Modular Control Plane Verification via Temporal Invariants

TIMOTHY ALBERDINGK THIJM, Princeton University, United States
RYAN BECKETT, Microsoft Research, United States
AARTI GUPTA, Princeton University, United States
DAVID WALKER, Princeton University, United States

Monolithic control plane verification cannot scale to hyperscale network architectures with tens of thousands of nodes, heterogeneous network policies and thousands of network changes a day. Instead, *modular verification* offers improved scalability, reasoning over diverse behaviors, and robustness following policy updates. We introduce Timepiece, a new modular control plane verification system. While one class of verifiers, starting with Minesweeper, were based on analysis of stable paths, we show that such models, when deployed naïvely for modular verification, are unsound. To rectify the situation, we adopt a routing model based around a logical notion of time and develop a sound, expressive, and scalable verification engine.

Our system requires that a user specifies interfaces between module components. We develop methods for defining these interfaces using predicates inspired by temporal logic, and show how to use those interfaces to verify a range of network-wide properties such as reachability or access control. Verifying a prefix-filtering policy using a non-modular verification engine times out on an 80-node fattree network after 2 hours. However, Timepiece verifies a 2,000-node fattree in 2.37 minutes on a 96-core virtual machine. Modular verification of individual routers is embarrassingly parallel and completes in seconds, which allows verification to scale beyond non-modular engines, while still allowing the full power of SMT-based symbolic reasoning.

CCS Concepts: • **Networks** → **Protocol testing and verification**; **Formal specifications**; • **Theory of computation** → **Verification by model checking**; **Automated reasoning**.

Additional Key Words and Phrases: formal network verification, compositional reasoning, modular verification

ACM Reference Format:
Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. 2023. Modular Control Plane Verification via Temporal Invariants. *Proc. ACM Program. Lang.* 7, PLDI, Article 108 (June 2023), 26 pages. https://doi.org/10.1145/3591222

1 INTRODUCTION

Major cloud providers are seeing sustained financial growth in response to mounting demand for reliable networking [Miller 2022]. This demand suggests a commensurate network *infrastructure growth* will take place to accommodate more and more users. These networks can already have hundreds of data centers, each with hundreds of thousands of devices running thousands of heterogeneous policies, and receiving thousands of updates a day [Jayaraman et al. 2019b]. Network operators program this infrastructure using distributed routing protocols, where each router in a network may run thousands of lines of configuration code. Despite operators' care, routine

Authors' addresses: Timothy Alberdingk Thijm, Princeton University, Princeton, NJ, United States, tthijm@cs.princeton.edu; Ryan Beckett, Microsoft Research, Redmond, WA, United States, ryan.beckett@microsoft.com; Aarti Gupta, Princeton University, Princeton, NJ, United States, aartig@cs.princeton.edu; David Walker, Princeton University, Princeton, NJ, United States, dpw@cs.princeton.edu.

2475-1421/2023/6-ART108
https://doi.org/10.1145/3591222

configuration updates have inadvertently rendered routers unreachable [Strickx and Hartman 2022] or violated isolation requirements that prevent flooding [Vigliarolo 2022].

To prevent costly errors, operators can use *control plane verification* to analyze their networks [Abhashkumar et al. 2020; Beckett et al. 2017a, 2018, 2019; Fayaz et al. 2016; Gember-Jacobson et al. 2016; Lopes and Rybalchenko 2019; Prabhu et al. 2020; Weitz et al. 2016; Ye et al. 2020]. Until recently, research has focused on *monolithic* verification of the entire network at once, which is infeasible for large cloud provider networks. Such networks demand *modular* techniques that divide the network into components to verify in isolation. This approach has proven successful for software verification [Alur and Henzinger 1999; Flanagan and Qadeer 2003; Giannakopoulou et al. 2018; Grumberg and Long 1994; Henzinger et al. 1998] and network data plane verification [Jayaraman et al. 2019a]. We annotate the *interfaces* between network components with *invariants* that describe each component's routing behavior. Given the interfaces of a component's neighbors, we can verify that the component respects its own interface. When the interfaces imply a useful property, *e.g.,* reachability or access control, we can conclude that the monolithic network satisfies that property.

We propose Timepiece, the first modular technique with *abstract network interfaces to verify a wide range of properties* (including route reachability). Kirigami [Alberdingk Thijm et al. 2022a] proposed an architecture for modular control plane verification, but restricted its interfaces to only *exact* routes. Lightyear [Tang et al. 2022] presented an alternative verification technique with more expressive interfaces, but can only check that a network never receives a route (*e.g.,* for access control properties) — it cannot check reachability, a keen property of interest.

A temporal model. The basis of Timepiece's approach is a *temporal model of network execution*, where we reason over the states of nodes *at all times*. This model came as a surprise to us: one branch of prior work, starting with Minesweeper [Beckett et al. 2017a], sought to avoid the burden of reasoning over all transient states of the network by focusing on the *stable states* of the routing protocol once routing converges. Unfortunately, a naïve combination of modular reasoning and Minesweeper-style analysis of stable states *is unsound*. We discovered that the best way to recover soundness, while maintaining the system's generality, is to move to a temporal model.

This temporal model appears to ask the verification engine to do a lot more work: the system must verify that all the messages produced *at all times* are consistent with a user-supplied interface for each network component. Nevertheless, because reasoning is modular, ensuring individual problems are small, the system scales with the size of the largest *component* rather than the size of the network. This modular reasoning is general and any symbolic method (*e.g.,* symbolic simulation, model checking) could use it to verify individual components. We use a Satisfiability Modulo Theories (SMT)-based method in this work [Barrett and Tinelli 2018]. As a preview of modularity's benefits, Figure 1 shows the time it takes Timepiece to verify connectivity for variable-sized fattree topologies [Al-Fares et al. 2008] with external route announcements using the eBGP routing protocol, compared with a Minesweeper-style network-wide stable paths encoding.

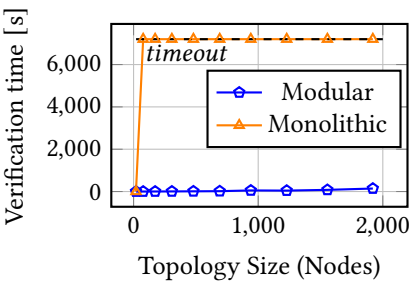


Fig. 1. Verification time comparison between Timepiece and Minesweeper-style verification.

Timepiece does require more work of users than monolithic, non-modular systems: users must supply interfaces that characterize the routes each network component may generate at each time. Still, these interfaces, once constructed, provide the typical benefits of interfaces in any

software engineering context. First, they localize exactly where an error occurs: if a component is not consistent with its interface, then one must search *only* that component for the mistake, and a counterexample from the SMT solver can help pinpoint it. Second, router configurations change rapidly, and these changes are often the source of network-wide problems [Zhang et al. 2022]. Well-defined interfaces will be stable over time. As users update their configurations, they may easily recheck them against the stable, local interface for problems.

Inspired by temporal logic [Pnueli 1984], we developed a simple language to help users specify their interfaces. Through this language, users may state that they expect to see certain sets of routes *always*, *eventually* (by some specified time t, to be more precise), or *until* (some approximate specified time). Moreover, the interface language allows users to write abstract specifications that need not characterize irrelevant features of routes, and instead only what is necessary to prove a desired property. For instance, a user might specify a reachability property simply by stating a node must "eventually receive some route at time t," without saying which route it must receive. Our formal model is based on a synchronous semantics of time, where nodes receive updates in lockstep. As discussed in prior work [Daggitt et al. 2018], this simplifies reasoning over the routing behavior of networks which converge to unique solutions. One may extend this model to consider a bounded number of steps of *delay* at the cost of increasing the complexity of our invariants.

To summarize, the key contributions of this paper are:

- We demonstrate in depth why a natural, but naïve modular control plane analysis based on an analysis of stable states is unsound (§2).
- We develop a new theory for modular control plane analysis based on time (§3). We prove it sound with respect to the semantics of a network simulator, and complete with respect to the closed network semantics (starting from fixed initial values). This theory is general, and can verify individual components using any verification method.
- We define an SMT-based verification procedure to reason about all possible routes at all times, which can analyze networks with symbolic representations of, *e.g.,* external announcements or destination routers. (§4)
- We design and implement a new, modular control plane verification tool, Timepiece, based directly on this procedure (§5). We evaluate Timepiece and check a variety of policies at individual nodes in hundreds of milliseconds. Thanks to its embarrassingly parallel modular procedure, Timepiece scales to networks with thousands of nodes (§6).

2 KEY IDEAS

This section introduces the stable routing model of network control planes, which serves as a foundation for many past network verification tools [Beckett et al. 2017a, 2018, 2019; Prabhu et al. 2020]. It illustrates in depth why naïvely adopting this model for modular verification is unsound. It then introduces a new temporal model for control plane verification and provides the intuition for why the revised model is superior. This section is long but contains a substantial payoff: the essence of why a sound and general modular control plane analysis should be based off a temporal model of control plane behavior.

2.1 Background

To determine how to deliver traffic between two endpoints, routers (also called nodes) run distributed routing protocols such as BGP [Lougheed and Rekhter 1991], OSPF [Moy 1998], RIP [Hedrick 1988], or ISIS [Oran 1990]. Each node participating in a protocol receives *messages* (also called *routes*) from its neighboring nodes. After receiving routes from its neighbors, a node will select its "best" route—the route it will use to forward traffic. Different protocols use different metrics to compare

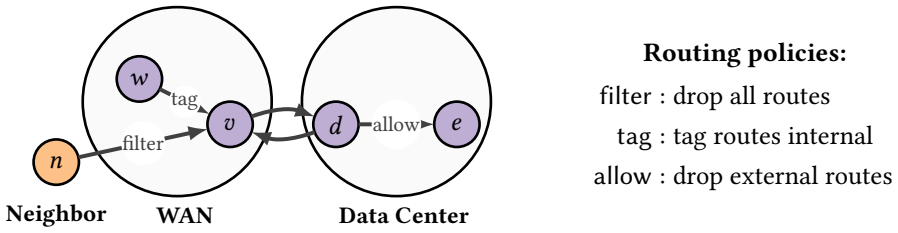


Fig. 2. Our idealized example cloud provider network.

routes and select the best among those received. For instance, RIP compares hop count; OSPF uses the shortest weighted-length path; and BGP uses a complex, user-configurable combination of metrics. Finally, each router sends its chosen route to its neighbors, possibly modifying the route along the way (for instance, by prepending its identifier to the path represented by the route).

Routing algebras. Routing algebras [Griffin and Sobrinho 2005; Sobrinho 2005] are abstract models that capture the similarities between different distributed routing protocols. Prior work on control plane verification [Beckett et al. 2017a; Giannarakis et al. 2020; Griffin et al. 2002] uses similar abstract models to formalize route computation. We adopt this standard abstract model of routing protocols, which specifies the following components.

- A directed graph G that defines the network topology's nodes (V) and edges (E). We use lowercase letters (u, v, w, *etc.*) for nodes and pairs (uv) to indicate directed edges.
- A set S of *routes* that communicate routing information between nodes. Routes abstract the routing announcements and metarouting information used in different routing protocols. Depending on the problem under consideration, S may be the set of Booleans $\mathbb{B}$ or natural numbers $\mathbb{N}$ (*e.g.,* checking reachability or path length properties), a set of flags (*e.g.,* checking access control), or a record with multiple fields (more complex policies and/or properties).
- An initialization function I that provides an initial route $I_v \in S$ for each node v.
- A function F that maps edges to transfer functions. Each transfer function $F(e) = f_e$ transforms routes as they traverse the edge e.
- A binary associative and commutative function $\oplus$ (*a.k.a. merge* or the *selection function*) selects the best route between two options.

An idealized example. Many large cloud providers deploy data center networks to scale up their compute capacity. They connect those data centers to each other and the rest of the Internet via a wide-area network (WAN). To illustrate the challenges of modular network verification, we will explore verification of an idealized cloud provider network with WAN and data center components. Figure 2 presents a highly abstracted view of our network's topology. The data center network contains routers d and e where d connects to the corporate WAN and e connects to data center servers. The WAN consists of routers w and v. Router v connects to the data center as well as to a neighboring network n, which is not controlled by our cloud provider.[1]

The default routing policy uses shortest-paths. However, in addition, the network administrators want e to be reachable from all cloud-provider-owned devices (*i.e.,* w, v, d), but not to be reachable from outsiders (*i.e.,* n). They intend to enforce this property by tagging all routes originating from their network (w) as "internal" (*e.g.,* using BGP community tags [CISCO 2005]) and allowing those

[1]Any of the edges could be bi-directional, allowing routes to pass in both directions, but for pedagogic reasons we strip down the example to the barest minimum, retaining edges that flow from left-to-right except for at v and d where routes may flow back and forth.

routes to traverse the *de* edge. Doing so should allow e to communicate with internal machines but not external machines. Furthermore, to protect nodes from outside interference, the cloud provider applies route filters to external peers to drop erroneous advertised routes that may "hijack" [Feamster and Balakrishnan 2005] internal routing.

Modelling the example. To model our example network, we define the network topology as the graph G pictured earlier. We assume all routers participate in an idealized variant of eBGP [Lougheed and Rekhter 1991], which is commonly used in both wide-area networks and data centers [Abhashkumar et al. 2021]. We abstract away some of the fields of eBGP routing announcements to define the set of routes S as records with 3 fields: *(i)* an integer "local preference" that lets users overwrite default preferences, *(ii)* an integer path length, and *(iii)* a boolean tag field that is set to true if a route comes from an internal source and false otherwise. S also includes ∞, a message that indicates *absence* of a route.

Let's consider what happens when starting with a specific route at WAN node w, $\langle 100, 0, \text{false}\rangle$ (local preference 100, path length of 0, not tagged internal). The I function assigns w that route, and assigns the ∞ route to all other nodes.

The transfer function f_e increments the length field of every route by one across every edge e. In addition, edge wv sets the internal tag field to true and edge nv drops all routes (transforms them into ∞). Finally, edge de drops all routes not tagged internal/true.

The merge function $\oplus$ always prefers some route over the ∞ route, and prefers routes with higher local preference over lower local preference. If the local preference is the same, it chooses a route with a shorter path length. $\oplus$ ignores the tag field. For example, $\oplus$ operates as follows:

$$\begin{array}{lcll} \langle 100, 2, \text{false}\rangle & \oplus & \infty & = \langle 100, 2, \text{false}\rangle \\ \langle 100, 2, \text{false}\rangle & \oplus & \langle 200, 5, \text{true}\rangle & = \langle 200, 5, \text{true}\rangle \\ \langle 200, 2, \text{false}\rangle & \oplus & \langle 200, 5, \text{true}\rangle & = \langle 200, 2, \text{false}\rangle \end{array}$$

Network simulation. A *state* of a network is a mapping from nodes to the "best routes" they have computed so far. One may simulate a network by starting in its initial state and repeatedly computing new states (*i.e.,* new "best routes" for particular nodes). Well-behaved networks eventually converge to *stable states* where no node can compute a better route, given the routes provided by its neighbors.

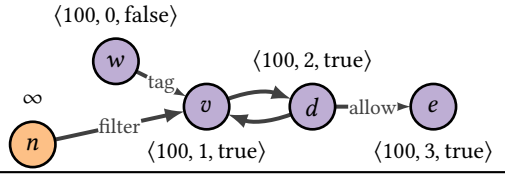
To compute a new best route at a particular node, say v, we apply the f function to each best route computed so far at its neighbors w, n, and d, and then select the best route among the results and the initial value at v, using the merge ($\oplus$) function. More precisely:

$$v_{new} = f_{wv}(w_{old}) \oplus f_{nv}(n_{old}) \oplus f_{dv}(d_{old}) \oplus I_v$$

The table in Figure 3 presents an example simulation. At each time step, all nodes compute their best route given the routes sent by their neighbors at the previous time step. Our model assumes a synchronous time semantics for simplicity: this simulation is hence one possible asynchronous execution.[2] After time step 3, no node computes a better route—the system has reached a *stable state*. The picture in Figure 3 annotates each node in the diagram with the stable route it computes.

Network verification. Since the edge from d to e only allows routes tagged internal, w's route would not reach e if v were to receive a better route from n (*e.g.,* if the route filter from n was implemented incorrectly). In other words, the simulation demonstrates that the network correctly operates when n sends no route (∞). But what about other routes? Will f_{nv} filter all routes from n correctly? SMT-based tools like Minesweeper [Beckett et al. 2017a] and Bagpipe [Weitz et al. 2016] can answer such questions by translating the routing problem into constraints for a Satisfiability Modulo Theory (SMT) solver to solve. An SMT-based encoding of our network could represent

[2]See §4 for a discussion of how we can extend our model to consider networks with delay.



time	n	w	v	d	e
0	∞	⟨100, 0, false⟩	∞	∞	∞
1	∞	⟨100, 0, false⟩	⟨100, 1, true⟩	∞	∞
2	∞	⟨100, 0, false⟩	⟨100, 1, true⟩	⟨100, 2, true⟩	∞
3	∞	⟨100, 0, false⟩	⟨100, 1, true⟩	⟨100, 2, true⟩	⟨100, 3, true⟩
4	∞	⟨100, 0, false⟩	⟨100, 1, true⟩	⟨100, 2, true⟩	⟨100, 3, true⟩

Fig. 3. Simulation of the example network for a fixed set of initial routes. Node *e* receives a route from *d* since its route is tagged as internal, and the network stabilizes at time 3.

any possible external route announcement from n by representing its initial value with a symbolic variable: the solver can then search for a concrete route captured by this variable that *violates* our desired property, *i.e.,* a stable state where e never receives a route from w.

2.2 The Challenge of Modular Verification

A system for modular verification will partition a network into components and verify each component separately, possibly in parallel. However, since routes computed at a node in one component depend on the routes sent by nodes in neighboring components, each component must make some assumptions about the routes produced by its neighbors.

Interfaces. In our case, for simplicity (though this is not necessary), we place every node in its own component and define for it an *interface* that attempts to *overapproximate* (or equal) the set of routes that the node might produce in a stable state. The interface for the network as a whole is a function A from nodes to sets of routes where $A(x)$ is the interface for node x.

The person attempting to verify the network will supply these interfaces. Of course, interfaces may be *wrong*—that is, they might not include some route computed by a simulation (and hence might not be a proper overapproximation). Indeed, when there are bugs in the network, the interfaces a user supplies are likely to be wrong! The user *expects* the network to behave one way, producing a certain set of routes, but the network behaves differently due to an error in its configuration. A sound modular verification procedure must detect such errors and indicate if we must strengthen the interface to prove the property. On the other hand, a useful modular verification procedure should allow interfaces to overapproximate the routes produced, when users find it convenient. Overapproximations are sound for verifying properties over all routing behaviors of a network, and they often simplify reasoning, allowing users to think more abstractly.

Throughout the paper, we use predicates φ to define interfaces, where φ stands in for the set of routes $\{s \mid s \in S, \varphi(s)\}$. Returning to our running example, one might define the interface for w using the predicate $s.\text{lp} = 100 \wedge s.\text{len} = 0 \wedge \neg s.\text{tag}$. Such an interface would include exactly the one route generated by w in our example: $\langle 100, 0, \text{false}\rangle$. However, path length is unimportant in the current context; to avoid thinking about it, a user could instead provide a weaker interface representing infinitely many possible routes, such as $s.\text{lp} = 100 \wedge \neg s.\text{tag}$. This interface relieves the user of having to figure out the exact path length (not so hard in this simple example, but potentially challenging in an arbitrary wide-area network), and instead specifies only the local

preference and the tag. In general, admitting overapproximations make it possible for users to ignore any features of routing that are not actually relevant for analyzing the properties of interest.

The strawperson verification procedure. For a given node x, the *component centered at* x is the subgraph of the network that includes node x and all edges that end at x. Given a network interface A, our strawperson verification procedure (SV) will consider the component centered at each node x independently. Suppose a node x has neighbors $n_1, \ldots, n_k$. For that node x, SV checks that

$$\forall s_1 \in A(n_1), \ldots, \forall s_k \in A(n_k),\ f_{n_1 x}(s_1) \oplus \cdots \oplus f_{n_k x}(s_k) \oplus I_x \in A(x) \tag{1}$$

This check is akin to performing one local step of simulation, checking that all possible inputs from neighbors produce an output route satisfying the interface. We might *hope* that by performing such a check on *all* components independently, we could guarantee that all nodes converge to stable states satisfying their interfaces. If that were the case, then we could verify properties by:

(1) Checking that all components guarantee their interfaces, under the assumption their neighbors do as well; and
(2) Checking that the interfaces imply the network property of interest (*e.g.,* reachability, access control, no transit).

The problem: execution interference. It turns out this simple and natural verification procedure is unsound: users can supply interfaces that, when analyzed in isolation, satisfy equation (1) above, but *exclude* stable states computed by simulation. Hence, the second verification step is pointless: a destination that appears reachable according to an interface may not be; conversely, a route that appears blocked may not be.

Let us reconsider the running example, where we assign w an initial route with local preference 100, and assume the external neighbor n can send us any route (true). A user could provide the interfaces shown in Figure 4 to falsely conclude that e will not receive a route from w.

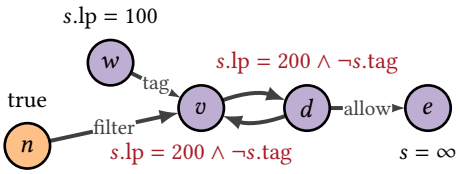


Fig. 4. Running example with bad interfaces.

Here, it is easy to check that nodes n and w satisfy equation (1). Node n's interface is simply any route. Node w's route can be any route with a local preference of 100.[3]

The surprise comes at node v where its interface *only includes* routes that satisfy $\neg s.\text{tag}$, *i.e.,* routes not tagged as internal. Those routes have $s.\text{lp} = 200$ and may have any path length. But the route from w is tagged *true* along the edge wv — why is such a route erroneously excluded from v's interface? We show the component centered at v in Figure 5.

[3]It could be any route (true), as the edge wv applies the default preference of 100, but for clarity we label the routes at w with preference 100.

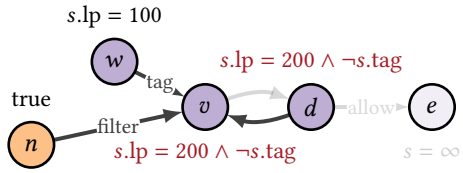


Fig. 5. Running example centered on v's component.

When computing its stable state, v will compare the routes it receives from w and d: because all routes from w have a local preference set to 100 by f_{wv}, whereas all routes from d have a better local preference of 200, v will always wind up selecting the route from d over the route from w.

But how then did d acquire these preferential routes tagged false? Such routes came in turn from v's interface. Figure 6 shows the component centered at d.

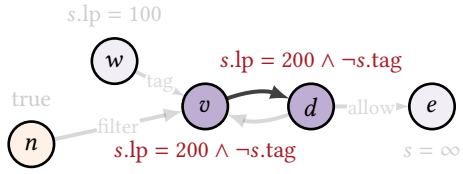


Fig. 6. Running example centered on d's component.

What has happened is that v transmits its spurious routes to d, enabling d to justify its own spurious routes. d transmits these back again to v, where d's routes interact with the legitimate routes from w. Since w's routes have lower local preference, v discards them during computation of stable states. In a nutshell, our interface proposed routes that do *not* soundly overapproximate the legitimate routes from the true simulation, but our verification procedure *accepted* this bad interface as it circularly justified itself at v and d. How might we prevent this *execution interference*?

Other approaches. We can modify this verification procedure to make it sound, but these solutions will limit the verification procedure's power or the expressiveness of the properties it can prove.

One approach is to limit every interface to *exactly one route*. Doing so avoids introducing any imaginary executions in the first place. Kirigami [Alberdingk Thijm et al. 2022a] takes this approach, but the cost is that a user analyzing their network must know *exactly* which routes appear at which locations. Computing routes exactly can be difficult in practice, and would seem unnecessary if all one cares about is a high-level property such as reachability. Moreover, it makes the interfaces brittle in the face of change—any change in network configuration likely necessitates a change in interface. A superior system would allow operators to define *durable* and *abstract* interfaces that imply key properties, and to check configuration updates against those interfaces.

Another approach is to limit the set of properties that the system can check to only those that say what *does not happen* in the network rather than what does happen. This is the approach Lightyear [Tang et al. 2022] takes. For instance, Lightyear can check that node a will *not* be able to reach node b, but not that a and b will have connectivity — a common requirement in networks.

A final approach is to *statically order* the components, and verify each component according to this ordering, using no information from the not-yet-verified components. By ordering v before d, we would need to satisfy v's invariant without routes from d (treating d's invariant as false) — this would fail for the bad invariant $s.\text{lp} = 200 \wedge \neg s.\text{tag}$ using only routes from n and w. In practice,

this is still unnecessarily conservative. The running example is overly simple as it shows routes propagated through a network in a single direction from left to right. In realistic networks, multiple destinations may broadcast routes in multiple directions at once. In such situations, there may be no way to order the components, and verification may not be possible.

2.3 The Solution: A Temporal Model

Our key insight is to change the model: rather than focus exclusively on the final stable states of a system, as a Minesweeper-style verifier would, we ensure that the model preserves the *entirety* of every step-by-step execution. To make this work, we need to add information to the model: a notion of *logical time*. By associating every route with the time at which a node computes it, we can *(i)* ensure that *all* routes at a particular time are properly considered, and their executions extended a time step, and *(ii)* ensure that we avoid collisions between routes computed at different times.

To verify such routing systems modularly, we once again must specify interfaces, but this time the interface for each node will specify the set of routes that may appear *at any time*. We write this now as an interface $A(x)(t)$ that takes both a node x *and a time* t and returns an overapproximation of the set of routes that may appear at x *at that time* t. To check the interfaces, we use a verification procedure structured inductively with respect to time, as follows:

- At every node x, check I_x is included in $A(x)(0)$
- Consider each node x with neighbors $n_1, \ldots, n_k$. At time $t+1$, check that merging any combination of routes $s_1 \in A(n_1)(t), \ldots, s_k \in A(n_k)(t)$ from neighbors' interfaces at time t produces a route in $A(x)(t+1)$:

$$f_{n_1 x}(s_1) \oplus \cdots \oplus f_{n_k x}(s_k) \oplus I_x \in A(x)(t+1) \tag{2}$$

Because this procedure is structured inductively, we can prove, by induction on time, that all states at all times are included in their respective interfaces—the procedure is *sound*.

For brevity, we specify our interfaces using *temporal operators*. These operators are functions that take a time t as an argument, compare it to an explicit time variable τ, and return a predicate. We write $\mathcal{G}(P)$ ("globally P") when a node's interface includes the routes that satisfy predicate P for all times t. We write $P_1 \; \mathcal{U}^\tau \; Q_2$ ("P_1 until Q_2") when a node may have routes satisfying P_1 until time $\tau - 1$ and operator $Q_2(\tau)$ holds afterwards. Finally, we write $\mathcal{F}^\tau(Q)$ ("finally Q") to mean that eventually at time τ routes start satisfying $Q(\tau)$.

Verifying correct interfaces. Figure 7 below presents an interface we may verify with this model.

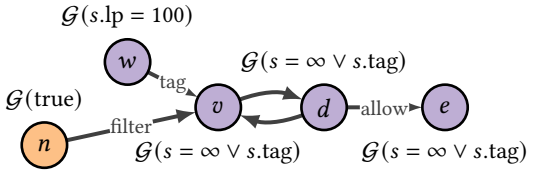


Fig. 7. Running example with interfaces proving that if e has a route, it is tagged.

We again assume that n sends any route at any time, denoted by the interface: $\mathcal{G}(\text{true})$. We assume w has a route with default local preference: $\mathcal{G}(s.\text{lp} = 100)$. The interesting part is at nodes v and d where the interfaces state that there is always no route (*e.g.,* at time 0), or a tagged route: $\mathcal{G}(s = \infty \vee s.\text{tag})$. We can then prove a weak property about node e: *if it receives a route*, then the route will be tagged internal. We can prove node v's route satisfies its interface since f_{wv} tags routes

on import from node w, routes from n are correctly dropped, and routes from d must also have a tag per its interface. In fact, all the nodes satisfy their interface given their neighbors' interfaces.

Proving reachability. Figure 7's interfaces were too weak to prove that w can reach e. The problem is that they reason about *all* times (*i.e.,* from time 0 onward), yet e will only *eventually* have a route from w at some time in the future. Consider now the stronger interfaces shown in Figure 8:

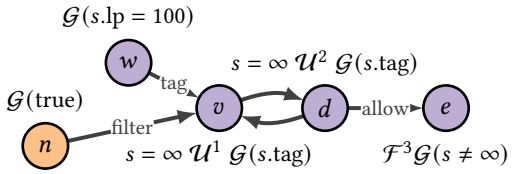


Fig. 8. Running example with interfaces proving e can reach w.

As before, we allow n and w to send any route. However, now nodes v and d declare that they will not have a route *until* a specified (logical) time, at which point they receive a tagged route. We give precise witness times for v and d's interfaces, as otherwise v could give d a non-null route (or vice-versa) that would violate the interface before its witness time. e's interface simply requires that e receives some route at the witness time (allowing arbitrary routes before the witness time). These interfaces are sufficient to prove that e will eventually receive a route to w, since d will eventually have a route tagged as internal, and hence e will allow it.

Debugging erroneous interfaces. Let us revisit the example where a user gave unsound interfaces using spurious routes with local preference 200. Figure 9 presents the equivalent temporal interfaces.

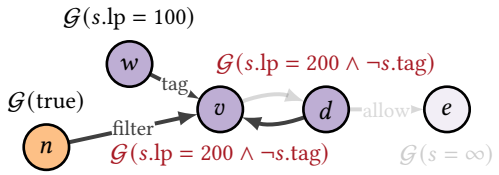


Fig. 9. Running example with bad temporal interfaces.

Unlike before, the verification procedure detects an error: the interfaces at nodes v and d do not include the initial route ∞ at time 0. As a result, the user will receive a counterexample for time $t = 0$ when verifying v or d. Suppose our imaginative user tries to circumvent this issue by also including the initial route in the interfaces for v and d with the interface:

$$\mathcal{G}\big((s.\text{lp} = 200 \wedge \neg s.\text{tag}) \vee (s = \infty)\big)$$

However, doing so merely pushes the problem "one step forward in time"—there is no way to circumvent our temporal analysis. If d's route may be ∞, v's interface must also consider what routes it selects when that is the case, including tagged routes such as $\langle 100, 1, \text{true}\rangle$. The user might receive a counterexample at time $t = 1$ where v's route is the following:

$$f_{wv}(\langle 100, 0, \text{false}\rangle) \oplus f_{nv}(\infty) \oplus f_{dv}(\infty) = \langle 100, 1, \text{true}\rangle$$

where $A(v)(1)$ does not contain the result $\langle 100, 1, \text{true}\rangle$. This counterexample reveals the fact that there is an error in either the specification (as in this case) or the configuration (*e.g.,* if a buggy configuration tagged routes from w false rather than true as expected).

Table 1. Ghost state for selected example properties.

Property	Added ghost state
reachability to d [Fogel et al. 2015]	*1 bit to mark routes from d*
isolation [Beckett et al. 2017a]	*1 bit per isolation domain*
ordered waypoint [Kazemian et al. 2013].	$log_2(k)$ *bits for k waypoints*
unordered waypoint [Beckett et al. 2017a]	*k bits for k waypoints*
routing loops [Beckett et al. 2017a]	*up to* $\|V\|$ *bits to track visited nodes*
no-transit [Beckett et al. 2016]	*mark with* {peer, prov, cust}
fault tolerance [Beckett et al. 2017a]	*up to* $\|E\|$ *bits to track failed edges*
bounded path length [Lopes et al. 2015]	*integer length field*

Properties and ghost state. Although our modular properties reference node-local routes, we can verify many end-to-end control plane properties using *ghost state*. Users may model routes with additional "ghost" fields (*cf.* ghost fields in Dafny [Leino 2010]) that play no role in a protocol's routing behavior, yet may capture end-to-end properties. For instance, suppose we added a boolean "ghost" field "fromw" to indicate if a route originated from node w (see Figure 10). We assume this field is initially true at w, false at all other nodes, and that transfer functions preserve its value. With this addition, we can now check that e receives a route from w and no other node.

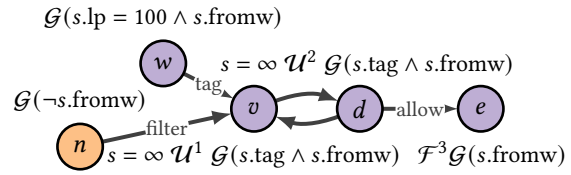


Fig. 10. Running example augmented with a "fromw" ghost variable.

Ghost state allows us to specify and check many network properties; Table 1 presents a variety of other possibilities. For example, to check for routing loops, we can use a multi-set of visited nodes (up to $|V|$) and mark if a node is ever visited more than once. That said, while ghost state is general and flexible, it can only capture information about the history of a route at a *single* node and is thus not a panacea. For instance, properties involving the routes at more than one node, such as a formulation of local equivalence [Beckett et al. 2017a], where $\sigma(u)(t) = \sigma(v)(t)$ for some arbitrary u, v and t, is inexpressible using our verifier. We focus on properties of the network's *control plane*, without considering the data plane forwarding behavior. This rules out data plane properties such as load balancing and multipath consistency.

3 FORMAL MODEL WITH TEMPORAL INVARIANTS

Figure 11 presents the key definitions and notation needed to formalize our verification procedure. The notation follows from the previous section, *e.g.*, a network instance $N = (G, S, I, F, \oplus)$ contains the key components introduced earlier. To refer to the route computed by a network simulator at node v at time t, we use the notation $\sigma(v)(t)$ (defined as before—see Figure 11).

Figure 12 presents our interfaces and language of temporal operators. As before, we use A to denote network interfaces. The temporal operators Q are syntactic sugar for *functions* from a time t to a predicate. $\varphi\ \mathcal{U}^\tau\ Q$ and $\mathcal{F}^\tau(Q)$ take a concrete natural number τ as a *witness time* parameter

Network instances $\boxed{N = (G, S, I, F, \oplus)}$

$G = (V, E)$	network topology
V	topology nodes
$E \subseteq V \times V$	topology edges
S	set of network routes
$s \in S$	a route
$I : V \to S$	node initialization function
$I_v \in S$	initial route at node v
$F : E \to (S \to S)$	edge transfer functions
$f_e : S \to S$	transfer function for edge e
$\oplus : S \times S \to S$	merge function

Network semantics $\boxed{\sigma : V \to (\mathbb{N} \to S)}$

$\sigma(v)(t) \in S$	state at node v at time t
$preds(v) = \{u \mid u \in V, uv \in E\}$	in-neighbors of v

$$\sigma(v)(0) = I_v \tag{3}$$

$$\sigma(v)(t+1) = I_v \oplus \bigoplus_{u \in preds(v)} f_{uv}(\sigma(u)(t)) \tag{4}$$

Fig. 11. Summary of our formal routing model and notation.

to compare against the argument time t. These witness times are *absolute* times, specifying the time steps since the initial time 0. We can nest $\mathcal{U}^\tau$ and $\mathcal{F}^\tau$ operators to represent arbitrarily-many *intervals* of time, *e.g.*, $\mathcal{F}^2(\varphi_1\ \mathcal{U}^4\ \mathcal{G}(\varphi_2))$ is a function that returns S (true) given input time 0 or 1; φ_1 given time 2 or 3; and φ_2 given a time of 4 or more. We also lift set union, intersection and negation to succinctly combine temporal operators. In our evaluation (§6), we found this (intentionally small) language sufficient to express a wide variety of reachability and security properties.

A valid interface is an *inductive invariant* [Giannakopoulou et al. 2018]. Such interfaces satisfy the *initial and inductive conditions* specified in Figure 12. Valid interfaces may be used to prove node properties, as specified by the *safety condition* in Figure 12. As the network has a finite number of nodes, we can enumerate them to check these three conditions on every node in the network.

The most important property of our system is *soundness*: the simulation states are included in any interface A that satisfies the initial and inductive conditions.

Theorem 3.1 (Soundness). *Let A satisfy the initial and inductive conditions for all nodes. Then A always includes the simulation state σ, meaning $\forall v \in V, \forall t \in \mathbb{N}, \sigma(v)(t) \in A(v)(t)$.*

Proof. By induction on t. See [Alberdingk Thijm et al. 2022b] for the full proof. □

Since initial and inductive conditions suffice to prove that simulation states are included within interfaces, it is safe in turn to use interfaces to check node properties.

Corollary 3.2 (Safety). *Let A satisfy the initial and inductive conditions for all nodes. Let P satisfy the safety condition with respect to A for all nodes. Then $\forall v \in V, \forall t \in \mathbb{N}, \sigma(v)(t) \in P(v)(t)$.*

Proof. From definitions. See [Alberdingk Thijm et al. 2022b]. □

A *closed network* is a network instance where all initial routes are fixed routes, such that all states of the network σ are captured by a concrete simulation. Our verification procedure is *complete* with

Interfaces, Properties and Metavariables

$$\begin{array}{lll} A & : V \to (\mathbb{N} \to 2^S) & \text{node interfaces/invariants} \\ P & : V \to (\mathbb{N} \to 2^S) & \text{node properties} \\ \varphi & : 2^S & \text{sets of states} \\ \tau & : \mathbb{N} & \text{witness times} \end{array}$$

Temporal operators $\boxed{Q : \mathbb{N} \to 2^S}$

$$\begin{array}{lcll} \mathcal{G}(\varphi) & = & \lambda\, t.\, \varphi & \text{globally} \\ \varphi\ \mathcal{U}^\tau\ Q & = & \lambda\, t.\, \mathit{if}\ t < \tau\ \mathit{then}\ \varphi\ \mathit{else}\ Q(t) & \text{until} \\ \mathcal{F}^\tau(Q) & = & S\ \mathcal{U}^\tau\ Q & \text{finally} \\ Q_1 \sqcap Q_2 & = & \lambda\, t.\, Q_1(t) \cap Q_2(t) & \text{intersection (lifted)} \\ Q_1 \sqcup Q_2 & = & \lambda\, t.\, Q_1(t) \cup Q_2(t) & \text{union (lifted)} \\ \sim Q & = & \lambda\, t.\, S \setminus Q(t) & \text{negation (lifted)} \end{array}$$

Verification Conditions

Initial condition for node v:

$$I_v \in A(v)(0) \tag{5}$$

Inductive condition for node v with in-neighbors $u_1, u_2, \ldots, u_n$:

$$\forall t \in \mathbb{N}, \forall s_1 \in A(u_1)(t), \forall s_2 \in A(u_2)(t), \ldots, \forall s_n \in A(u_n)(t),$$
$$\left(I_v \oplus \bigoplus_{i \in \{1,..,n\}} f_{u_i v}(s_i) \right) \in A(v)(t+1) \tag{6}$$

Safety condition for node v:

$$\forall t \in \mathbb{N}, A(v)(t) \subseteq P(v)(t) \tag{7}$$

Fig. 12. Summary of our interfaces and properties, temporal operators and verification conditions.

respect to a closed network: for any closed network, there exists an interface that characterizes its simulation states exactly. One of the consequences of completeness is that our modular verification procedure is powerful enough to prove any property that we could prove via a concrete simulation.[4]

Theorem 3.3 (Closed Simulation Completeness). *Let σ be the state of the closed network. Then for all $v \in V$ and $t \in \mathbb{N}$, $A(v)(t) = \{\sigma(v)(t)\}$ satisfies the initial and inductive conditions for all nodes.*

Proof. By construction of the interface. See [Alberdingk Thijm et al. 2022b]. □

4 SMT ALGORITHMS FOR VERIFICATION

We can check our initial, inductive and safety verification conditions (VCs) independently for every node in the network using off-the-shelf SMT solvers. To check an instance of a VC, the solver will attempt to prove the condition is valid (*i.e.,* true for all choices of t and $s_1, s_2, \ldots, s_n$) by checking whether the *negation* of our original VC is satisfiable. If the solver can satisfy the negation, it will provide us with a *counterexample*—the state(s) of the node(s) at a particular time such that the VC does not hold. Counterexamples to initial or inductive conditions indicate that the interface does not capture the network's behavior, while a counterexample to the safety condition indicates that the interface is not strong enough to prove the property. The latter case may occur because the property is simply not true (indicating a bug), or because we must strengthen the given interface to prove the property. If the negation is unsatisfiable, then we know the condition is valid.

[4]In open networks, an external node may take an arbitrary route at any point in time: this case is not captured by σ.

Algorithm 1 The modular checking algorithm.

```
proc CheckMod(network (G, S, I, F, ⊕), interface A, property P)
  for all v ∈ V do in parallel
    ψ1 ← EncodeInitCond(v, S, I, A)                          ▷ (5)
    ψ2 ← EncodeIndCond(v, preds(v), S, I, F, ⊕, A)           ▷ (6)
    ψ3 ← EncodeSafeCond(v, S, A, P)                          ▷ (7)
    for i ← 1, 2, 3 do
      if ¬IsValid(ψi) then return GetCounterExample(ψi)
  return Success
```

A network instance's routes and behavior determines its encoding in SMT. For example, one could encode the route triples in §2 as integer variables lp and len and a Boolean variable tag, and use Presburger arithmetic to encode F and $\oplus$ with + and <. To model networks with external peers (like n), multiple routing destinations, or other sources of nondeterminism, we may use *symbolic variables*. For external peers, rather than treating n as having a specific initial route such as ∞, we may use a symbolic variable s for I_n, and then ask the SMT solver to check the VCs *for all* choices of s. For multiple destinations, we can use a symbolic node variable to choose from a set of destination nodes when checking that any node in that set is reachable. We also can *assume* arbitrary preconditions for symbolic variables, *e.g.*, enforcing that s is not tagged with $\neg s$.tag. These assumptions are not checked when we encode them to SMT.

The modular checking procedure. We present our modular checking procedure in Algorithm 1. The CheckMod procedure iterates over each node of the network and encodes the underlying formula (for the current node) of our three verification conditions by calling EncodeInitCond, EncodeIndCond and EncodeSafeCond as defined in (5), (6) and (7), respectively.[5] Note that we encode t as an explicit symbolic variable: our temporal operators expand to a case analysis over this variable to determine what predicate holds on the particular node's route. We then ask the solver if every encoded formula is valid using IsValid. If IsValid returns false for any check, we ask for the relevant counterexample using GetCounterExample, which returns a variable assignment that violates the formula. Otherwise, we report success (A and P hold).

Encoding the initial and inductive conditions is roughly proportional in size to the complexity of the policy at the given node, which in turn is related to the in-degree of the node—denser networks that include nodes with higher in-degree are more expensive to check. Encoding the safety condition is proportional to the size of the formulae describing the interface and property (generally tiny). In addition to reducing the size of each SMT formula, the factoring of the problem into independent conditions makes it possible to check conditions on nodes *simultaneously in parallel*. We will discuss the performance implications of our procedure further in §6.

Handling counterexamples. When GetCounterExample returns a counterexample for a modular check, it specifies (a) routes of the node(s); (b) a concrete time; and (c) concrete values of any symbolic variables in the formula. These counterexamples can guide us in strengthening the invariants or pinpointing bugs. This is akin to other invariant-checking tools like Dafny [Leino 2010] where users must manually refine their invariants. Anecdotally, our practical approach to designing the interfaces in §6 was to start by choosing $A(v)(t) = P(v)(t)$ (trivially satisfying our safety condition). A counterexample would then identify a time instance where v's invariant

[5] For simplicity, our encoding uses predicates $V \to (\mathbb{N} \to (S \to \mathbb{B}))$ to represent A and P—other than this change, the formulae are identical to the given VCs.

violated its initial or inductive condition. To resolve this violation, we often had to add a $\mathcal{G}(\varphi)$ invariant to capture additional behavior (*e.g.,* no node receives a better route than the legitimate route). This "rule-of-thumb" suggests that counterexample-guided techniques [Clarke et al. 2000] may be capable of *inferring invariants*. We leave this as future work.

Incorporating delay. σ and A define a synchronous network semantics. As shown in prior work on routing algebras [Daggitt et al. 2018; Griffin and Sobrinho 2005], when F and $\oplus$ are *strictly monotonic* — informally, $\oplus$ prefers a route r over any transferred route $f_e(r)$ — there will only be one converged state of the network, which our synchronous model will certainly capture.[6] In other cases, the synchronous model captures a possible execution of the network.

We may extend our model to consider routes up to a bounded number of units of delay. To account for one unit of delay, we can extend our inductive condition to check *all* routes sent in the last *two* time steps t and $t+1$ satisfy the invariant at time $t+2$, becoming (changes in boxes):

$$\forall t \in \mathbb{N}, \forall s_1 \in \boxed{A(u_1)(t) \cup A(u_1)(t+1)}, \ldots, \forall s_n \in \boxed{A(u_n)(t) \cup A(u_n)(t+1)},$$
$$\left(I_v \oplus \bigoplus_{i \in \{1,..,n\}} f_{u_i v}(s_i)\right) \in \boxed{A(v)(t+2)}$$

We may extend the condition further to consider more units of delay. Doing so may also increase the complexity of our invariants.

5 IMPLEMENTATION

We implemented Timepiece's modular verification procedure as a library written in C#. The library allows users to construct models of networks and then modularly verify them. Like the network modelling framework NV [Giannarakis et al. 2020], Timepiece allows users to customize their models by providing the type of routes (which may involve integers, strings, booleans, bitvectors, records, optional data, lists, or sets) and the initialization, transfer and merge functions that process them. This modelling language makes it easy to add ghost state to routes, as described earlier. It is also possible to declare and use symbolic values in the model. Hence, one may reason about all possible prefixes or more generally about all possible external routing announcements.

Under the hood, Timepiece uses Microsoft's Zen verification library [Beckett and Mahajan 2020] to generate SMT formulas from higher-level C# data structures and pass these formulas to the Z3 SMT solver [De Moura and Bjørner 2008]. Timepiece thus supports any network model that Zen can encode to Z3. For example, to model a network running eBGP, we would adopt many of the modelling choices made in Minesweeper [Beckett et al. 2017a]. We can use integers and Presburger arithmetic to model path length, and bitvectors to model local preference and MED.

Timepiece uses multi-threading to run modular checks in parallel.[7] As each check is independent, the time to set up additional threads is the only overhead for parallelization.

6 EVALUATION

To evaluate Timepiece and illustrate its scaling trends, we generated a series of synthetic fattree [Al-Fares et al. 2008] data center networks and verified four variations on reachability properties. We also verified an isolation property on a real wide-area network configuration with over 100,000 lines of Junos configuration code. These two types of networks demonstrate Timepiece's performance

[6] As common protocols (*e.g.,* OSPF) rely on shortest-paths algorithms with strictly monotonic F and $\oplus$, prior work has sometimes assumed the network converges to a unique solution [Gember-Jacobson et al. 2016; Lopes and Rybalchenko 2019].

[7] We use C#'s Parallel LINQ library [Microsoft 2021], which can run up to 512 concurrent threads.

Table 2. Lines of C# code to define the network instances, interfaces and properties for each benchmark. †: BlockToExternal's network instance is defined partly from Internet2's configuration files (topology G and F functions); the remaining elements (I, $\oplus$ and symbolic variables) are defined in C#. The reported lines are for the C# portion—the configuration files are over 100,000 lines of Junos configuration code.

Benchmark	Network LoC	Interface LoC	Property LoC
Reach	79	3	2
Len	83	7	5
Vf	87	12	2
Hijack	142	21	4
BlockToExternal	83†	5	5

for networks which are highly-connected (data centers) and have complex policies (WANs). Table 2 shows that writing the interfaces for each of our benchmarks is low-effort compared to the rest of the network in terms of lines of code. We generated interfaces for our experiments parametrically for any size of network, based on the distinct *roles* of nodes: for fattree networks, a node's pod and tier determined its role (5 roles, discussed later); for our wide-area network benchmark, we distinguished internal nodes from external neighbors (2 roles). As a node's role determined its invariant, it is easy to *extend or update* these networks (*e.g.,* adding a new pod or external neighbor) and reuse the appropriate existing invariant for a node of that role.

To compare our implementation against a baseline, we implemented a monolithic, network-wide Minesweeper-style [Beckett et al. 2017a] procedure Ms and compared its performance against Timepiece. Ms analyzes stable states, which are independent of time. Given a property P over stable states, Ms checks if P always holds (is valid) given the stable states of the network. These states are encoded as a single formula over all nodes, as described in §2.1. To compare Ms with Timepiece, we first crafted properties for Timepiece, which employs timed invariants. We then erased the temporal details from these invariants to generate properties that Ms could manage. For instance, when Timepiece would verify properties of the form $\mathcal{G}(\varphi)$, $\mathcal{F}^t\mathcal{G}(\varphi)$, or $\varphi_2\ \mathcal{U}^t\ \mathcal{G}(\varphi)$, Ms would instead verify that the network's stable states satisfy φ.[8]

We ran our benchmarks on a Microsoft Azure D96s v5 virtual machine with 96 vCPUs and 384GB of RAM. We used the machine's multi-core processor to run all modular checks in parallel, while monolithic checks necessarily ran on a single thread. We timed out any benchmark that did not complete in 2 hours. We report four times for each benchmark: *(i)* the total time until all Timepiece threads finished (Tp); *(ii)* the median node check time; *(iii)* the 99$^{\text{th}}$ percentile node check time (99% of checks completed in less than this much time); and *(iv)* the total time taken by Ms.

Fattrees. We parameterize our fattree networks by the number of pods k: a k-fattree has $1.25k^2$ nodes and k^3 edges: Figure 13 shows an example 4-fattree used in our Vf benchmark. We considered multiples of 4 for $4 \leq k \leq 40$ to assess Timepiece's scalability: whereas we expected a monolithic verifier to time out on larger topologies, we hypothesized that Timepiece would scale to these networks. We present how verification time grows with respect to the number of nodes in each fattree in Figure 14. The figure shows verification time on the y-axis on a logarithmic scale.

We considered four different properties (explained below): reachability (Reach), bounded path length (Len), valley freedom (Vf) and route filtering (Hijack). We considered each property when routing to a single, fixed destination edge node *dest* (Sp), and routing to a *dest* edge node determined

[8]None of our properties required that we specify more than one witness time, so all $\mathcal{F}$ and $\mathcal{U}$ operators took a $\mathcal{G}$ operator as their temporal argument.

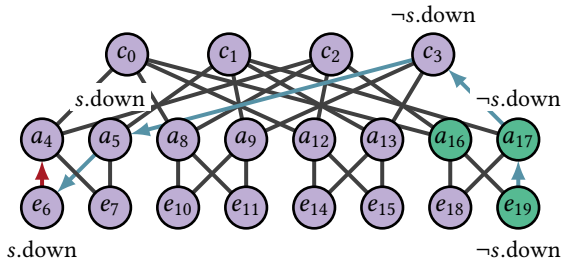


Fig. 13. An example fattree network, showing how Vf sets s.down along the path between the destination node e_{19} and e_6. e_6a_4 will drop the route from e_6 to prevent valley routing.

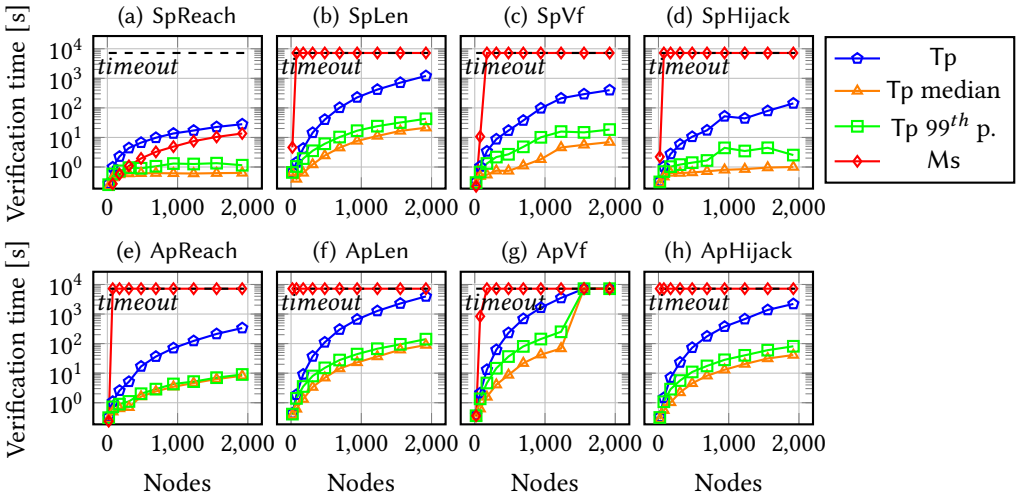


Fig. 14. Ms *vs.* Tp verification times for fattree benchmarks with 8 different policies.

by a symbolic variable (Ap) — modelling all-pairs routing to any edge node. Our routes modelled the eBGP protocol in these networks. Table 3 summarizes the eBGP fields represented and how we modelled them in SMT. We model the major common elements of eBGP routing: a route destination as a 32-bit integer (representing an IPv4 prefix); administrative distance, local preference, multi-exit discriminators as 32-bit integers (encoded as bitvectors); eBGP origin type as a ternary value; the AS path length as an (unbounded) integer; and BGP communities as a set of strings.

Witness times. Our temporal operators include witness times to specify the time when a node v has a route. To prove that v *eventually* has a route to the destination node *dest* at time t, we must show that one of v's neighbors sent it such a route at an earlier time $t-1$. The exact time v obtains a route thus depends on where *dest* is relative to v. This breaks down to five cases (or *roles*), following the fattree's structure, depending on whether v is: *(i)* the *dest* node (0 hops, has a route at $t=0$); *(ii)* an aggregation node in *dest*'s pod (1 hop, has a route at $t=1$); *(iii)* a core node, or an edge node in *dest*'s pod (2 hops, $t=2$); *(iv)* an aggregation node in another pod ($t=3$); or *(v)* an edge node in another pod ($t=4$). The largest choice of t is hence 4, the diameter of the fattree.

Table 3. eBGP route fields modelled by Timepiece in SMT for fattree benchmarks.

Route field	Modelled type in SMT
Route destination	bitvector [SMT-LIB 2010b]
Administrative distance	bitvector [SMT-LIB 2010b]
eBGP local preference	bitvector [SMT-LIB 2010b]
eBGP multi-exit discriminator	bitvector [SMT-LIB 2010b]
eBGP origin type	{egp, igp, unknown} [SMT-LIB 2010b]
eBGP AS path length	integer [SMT-LIB 2010c]
eBGP communities	set<string> [SMT-LIB 2010a, 2020]

These cases mirror those identified for local data center invariants in [Jayaraman et al. 2019a]. We use a function $dist(v)$ to encode these cases for each node v.

Reach. Reach demonstrates the simplest possible routing behavior and serves as a useful baseline. The policy simply increments the path length of a route on transfer. We initialized one destination edge node with a route to itself, and all other nodes with no route (∞). Our goal is to prove every node eventually has a route to the destination (*i.e.,* its route is not ∞). More precisely, because our network has diameter 4, each node v should acquire a route in 4 time steps.

$$P_{\mathsf{Reach}}(v) \equiv \mathcal{F}^4\mathcal{G}(s \neq \infty)$$

Interfaces for these benchmarks mirror the simplicity of the policy and property. If a node's route $s \neq \infty$ at time t, then its neighbors will in turn have a route $s \neq \infty$ at time $t+1$.

$$A_{\mathsf{Reach}}(v) \equiv \mathcal{F}^{dist(v)}\mathcal{G}(s \neq \infty)$$

SpReach's policy and property are so simple that Tp is actually slightly *slower* than Ms, as shown in Figure 14a. We conjecture the Ms encoding reduces to a particularly easy SAT instance. That said, we can already see that individual checks in Timepiece take only a fraction of the time that Ms takes, with 99% of node checks completing in at most 1.1 seconds, even for our largest benchmarks. For ApReach, Figure 14e shows that, perhaps from the burden of modelling the symbolic *dest*, Ms times out at $k = 8$. Tp verifies our largest benchmark ($k = 40$, with 2,000 nodes) in 5.5 minutes, with 99% of individual node checks taking under 9 seconds.

Len. Our next benchmark uses the same policy as Reach, but considers a stronger property: every node eventually has a route of at most 4 hops to the destination.

$$P_{\mathsf{Len}}(v) \equiv \mathcal{F}^4\mathcal{G}\big(s.\mathrm{len} \leq 4\big)$$

To prove this property, our interfaces specify that path lengths in routes should not exceed the distance to the destination: $s.\mathrm{len} \leq dist(v)$. In addition, because local preference influences routing, we fix the local preference to the default for all routes when present: $s.\mathrm{lp} = 100$.

$$A_{\mathsf{Len}}(v) \equiv \underbrace{\mathcal{G}\big(s = \infty \vee s.\mathrm{lp} = 100\big)}_{\textit{no better routes appear}} \sqcap \underbrace{\mathcal{F}^{dist(v)}\mathcal{G}\big(s.\mathrm{len} \leq dist(v)\big)}_{\textit{eventually the route appears}}$$

Reasoning over path lengths requires Z3 to use slower bitvector and integer theories. Figure 14b shows that monolithic verification times out at $k = 12$ for SpLen. By contrast, modular verification is able to solve $k = 40$ in just over 20 minutes, with 99% of nodes verified in under 43 seconds. Figure 14f shows that monolithic verification is not even possible for ApLen at $k = 4$; Tp completes for ApLen $k = 40$ in around 66 minutes, with 99% of nodes verified in 2.4 minutes.

Vf. Vf extends Reach with policy to prevent up-down-up (valley) routing [Beckett et al. 2016, 2017b; Pepelnjak 2018], where routes transit an intermediate pod. To implement this policy, we add a BGP community D along "down" edges in the topology (*i.e.,* from a core node or from an aggregation node to an edge node), and drop routes with D on "up" edges (see Figure 13). For brevity, we write "s.down" to mean "$D \in s$.tags". We test the same reachability property as Reach.

The legitimate routes in the fattree all start as routes travelling up from the destination node's pod, *e.g.,* the nodes in green (a_{16}, a_{17}, e_{19}) in Figure 13. We refer to these nodes as "adjacent nodes" with a shorthand $adj(v)$: they transmit routes to the core nodes (and thereby to the rest of the network) along their up edges. These edges will drop the routes if s.down, so we require that $adj(v) \rightarrow \neg s.\text{down}$. To ensure this, we add conjuncts to our interfaces requiring that nodes' final routes are no better than the shortest path's route: $s.\text{lp} = 100 \wedge s.\text{len} = dist(v)$. This ensures our inductive condition holds after every node has a route: otherwise, a core node (for instance) could offer a spurious route with $s.\text{len} < 1 \wedge s.\text{down}$ to an adjacent node.

$$A_{\text{Vf}}(v) \equiv s = \infty \; \mathcal{U}^{dist(v)} \; \mathcal{G}\Big(\underbrace{s.\text{lp}{=}100 \wedge s.\text{len}{=}dist(v)}_{\textit{no better routes appear}} \wedge \underbrace{(adj(v) \rightarrow \neg s.\text{down})}_{\textit{adjacent nodes will share routes}} \Big)$$

Figure 14c shows that Ms verifies up to $k = 8$ before timing out. As with SpLen, Tp time grows gradually in proportion to the number of nodes, topping out at 6.6 minutes for $k = 40$, with all node checks completing in under 20 seconds. Figure 14g shows that for all-pairs routing, monolithic verification times out again at $k = 12$, whereas Tp hits the 2-hour timeout at $k = 36$. We conjecture this may be due to the added complexity of encoding $adj(v)$ when the destination is symbolic.

Hijack. Hijack models a fattree with an additional "hijacker" node h connected to the core nodes. h represents a connection to the Internet from outside the network, which may advertise *any* route. We add a boolean ghost state tag to S for this policy to mark routes as external (from h) or internal. The destination node will advertise a route with $s.\text{prefix} = p$, where p is a symbolic value representing an internal address: the core nodes will then drop any routes from h for prefix p, but allow other routes through. Apart from this filtering, routing functions as in the Reach benchmarks. For this network, we verified that every internal node eventually has a route for prefix p and which is not via the hijacker ($\neg s.\text{tag}$), assuming nothing about the hijacker's route ($A_{\text{Hijack}}(h) \equiv \mathcal{G}(\text{true})$).

$$P_{\text{Hijack}}(v) \equiv \mathcal{F}^4 \mathcal{G}(s.\text{prefix}{=}p \wedge \neg s.\text{tag})$$

The Hijack interface is straightforward. We must simply re-affirm that nodes with internal prefixes never have external routes: $s.\text{prefix} = p \rightarrow \neg s.\text{tag}$. Once nodes have received a route from the destination at time $dist(v)$, they should keep that route forever, and hence their route will have both $s.\text{prefix} = p$ and $\neg s.\text{tag}$.

$$A_{\text{Hijack}}(v) \equiv \underbrace{\mathcal{F}^{dist(v)} \mathcal{G}(s.\text{prefix}{=}p \wedge \neg s.\text{tag})}_{\textit{route will be internally reachable}} \sqcap \underbrace{\mathcal{G}(s.\text{prefix}{=}p \rightarrow \neg s.\text{tag})}_{\textit{no hijack route is ever used}}$$

In SpHijack, monolithic verification times out at $k = 8$, whereas modular verification time scales to $k = 40$. 99% of nodes complete their checks in under 3 seconds, with our longest check taking 10.7 seconds at $k = 40$. As with our other benchmarks, verification time grows linearly with respect to the in-degree of each node (which determines the size of the SMT encoding of our inductive condition). Figure 14h shows similar patterns for the all-pairs case, with *no* monolithic benchmark completing on time, and modular verification taking at most 36.6 minutes.

Wide-area networks. To better investigate Timepiece's scalability for other types of networks, we evaluated it on the Internet2 [Internet2 2013] wide-area network.[9] Internet2's configuration files are one of the few publicly available examples of the complex policies of wide-area and cloud provider networks, as described in Propane [Beckett et al. 2016]. These files contain 1,552 Junos routing policies, which filter routes by tag or prefix and tag routes according to customer priorities (*e.g.,* commercial *vs.* academic peers). We converted the configuration files to Timepiece's model by extracting the policy details using Batfish [Fogel et al. 2015]. The resulting network has over 200 nodes: 10 internal nodes within Internet2's AS and 253 external peers. We did not model all components of Internet2's routing policies: we focused on IPv4 and BGP routing, and treated some complex behaviors as "havoc" (soundly overapproximating the true behavior).[10] We do not know Internet2's intended routing behavior: because of this, we cannot be certain that a counterexample found by Timepiece represents a real violation of the network's behavior; nonetheless, we may still use this network to assess how well Timepiece enables modular verification.

It appears that Internet2 uses a **BTE** community tag to identify routes that must not be shared with external neighbors. We checked that, if the internal nodes initially have any possible route, then no external neighbor of Internet2 should ever obtain a route with the **BTE** tag set, assuming the external neighbors do not start with such routes.

$$P_{\mathsf{BlockToExternal}}(v) \equiv \begin{cases} \mathcal{G}\big(s \neq \infty \rightarrow \mathbf{BTE} \notin s.\mathsf{tags}\big) & \text{if } v \text{ is external} \\ \mathcal{G}(\mathsf{true}) & \text{otherwise} \end{cases}$$

Our interface is the property, *i.e.,* $\forall v.\ A_{\mathsf{BlockToExternal}}(v) \equiv P_{\mathsf{BlockToExternal}}(v)$. Modular checking remains fast despite the network's more complex policies: on a 6-core Macbook Pro with 16GB of RAM, modular verification completes in 38.3 seconds, with a median check time of 0.6 seconds and a 99$^{\text{th}}$ percentile check time of 4.2 seconds. Monolithic verification does not complete after 2 hours.

7 RELATED WORK

Our work is most closely related to other efforts in control plane verification [Abhashkumar et al. 2020; Alberdingk Thijm et al. 2022a; Beckett et al. 2017a, 2018, 2019; Fogel et al. 2015; Gember-Jacobson et al. 2016; Giannarakis et al. 2020; Lopes and Rybalchenko 2019; Prabhu et al. 2020; Tang et al. 2022; Weitz et al. 2016; Ye et al. 2020]. We separate these tools into classes based on the specifics of one's verification problem.

SMT-based verification. For networks which are small (in the tens of nodes), SMT-based tools such as Minesweeper [Beckett et al. 2017a] or Bagpipe [Weitz et al. 2016] offer ease-of-use, generality, and symbolic reasoning. Minesweeper supports a broad range of properties including reachability, waypointing, no blackholes and loops, and device equivalence.

Simulation-based verification. For larger networks that do not necessitate incremental recomputation after device update nor fully symbolic reasoning, one can use simulation-based tools [Beckett et al. 2019; Fogel et al. 2015; Giannarakis et al. 2020; Lopes and Rybalchenko 2019; Prabhu et al. 2020; Ye et al. 2020]. Some of these tools also employ symbolic reasoning in limited ways to provide useful capabilities. For example, inspired by effective work on data plane analysis [Khurshid et al. 2013], Plankton [Prabhu et al. 2020] first analyzes configurations to identify IP prefix equivalence classes. Identified equivalence classes may be treated symbolically in the rest of the computation. Plankton might be able to reason symbolically about other attributes, but doing so would require additional custom engineering to find the appropriate sort of equivalence class ahead of time (*e.g.,*

[9] We used the versions of Internet2's configuration files available here [Weitz 2016].
[10] These include prefix matching, community regex matching and AS path matching. We also did not model BGP nexthop.

for BGP AS paths or communities). Timepiece can represent any or all route fields symbolically (*e.g.,* for external actors), or determine the behavior of I, F and $\oplus$ functions using symbolic variables (*e.g.,* for all-pairs properties). The solver effort is then passed off to the underlying SMT engine.

Scalable SMT-based verification. Fewer tools exist for networks which are large and where symbolic reasoning is important. Bonsai exploits symmetry to derive smaller abstractions of a network [Beckett et al. 2018], but does not work if the network has topology or policy asymmetries or one considers failures (which break topological symmetries). Other tools exploit modularity in network designs. Kirigami [Alberdingk Thijm et al. 2022a] verifies networks using assume-guarantee reasoning, but requires interfaces to specify the exact routes passed between any two components. As such, it is impossible to craft interfaces that are robust to minor changes in network policies. Lightyear [Tang et al. 2022] allows users to craft more general interfaces, but can only prove properties that capture the *absence* of some "bad" route (*e.g.,* our BlockToExternal property). Lightyear does not model route interaction (*e.g.,* selecting routes by path length), which we do via our $\oplus$ function: Lightyear's verification queries are therefore smaller, as it can check invariants on every edge independently. We conjecture (but have not proven) that Lightyear verifies properties that Timepiece expresses as $\mathcal{G}(\varphi)$, but not properties requiring $\mathcal{U}^t$ or $\mathcal{F}^t$ temporal operators.

Other efforts in network analysis. Daggitt *et al.* also use a timed model [Daggitt et al. 2018], but focus on convergence properties of routing protocols. We analyze properties that depend upon a network's topology and configuration such as reachability.

Other inspirations for our work are SecGuru and RCDC [Jayaraman et al. 2019a] in data plane verification. Unlike our work, they use non-temporal invariants, which they extract from the network topology and assume as ground truth for policies on individual devices. Whereas our experiments likewise used the topology to define local invariants, our verification procedure checks that these invariants are in fact guaranteed by the other devices in the network.

Compositional reasoning. Our work is inspired by the success of automated methods for compositional verification of concurrent systems – a recent handbook chapter [Giannakopoulou et al. 2018] provides many useful pointers to the rich literature on this topic. Automated methods using compositional reasoning have been successfully applied in many application domains – concurrent programs (*e.g.,* [Flanagan and Qadeer 2003; Gupta et al. 2011; Owicki and Gries 1976]), hardware designs (*e.g.,* [Henzinger et al. 2000; Kurshan 1988; McMillan 1997], reactive systems (*e.g.,* [Alur and Henzinger 1999]) — and for a range of properties including safety and liveness, as well as for refinement checking. [Lomuscio et al. 2010] applies assume-guarantee reasoning for verifying stability of network congestion control systems. Many such applications use temporal logic for specifying properties, as well as assumptions and guarantees at component interfaces [Pnueli 1984]. One main challenge is to come up with suitable assumptions that are strong enough to prove the properties of interest. Toward this goal, Timepiece uses a language of temporal invariants inspired by temporal logic to support checking local (*i.e.,* per-router) properties. However, it carefully limits the expressiveness of this language, *e.g.,* by not allowing arbitrary nesting of temporal operators, while allowing efficient verification of the proof obligations using SMT solvers.

Another main difference is that unlike most existing methods, Timepiece uses time as an *explicit* variable t in the language of invariants. This serves two distinct but related purposes. First, t provides a well-founded ordering to ensure that our proof rule is sound. Second, using t explicitly in the language of invariants avoids choosing some static ordering over the components, which could be otherwise used to break a circular chain of dependencies between their assumptions (*cf.* the Circ rule in [Giannakopoulou et al. 2018]). Unfortunately, it is not always possible to determine a static ordering, especially in cases where we consider multiple destinations at once symbolically.

Others have used induction over time [Misra and Chandy 1981] or over traces in specific models such as compositions of Moore/Mealy machines [Henzinger et al. 2002; McMillan 1997] and reactive modules [Alur and Henzinger 1999; Henzinger et al. 2000] to prove soundness of circular assume-guarantee proof rules. However, to the best of our knowledge, no prior efforts use time explicitly in the language of assumptions. Although handling time explicitly could be more costly for decision procedures, in practice we use abstractions (via temporal operators) that result in fairly compact formulas. Our evaluations show that these formulas can be handled well by modern SMT solvers.

There have also been many efforts that *automatically* derive assumptions for compositional reasoning [Giannakopoulou et al. 2018]. Representative techniques include computing fixed points over localized assertions called *split* invariants [Cohen and Namjoshi 2007], learning-based methods [Cobleigh et al. 2003], and counterexample-guided abstraction refinement [Bobaru et al. 2008; Elkader et al. 2018]. We can view our interfaces as split invariants, since they refer to only the local state of a component. However, we depend on the users to provide them as annotations.

Modular verification of distributed systems. There have been many prior efforts [Desai et al. 2018; Hawblitzel et al. 2015; Jung et al. 2015; Ma et al. 2019; Padon et al. 2016; Sergey et al. 2018; Yao et al. 2021] for modular verification of distributed systems – see a recent work [Sergey et al. 2018] for other useful pointers. In general, they handle much richer program logics or computational models than the network routing algebras we target; hence the required assumptions and verification tasks are more complex, and often require interactive theorem-proving. The synchronous semantics of network routing algebras [Daggitt et al. 2018] that underlies our work is more closely related to hardware designs modelled as compositions of finite state machines (FSMs), where a component FSM's state at time $t + 1$ depends on its state at time t and new inputs at time $t + 1$, some of which could be outputs from other FSM components, *i.e.,* their state at time t. No existing efforts for such models (*e.g.,* [Henzinger et al. 2002; McMillan 1997]) consider time explicitly in the assumptions.

8 CONCLUSION

Ensuring correct routing is critical to the operation of reliable networks. To verify today's hyperscale networks, we need modular control plane verification techniques that are general, expressive and efficient. We propose Timepiece, a radical new approach for verifying control planes based on a temporal foundation, which splits the network into small modules to verify efficiently in parallel. To carry out verification, users provide Timepiece with local interfaces using temporal operators. We proved that Timepiece is sound with respect to the network semantics and complete for closed networks, and argue that its temporal foundation is an excellent choice for modular verification.

ACKNOWLEDGMENTS

We thank our anonymous reviewers and our shepherd, Jedidiah McClurg, for their helpful feedback. This work was supported in part by grants from the Network Programming Initiative and the National Science Foundation: 1837030, 2107138.

AVAILABILITY

Timepiece is publicly available via GitHub [Alberdingk Thijm and Beckett 2023a] as well as Zenodo [Alberdingk Thijm and Beckett 2023b]. These contain the necessary code to run the benchmarks in our evaluation or write new ones in C#. We include a Docker image and Makefile to assist in running the benchmarks.

REFERENCES

Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast multilayer network verification. In *NSDI*. USENIX Association, Santa Clara, CA, USA, 201–219. https://www.usenix.org/system/files/nsdi20-paper-abhashkumar.pdf

Anubhavnidhi Abhashkumar, Kausik Subramanian, Alexey Andreyev, Hyojeong Kim, Nanda Kishore Salem, Jingyi Yang, Petr Lapukhov, Aditya Akella, and Hongyi Zeng. 2021. Running BGP in Data Centers at Scale. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, USA, 65–81. https://www.usenix.org/conference/nsdi21/presentation/abhashkumar

Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*. ACM, Seattle, WA, USA, 63–74. https://doi.org/10.1145/1402946.1402967

Timothy Alberdingk Thijm and Ryan Beckett. 2023a. *Timepiece*. GitHub. Retrieved April 6, 2023 from https://github.com/alberdingk-thijm/Timepiece

Timothy Alberdingk Thijm and Ryan Beckett. 2023b. *Timepiece*. Zenodo. Retrieved April 6, 2023 from https://zenodo.org/record/7799158

Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. 2022a. Kirigami, the Verifiable Art of Network Cutting. In *2022 IEEE 30th International Conference on Network Protocols (ICNP)*. IEEE, Lexington, KY, USA, 1–12. https://doi.org/10.1109/ICNP55882.2022.9940333

Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. 2022b. Modular Control Plane Verification via Temporal Invariants. https://doi.org/10.48550/arXiv.2204.10303 arXiv:2204.10303 [cs.LO]

Rajeev Alur and Thomas A Henzinger. 1999. Reactive modules. *Formal methods in system design* 15, 1 (1999), 7–48. https://doi.org/10.1023/A:1008739929481

Clark Barrett and Cesare Tinelli. 2018. Satisfiability modulo theories. In *Handbook of model checking*. Springer, Cham, Switzerland, 305–343. https://doi.org/10.1007/978-3-319-10575-8_11

Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017a. A General Approach to Network Configuration Verification. In *SIGCOMM*. ACM, New York, NY, USA, 155–168. https://doi.org/10.1145/3098822.3098834

Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2018. Control Plane Compression. In *SIGCOMM*. ACM, Budapest, Hungary, 476–489. https://doi.org/10.1145/3230543.3230583

Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2019. Abstract Interpretation of Distributed Network Control Planes. *Proc. ACM Program. Lang.* 4, POPL, Article 42 (dec 2019), 27 pages. https://doi.org/10.1145/3371110

Ryan Beckett and Ratul Mahajan. 2020. A General Framework for Compositional Network Modeling. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. Association for Computing Machinery, New York, NY, USA, 8–15. https://doi.org/10.1145/3422604.3425930

Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2016. Don'T Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *SIGCOMM*. ACM, New York, NY, USA, 328–341. https://doi.org/10.1145/2934872.2934909

Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitu Padhye, and David Walker. 2017b. Network Configuration Synthesis with Abstract Topologies. In *PLDI*. ACM, New York, NY, USA, 437–451. https://doi.org/10.1145/3062341.3062367

Mihaela Gheorghiu Bobaru, Corina S. Pasareanu, and Dimitra Giannakopoulou. 2008. Automated Assume-Guarantee Reasoning by Abstraction Refinement. In *CAV (Lecture Notes in Computer Science, Vol. 5123)*. Springer-Verlag, Berlin, Heidelberg, 135–148. https://doi.org/10.1007/978-3-540-70545-1_14

CISCO. 2005. Using BGP Community Values to Control Routing Policy in Upstream Provider Network. https://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/28784-bgp-community.html

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *CAV*. Springer, Berlin, Heidelberg, 154–169. https://doi.org/10.1007/10722167_15

Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. 2003. Learning Assumptions for Compositional Verification. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS, Proceedings (Lecture Notes in Computer Science, Vol. 2619)*. Springer, Berlin, Heidelberg, 331–346. https://doi.org/10.1007/3-540-36577-X_24

Ariel Cohen and Kedar S. Namjoshi. 2007. Local Proofs for Global Safety Properties. In *Computer Aided Verification (CAV), Proceedings (Lecture Notes in Computer Science, Vol. 4590)*. Springer, USA, 55–67. https://doi.org/10.5555/1519231.1519265

Matthew L Daggitt, Alexander JT Gurney, and Timothy G Griffin. 2018. Asynchronous convergence of policy-rich distributed Bellman-Ford routing protocols. In *SIGCOMM*. ACM, Budapest, Hungary, 103–116. https://doi.org/10.1145/3230543.3230561

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*. Springer, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. 2018. Compositional programming and testing of dynamic distributed systems. *PACMPL* 2, OOPSLA (2018), 159:1–159:30. https://doi.org/10.1145/3276529

Karam Abd Elkader, Orna Grumberg, Corina S. Pasareanu, and Sharon Shoham. 2018. Automated circular assume-guarantee reasoning. *Formal Aspects of Computing* 30, 5 (2018), 571–595. https://doi.org/10.1007/s00165-017-0436-0

Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient Network Reachability Analysis using a Succinct Control Plane Representation. In *OSDI*. USENIX Association, Savannah, GA, USA, 217–232. https://dl.acm.org/doi/10.5555/3026877.3026895

Nick Feamster and Hari Balakrishnan. 2005. Detecting BGP Configuration Faults with Static Analysis. In *NSDI*. USENIX Association, USA, 43–56. https://doi.org/10.5555/1251203.1251207

Cormac Flanagan and Shaz Qadeer. 2003. Thread-modular model checking. In *International SPIN Workshop on Model Checking of Software*. Springer, Berlin, Heidelberg, 213–224. https://doi.org/10.1007/3-540-44829-2_14

Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *NSDI*. USENIX Association, USA, 469–483. https://doi.org/10.5555/2789770.2789803

Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *SIGCOMM*. ACM, USA, 300–313. https://doi.org/10.1145/2934872.2934876

Dimitra Giannakopoulou, Kedar S Namjoshi, and Corina S Păsăreanu. 2018. Compositional reasoning. In *Handbook of Model Checking*. Springer, Cham, Switzerland, 345–383. https://doi.org/10.1007/978-3-319-10575-8_12

Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. 2020. NV: An Intermediate Language for Verification of Network Control Planes. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 958–973. https://doi.org/10.1145/3385412.3386019

Timothy G. Griffin, F. Bruce Shepherd, and Gordon Wilfong. 2002. The Stable Paths Problem and Interdomain Routing. *IEEE/ACM Trans. Networking* 10, 2 (April 2002), 232–243. https://ieeexplore.ieee.org/abstract/document/993304

Timothy G. Griffin and Joäo Luís Sobrinho. 2005. Metarouting. In *SIGCOMM*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1080091.1080094

Orna Grumberg and David E Long. 1994. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 3 (1994), 843–871. https://doi.org/10.1145/177492.177725

Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. 2011. Threader: A Constraint-Based Verifier for Multi-threaded Programs. In *Computer Aided Verification (CAV). Proceedings (Lecture Notes in Computer Science, Vol. 6806)*. Springer, Berlin, Heidelberg, 412–417. https://doi.org/10.1007/978-3-642-22110-1_32

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *SOSP*. ACM, New York, NY, USA, 1–17. https://doi.org/10.1145/2815400.2815428

C. Hedrick. 1988. Routing Information Protocol. Internet Request for Comments. https://datatracker.ietf.org/doc/html/rfc1058

Thomas A Henzinger, Shaz Qadeer, and Sriram K Rajamani. 1998. You assume, we guarantee: Methodology and case studies. In *International Conference on Computer Aided Verification*. Springer, Berlin, Heidelberg, 440–451. https://doi.org/10.1007/BFb0028765

Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. 2000. Decomposing Refinement Proofs Using Assume-Guarantee Reasoning. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE Computer Society, San Jose, CA, USA, 245–252. https://doi.org/10.5555/602902.602958

Thomas A. Henzinger, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. 2002. An assume-guarantee rule for checking simulation. *ACM Transactions on Programming Languages and Systems* 24, 1 (2002), 51–64. https://doi.org/10.1145/509705.509707

Internet2. 2013. About Internet2. https://meetings.internet2.edu/media/medialibrary/2013/08/01/AboutInternet2.pdf

Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. 2019a. Validating Datacenters at Scale. In *SIGCOMM*. Association for Computing Machinery, Beijing, China, 200–213. https://doi.org/10.1145/3341302.3342094

Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. 2019b. Validating Datacenters at Scale (Presentation at SIGCOMM 2019). https://conferences.sigcomm.org/sigcomm/2019/files/slides/paper_5_1.pptx

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. ACM, New York, NY, USA, 637–650. https://doi.org/10.1145/2775051.2676980

Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real Time Network Policy Checking Using Header Space Analysis. In *NSDI*. USENIX Association, USA, 99–112. https://doi.org/10.5555/2482626.2482638

Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *NSDI*. USENIX Association, USA, 15–28. https://doi.org/10.5555/2482626.2482630

R. P. Kurshan. 1988. Reducibility in analysis of coordination. In *Discrete Event Systems: Models and Applications (LNCIS, Vol. 103)*. Springer, Berlin, Heidelberg, 19–39. https://doi.org/10.1007/BFb0042302

K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, Berlin, Heidelberg, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

Alessio Lomuscio, Ben Strulo, Nigel Walker, and Peng Wu. 2010. Assume-guarantee reasoning with local specifications. In *ICFEM*. Springer, Berlin, Heidelberg, 204–219. https://doi.org/10.1007/978-3-642-16901-4_15

Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking Beliefs in Dynamic Networks. In *NSDI*. USENIX Association, Oakland, CA, 499–512. https://doi.org/10.5555/2789770.2789805

Nuno P Lopes and Andrey Rybalchenko. 2019. Fast BGP simulation of large datacenters. In *VMCAI*. Springer, Cham, Switzerland, 386–408. https://doi.org/10.1007/978-3-030-11245-5_18

K. Lougheed and Y. Rekhter. 1991. A Border Gateway Protocol 3 (BGP-3). Internet Request for Comments. https://datatracker.ietf.org/doc/html/rfc1267

Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A. Sakallah. 2019. I4: incremental inference of inductive invariants for verification of distributed protocols. In *SOSP*. ACM, Huntsville, ON, Canada, 370–384. https://doi.org/10.1145/3341301.3359651

Kenneth L. McMillan. 1997. A Compositional Rule for Hardware Design Refinement. In *Computer Aided Verification (CAV), Proceedings (Lecture Notes in Computer Science, Vol. 1254)*. Springer, Berlin, Heidelberg, 24–35. https://doi.org/10.1007/3-540-63166-6_6

Microsoft. 2021. Introduction to PLINQ. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq.

Ron Miller. 2022. As overall cloud infrastructure market growth dips to 24%, AWS reports slowdown. https://techcrunch.com/2022/10/28/as-overall-cloud-infrastructure-market-growth-dips-to-24-aws-reports-slowdown/

Jayadev Misra and K. Mani Chandy. 1981. Proofs of Networks of Processes. *IEEE Transactions on Software Engineering* 7, 4 (1981), 417–426. https://doi.org/10.1109/TSE.1981.230844

J. Moy. 1998. Open Shortest Path First Protocol Version 2. Internet Request for Comments. https://datatracker.ietf.org/doc/html/rfc2328

D. Oran. 1990. OSI IS-IS Intra-domain Routing Protocol. Internet Request for Comments. https://datatracker.ietf.org/doc/html/rfc1142

Susan S. Owicki and David Gries. 1976. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Commun. ACM* 19, 5 (1976), 279–285. https://doi.org/10.1145/360051.360224

Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *PLDI*. ACM, New York, NY, USA, 614–630. https://doi.org/10.1145/2908080.2908118

Ivan Pepelnjak. 2018. Valley-Free Routing in Data Center Fabrics. https://blog.ipspace.net/2018/09/valley-free-routing-in-data-center.html.

Amir Pnueli. 1984. In Transition From Global to Modular Temporal Reasoning about Programs. In *Logics and Models of Concurrent Systems - Conference proceedings (NATO ASI Series, Vol. 13)*. Springer, Berlin, Heidelberg, 123–144. https://doi.org/10.1007/978-3-642-82453-1_5

Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking. In *NSDI*. USENIX Association, Santa Clara, CA, USA, 953–967. https://doi.org/10.5555/3388242.3388310

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and proving with distributed protocols. *Proceedings of ACM Programming Languages* 2, POPL (2018), 28:1–28:30. https://doi.org/10.1145/3158116

SMT-LIB. 2010a. ArraysEx. https://smtlib.cs.uiowa.edu/theories-ArraysEx.shtml.

SMT-LIB. 2010b. FixedSizeBitVectors. https://smtlib.cs.uiowa.edu/theories-FixedSizeBitVectors.shtml.

SMT-LIB. 2010c. Ints. https://smtlib.cs.uiowa.edu/theories-Ints.shtml.

SMT-LIB. 2020. Unicode Strings. http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml.

João Luís Sobrinho. 2005. An Algebraic Theory of Dynamic Network Routing. *IEEE/ACM Trans. Netw.* 13, 5 (October 2005), 1160–1173. https://ieeexplore.ieee.org/abstract/document/1528502

Tom Strickx and Jeremy Hartman. 2022. Cloudflare outage on June 21, 2022. https://blog.cloudflare.com/cloudflare-outage-on-june-21-2022/.

Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, and George Varghese. 2022. LIGHTYEAR: Using Modularity to Scale BGP Control Plane Verification. arXiv:2204.09635 [cs.NI] https://arxiv.org/abs/2204.09635

Brandon Vigliarolo. 2022. After config error takes down Rogers, it promises to spend billions on reliability. https://www.theregister.com/2022/07/25/canadian_isp_rogers_outage/.

Konstantin Weitz. 2016. Getting Started With Bagpipe. http://www.konne.me/bagpipe/started.html

Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver. In *OOPSLA* (Amsterdam, Netherlands). Association for Computing Machinery, New York, NY, USA, 765–780. https://doi.org/10.1145/2983990.2984012

Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. 2021. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols. In *OSDI*. USENIX Association, USA, 405–421. https://www.usenix.org/conference/osdi21/presentation/yao

Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. 2020. Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN. In *SIGCOMM (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 599–614. https://doi.org/10.1145/3387514.3406217

Peng Zhang, Aaron Gember-Jacobson, Yueshang Zuo, Yuhao Huang, Xu Liu, and Hao Li. 2022. Differential Network Analysis. In *NSDI*. USENIX Association, Renton, WA, USA, 601–615. https://www.usenix.org/conference/nsdi22/presentation/zhang-peng

Received 2022-11-10; accepted 2023-03-31