels each vertex at most once (Lemma 15). Therefore, we only need to show that any single invocation of the ANNOTATE procedure in the algorithm from Figure 5.6 correctly labels vertexes with respect to Definition 18.

Next, we observe that the ANNOTATE procedure in Figure 5.6 partitions the interval to explain into annotation intervals of other vertexes in exactly the same way that the definition requires. Therefore, $I_v^\alpha$ is exactly the direct or transitive delay of $v$. We discuss the partition logic of the ANNOTATE procedure in more detail below.

The children of a DRV vertex in the provenance graph would be DRV, INS, or RCV vertexes for its preconditions, and lines 10–18 iterate over these vertexes in the order of their end times. (The original trace only records the preconditions of an event at the point when its derivation starts; thus, if a precondition had temporarily become true and then false again, the corresponding DRV vertex would not appear as children here.) The loop calls ANNOTATE on vertex $v'$. with a subinterval of $[t_s, t_e]$ that ends at the point where the precondition is fully derived, and starts either at $t_s$ itself or the end of the previous interval. This subinterval is the annotation interval $I_{v'}^\alpha$ for $v'$ (Definition 20). Preconditions that were already true at $t_s$ and remained true during the entire interval do not enter the IF block and thus do not generate a recursive call. The first WHILE loop exits with $T$ set to the end time of the last precondition; the WHILE loop that follows it (in lines 20–26) subdivides any non-empty interval between the last satisfied precondition and the start of the derivation of $v$, just as the definition requires. Again, here each of the divided intervals is the annotation interval $I^{\alpha}_{v'}$ for another vertex $v'$ (Definition 20). In particular, noticed that recursive calls happen only for vertexes that directly delayed $v$ (and, hence, directly or transitively delayed the vertex in the original query).

Finally, we observe that each vertex $v$ gets labeled with the length of $I^{\alpha}_{v}$ in line 8. The labeled value is also the amount of direct or transitive delay that $v$ contributes, because we have proved above that the $I^{\alpha}_{v}$ is exactly the direct or transitive delay of $v$. For example, the intervals annotated beside vertexes in Figure C.5 are also their direct or transitive delay. □

### C.1.6 Semantics of delay annotations

Although the definitions from Section C.1.5 do capture the intuitive notion of "delay", we want to reinforce this by formalizing another aspect of this concept: if a vertex $v$ really did delay a derivation by some time $T_v$, then it should be possible to "speed up" the derivation by $T$ (i.e., cause it to happen $T_v$ units of time sooner) by reducing the duration of $v$ by $T_v$. In other words, we should be able to construct a valid (hypothetical) trace that differs from the actual trace in that $v$ takes less time, such that the hypothetical trace finishes $T_v$ units of time earlier. Note that the hypothetical trace might not be "realistic" in a practical sense because: (a) some of the events in it may take zero time, and thus be instantaneous; (b) the trace only includes all events that are present in the original temporal provenance, whose vertexes represent a valid and complete subtrace of the original execution (Section C.1.3). The goal is merely to demonstrate that $v$ is really "responsible for" $T_v$ units of delay. For example, Figures C.7 shows the steps of "speeding up" vertexes based on their annotations

```
A@X :- B@X,E@Y
B@X :- C@X
E@Y :- C@X
C@X :- Z@X
```

Figure C.6: An example NDlog program.

((a) $\rightarrow$ ... $\rightarrow$ (g)). This procedure shortens the overall (hypothetical) execution at each step and eventually eliminates any delay.

For this discussion, the annotation intervals that are computed by the algorithm in Figure 5.6 are not directly useful, because they describe the delay that was caused by an entire subgraph of the provenance. Hence, we first describe how we have derived a more fine-grain form of annotation, which describes the delay that is contributed by a vertex itself (Definition 21). We then discuss two properties of the derived annotation (Lemmas 17-18). We continue by defining the procedure of "speeding up" an execution based on derived annotations (Definitions 22-23). We conclude by presenting the main theorem which states that if there is a vertex $v$ in a temporal provenance tree with a (derived) annotation of $T$, then it is possible to construct another valid (but hypothetical) execution in which $v$'s finished time is reduced by $T$ and in which the derivation finishes $T$ units of time earlier (Definition 24 and Theorem 19).

**Definition 21. (Speedup interval):** *The speed interval $I_v^\delta = [t_s, t_e]$ of vertex v is the difference between v's annotation interval, as computed by the algorithm from Figure 5.6, and the union of the annotation intervals of the vertexes directly annotated by v (via the recursive calls in the* ANNOTATE *procedure).*

Intuitively, $I_v^\delta$ represents the interval during which the execution of $v$ itself delays $e$. For example, in the provenance from Figure C.7(a), red intervals represent annotation intervals and blue intervals represent speedup intervals. The speed up interval of DRV(A) is $[6s, 7s]$, which is the difference from its annotation interval ($[0s, 7s]$), and the union of the annotations intervals of DRV(B) and RCV(+E) ($[0s, 4s] \bigcup [4s, 6s]$). Speed up intervals have the following two properties:

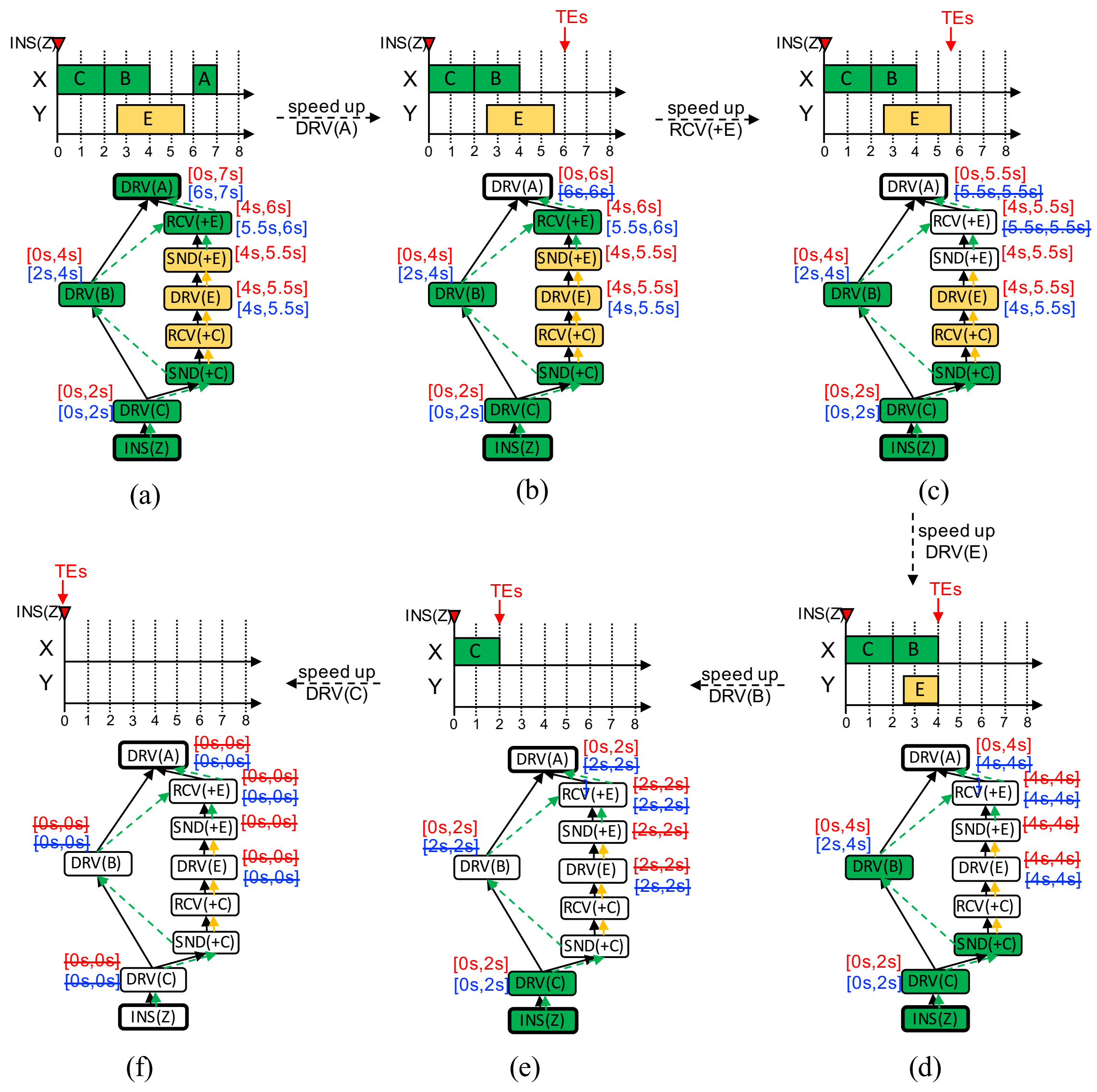


Figure C.7: An example of "speeding up" temporal provenance using a series of transformations (Definition 23). The NDlog rules are in Figure C.6. Each sub-figure shows a step of the transformation ((a) → ... → (g)). In each sub-figure, the execution trace is at the top, and the resulting temporal provenance at the bottom. The query is T-QUERY(INS(Z), DRV(A)) in all sub-figures; the start and end vertexes are marked in bold. Vertex names have been shortened and some fields have been omitted for clarity. Each vertex is associated with its annotation interval (red, Definition 20) and speed up interval (blue, Definition 21). Crossed intervals represent that the interval becomes empty but the annotation is preserved. White vertexes are terminal events.

**Lemma 17.** *The speedup interval $I_v^\delta$ of vertex v always ends at $t_e(v)$, where $t_e(v)$ is when the execution of v finishes or the end timestamp of v (Section 5.3.2).*

**Proof.** The annotation interval of $v$ always ends at $t_e(v)$ (Lemma 13). We prove that that $I_v^\delta$ ends at the same moment when $I_v^\alpha$ ends. If $v$ has no child, $I_v^\delta = I_v^\alpha$; if $v$ has children, the two WHILE loops in the algorithm in Figure 5.6 distribute the interval between $t_s$ and $t_s(v)$ to other vertexes via recursive ANNOTATE calls, and the remaining interval in $[t_s, t_e]$ is the speedup interval; in either case, $I_v^\delta$ ends when $I_v^\alpha$ ends, and therefore, $I_v^\delta$ ends at $t_e(v)$. □

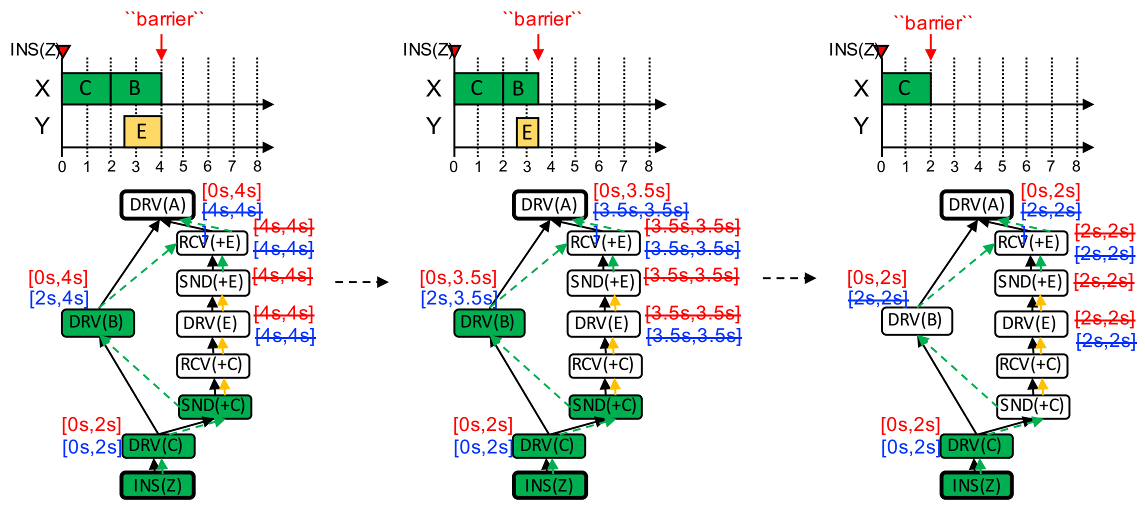
**Lemma 18.** *Given the timing provenance that explains* T-QUERY$(e', e)$, *consider the set of all speedup intervals $\{I_{v^i}^\delta\}$: (a) $\bigcap_i I_{v^i}^\delta = \emptyset$; (b) $\bigcup_i I_{v^i}^\delta = I_e^\alpha$.*

**Proof.** In a timing provenance graph, vertexes with annotation intervals form a tree, because vertex $v$ is annotated at most once (Lemma 15) by a parent of $v$. Consequently, vertexes with speedup intervals form a tree (Definition 21). We prove by structural induction on the height of the tree.

**Base case**: The height of the tree is one. Denote the root vertex as $v$. The set of speedup intervals has one element ($\{I_{v^i}^\delta\} = I_v^\delta$), condition (a) holds. $I_v^\delta = I_v^\alpha$ because $v$ has no child (Definition 21), condition (b) holds.

**Induction case**: Suppose the conditions hold for trees (of vertexes with annotation or speedup intervals) with height less than $k$ ($k \geq 1$). Consider a tree with depth $k+1$, rooted at vertex $v$. Without loss of generality, denote $T(v_1)$ and $T(v_2)$ as subtrees of $v$ that have annotation intervals. Note that the speedup intervals of vertexes in $T(v)$ must be a subinterval of the annotation interval of $v$, because: the speedup interval is simply the difference of the annotation interval of $v$ and the annotation intervals of the children of $v$ (Definition 21); the annotation interval of $v$ is a subinterval of the annotation interval of its parent in the tree (Lemma 14).

It follows from Definition 21 that the speedup interval of $v$ and that of $T(v_1)$ (or $T(v_2)$) cannot overlap. It follows from Lemma 12 that the speedup intervals of $T(v_1)$



(a) $t_b = 4s$ (b) $t_b = 3.5s$ (c) $t_b = 2s$

Figure C.8: An example of "speeding up" temporal provenance using an annotated vertex DRV(B) (Definition 23). The NDlog rules are in Figure C.6. In each subfigure, the execution trace is at the top, and the resulting temporal provenance at the bottom. The query is T-QUERY(INS(Z), DRV(A)) in all sub-figures; the start and end vertexes are marked in bold. Vertex names have been shortened and some fields have been omitted for clarity. Each vertex is associated with its annotation interval (red, Definition 20) and speed up interval (blue, Definition 21). Crossed intervals represent that the interval becomes empty but the annotation is preserved. White vertexes are terminal events.

and $T(v_2)$ cannot overlap. It follow from the induction hypothesis that the speedup intervals within $T(v_1)$ (or $T(v_2)$) cannot overlap. Therefore, condition (a) holds. It follows from Definition 21 that condition (b) holds. $\square$

Intuitively, the above two lemmas states that, given a timing provenance that explains T-QUERY$(e',e)$, the overall delayed interval – that is, $[t_s(e'),t_e(e)]$ – can be subdivided into a sequence of all speedup intervals $\{I^{\delta}_{v_i}\}$. In the example provenance from Figure C.7(a), such a sequence of speedup intervals is $[0s,2s]$@DRV(C), $[2s,4s]$@DRV(B), $[4s,5.5s]$@DRV(E), $[5.5s,6s]$@RCV(+E), and $[6s,7s]$@DRV(A).

**Definition 22. (Terminal event):** *A vertex v is a terminal event if any of the following conditions holds:*

- *(a) if $v$ is annotated, $v$ ends at $t_e(e)$, and $I_v^\delta = \emptyset$ (a vertex is annoated if it is associated with an annotation interval in the original G); and*
- *(b) if $v$ is not annotated, on any path in G from $v$ to $e$, select $u$ as the first annotated vertex, $I_u^\alpha = \emptyset$.*

Intuitively, terminal events represent executions that no longer contribute any delay (in a hypothetical execution). Condition (a) describes an event that finishes at the end of the entire execution and that no longer contributes any delay. For example, RCV(+E) in Figure C.7(c) is a terminal event: it was annotated in the original provenance (Figure C.7(a)); it ends at $t = 5.5s$, which is the end timestamp of DRV(A); and its speedup interval is empty. Condition (b) describes an event that only belongs to subgraphs that no longer contribute any delay. For example, RCV(C) in Figure C.7(e) is a terminal event: it was not annotated in the original provenance (Figure C.7(a)); on its (only) path to DRV(A), the first annotated vertex is DRV(E), whose annotation interval is already empty ($[4s, 4s]$). Next, we describe steps to transform the original execution to hypothetical executions.

**Definition 23. (Speed up):** *Given a vertex $v$ in $G(e', e, \mathscr{E})$, where $t_e(v) = t_e(e)$ and $I_v^\delta > 0$, $v$ speeds up $G$ by $I_v^\delta$ using the following procedure. Consider a "barrier" $t_b$ that moves on the timeline; it starts from the right boundary of $I_v^\delta$ and moves leftwards (and thus $t_b$ becomes smaller); it stops when it reaches the left boundary of $I_v^\delta$. For ease of exposition, we say that the "barrier" pushes a timestamp $t$ when we set $t$ to $\text{MIN}(t, t_b)$. During its move, if the "barrier" encounters a vertex $v_i$ that is either $v$ or a terminal event, it transforms $v_i$ by pushing these timestamps: (a) the starting timestamp (or the ending timestamp) of $v_i$, (b) the left boundary (or the right boundary) of $I_{v_i}^\alpha$ (if any); and (c) the left boundary (or the right boundary) of $I_{v_i}^\delta$ (if any).*

Intuitively, the "speed up" operation represents a transformation step that essentially "squeezes" a set of vertexes to the left. Note that, while $v$ speeds up $G$, only $v$ itself and terminal events – vertexes that no longer contribute any delay – are pushed leftwards. For example, Figure C.8 shows the process of speeding up the provenance using $\textsc{drv}$(B): the "barrier" starts from the right boundary of $I^{\delta}_{\text{DRV(A)}}$ (Figure C.8(a)); while it moves, the "barrier" pushes $\textsc{drv}$(B) as well as terminal events $\textsc{drv}$(E) and $\textsc{rcv}$(C) leftwards (Figure C.8(b) shows the snapshot of $t_b = 3.5s$); the "barrier" stops at the left boundary of $I^{\delta}_{\text{DRV(A)}}$ (Figure C.8(c)).

Figure C.7 shows the process of speeding up an entire provenance graph until it becomes instantaneous. Next, we describe the sequence of "speed up" operations:

- (a) $\rightarrow$ (b), $\textsc{drv}$(A) speeds up $G$ by $I^{\delta}_{\text{DRV(A)}} = [6s, 7s]$: the execution is shortened to $[0s, 6s]$, $\textsc{drv}$(A) becomes a terminal event;
- (b) $\rightarrow$ (c), $\textsc{rcv}$(+E) speeds up $G$ by $I^{\delta}_{\text{RCV(+E)}} = [5.5s, 6s]$: the execution is shortened to $[0s, 5.5s]$, $\textsc{rcv}$(+E) and $\textsc{snd}$(+E) become terminal events;
- (c) $\rightarrow$ (d), $\textsc{drv}$(E) speeds up $G$ by $I^{\delta}_{\text{DRV(E)}} = [4s, 5.5s]$: the execution is shortened to $[0s, 4s]$, $\textsc{drv}$(E) and $\textsc{rcv}$(+C) become terminal events;
- (d) $\rightarrow$ (e), $\textsc{drv}$(B) speeds up $G$ by $I^{\delta}_{\text{DRV(B)}} = [2, 4s]$: the execution trace is shortened to $[0s, 2s]$, $\textsc{drv}$(B) becomes terminal events;
- (e) $\rightarrow$ (f), $\textsc{drv}$(C) speeds up $G$ by $I^{\delta}_{\text{DRV(C)}} = [0, 2s]$: the execution trace is shortened to $[0s, 0s]$, all events are now terminal events.

**Definition 24. (Well-annotated):** *Consider an annotated temporal provenance graph $G(e', e, \mathscr{E})$. $G$ is well-annotated iff either (a) $t_s(e') = t_e(e)$, that is, the entire execution is instantaneous; (b) we can transform $G$ into another valid and well-annotated temporal provenance graph $G'$ by locating an unique vertex $v$, where $t_e(v) = t_e(e)$ and $I^{\delta}_v > 0$, and speeding up $G$ by $v$ (Definition 23).*

**Theorem 19.** *Temporal provenance is well-annotated.*

**Proof.** Consider the speedup intervals $\{I^{\delta}_{v_i}\}$ of $G$. It follows from Lemma 18 that $\{I^{\delta}_{v_i}\}$ do not overlap and unions to $[t_s(e'), t_e(e)]$. Therefore, we can sort intervals in $\{I^{\delta}_{v_i}\}$ by descending (ending) timestamp. At the $i$th step, we speed up $G$ by $v_i$.

We need to prove that: (a) each "speed up" operation pushes the timestamps of all events that ends during $I^{\delta}_{v_i}$; (b) the length of the execution $[t_s(e'), t_e(e)]$ is reduced by the length of $I^{\delta}_{v_i}$; (c) the temporal provenance remains valid.

To prove condition (a), given any vertex $v'_i$ whose execution ends during $I^{\delta}_{v_i}$, we perform a case analysis on $v'_i$:

- $v'_i = v_i$: the timestamps of $v'_i = v_i$ is pushed, by the construction of Definition 23.
- $v'_i \neq v_i$ and $v'_i$ is annotated in the original provenance: $v'_i$ must be a terminal event, and therefore, its timestamps is pushed. Because $I^{\delta}_{v'_i}$ ends when $v'_i$ ends (Lemma 17); consequently, $I^{\delta}_{v'_i}$ must end during $I^{\delta}_{v_i}$; if $I^{\delta}_{v'_i}$ is not empty, $I^{\delta}_{v'_i}$ will overlap with $I^{\delta}_{v_i}$, which contradicts with Lemma 18.
- $v'_i \neq v_i$ and $v'_i$ is not annotated in the original provenance: $v'_i$ must be a terminal event, and therefore, its timestamps is pushed. Because, given any path from $v'_i$ to $e$, consider the first annotated ancestor $u$ and its child on the path $w$; if we assume that $I^{\alpha}_{u}$ is not empty when the "barrier" reaches the end of $w$, then $I^{\alpha}_{u}$ must start before the end of $w$; by construction of the algorithm from Figure 5.6, $w$ must be annotated by $u$, which contradicts the fact that $w$ is not annotated in the original provenance.

Condition (b) follows directly from the statement above: the execution is shortened by the length of $I^{\delta}_{v_i}$, because all events that end during $I^{\delta}_{v_i}$ are pushed leftwards until the left boundary of $I^{\delta}_{v_i}$.

Condition (c) holds because the "speed up" operation does not invert causality: if an event $a$ caused another event $b$, it does not alter the ordering of $a$ and $b$; nor does it delete any event. □

Note that Definition 24 weeds out some annotation approaches. For example, Figure C.9 shows how a straw-man approach that associates the entire delay with the last precondition would have annotate the same provenance graph in Figure C.7. The result is not well annotated: while DRV(E) speeds up $G$ ((d) $\rightarrow$ (e)), another

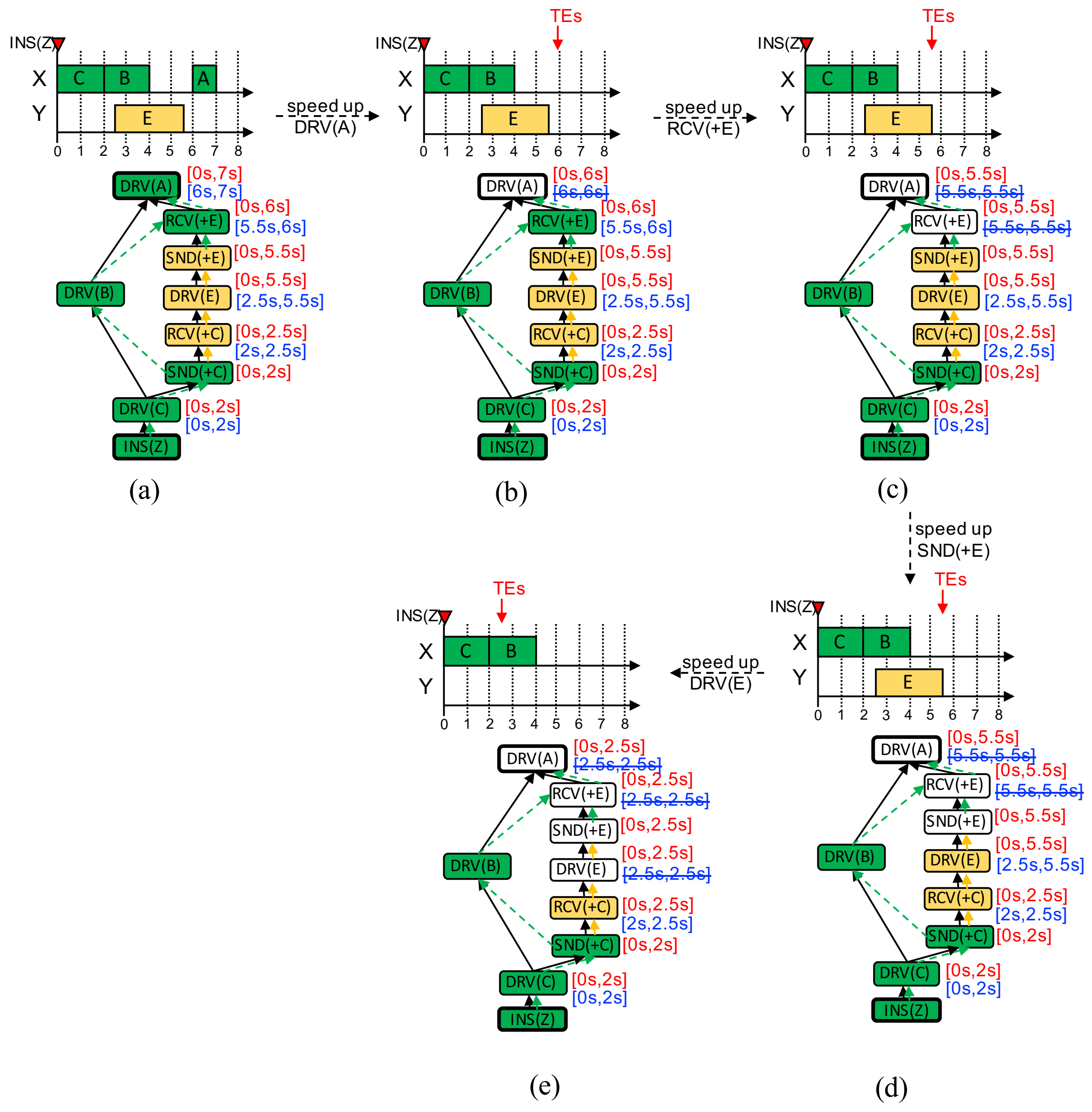
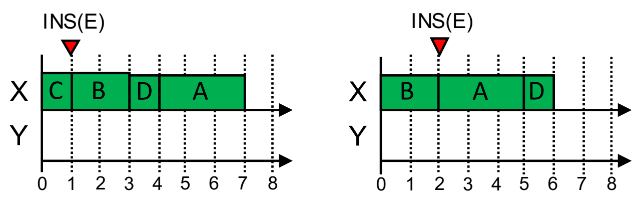


Figure C.9: An example of "poorly annotated" temporal provenance. The NDlog rules are in Figure C.6. Each sub-figure shows a step of the transformation ((a) → ... → (g)). In each sub-figure, the execution trace is at the top, and the resulting temporal provenance at the bottom. The query is T-QUERY(INS(Z), DRV(A)) in all sub-figures; the start and end vertexes are marked in bold. Vertex names have been shortened and some fields have been omitted for clarity. Each vertex is associated with its annotation interval (red, Definition 20) and speed up interval (blue, Definition 21). Crossed intervals represent that the interval becomes empty but the annotation is preserved. White vertexes are terminal events.

```
A@X :- B@X
B@X :- C@X
D@X :- E@X
```



(a) Original execution. (b) After speeding up DRV(C).

Figure C.10: An example scenario where speeding up a single event delays the completion of another event. The NDlog rules are at the left and the timing of concrete executions are in the right. Speeding up DRV$(C)$ changes the ordering of events and delays the completion of DRV$(D)$. Note that the (pending) insertion of tuple $E$ enters the update queue at $t = 0.5s$ (Section 5.3.1).

vertex DRV(B) becomes the bottleneck; however, DRV(B) cannot be pushed leftwards, because it is not a terminal event, that is, it has not been "sped up".

Theorem 19 demonstrates that the annotations on temporal provenance are intuitive: if one follows the annotations $\{I_{v_i}^{\delta}\}$ and speed up events $\{v_i\}$ one by one, the overall delay of the (hypothetical) execution is reduced by the length of $I_{v_i}^{\delta}$ at the $i$-th step and eventually becomes instantaneous. However, note that if one *only* speeds up a *single* event $v_i$ by $I_{v_i}^{\delta}$ (and all other events take same amounts of time), the overall delay is not guaranteed to decrease. Such anomalies – where speeding up an event can sometimes increase the overall delay – is well-known in scheduling theory [51, 86]. Figure C.10 shows an example of such anomalies: speeding up DRV$(C)$ advances DRV$(B)$ and reverses the ordering of DRV$(B)$ and INS$(E)$; consequently, the ordering of their dependent computations – DRV$(A)$ and DRV$(D)$ – is reversed; as a result, DRV$(D)$ is delayed. Our definition is different: instead of speeding up a single event, the procedure in Theorem 19 speeds up a sequence of events in reverse order of occurrence (i.e., $\{v_i\}$ such that $\{I_{v_i}^{\delta}\}$ have descending timestamps); each step of the process preserves the ordering of all events in the temporal provenance and the aforementioned anomaly cannot happen in each of the (hypothetical) executions.

Theorem 19 suggests that the annotations on temporal provenance do correspond to the "potential for speedup" that one may intuitively associate with the concept of delay. This is useful, because, while the temporal provenance maybe gigantic and complex, operators can focus on vertexes with annotations and gain a comprehensive understanding of the end-to-end delay, including potential operations to speed up.

# Bibliography

[1] K. Agarwal, E. Rozner, C. Dixon, and J. Carter. SDN traceroute: Tracing SDN forwarding without changing network behavior. In *Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2014. [Cited on page 15.]

[2] B. Aggarwal, R. Bhagwan, T. Das, S. Eswaran, V. N. Padmanabhan, and G. M. Voelker. NetPrints: Diagnosing home network misconfigurations using shared knowledge. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009. [Cited on page 14.]

[3] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 1990. [Cited on page 13.]

[4] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003. [Cited on pages 1 and 14.]

[5] E. Al-Shaer and S. Al-Haj. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *ACM workshop on Assurable and Usable Security Configuration*, 2010. [Cited on page 12.]

[6] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. ElBadawi. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *IEEE International Conference on Network Protocols (ICNP)*, 2009. [Cited on page 12.]

[7] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2014. [Cited on page 12.]

[8] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred. Taking the blame game out of data centers operations with NetPoirot. In *ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, 2016. [Cited on page 14.]

[9] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012. [Cited on page 14.]

[10] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004. [Cited on pages 1 and 15.]

[11] A. Bates, D. Tian, K. R. Butler, and T. Moyer. Trustworthy whole-system provenance for the Linux kernel. In *USENIX Security Symposium*, 2015. [Cited on page 10.]

[12] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, 2017. [Cited on page 12.]

[13] R. Beckett, X. K. Zou, S. Zhang, S. Malik, J. Rexford, and D. Walker. An assertion language for debugging SDN applications. In *Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2014. [Cited on page 80.]

[14] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *The Internet Measurement Conference (IMC)*, 2010. [Cited on pages 1, 79, 83, and 85.]

[15] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review (CCR)*, 2014. [Cited on pages 54 and 111.]

[16] J.-Y. L. Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, volume LNCS 2050. Springer, 2001. [Cited on page 121.]

[17] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *International Conference on Database Theory (ICDT)*, 2001. [Cited on pages 2, 10, and 94.]

[18] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE way to test OpenFlow applications. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012. [Cited on pages 1, 12, and 80.]

[19] A. Chanda, A. L. Cox, and W. Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007. [Cited on pages 1 and 15.]

[20] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *International Conference on Software Engineering (ICSE)*, 2011. [Cited on page 88.]

[21] A. Chapman and H. Jagadish. Why not? In *ACM International Conference on Management of Data (SIGMOD)*, 2009. [Cited on pages 2, 10, 17, and 54.]

[22] A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. One primitive to diagnose them all: Architectural support for internet diagnostics. In *ACM European Conference on Computer Systems (EuroSys)*, 2017. [Cited on pages 2 and 11.]

[23] A. Chen, W. B. Moore, H. Xiao, A. Haeberlen, L. T. X. Phan, M. Sherr, and W. Zhou. Detecting covert timing channels with time-deterministic replay. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014. [Cited on pages 121 and 125.]

[24] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance. In *ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, 2016. [Cited on pages 2, 11, 78, and 94.]

[25] A. Chen, H. Xiao, L. T. X. Phan, and A. Haeberlen. Fault tolerance and the five-second rule. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2015. [Cited on page 121.]

[26] C. Chen, H. T. Lehri, L. K. Loh, L. Jia, B. T. Loo, W. Zhou, and A. Alur. Distributed provenance compression. In *ACM International Conference on Management of Data (SIGMOD)*, 2017. [Cited on page 11.]

[27] Y.-Y. M. Chen, A. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E. A. Brewer. Path-based failure and evolution management. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2004. [Cited on pages 1 and 15.]

[28] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 2009. [Cited on page 10.]

[29] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: end-to-end performance analysis of large-scale internet services. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014. [Cited on pages 1, 14, 106, and 125.]

[30] N. Dalvi, C. Ré, and D. Suciu. Probabilistic databases: diamonds in the dirt. *Communications of the ACM*, 2009. [Cited on page 126.]

[31] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, 2008. [Cited on page 78.]

[32] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, 2011. [Cited on page 10.]

[33] R. Durairajan, J. Sommers, and P. Barford. Controller-agnostic SDN debugging. In *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2014. [Cited on page 80.]

[34] S. A. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. In *ACM Design Automation Conference (DAC)*, 2007. [Cited on pages 121 and 125.]

[35] D. Erickson. The Beacon OpenFlow controller. In *Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2013. [Cited on page 47.]

[36] D. Erickson. The Beacon OpenFlow controller. In *Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2013. [Cited on page 82.]

[37] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar. BUZZ: Testing context-dependent policies in stateul networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016. [Cited on page 12.]

[38] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005. [Cited on page 12.]

[39] A. Feldmann, O. Maennel, Z. M. Mao, A. Berger, and B. Maggs. Locating Internet routing instabilities. In *ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, 2004. [Cited on pages 1 and 14.]

[40] FireEye. Recent Zero-Day Exploits, 2016. `https://www.fireeye.com/current-threats/recent-zero-day-attacks.html`. [Cited on pages 1 and 11.]

[41] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015. [Cited on page 12.]

[42] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007. [Cited on pages 1 and 15.]

[43] I. T. Foster, J.-S. Vöckler, M. Wilde, and Y. Zhao. The virtual data grid: A new model and architecture for data-intensive collaboration. In *Conference on Innovative Data Systems Research (CIDR)*, 2003. [Cited on page 10.]

[44] A. Gehani and D. Tariq. SPADE: Support for provenance auditing in distributed environments. In *International Middleware Conference (Middleware)*, 2012. [Cited on page 10.]

[45] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing one thousand queries with one stone. In *International Conference on Very Large Databases (VLDB)*, 2012. [Cited on page 76.]

[46] Google. Google Compute App Incident 14005, 2014. `https://status.cloud.google.com/incident/appengine/14005`. [Cited on page 113.]

[47] Google. Google Compute Engine Incident 15039, 2014. `https://status.cloud.google.com/incident/compute/15039`. [Cited on page 112.]

[48] Google. Google Compute App Incident 15005, 2015. `https://status.cloud.google.com/incident/appengine/15005`. [Cited on page 113.]

[49] Google. Google Compute Engine Incident 15057, 2015. https://status.cloud.google.com/incident/compute/15057. [Cited on page 113.]

[50] Google. Google Cloud Platform Dashboard, 2016. https://status.cloud.google.com/summary. [Cited on pages 93 and 112.]

[51] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969. [Cited on page 196.]

[52] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Principles of Database Systems (PODS)*, 2007. [Cited on page 10.]

[53] T. J. Green, G. Karvounarakis, N. E. Taylor, O. Biton, Z. G. Ives, and V. Tannen. ORCHESTRA: Facilitating collaborative data sharing. In *ACM International Conference on Management of Data (SIGMOD)*, 2007. [Cited on page 10.]

[54] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, M. Musuvathi, Z. Zhang, and L. Zhou. Failure recovery: When the cure is worse than the disease. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2013. [Cited on page 75.]

[55] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM International Conference on Management of Data (SIGMOD)*, 1984. [Cited on page 34.]

[56] M. Hadjieleftheriou. libspatialindex, 2014. https://libspatialindex.github.io/. [Cited on page 40.]

[57] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. Where is the debugger for my software-defined network? In *Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2012. [Cited on pages 15 and 36.]

[58] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014. [Cited on pages 1 and 15.]

[59] R. Hasan, R. Sion, and M. Winslett. The case of the fake picasso: Preventing history forgery with secure provenance. In *USENIX Conference on File and Storage Technologies (FAST)*, 2009. [Cited on page 10.]

[60] H. Hojjat, P. Reummer, J. McClurgh, P. Cerny, and N. Foster. Optimizing Horn solvers for network repair. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2016. [Cited on pages 13, 14, and 88.]

[61] A. Horn, A. Kheradmand, and M. R. Prasad. Delta-net: Real-time network verification using atoms. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017. [Cited on page 12.]

[62] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *International Conference on Very Large Databases (VLDB)*, 1(1), 2008. [Cited on pages 10, 16, 17, and 54.]

[63] H. Jagadish and F. Olken. Database management for life sciences research. *ACM SIGMOD Record*, 2004. [Cited on page 10.]

[64] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, et al. Canopy: An end-to-end performance tracing and analysis system. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2017. [Cited on page 15.]

[65] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. In *ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, 2009. [Cited on page 14.]

[66] E. Katz-Bassett, H. V. Madhyastha, J. P. John, A. Krishnamurthy, D. Wetherall, and T. Anderson. Studying black holes in the Internet with Hubble. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008. [Cited on pages 1 and 12.]

[67] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013. [Cited on page 12.]

[68] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012. [Cited on pages 1 and 12.]

[69] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013. [Cited on page 12.]

[70] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering (ICSE)*, 2013. [Cited on page 57.]

[71] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable dynamic network control. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015. [Cited on page 12.]

[72] Kim, Changhoon and Bhide, Parag and Doe, Ed and Holbrook, Hugh and Ghanwani, Anoop and Daly, Dan and Hira, Mukesh and Davie, Bruce. In-band network telemetry, 2016. `http://p4.org/wp-content/uploads/fixed/INT/INT-current-spec.pdf`. [Cited on page 111.]

[73] L. Kleinrock. *Queueing Systems, Volume 1: Theory*. Wiley-Interscience, 1975. [Cited on page 121.]

[74] L. Kleinrock. *Queueing Systems, Volume 2: Computer Applications*. Wiley-Interscience, 1976. [Cited on page 121.]

[75] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network virtualization in multi-tenant datacenters. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014. [Cited on page 12.]

[76] E. Koskinen and J. Jannotti. Borderpatrol: isolating events for black-box tracing. In *ACM European Conference on Computer Systems (EuroSys)*, 2008. [Cited on pages 1 and 14.]

[77] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 1978. [Cited on page 101.]

[78] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Workshop on Hot Topics in Networks (HotNets)*, 2010. [Cited on pages 40, 77, and 79.]

[79] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering (ICSE)*, 2012. [Cited on pages 12 and 88.]

[80] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004. [Cited on page 80.]

[81] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *Workshop on Architectural and system support for improving software dependability*, 2006. [Cited on page 71.]

[82] D. Logothetis, S. De, and K. Yocum. Scalable lineage capture for debugging DISC analysis. Technical Report CSE2012-0990, UCSD, 2012. [Cited on pages 40, 78, and 111.]

[83] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Communications of the ACM*, 2009. [Cited on pages xii, 6, 22, 54, 59, 98, 143, 145, and 146.]

[84] B. T. Loo, S. C. Muthukumar, X. Li, L. Changbin, J. B. Kopena, M. Oprea, T. Tao, and W. Zhou. RapidNet, 2009. `http://netdb.cis.upenn.edu/rapidnet/`. [Cited on pages 18, 40, 78, 110, and 111.]

[85] N. P. Lopes, N. Bjørner, P. Godefroid, and K. Jayaraman. Checking beliefs in dynamic networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015. [Cited on page 12.]

[86] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *IEEE Real-Time Systems Symposium (RTSS)*, 1999. [Cited on page 196.]

[87] P. Macko and M. Seltzer. Provenance Map Orbiter: Interactive exploration of large provenance graphs. In *USENIX Workshop on the Theory and Practice of Provenance (TaPP)*, 2011. [Cited on page 32.]

[88] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *ACM International Conference on Management of Data (SIGMOD)*, 2002. [Cited on page 76.]

[89] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, 2011. [Cited on pages 1 and 12.]

[90] J. McClurg, H. Hojjat, P. Černý, and N. Foster. Efficient synthesis of network updates. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2015. [Cited on page 13.]

[91] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review (CCR)*, 2008. [Cited on page 54.]

[92] A. Meliou, W. Gatterbauer, S. Nath, and D. Suciu. Tracing data errors with view-conditioned causality. In *ACM International Conference on Management of Data (SIGMOD)*, 2011. [Cited on page 2.]

[93] A. Meliou and D. Suciu. Tiresias: The database oracle for how-to queries. In *ACM International Conference on Management of Data (SIGMOD)*, 2012. [Cited on page 10.]

[94] A. Meliou and D. Suciu. Tiresias: the database oracle for how-to queries. In *ACM International Conference on Management of Data (SIGMOD)*, 2012. [Cited on pages 10, 17, and 54.]

[95] B. P. Miller. Dpm: A measurement system for distributed programs. *IEEE Transactions on Computers*, 37(2):243–248, 1988. [Cited on pages 1 and 15.]

[96] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013. [Cited on pages x, xii, 6, 12, 22, 36, 37, 38, 78, 87, 143, 151, 152, 153, 154, and 155.]

[97] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *USENIX Annual Technical Conference (ATC)*, 2006. [Cited on pages 33 and 36.]

[98] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer. Provenance-aware storage systems. In *USENIX Annual Technical Conference (ATC)*, 2006. [Cited on page 10.]

[99] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012. [Cited on page 14.]

[100] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014. [Cited on page 12.]

[101] B. Networks. Barefoot Kitsilano project, 2017. Private communication. [Cited on page 111.]

[102] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *International Conference on Software Engineering (ICSE)*, 2013. [Cited on pages 12 and 88.]

[103] Node.js, 2017. `https://nodejs.org/`. [Cited on page 111.]

[104] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *USENIX Annual Technical Conference (ATC)*, 2013. [Cited on page 14.]

[105] outages.org. Outages Mailing List, 2014. `http://wiki.outages.org/index.php/Main_Page`. [Cited on pages 19 and 20.]

[106] K. Pan, S. Kim, and E. J. Whitehead Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009. [Cited on pages 57, 61, 71, and 79.]

[107] A. Panda, O. Lahav, K. J. Argyraki, M. Sagiv, and S. Shenker. Verifying reachability in networks with mutable datapaths. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017. [Cited on page 12.]

[108] R. Perera, U. A. Acar, J. Cheney, and P. B. Levy. Functional programs that explain their work. In *ACM International Conference on Functional Programming (ICFP)*, 2012. [Cited on page 13.]

[109] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, et al. Automatically patching errors in deployed software. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2009. [Cited on pages 12 and 88.]

[110] Pyramid, 2017. `https://trypyramid.com/`. [Cited on page 111.]

[111] C. Ré, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *International Conference on Data Engineering (ICDE)*, 2007. [Cited on page 10.]

[112] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2006. [Cited on page 14.]

[113] L. Ryzhyk, N. Bjørner, M. Canini, J.-B. Jeannin, C. Schlesinger, D. B. Terry, and G. Varghese. Correct by construction networks using stepwise refinement. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017. [Cited on page 12.]

[114] S. Saha, S. Prabhu, and P. Madhusudan. NetGen: Synthesizing data-plane configurations for network policies. In *ACM Symposium on SDN Research (SOSR)*, 2015. [Cited on pages 13, 14, and 88.]

[115] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011. [Cited on pages 14 and 109.]

[116] B. Schlinker, R. N. Mysore, S. Smith, J. C. Mogul, A. Vahdat, M. Yu, E. Katz-Bassett, and M. Rubin. Condor: Better topologies through declarative design. In *ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, 2015. [Cited on page 13.]

[117] C. Scott, A. Panda, V. Brajkovic, G. Necula, A. Krishnamurthy, and S. Shenker. Minimizing faulty executions of distributed systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016. [Cited on page 12.]

[118] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker. Troubleshooting blackbox SDN control software with minimal causal sequences. In *ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, 2014. [Cited on pages 1, 11, and 12.]

[119] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. In *IEEE Symposium on Security and Privacy*, 2005. [Cited on pages 13 and 88.]

[120] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. In *Google Technical Report*, 2010. [Cited on pages 1, 15, 92, 93, 111, 118, and 125.]

[121] A. Singh, P. Maniatis, T. Roscoe, and P. Druschel. Using queries for distributed monitoring and forensics. In *ACM European Conference on Computer Systems (EuroSys)*, Apr. 2006. [Cited on page 15.]

[122] K. Subramanian, L. D'Antoni, and A. Akella. Genesis: Synthesizing forwarding tables in multi-tenant networks. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2017. [Cited on page 13.]

[123] P. Tammana, R. Agarwal, and M. Lee. Cherrypick: Tracing packet trajectory in software-defined datacenter networks. In *ACM Symposium on SDN Research (SOSR)*, 2015. [Cited on page 15.]

[124] R. Teixeira and J. Rexford. A measurement framework for pin-pointing routing changes. In *Workship on Network troubleshooting (NetTS)*, 2004. [Cited on pages 39 and 43.]

[125] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter. The netlogger methodology for high performance distributed systems performance analysis. In *ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 1998. [Cited on pages 1 and 15.]

[126] Q. T. Tran and C.-Y. Chan. How to conquer why-not questions. In *ACM International Conference on Management of Data (SIGMOD)*, 2010. [Cited on pages 2, 54, and 88.]

[127] Trema. Trema, 2014. `http://trema.github.io/trema/`. [Cited on pages xii, 6, 18, 40, 78, 87, 143, 146, 147, 148, and 149.]

[128] A. Wang, L. Jia, W. Zhou, Y. Ren, B. T. Loo, J. Rexford, V. Nigam, A. Scedrov, and C. L. Talcott. FSR: Formal analysis and implementation toolkit for safe inter-domain routing. *IEEE/ACM ToN*, 20(6):1814–1827, Dec. 2012. [Cited on page 39.]

[129] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004. [Cited on pages 1, 11, and 14.]

[130] M. Weiser. Program slicing. In *International Conference on Software Engineering (ICSE)*, 1981. [Cited on page 13.]

[131] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *Conference on Innovative Data Systems Research (CIDR)*, 2005. [Cited on page 10.]

[132] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem. *ACM Transactions on Embedded Computing Systems*, 2008. [Cited on page 121.]

[133] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. Automated bug removal for software-defined networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017. [Cited on pages iii and 94.]

[134] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo. Diagnosing missing events in distributed systems negative provenance. In *ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, 2014. [Cited on pages iii, 78, 94, and 111.]

[135] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling record and replay troubleshooting for networks. In *USENIX Annual Technical Conference (ATC)*, 2011. [Cited on page 15.]

[136] C. Yang and B. P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *IEEE International Conference on Distributed Computing Systems (DCS)*, 1988. [Cited on page 106.]

[137] Y. Yuan, D. Lin, R. Alur, and B. T. Loo. Scenario-based programming for SDN policies. In *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2015. [Cited on page 13.]

[138] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2012. [Cited on pages 1, 12, 79, and 80.]

[139] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014. [Cited on page 12.]

[140] S. Zhang and M. D. Ernst. Automated diagnosis of software configuration errors. In *International Conference on Software Engineering (ICSE)*, 2013. [Cited on pages 12 and 88.]

[141] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI 2: Cpu performance isolation for shared compute clusters. In *ACM European Conference on Computer Systems (EuroSys)*, 2013. [Cited on page 14.]

[142] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014. [Cited on pages 1 and 14.]

[143] W. Zhou. *Secure Time-Aware Provenance For Distributed Systems*. PhD thesis, University of Pennsylvania, 2012. [Cited on pages 165 and 166.]

[144] W. Zhou, L. Ding, A. Haeberlen, Z. Ives, and B. T. Loo. TAP: Time-aware provenance for distributed systems. In *USENIX Workshop on the Theory and Practice of Provenance (TaPP)*, 2011. [Cited on page 121.]

[145] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure network provenance. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2011. [Cited on pages 2, 11, 23, 33, 39, 94, 98, and 126.]

[146] W. Zhou, S. Mapara, Y. Ren, Y. Li, A. Haeberlen, Z. Ives, B. T. Loo, and M. Sherr. Distributed time-aware provenance. In *International Conference on Very Large Databases (VLDB)*, 2013. [Cited on pages xi, 2, 11, 22, 94, 95, 99, 100, 101, 107, 111, 121, 130, and 133.]

[147] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at Internet-scale. In *ACM International Conference on Management of Data (SIGMOD)*, 2010. [Cited on pages 2, 11, 26, 33, 78, 94, and 111.]

[148] Zipkin, 2017. `http://zipkin.io/`. [Cited on pages 6, 92, 110, and 111.]