Kirigami, the Verifiable Art of Network Cutting

Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta and David Walker

***Abstract*—Satisfiability Modulo Theories (SMT)-based analysis allows exhaustive reasoning over complex distributed control plane routing behaviors, enabling verification of converged routing states under arbitrary conditions. To improve scalability of SMT solving, we introduce a modular verification approach to network control plane verification, where we cut a network into smaller fragments. Users specify an annotated cut which describes how to generate these fragments from the monolithic network, and we verify each fragment independently, using these annotations to define assumptions and guarantees over fragments akin to assume-guarantee reasoning. We prove that any converged states of the fragments are converged states of the monolithic network, and there exists an annotated cut that can generate fragments corresponding to any converged state of the monolithic network. We implement this procedure as Kirigami, an extension of the network verification language and tool NV, and evaluate it on industrial topologies with synthesized policies. We observe a 10x improvement in end-to-end NV verification time, with SMT solve time improving by up to 6 orders of magnitude.**

***Index Terms*—modular verification, network control plane, control plane verification, routing protocols**

I. Introduction

Today's networks are labyrinthine and hard-to-analyze systems. To determine the best paths routers may use to forward traffic, networks typically run distributed routing protocols. Despite advances like software-defined networking, these protocols remain widely used in data centers [1] and wide-area networks. Millions of lines of decentralized, low-level router configuration code control protocol behaviors, and operators must update these device configurations over time. This overwhelming complexity has led to several notable and costly outages [2]–[5]. Often, the culprits behind these incidents are subtle network misconfigurations.

In response, researchers have developed a variety of verification tools and techniques to catch errors before outages occur. Some [6]–[13] have targeted the network *data plane*, which is responsible for forwarding traffic from point A to point B. This work has produced scalable, efficient methods for modeling the data plane and checking properties of how packets traverse it.

The data plane is produced by the *control plane*. It uses the aforementioned routing protocols to decide which routes forwarding should use. Occasionally, these protocols may update their choice of routes — *e.g.,* following a device failure — and recompute new paths. When this happens, the data plane is regenerated, and the user must repeat any data plane analysis. Control plane errors can lead to further issues like route flapping, leaving human operators to hunt for subtle bugs in a Kafkaesque morass of router configurations.

To address this problem, researchers have developed another suite of tools to analyze the control plane [14]–[25]. Control plane analyses consider which routes the data plane will use in given network environments, and check properties of the network in such environments. One branch of control plane verification, starting from Minesweeper [15], encodes a network as a Satisfiability Modulo Theories (SMT) formula and then asks an SMT solver [26] to check properties of the encoded network. SMT-based verification has some advantages over other approaches: it is expressive and can reason symbolically about network behavior, allowing analyses about *all possible routes* a neighbor might announce; it also may form a basis for network synthesis and repair [27]. Unfortunately, it suffers from scalability issues. Prior work has explored using abstractions to resolve this problem, *e.g.,* using symmetries in topologies to compress networks [19], [20]. These abstractions offer some relief, but cannot always handle arbitrary networks.

Control plane verification users thus face a trade-off: they may use semi-symbolic or simulation-based tools [16], [21]–[25], [28] to analyze industrial-sized networks when the flexibility of SMT-based symbolic reasoning is not necessary; or they must contend with SMT-based verifiers which may not scale to networks with more than a few hundred nodes. This paper offers another option: using a user's own insights about their network's behavior, we leverage the inherent modularity of the control plane to *cut* a monolithic network into multiple fragments to verify independently. Networks' modular structure — where end-to-end behaviors emerge from individual routers' local decisions — makes cutting an intuitive way to scale verification. In an SMT-based context, it allows us to verify properties in the presence of faults or arbitrary external announcements, which is not shown with prior abstraction approaches [19], [21]. Building on assume-guarantee verification of modular programs [29], [30], we present a new technique for modular verification of control planes and implement it as Kirigami, an extension for the NV [23] network verification tool. While we focus on SMT-based verification, one could combine our cutting technique with other methods *e.g.,* model checking, simulation.

In a typical assume-guarantee verification approach, one can verify a safety property over a system of concurrent processes by verifying local properties of each process independently, using *assumptions* over the process's inputs and *guarantees* over its outputs. The verifier will check the required proof obligations on each component (formulated as assume-guarantee rules): if all checks pass, then the property holds for the monolithic system. Our verification technique mirrors this idea: we verify a property over network fragments (*cf.*

This work was supported in part by the National Science Foundation awards NeTS 1704336, FMitF 1837030, SHF 2107138, and Facebook Research Award on "Network control plane verification at scale."

978-1-6654-8234-9/22/$31.00 ©2022 IEEE

processes), given assumptions over the rest of the network and guarantees over our fragments, to conclude that the property holds for the monolithic network.

We start from an existing model for distributed routing, the Stable Routing Problem (SRP) [19]. In an SRP, each node of the network exchanges routes with its neighbors to compute a locally-stable solution. To define an SRP, we require complete knowledge of the network and its configurations. In theory, our work could apply to interdomain routing — if multiple organizations gave us their configurations, we could jointly analyze those configurations. In practice, operators are reluctant to share their configurations outside their organization. Thus, our work's main practical application is on networks controlled by a single entity. This is frequently the case in large data centers, many of which run distributed routing protocols such as BGP [31].

We first generalize SRPs to "open SRPs", in which a network receives routes along a set of *input nodes* and sends out routes along a different set of *output nodes*. We identify the input node solutions as our open SRP's assumptions, and the output node solutions as its guarantees. We present a procedure CUT which, given an *interface* — a mapping from a cut-set of edges to routes — cuts an open SRP S into two open SRPs T_1 and T_2 covering S, and where we replace each cut edge with a route assumed in one SRP and guaranteed in the other. Interfaces can follow a network's natural boundaries, *e.g.,* tiers or hierarchies in a data center topology [32]–[34].

As with the traditional (closed) SRP, we can check that an open SRP satisfies a given safety property $prop$ by verifying that $prop$ holds for the SRP's solutions. We prove that if P holds on T_1 and T_2's solutions, then it holds on S's. This is the basis for our modular network verification technique. Starting from a network S, an interface I, and a safety property P, we use $\text{CUT}(S, I)$ to obtain a set of n open SRPs $T_1, \ldots, T_n$ that we verify independently. We verify P for each open SRP T_i: if P does not hold, we return a counterexample demonstrating a solution that violates P.

We consider SRPs with any number of solutions. Our CUT procedure constructs fragments such that, given an interface which captures *some* solutions of the monolithic SRP, the fragments will have the *same* solutions. This allows us to check properties of all solutions that satisfy our interface's guarantees given its assumptions – we call these *solutions "modulo" the interface*. If the monolithic network has a unique solution, we can check properties of this solution using a single interface. If the network has multiple solutions, we can check properties of all solutions modulo a given interface, and compute additional interfaces to cover other solutions. If the interface's guarantees do not hold, or the network has no solution, we report this case to the user to diagnose.

Checking multiple smaller open SRPs rather than a single closed SRP offers significant scalability improvements. SMT-based verification time can — depending on the policy and property — grow exponentially with the size of the network [15]. Hence, verifying P on each open SRP T_i takes a fraction of the time to verify P directly on S, and is embarrassingly parallel. Our experiments demonstrate that this modular verification technique works well for a variety of data center, random and backbone networks, with significant improvements in SMT solve time: we show for one set of fattree [32] benchmarks that verifying the fattree pod-by-pod cuts SMT time from 90 minutes to under 2 seconds; verifying every node individually reduces SMT time to around 10 milliseconds. Taking advantage of parallelism also cuts down NV end-to-end verification time for our largest benchmarks from over 2 hours to under 15 minutes. This modularity can scale verification to tomorrow's networks, and produce localized errors when verification fails, empowering network operators with stronger safety and reliability guarantees.

In summary, we make the following contributions:

- **A Theory of Network Fragments.** We develop an extension of the Stable Routing Problem (SRP) model [19] for network fragments. Our extension provides a method to cut monolithic SRPs into a set of fragments. We define *interfaces* to cut SRPs and map the cut edges to annotations which then define *assumptions* and *guarantees* of our fragments. We prove that under these assumptions, if these guarantees hold, then a property that holds in every fragment also holds in the monolithic network. (§IV)
- **A Modular Network Verification Technique.** We present a checking procedure to verify SRP properties. Given a property P we wish to verify, we cut an SRP S according to a given interface I into fragments, and generate checks on each fragment to both verify that our interface captures the monolithic network behavior, and verify P on every fragment. This enables a novel approach for modular control plane verification based on assume-guarantee reasoning. (§V)
- **Fast, Scalable and Modular SMT Verification.** We implement our technique as Kirigami, an extension for NV, a network verification language and tool [23]. Kirigami improves on NV verification scalability and performance, with an SMT solve time up to *six orders of magnitude* faster for a selection of NV benchmarks. (§VI and §VII)
- **An Algorithm for Modular Verification for Multi-Solution Networks.** We present a second checking procedure to verify properties of networks with multiple solutions. This procedure takes multiple interfaces as arguments to verify properties of SRPs with multiple solutions. We illustrate how it functions via the classic multi-solution DISAGREE example from the routing algebra literature [35]. (§VIII)

This paper is based on the Kirigami modular verification method presented earlier [36], which considered networks with unique converged states. This work extends the method to consider networks with multiple converged states.

II. OVERVIEW

A. The Stable Routing Problem

A network is a graph with nodes V representing routers and edges E representing the links between them. A distributed control plane uses routing protocols to determine paths to routing destinations. Each router deploys its own local rules to broadcast routing announcements (or *routes*) and select a

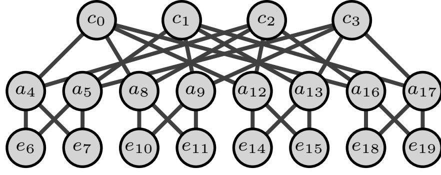


Fig. 1: A 4-pod fattree topology.

```
1  type attribute = { id: int; cost: int }
2
3  symbolic d : int (* symbolic id *)
4  require (d = 6 || d = 7 || d = 10 || d = 14 || d
       = 15 || d = 18 || d = 19)
5  let nodes = 20 (* topology *)
6  let edges = {
7    0=4; 0=8; 0=12; 0=16;(*...*)
8    16=18; 16=19; 17=18; 17=19;
9  }
10
11 let merge node x y = if x.cost < y.cost then x
     else y
12 let trans edge x = {x with cost = x.cost + 1}
13 let init node = match node with
14   | 0n -> if (d = 0) then {id=d; cost=0;} else
     NULL
15   | 1n -> if (d = 1) then {id=d; cost=0;} else
     NULL (*...*)
```

Fig. 2: An NV program `fat.nv` representing Fig. 1.

"best" route: the details of these rules vary with the protocol, but generally protocols focus on minimizing routing costs.

These elements — nodes and edges, a set of routes, and a set of rules to initialize, compare and broadcast them — form the basis for our control plane routing model, the SRP [19] (defined formally in §III). In a well-designed network, this exchange of routes eventually converges to a *stable state*, where no node may improve on its current best route by selecting another offered by a neighbor. A *solution* $\mathcal{L}$ to the SRP is a mapping from nodes to these stable routes.

a) An Example SRP: Consider a fattree [32] data center network, as shown in Fig. 1. Routing in fattree networks typically follows a Λ shape: traffic that starts at an edge layer switch ($e_6, \ldots, e_{19}$) travels up a link to an aggregation layer switch ($a_4, \ldots, a_{17}$), then ascends from the pod to a core layer switch ($c_0, \ldots, c_3$) in the spine and descends into another pod.

Suppose we wish to verify that every node in a fattree SRP instance S can reach every edge layer node of the fattree, where S is running BGP (the Border Gateway Protocol) [1]. We can do so by first modeling S's routes as highly-simplified BGP announcements $\langle\, p, x\,\rangle$ with 2 fields: an identifier p (*cf.* a prefix) and a cost metric x (abstracting, *e.g.,* local preference, AS path length [37], [38], *etc.*).[1] Each node has an identifier p, where every node has an initial route $\langle\, p, 0\,\rangle$ for its identifier, and no route to other identifiers. Nodes will broadcast their current routes to their neighbors: in this simple example, if a node has a route $\langle\, p, x\,\rangle$, it will send a route $\langle\, p, x+1\,\rangle$ to its neighbors (incrementing the metric). Nodes compare each received route with their current choice and select the one with the smallest cost, and then re-broadcast if their route changes.

[1]In a real network, routes could represent many more BGP fields, but this example provides the necessary detail to demonstrate the basics of SRPs.

```
1 include "fat.nv"
2
3 (* map nodes to solutions (stable routes) *)
4 let sol = solution {init = init; trans = trans;
     merge = merge}
5 (* check a property of every node's solution *)
6 assert foldNodes (fun n r acc -> acc && r.id = d
       && r.cost <= 4) sol true
```

Fig. 3: An NV program asserting that every node can reach every prefix advertised by an edge layer switch.

A node u's solution $\mathcal{L}(u)$ is the best route between u's initial route and the solutions broadcast by each of u's neighbors.

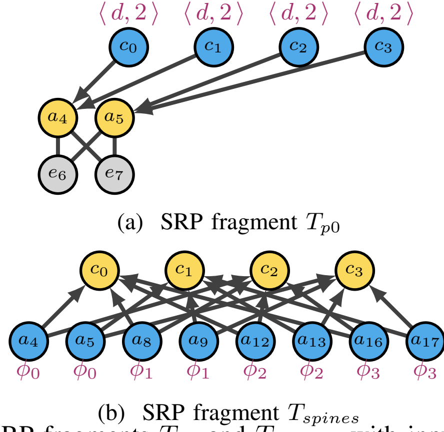
b) Verifying SRPs with NV [23]: To verify all-edge reachability in S, we must check that for any choice of identifier p of an edge layer node, all nodes of the network have a path of cost at most 4 to that node. Naively enumerating all possible identifiers is obviously inefficient, if not infeasible in practice. In some scenarios, an equivalence class-based approach like that of Plankton [24] may make the space of all identifiers small enough to efficiently enumerate. We will use a symbolic approach, where we treat the identifier as a symbolic variable d: we then will verify that for any concrete identifier instantiating d, every node can reach that identifier. Symbolic variables can also help verify properties in the presence of link failures or arbitrary external announcements from outside one's network. One verification tool supporting symbolic reasoning is NV [23]. NV is a functional programming language for modeling control planes and verifying their properties using SMT. An NV program's components resemble an SRP's: it has a topology with `nodes` and `edges`; a type of routes `attribute`; a function `init` to initialize routes; a function `trans` to broadcast routes; and a function `merge` to compare routes. NV provides `symbolic` and `require` expressions to declare and constrain symbolic values, respectively. Fig. 2 presents a condensed NV program for Fig. 1.

Fig. 3 demonstrates how to verify a safety property P over all nodes and their solutions in NV, where $P(u, \mathcal{L}(u))$ holds for node u and solution $\mathcal{L}(u)$ *iff* $\mathcal{L}(u).p = d \wedge \mathcal{L}(u).x \leq 4$. We define a solution (line 4) using `init`, `trans` and `merge` from Fig. 2. We then assert (line 6) that P holds on this solution. When we ask NV to verify Fig. 3, it encodes S and P as an SMT query, and confirms that P holds.

B. Modular SRP Verification

SMT-based verification is flexible, but has issues when it comes to scalability. Our evaluation in §VII shows it scales superlinearly for larger fattrees with more complex policies: from 0.03 seconds for a 20-node network, to 1.4 seconds for an 80-node network, and 1833.7 seconds for a 320-node network! To verify industrial fattree networks with 10^4 or more switches [13], we need a way to scale this technique up.

Suppose then that we took a large network and *cut it into fragments*, then verified a safety property P on each fragment independently. If P holds for every fragment, then we want it to hold for the monolithic network; otherwise, we want to observe real counterexamples as in the monolithic network. To



(a) SRP fragment T_{p0}

(b) SRP fragment T_{spines}

Fig. 4: SRP fragments T_{p0} and T_{spines}, with input nodes in blue, output nodes in yellow and assumptions in purple.

achieve this goal, our cutting procedure must also summarize the network behavior external to each fragment.

We incorporate these summaries into the SRP model by generalizing it to open SRPs. Open SRPs extend the SRP model by designating some nodes as *input nodes* and some others as *output nodes*. We annotate input and output nodes with routes representing solutions *assumed* on the inputs and *guaranteed* on the outputs. We express these annotations using an *interface*: a mapping from each cut edge to a route annotation. Given an open SRP S and an interface I, we cut S into open SRP fragments, where each fragment identifies assumptions on its inputs and guarantees on its outputs.

a) Cutting Down Fattrees: We will now move on to demonstrating this idea for our example. Let's cut each pod of our network into its own fragment T_{p0} through T_{p3}, leaving the spine nodes as a fifth fragment T_{spines}.

Figs. 4a and 4b show pod 0 and the spines of Fig. 1 as open SRPs T_{p0} and T_{spines}, respectively. In T_{p0}, we assume routes from the spines and check guarantees on a_4 and a_5. Every route guaranteed by one fragment is assumed by another (and vice-versa): if we guarantee that c_0 has a route $\langle d, 2 \rangle$ in T_{spines}, we assume it has a route $\langle d, 2 \rangle$ in T_{p0}. The exact route advertised by an aggregation node a depends on if the destination identifier d lies in a's pod or not. For instance, if $d = 6$, nodes a_4, a_5 of pod 0 have a route $\langle d, 1 \rangle$ from their neighbor e_6, while all other aggregation nodes have a route $\langle d, 3 \rangle$ (via the core nodes). We write ϕ_i as a shorthand for this reasoning over costs in Fig. 4b, where

$$\phi_i = \text{if } d \text{ in pod } i \text{ then } \langle d, 1 \rangle \text{ else } \langle d, 3 \rangle$$

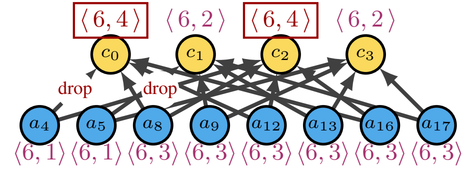
b) Verifying Network Fragments: In modular verification, we perform an independent verification query for each fragment: we encode the open SRP and property, along with an assumptions formula assuming a state of the inputs and a guarantees formula to check on the state of the outputs. We then submit every query to our solver and ask if the network has a solution where, under the given assumptions, either the property is false or the guarantee do not hold. The solver searches for a counterexample demonstrating a concrete violation of the property or our guarantees. Guarantee violations

```
1  include "fat.nv"
2
3  let partition node = match node with
4    | 0n | 1n | 2n | 3n -> 0 (* spines *)
5    | 4n | 5n | 6n | 7n -> 1 (* p0 *)(*...*)
6
7  let interface edge x = match edge with
8    | 0~_ | 1~_ | 2~_ | 3~_ -> x = { id = d; cost
       = 2; }
9    | 4~_ | 5~_ -> x = { id = d; cost = if d > 3
       && d < 8 then 1 else 3; }
10   | 8~_ | 9~_ -> x = { id = d; cost = if d > 7
       && d < 12 then 1 else 3; }
```

Fig. 5: An (abbreviated) NV program to cut Fig. 2 into pods.

demonstrate possible bugs in our network implementation or mistakes in our beliefs, just as property violations do.

Let us consider our fattree network again. Suppose we misconfigured a_4 to black hole (silently drop) outgoing traffic. Consider what happens when $d = 6$, meaning the destination is e_6: then d is in a_4's pod. Because a_4 is dropping outgoing traffic, the best route c_0 will receive will be a $\langle 6, 3 \rangle$ route from one of a_8, a_{12}, a_{16}, and hence $\mathcal{L}(c_0) = \langle 6, 4 \rangle$. The solution in T_{spines} will be as follows:



Our interface maps $c_0 a_8$ to $\langle d, 2 \rangle$, so we must guarantee that $\mathcal{L}(c_0) = \langle d, 2 \rangle$ when verifying T_{spines}. Due to the bug, this check fails ($\langle 6, 2 \rangle \neq \langle 6, 4 \rangle$) and our solver returns the solution above as a counterexample. This localizes the counterexample to this part of the network; our other fragments will pass verification. This means that, so long as our interface remains the same, we only need to correct the bug and re-verify T_{spines}, without needing to re-verify *any other fragment*. In §IV, we prove that if our guarantees and P hold for all fragments of S, then P holds for the monolithic network S.

c) Cutting with Kirigami: As part of our work, we implemented an extension Kirigami to NV for cutting and verifying networks. Fig. 5 shows an NV file with new `partition` and `interface` functions. `partition` maps each node to a fragment, while `interface` adds assertions to check that a route x along a cross-fragment edge equals the given annotation, *e.g.,* that the route from `0n` to `4n` has id d and cost 2. These functions provide the necessary detail to construct our fragments and modularly verify them.

d) Alternative Cuts: Pod-based cuts suit our hierarchical view of fattrees, but we can consider alternative cuts. We could cut Fig. 2 so that every node is in its own fragment. Verifying a single node in SMT can take milliseconds, and hence leads to significant performance improvements. The corresponding NV program resembles Fig. 5, except every node maps to its own fragment and we annotate every edge.

III. Background on the Stable Routing Problem

We summarize prior work [19] on the Stable Routing Problem (SRP) network model. Its components resemble routing algebras used for reasoning about convergence of routing protocols [39]–[41], but SRPs also include a network topology for reasoning about properties such as reachability between nodes.

An SRP instance S is a 6-tuple $(V, E, R, \mathsf{init}, \oplus, \mathsf{trans})$, defined as follows.

a) Topology: V is a set of nodes and $E \subseteq V \times V$ is a set of directed edges. We write uv for an edge from node u to node v. Edges may not be self-loops: $\forall v \in V.\ vv \notin E$.

b) Routes: R is a set of routes that describe the fields of routing messages. For example, when modeling BGP, R might be a set of tuples of an integer local preference, a set of community tags, and a sequence of AS numbers representing the AS path [37], [38].

c) Node Initialization: The initialization function $\mathsf{init} : V \rightarrow R$ describes the initial route of each node. When modeling single destination routing, init may map a destination node e_{19} to some initial route r_d, and all other nodes to a null route; in multiple destination routing, we may have many initial routes.

d) Route Update: The merge function $\oplus : R \times R \rightarrow R$ defines how to compare routes. $\oplus$ represents updates of a node's selected route: we assume $\oplus$ is *associative* and *commutative*, *i.e.,* the order in which routes are merged does not matter.

e) Route Transfer: The transfer function $\mathsf{trans} : E \times R \rightarrow R$ describes how routes are modified between nodes. Given an edge uv and a route r from node u, $\mathsf{trans}(uv, r)$ determines the route received at v.

f) Solutions: A solution $\mathcal{L} : V \rightarrow R$ is a mapping from nodes to routes. Intuitively, a solution is defined such that each node is *locally stable*, *i.e.,* it has no incentive to deviate from its currently chosen neighbors. Nodes compute their solution via message exchange, where each node in the SRP advertises its chosen route to each of its neighbors. Formally, an SRP solution $\mathcal{L}$ satisfies the constraint:

$$\mathcal{L}(v) = \mathsf{init}(v) \oplus \bigoplus_{uv \in E} \mathsf{trans}(uv, \mathcal{L}(u)) \tag{1}$$

where $\bigoplus$ is the sequence of $\oplus$ operations on each transferred route $\mathsf{trans}(uv, \mathcal{L}(u))$ from each neighbor u of v. These received routes are merged with v's initial value $\mathsf{init}(v)$.

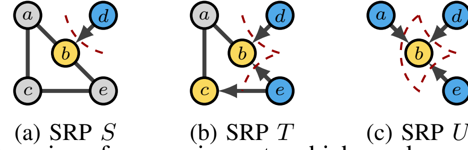
If an SRP has at least one solution, we say it *converges*. Convergence has been studied in past work [35], [39]–[41], which identified monotonicity properties that guarantee that the SRP converges to a *unique* solution. In particular, if the network has a finite R, and satisfies the constraints below $\forall r, r_1, r_2 \in R, \forall e \in E$:

$$r_1 \oplus r_2 \in \{r_1, r_2\} \tag{2}$$

$$\exists \infty \in R.\ r \oplus \infty = r = \infty \oplus r \tag{3}$$

$$\exists 0 \in R.\ r \oplus 0 = 0 = 0 \oplus r \tag{4}$$

$$r \neq \infty \rightarrow \mathsf{trans}(e, r) \oplus r = r \wedge \mathsf{trans}(e, r) \neq r \tag{5}$$



(a) SRP S (b) SRP T (c) SRP U

Fig. 6: A series of successive cuts which produce open SRPs S, T and U. Base nodes are grey, input nodes blue and output nodes yellow. Each cut (in red) slices off part of the SRP, leaving input nodes to represent the cut components.

then it converges to a unique solution.[2] Constraint (2) defines $\oplus$ to be *selective*, meaning it always chooses between one of the two given routes, and never introduces "new" routes. Constraint (3) defines a route ∞ to be an *identity* for $\oplus$, meaning that other routes are always preferred over ∞. This typically represents an "invalid" or null route. Constraint (4) defines a trivial route 0 to be an *annihilator* for $\oplus$, meaning that 0 is always preferred over any other route. This typically represents a node's route to itself. Constraint (5) defines trans to be *strictly monotonic* (*a.k.a. strictly increasing*) over $\oplus$, meaning that applying a trans function along any edge to a route r always makes a route that is less preferable according to $\oplus$ than r.

SRPs with *no* solution are said to *diverge*. An SRP has no solution if there is no mapping that satisfies (1). Griffin *et al.* [35] present a variety of "gadgets" which demonstrate these cases, which do not satisfy (5). We consider SRPs with any number of solutions, meaning there may exist multiple mappings $\mathcal{L}$ satisfying (1).

A solution may determine an SRP's forwarding behavior or another decision-making procedure, as shown in [19]. We omit discussing forwarding behavior to focus on a general SRP definition without restricting ourselves only to forwarding.

IV. Cutting SRPs

We now introduce our original contributions, starting with *open SRPs*. We define a Cut procedure to partition an open SRP into fragments. We prove that fragment have solutions corresponding to the larger SRP's solutions and vice-versa.

We introduce some notation in this section. $\mathsf{dom}(f)$ is the *domain* of the function f, and $f|_X$ is the *restriction* of f to $X \subseteq \mathsf{dom}(f)$. We use subscripts to specify SRP components, *e.g.,* init_S refers to SRP S's init component.

A. Open SRPs

An *open SRP* generalizes our earlier SRP definition to include *assumptions* and *guarantees*. An open SRP instance S is an 8-tuple $(V, E, R, \mathsf{init}, \oplus, \mathsf{trans}, \mathsf{ass}, \mathsf{guar})$.

The first six elements are exactly as for regular (closed) SRPs. The final two elements, ass ("assumptions") and guar ("guarantees"), are *partial functions* ($V \hookrightarrow R$) mapping mutually disjoints subsets $V^{in}, V^{out} \subseteq V$ to routes. We use V^{in} (input nodes) as a shorthand for $\mathsf{dom}(\mathsf{ass})$ and V^{out} (output nodes) as a shorthand for $\mathsf{dom}(\mathsf{guar})$. All nodes that are neither input nor output nodes are "*base nodes*" V^{base}. A

[2] These constraints are sufficient to ensure uniqueness, but not necessary.

closed SRP is an open SRP where $V^{in} = V^{out} = \varnothing$. Going forward, we assume an open SRP wherever we write "SRP".

Input nodes are *source nodes* (*i.e.,* they have in-degree 0). Hence, they act as auxiliary nodes, indicating where a fixed incoming route "arrives" from *outside* the SRP, as specified by the assumptions ass. Output nodes correspondingly mark where routes "depart" the SRP, per the guarantees guar. We do not require any connectivity properties of output nodes: they simply identify an outgoing route we wish to guarantee, without detailing where the SRP is announcing that route to.[3] Fig. 6 illustrates this concept.

Definition IV.1 (Open SRPs). *An* open SRP instance $S = (V, E, R, \mathsf{init}, \oplus, \mathsf{trans}, \mathsf{ass}, \mathsf{guar})$ *has the following properties:*

- $V = V^{in} \cup V^{out} \cup V^{base}$ *and* $V^{in}, V^{out}, V^{base}$ *are pairwise-disjoint;*
- $\mathsf{ass} : V^{in} \to R$ *and* $\mathsf{guar} : V^{out} \to R$*; and*
- $\forall v \in V^{in}.$ *in-degree*$(v) = 0$.

1) Open SRP Solutions: A mapping $\mathcal{L} : V \to R$ is a *solution* to an open SRP *iff*:

$$\mathcal{L}(u) = \mathsf{init}(u) \oplus \bigoplus_{vu \in E} \mathsf{trans}(vu, \mathcal{L}(v)) \qquad \forall v \notin V^{in} \tag{6}$$

$$\mathcal{L}(u) = \mathsf{ass}(u) \qquad \forall v \in V^{in} \tag{7}$$

$$\mathcal{L}(u) = \mathsf{guar}(u) \qquad \forall v \in V^{out} \tag{8}$$

Note that Equations (6) and (8) both apply for all outputs $v \in V^{out}$. Solutions for open SRPs resemble closed SRP solutions, with the addition of constraints based on the values of ass and guar. For any input node u, its assumption $\mathsf{ass}(u)$ determines the node's solution directly; for an output node u, its solution $\mathcal{L}(u)$ must be consistent with both the right-hand side of (6) *and* the right-hand side of (8). Hence, if $\exists u \in V^{out}.\ \mathsf{init}(u) \oplus \bigoplus_{vu \in E} \mathsf{trans}(vu, \mathcal{L}(v)) \neq \mathsf{guar}(u)$, there is no solution to the open SRP. As in closed SRPs, open SRPs may converge or diverge. We discuss open SRPs with more than one solution in more depth later in this section (see §IV-D).

We consider *safety properties* on SRP solutions. A property $P : V \times R \to \mathbb{B}$ holds on a solution $\mathcal{L}_S$ *iff*

$$\forall v \in V_S^{base} \cup V_S^{out}.\ P(v, \mathcal{L}_S(v)) \tag{9}$$

We ignore input nodes as they are "outside" the SRP. We can express many properties this way, including reachability, isolation, path length, waypointing and fault tolerance [15], but not convergence properties or properties over multiple nodes, *e.g.,* that two nodes u, v have the same solutions $\mathcal{L}(u) = \mathcal{L}(v)$ (useful for checking consistency across nodes, irrespective of their exact solutions).

B. Interfaces and Cutting SRPs

We now consider how to cut an SRP S into two SRPs T_1 and T_2, where T_1 and T_2 cover S and replicate its behavior using their assumptions and guarantees. We select a cut-set $C \subseteq E$ of edges in S and annotate each cut edge uv with a route that describes the solution transferred from u to v. This cut-set divides the *non-input nodes* $V_S \setminus V_S^{in}$ into two disjoint subsets, W_1 and W_2 (we will treat input nodes separately). We call this annotated cut-set an *interface* I.

Definition IV.2 (Interface). *Let S be an SRP and let $C \subseteq E$ be a cut-set partitioning $V_S \setminus V_S^{in}$. $I : C \to R_S$ is an* interface *if it maps every element uv of C to a route $I(uv)$ in R_S.*

We now define a CUT procedure. Given an SRP S and an interface I, $\textsc{Cut}(S, I)$ partitions S into two SRPs, T_1 and T_2, which we call *fragments*. We can CUT an SRP into arbitrarily many SRPs by recursively cutting the resulting fragments.

Definition IV.3 (CUT). *Let S be an SRP and let I be an interface over S. Let (W_1, W_2) be disjoint subsets of $V_S \setminus V_S^{in}$ as cut by $\mathsf{dom}(I)$. Given S and I, $\textsc{Cut}(S, I) = (T_1, T_2)$, where T_1 and T_2 are open SRPs where, for $i \in \{1, 2\}$:*

$$\begin{aligned}
V_i^{in} &= \{u \mid u \in V_S^{in} \wedge \exists uv \in E_S.\ v \in W_i\} \\
&\quad \cup \{u \mid \exists uv \in \mathsf{dom}(I).\ v \in W_i\} \\
V_i^{out} &= \{u \mid u \in W_i \wedge u \in V_S^{out}\} \\
&\quad \cup \{u \mid \exists uv \in \mathsf{dom}(I).\ u \in W_i\} \\
V_i &= W_i \cup V_i^{in} \\
E_i &= \{uv \mid u, v \in V_i \wedge uv \in E_S\} \\
R_i &= R_S \\
\oplus_i &= \oplus_S \\
\mathsf{init}_i &= \mathsf{init}_S|_{V_i} \\
\mathsf{trans}_i &= \mathsf{trans}_S|_{E_i} \\
\mathsf{ass}_i(u) &= \begin{cases} \mathsf{ass}_S(u) & \text{if } u \in V_S^{in} \\ I(uv) & \text{if } uv \in \mathsf{dom}(I) \wedge v \in V_i \end{cases} \\
\mathsf{guar}_i(u) &= \begin{cases} \mathsf{guar}_S(u) & \text{if } u \in (V_S^{out} \setminus V_i^{in}) \\ I(uv) & \text{if } uv \in \mathsf{dom}(I) \wedge v \notin V_i \end{cases}
\end{aligned}$$

The resulting SRPs T_1 and T_2 have the following properties:

- **Covering:** $V_1 \cup V_2 = V_S$ and $E_1 \cup E_2 = E_S$, with $V_1^{in} \cup V_2^{in} \supseteq V_S^{in}$ and $V_1^{out} \cup V_2^{out} \supseteq V_S^{out}$;
- **Policy preservation:** init, trans and $\oplus$ produce the same routes as in S for all possible routes in R_S;
- **Input-output nodes:** every input node u in T_1 or T_2 that is *not* an input node "inherited" from S has a corresponding output node in the other fragment, and such that $\mathsf{ass}_1(u) = \mathsf{guar}_2(u)$ or $\mathsf{ass}_2(u) = \mathsf{guar}_1(u)$;
- **Input-output nodes produced by I:** $\forall uv \in \mathsf{dom}(I)$, u is an input-output node with the above property;
- **Shared inherited inputs:** the only other nodes shared by T_1 and T_2 are input nodes into *both* fragments inherited from S: $V_1 \cap V_2 = (V_1^{in} \cap V_2^{in}) \cup (V_1^{in} \cup V_2^{in}) \setminus V_S^{in}$.

Importantly, CUT defines T_1 and T_2 to have *equal* assumptions and guarantees along each cut edge using our interface I. For each edge $uv \in \mathsf{dom}(I)$, $\textsc{Cut}(S, I)$ establishes a guarantee $\mathsf{guar}(u) = I(uv)$ in T_1 and an assumption $\mathsf{ass}(u) = I(uv)$ in T_2 (or vice-versa). By requiring guarantees and assumptions to be equal, we rely on the stability of an open SRP's solution to rule out circular (self-justifying) assumptions. As u's solution is both assumed in one fragment and guaranteed in the other, we refer to it as an *input-output node*.

[3] We could have alternatively attached auxiliary nodes to output nodes to show where routes went, but we found this definition more intuitive.

C. Fragment Solutions

1) Solutions Modulo Interfaces: The solutions of fragments produced by CUT may be a *subset* of the solutions of S. The interface I imposes new constraints on the solutions of input nodes (7) and output nodes (8), which *restrict* the solutions of these nodes to individual routes. We call these solutions that are restricted by the interface-generated constraints *solutions modulo an interface* I. Recall that an SRP has *no* solution if the conjunction of the network semantics constraint (6) and the guarantee constraint (8) is unsatisfiable at any output node. If $\textsc{Cut}(S, I)$ produces fragments with *no* solutions from a monolithic SRP S with solutions, we call its interface I an *incorrect* interface, as it enforces guarantees that do not respect the monolithic solutions and lead to divergence.

We prove that if we use CUT to produce fragments T_1 and T_2 from S, then the joined solutions of T_1 and T_2 are a solution of S; and that if S has a solution, then there *always exists* a (correct) interface I such that $\textsc{Cut}(S, I)$ produces two fragments T_1 and T_2 where the solution of S is a solution modulo I (when appropriately restricted) for T_1 and T_2. Proofs of our theorems are available in the appendix. By showing that the fragments' solutions are the same as the monolithic SRP's modulo a given interface, we can use the fragments *in place of* the monolithic SRP during verification of a property P.

We first prove that the solutions modulo a given interface I of the fragments T_1, T_2 correspond to a solution of the monolithic SRP S: each node of S maps to its fragment solution, with S's input nodes mapping to their expected assumptions.

Theorem IV.1 (Fragment Solutions Are Non-Spurious)**.** *Let S be an open SRP, and let I be an interface over S. Let $\textsc{Cut}(S, I) = (T_1, T_2)$. Suppose T_1 has a solution $\mathcal{L}_1$ and T_2 has a solution $\mathcal{L}_2$. Consider a mapping ${\mathcal{L}_S}' : V_S \to R$, defined such that:*

$$\forall v \in V_1.\ {\mathcal{L}_S}'(v) = \mathcal{L}_1(v)$$
$$\forall v \in V_2.\ {\mathcal{L}_S}'(v) = \mathcal{L}_2(v)$$
$$\forall v \in V_S^{in}.\ {\mathcal{L}_S}'(v) = \mathsf{ass}_S(v)$$

Then ${\mathcal{L}_S}'$ is a solution of S.

Proof Sketch. By case analysis on a given node v's group (base nodes, input nodes, output nodes) in S. There are three cases to consider:

- v is a base node in S: must satisfy (6);
- v is an input node in S: must satisfy (7);
- v is an output node in S: must satisfy (6) and (8).

Each case determines particular equations which must hold on a solution at v, according to the definition of an open SRP solution. For each case, the proof proceeds by showing that, since v has a solution $\mathcal{L}_i(v)$ in the fragments, that solution will satisfy the constraints to be a solution $\mathcal{L}'_S$ in S. □

Given a solution $\mathcal{L}_S$ of S, we can always find a suitable (correct) interface I to cut S, such that T_1 and T_2 have the same solution as in $\mathcal{L}_S$ for each node: we simply annotate each cut edge uv with the solution $\mathcal{L}_S(u)$, which is the solution transferred from u to v in S.

Theorem IV.2 (Correct Interface Existence)**.** *Let S be an open SRP, and let I be an interface over S. Let $\textsc{Cut}(S, I) = (T_1, T_2)$. Assume S has a solution $\mathcal{L}_S$. Assume that $\forall uv \in \mathsf{dom}(I).\ I(uv) = \mathcal{L}_S(u)$. Consider the following two mappings ${\mathcal{L}_1}' : V_1 \to R$ and ${\mathcal{L}_2}' : V_2 \to R$, defined such that:*

$$\forall v \in V_1.\ {\mathcal{L}_1}'(v) = \mathcal{L}_S(v)$$
$$\forall v \in V_2.\ {\mathcal{L}_2}'(v) = \mathcal{L}_S(v)$$

Then ${\mathcal{L}_1}'$ is a solution for T_1 and ${\mathcal{L}_2}'$ is a solution for T_2.

Proof Sketch. As with our non-spuriousness theorem above, we prove the existence of correct interfaces by case analysis on the group (base, input, output) of a given node v. We start from a solution $\mathcal{L}_S$ and show that this solution will satisfy all the same equations as a solution for T_1 and T_2, using the fact that I will define assumptions and guarantees such that $\mathsf{ass}(v) = \mathcal{L}(v)$ and $\mathsf{guar}(v) = \mathcal{L}(v)$. □

Theorem IV.1 implies that any property P that holds over our fragments' solutions modulo the interface I will hold over the corresponding solution in the monolithic network.

Corollary IV.3 (Property Preservation Modulo Interfaces)**.** *Let S be an open SRP, and let I be an interface over S. Let $\textsc{Cut}(S, I) = (T_1, T_2)$. Let P be a safety property. Assume that T_1 has a solution $\mathcal{L}_1$ and T_2 has a solution $\mathcal{L}_S$. Then if P holds on $\mathcal{L}_1$ and P holds on $\mathcal{L}_2$, P holds on the corresponding solution $\mathcal{L}_S$ of S.*

Proof Sketch. Directly by Theorem IV.1 and the definition of a safety property. □

Similarly, Theorem IV.2 implies that a property P that holds over a given solution in the monolithic network will hold over the solution modulo a (correct) interface I of each fragment. This gives a useful contrapositive for *property checking*: if a property P does *not* hold on the solution modulo I of any fragment, then P will also *not* hold on the monolithic solution.

Corollary IV.4 (Property Violation Modulo Interfaces)**.** *Let S be an open SRP, and let I be an interface over S. Let $\textsc{Cut}(S, I) = (T_1, T_2)$. Let P be a safety property. Assume that S has a solution $\mathcal{L}_S$, and T_1 and T_2 have solutions modulo I: $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively. If P does* not *hold on $\mathcal{L}_1$ or $\mathcal{L}_2$, then P does* not *hold on $\mathcal{L}_S$.*

Proof Sketch. The contrapositive follows directly by Theorem IV.2 and the definition of a safety property. □

D. SRPs with No or Multiple Solutions

We now elaborate on the special cases when CUT takes SRPs with no or multiple solutions as input.

1) Divergent SRPs: When an SRP has no solution, at least one of its fragments will also have no solution. This follows directly from our non-spurious fragment solutions theorem: suppose (for contradiction) that an SRP has no solution but all of its fragments do (*i.e.,* they have spurious solutions). Then by Theorem IV.1, the combined solution of those fragments is a solution to the SRP.

Figure 7 presents an example SRP with no solution, for the BAD GADGET example taken from [35]. The routes in

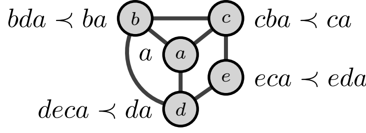


Fig. 7: The BAD GADGET SRP from [35]

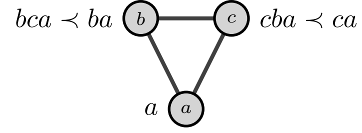


Fig. 8: The DISAGREE SRP from [35]

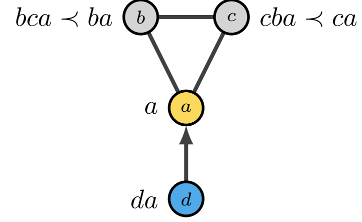


Fig. 9: A fragment of the DISAGREE+1 SRP with two solutions.

this network are paths (finite sequences of nodes) π terminating at a destination v_n, *i.e.,* $\pi : v_1 v_2 \dots v_n$, such that $\forall i < n.\ v_i v_{i+1} \in E$. When a node transfers a route, it prepends itself to the path: $\mathsf{trans}(uv, u\pi) = vu\pi$. $\oplus$ defines a partial order $\preceq$ over paths: $\pi_1 \oplus \pi_2 = \pi_1$ *iff* $\pi_1 \preceq \pi_2$. The reflexive reduction of $\preceq$ is a strict partial order $\prec$, where $\pi_1 \prec \pi_2 \Leftrightarrow \pi_1 \preceq \pi_2 \wedge \pi_1 \neq \pi_2$. Node a advertises a route to itself to nodes b, c and d, which then share these routes with their neighbors. a's neighbors all prefer to route via one another rather than directly to a, which leads the network to never converge to a stable solution.

Since there is no solution to BAD GADGET, there will be no interface possible such that all fragments have a solution. Suppose we cut e off from the rest of the network, and gave the interface:

$$I = \{de \mapsto deca, ce \mapsto ca, ec \mapsto eca, ed \mapsto eca\}$$

Then the fragment containing e will have a solution that precisely matches the interface routes, namely

$$\mathcal{L}(c) = ca \qquad \mathcal{L}(d) = deca \qquad \mathcal{L}(e) = eca$$

But the fragment containing a, b, c and d will have no solution! The problem stems from our guarantees: c must have a route ca, which means it must not have been offered a route ba by b; for that to be the case, b must have its more preferred route bda; but that means that d's route would have to be da, which violates the guarantee that d's route is $deca$! This captures the same divergent routing behavior as in the monolithic network.

A similar problem occurs if the interface specifies the next-best route at e:

$$I = \{de \mapsto da, ce \mapsto cba, ec \mapsto eda, ed \mapsto eda\}$$

The fragment containing e has a new solution now (again as defined by the interface), but the larger fragment still diverges. If d chooses the route da, then b will receive a route bda, which it prefers over ba, and therefore c will select the route ca, thereby breaking its guarantee to select the route cba.

2) SRPs with Multiple Solutions: SRPs with multiple solutions can produce fragments with multiple solutions. To illustrate this idea, consider the network shown in Figure 8, for the DISAGREE example taken from [35]. Once again, routes are paths. Node a advertises a route to itself to nodes b and c: both b and c prefer routes via one another over routing directly to a. As shown in [35], this network converges to two distinct solutions:

$$\textit{Solution 1:} \quad \mathcal{L}(a) = a \qquad \mathcal{L}(b) = bca \qquad \mathcal{L}(c) = ca$$
$$\textit{Solution 2:} \quad \mathcal{L}(a) = a \qquad \mathcal{L}(b) = ba \qquad \mathcal{L}(c) = cba$$

Suppose we cut the DISAGREE network into two fragments T_a and T_{bc} using the interface:

$$I_1 = \{ab \mapsto a, ac \mapsto a, ba \mapsto bca, ca \mapsto ca\}$$

The resulting fragment T_{bc} containing nodes b and c has a single solution: the first one given above. The second solution to the monolithic network cannot satisfy the guarantees that $\mathcal{L}(b) = bca$ and $\mathcal{L}(c) = ca$. If we cut the network using a different interface:

$$I_2 = \{ab \mapsto a, ac \mapsto a, ba \mapsto \boxed{ba}, ca \mapsto \boxed{cba}\}$$

Then the resulting fragment again has a single solution, the *second* one.

In both cases above, we obtained fragments with one solution from a monolithic network with two solutions. This was due to the interface fixing the routes at both nodes which have multiple solutions in the monolithic network. Other networks and cuts can produce fragments with multiple solutions. Suppose we had an alternative SRP "DISAGREE+1", which resembled the DISAGREE network, but with an additional node d connected to node a, that like b and c seeks to route to a. We may cut the network along the ad edge to produce a fragment like the one shown in Figure 9. This fragment will have (at most) two solutions (the two specified above), while its sibling fragment (containing only a and d) will have (at most) one (the path da).[4]

3) Properties of Fragment Solutions: Corollary IV.3 says that a property of DISAGREE's solutions modulo these interfaces holds on the corresponding solution(s) in the monolithic network. In this example, the fragment has two different solutions modulo the two interfaces I_1 and I_2. If we wanted to check a property such as "all nodes in DISAGREE have a path of at most 2 hops to a", we could check that property against both solutions by cutting DISAGREE once with I_1 and again with I_2. We could then conclude that, since this property holds on both possible fragments, it holds on both solutions of the monolithic network.[5] On the other hand, Corollary IV.4 says that, if P does *not* hold on a fragment's solution, then we know that P also does *not* hold on the monolithic network. We can

[4]We qualify with "at most" since, if the interface is incorrect, then the fragments will have no solution.

[5]In monolithic verification, when a network has multiple solutions, a property that may hold on one solution may not hold for the others. We must therefore be sure to have considered interfaces that capture all the network's solutions before arguing that a property holds on all solutions when performing modular verification.

Algorithm 1 The fragment checking algorithm.

```
1:  proc SOLVE(fragment T, property P)
2:      M ← ENCODE(T)                                        ▷ (6)
3:      A ← ⋀_{u∈V_T^in} L_T(u) = ass_T(u)                   ▷ (7)
4:      G ← ⋀_{u∈V_T^out} L_T(u) = guar_T(u)                 ▷ (8)
5:      Q ← ⋀_{u∈V_T^base∪V_T^out} P(u, L_T(u))     ▷ property check
6:      return ASKSAT(A ∧ M ∧ ¬(G ∧ Q))

7:  proc CHECK(SRP S, property P, interface I)
8:      T_1, ..., T_n ← CUT(S, I)
9:      for i ← 1, n do in parallel
10:         r ← SOLVE(T_i, P)
11:         if r ≠ UNSAT then
12:             return r
13:     return UNSAT
```

use an SMT solver to search for solutions to the fragments of S such that the negated property $\neg P$ holds for *any* fragment. We present our technique for doing so in the following section.

V. Checking Fragments in SMT

We now present our three-step modular verification methodology: *(i)* given an SRP S and an interface I, produce n fragments using $\textsc{Cut}(S, I)$; then *(ii)* in parallel, encode each fragment to SMT and check its guarantees and a safety property P under the given assumptions; and *(iii)* if any guarantees fail, let the user *refine* I or correct network bugs. If the SMT solver verifies P and all guarantees over S's fragments, we can conclude that it has verified P over S. Our algorithm below can verify guarantees and properties of networks with *at most one* solution for every output node: if given a multi-solution network, this algorithm may report false negatives (spurious counterexamples). These false negatives represent cases where an output node has multiple solutions, but the given interface's guarantee can only capture one of them. We discuss an alternative algorithm that does not suffer from such false negatives in §VIII.

A. The Fragment Checking Algorithm

Algorithm 1 shows how we cut an SRP and check the three constraints on open SRP solutions (described in §IV-A1) on each of the fragments. We start in the CHECK procedure on line 1.7. CHECK calls $\textsc{Cut}(S, I)$ to cut S into fragments, and then calls SOLVE (line 1.1) on each fragment, reporting any SAT result it receives back from the solver. SOLVE encodes (6) on line 1.2, (7) on line 1.3, (8) on line 1.4, and the check that P holds on line 1.5. Since we want to know if G or P are ever violated, our query formula conjoins M (*i.e.,* $\textsc{Encode}(T)$) and A with the negation of $G \wedge Q$ (line 1.6). ASKSAT asks an SMT solver if this formula is satisfiable, and returns either SAT with a model, or UNSAT. This model will be a counterexample mapping L from V_T to R_T where the $\textsc{Encode}(T)$ and A constraints hold, but $\exists u \in V_T^{out}.\ L_T(u) \neq \mathsf{guar}_T(u)$ (guarantee violation) or $\exists u \in V_T.\ L_T(u) \notin P(u)$ (property violation). Otherwise, if the solver returns UNSAT, then the guarantees and property always hold, or the network has no solution.

B. Refining Interfaces

Assuming the network has a solution, if every fragment's guarantee check and property check return UNSAT, by Theorem IV.1 we may conclude that our interface is *correct* (it produces fragments which capture a monolithic solution), and by Corollary IV.3 we have that P holds for this monolithic solution since it holds for the fragments' solutions. However, if any fragment returns SAT, we must determine why our guarantees or property were violated. For example, in §II, we considered if our interface correctly captured the intended network behaviour, but a bug in the network policy led to a guarantee violation. If the reverse were true — our network is configured correctly, but our interface is incorrect — we must *refine* our interface to correct it. Both cases may be common in practice, and point to the importance of *checking* our interfaces: counterexamples provide users with insight into why the network's actual behavior does not conform to their beliefs. Other annotation checking tools like Dafny [42] may use a similar interactive process of refining interfaces as the user identifies inconsistencies or bugs.

By Theorem IV.1, we know that any incorrect interface will not define a solution in T_1 and T_2, meaning our check in SOLVE will fail and return a counterexample. This counterexample may then suggest a new interface we can provide in a successive run of CHECK. Returning to our fattree fragments in Fig. 4, suppose we used the same interface except for an incorrect annotation $I(c_0a_4) = \langle\, d, 1\,\rangle$. To check the corresponding guarantee, we generate a constraint $\mathcal{L}_{spines}(c_0) = \langle\, d, 1\,\rangle$. $\textsc{Solve}(T_{spines}, P)$ returns SAT, providing $\mathcal{L}_{spines}(c_0) = \langle\, d, 2\,\rangle$ as a counterexample. We can then inspect T_{spines} to see that $\langle\, d, 1\,\rangle$ is not a possible route given the assumptions on c_0's input nodes, and correct our interface to specify $\langle\, d, 2\,\rangle$ instead.

VI. Implementation

We built Kirigami on top of the NV language [23]. NV represents routing protocols, such as BGP, OSPF and RIP, using an SRP-like model and a toolbox of types and data structures such as booleans, integers, tuples, records, maps and non-recursive functions. Our Kirigami extension adds `partition` and `interface` functions to NV: when we run NV on a file that declares these functions, NV cuts the SRP into fragments as described by Definition IV.3 of CUT. The `partition` function maps each node to a fragment, while the `interface` function defines the interface I.

Kirigami's SMT encoding follows Algorithm 1, using NV's monolithic SRP encoding as the encoding function ENCODE. Our implementation uses the OCaml Parmap library [43] to parallelize fragment-specific work, including decomposing properties and the embarrassingly-parallel SOLVE procedure. The only limit on parallelism is the number of CPU cores.

VII. Evaluation

We evaluated Kirigami on a variety of NV benchmarks representing fattree, random and Internet topologies. Our questions focus on the scalability and performance of Kirigami in comparison to NV, specifically: *(i)* does Kirigami improve on NV verification time across topologies and properties, and *(ii)* how do different cuts impact Kirigami performance? We consider two metrics for verification time: the maximum time reported to verify an SMT query encoding the monolithic network or fragment using the Z3 [44] SMT solver;[6] and the "total time" of NV, which is the time taken by NV's pipeline of network transformations, partitioning (for cut networks), encoding to SMT and solving the query or queries.

We ran each benchmark on a computing cluster node with a 2.4GHz CPU and up to 128GB of memory per CPU core. For cut benchmarks, we parallelized partitioning and solving over up to 32 cores.[7] Each benchmark tested verification of either the monolithic network or a cut network. We timed out any benchmark that did not finish solving a Z3 query in 2 hours.[8]

A. Fattrees

To evaluate Kirigami's performance for fattrees, we made use of the shortest path policy SP and valley-free policy FAT described in [23], along with two extensions: an all-edge reachability policy AP and an original fault-tolerance policy MAINT. Whereas SP checks reachability of a fixed prefix of an edge layer node, AP verifies that all prefixes of edge layer nodes are reachable using a symbolic variable for the set of possible prefixes. MAINT extends SP by requiring that nodes avoid routing through a non-destination node `down` which is currently down for maintenance. We check this property for any down node by encoding `down` as a symbolic value. The set of routes R modelled routing using a combination of eBGP, connected and static routes: for eBGP, we represented its fields with bitvectors representing local preference, AS path length, the multi-exit discriminator (MED), and a set of integer BGP community tags; for connected and static routes, we use integers representing the next hop; we model the choice between eBGP, connected and static routing using 2 bits.

As in [23], we parameterize fattree designs by k, the number of pods: we vary the topology size from $k = 4$ (20 nodes) to $k = 20$ (500 nodes) to assess scalability.[9] Furthermore, we consider four different cuts of our fattree networks:

- **Vertical**: creates 2 fragments, each with half the spines and half the pods ($\frac{5k^2}{8}$ nodes);
- **Horizontal**: creates 3 fragments: the pod containing the routing destination (k nodes), the spines ($\frac{k^2}{4}$ nodes), and all the other pods ($k^2 - k$ nodes);
- **Pods**: creates $k+1$ fragments (given k pods): the spine nodes, and each pod (k nodes) in its own fragment; and
- **Full**: creates $|V|$ fragments (given $|V|$ nodes), with every node in its own fragment.

To simplify the process of coming up with interfaces for evaluating these parametric networks, we wrote a script to generate interfaces for each of our policies. We computed BGP AS path lengths for each node using graph algorithms, and then generated the other route fields according to the policy's behavior.[10] For SP, we only computed shortest paths. For AP, we determined path lengths based on a node's tier relative to the symbolic destination node, as presented in §II. For FAT, we replicated how the policy sets community tags according to each pod's tier to block down-up-down routes (valleys) [45], [46]. For MAINT, we used Yen's 2-shortest paths algorithm [47]: this gives the shortest and second-shortest path (taken if `down` lies on the shortest path) to the destination from each node. We then assigned routes conditioned on the length of the shortest path avoiding `down`.

We compare SMT verification time for monolithic benchmarks versus their cut counterparts in Fig. 10. We plot the number of nodes in the monolithic benchmark against the maximum time spent by Z3 solving the SMT queries. Time is shown on a logarithmic scale. All policies show extreme improvements in SMT time as the number of fragments grows. The maximum SMT time for a full cut fragment of our largest SP benchmark is *six orders of magnitude* faster than the monolithic time. The FAT policy's SMT encoding is most complex, leading to timeouts for the monolithic FAT16 and FAT20 benchmarks: monolithic SMT solving also times out for AP20 and MAINT20. Most fragments are proportionally sized in terms of non-input nodes[11], but fragments with more input nodes such as spine fragments or with more complex policies tend to take longer to solve.

Fig. 11 plots the times for end-to-end NV verification, again on a logarithmic scale. These times include partitioning the network and encoding queries to SMT. SMT encoding and solving dominate all other operations in all benchmarks, except for fully-cut benchmarks, where partitioning is the longest-running operation (between 25–65% of NV time). Partitioning takes up to 25% of NV time for the pods and horizontal cuts: thanks to the reduced SMT time for these benchmarks, they see the best speedup relative to the monolithic benchmarks, *e.g.,* for $k = 16$ benchmarks, we see the pods cut finish 5–25 times faster than the monolithic benchmark. The pods and full cut benchmarks for FAT20 hit our memory limit with 128 GB per core. We increased available memory while decreasing cores (16 cores, 256 GB for pods; 8 cores, 512 GB for full) to handle them (results in Fig. 11c). This does not affect Z3 times, but NV times could improve with more available memory.

B. Random Networks

We next assess Kirigami with randomly-generated topologies of N nodes using the Erdős–Rényi–Gilbert model [48],

[6]As we can solve each fragment SMT query independently, the maximum query time is an upper bound on the total SMT solve time when we solve every fragment in parallel.

[7]Benchmarks with $i < 32$ fragments ran in parallel on i cores. As each core could use up to 128 GB of memory, the total memory available was $128i$ GB (up to 4 TB).

[8]Cut benchmarks call Z3 multiple times, and hence had a two-hour limit on individual calls: total NV time could then exceed 2 hours.

[9]A k-fattree has $\frac{5}{4}k^2$ nodes and k^3 edges.

[10]We used Kirigami's counterexamples to debug mistakes in our reasoning when writing our script, as described in §V.

[11]The horizontal benchmark is a notable exception, where the "non-destination pods" fragment is significantly larger.

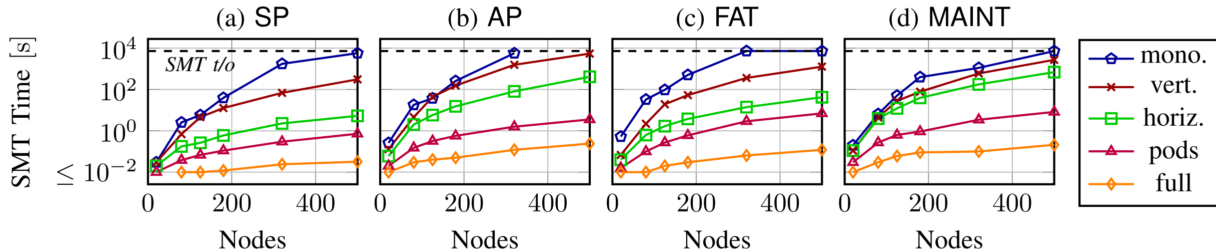


Fig. 10: Largest SMT solve times for fattree benchmarks.

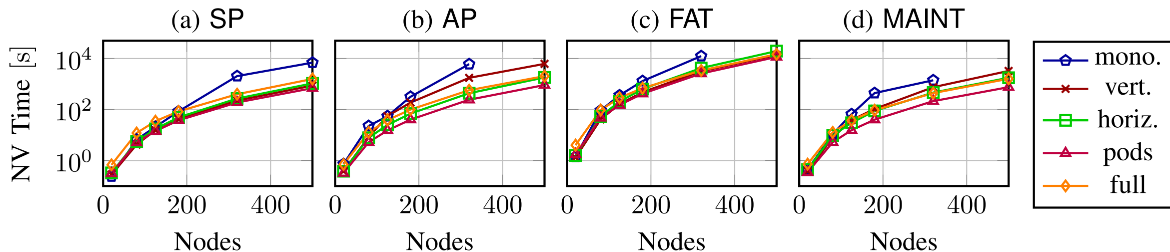


Fig. 11: NV times for fattree benchmarks.

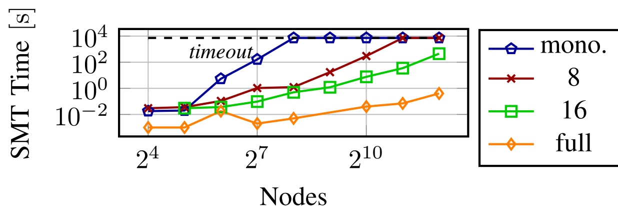


Fig. 12: Largest SMT solve times for random networks.

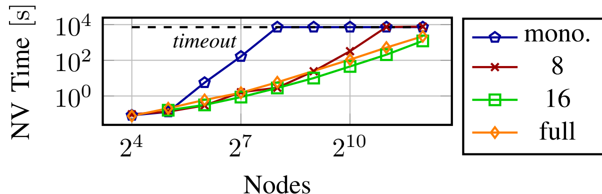


Fig. 13: NV times for random networks.

[49], where each edge has independent probability p of being present. To assess scalability, we vary N and p in our experiments according to a parameter x where $N = 2^x$ and $p = 2^{2-x}$ for $x \in [4, 12]$: these choices lead to networks where 2–6% of nodes are disconnected from the others. We use BGP routes and a pure shortest-path policy based on SP for these networks, and check that nodes can reach the destination: we then report counterexamples for disconnected nodes and a positive verification result for connected nodes. Once again, we used a script to generate interfaces using a shortest paths algorithm. To choose cuts, we use a graph partitioning tool, hMETIS [50], to compute i fragments for each network. The computed fragments minimize the number of edges cut between fragments, and capture clustering behavior of the topology, while minimizing variance in fragment size. We consider $i = 8$ and $i = 16$, and a full cut ($i = |V|$).

We show the maximal SMT solve times for these benchmarks in Fig. 12 and NV times in Fig. 13.[12] Monolithic verification hits our Z3 timeout at $N = 256$; with 8 fragments, verification times out on the 8 times larger $N = 2048$ network. With 16 fragments, Z3 verifies $N = 4096$ in under 8 minutes, and when fully partitioning, Z3 takes at most 0.4 seconds to finish, with NV terminating for $N = 4096$ after 37 minutes.

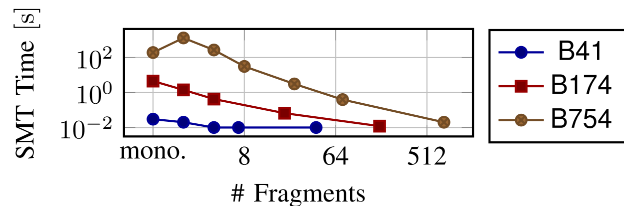


Fig. 14: Largest SMT solve times for TopologyZoo networks.

C. Backbone Networks

We evaluated Kirigami with backbone network topologies from the Internet Topology Zoo [51]. We consider three networks: a 41-node (50-edge) topology B41, a 174-node (205-edge) topology B174 and a 754-node (895-edge) topology B754. As before, we model BGP routing throughout. B41's policy enforces no-transit and drops routes transiting [45] through AS customers or peers (relationships inferred from the topology [52]). B174 and B754 use a shortest-path policy as in SP. As for random networks, we compute i fragments using hMETIS. We consider $i = 2, 4, 7, 41$ for B41, $i = 2, 4, 20, 174$ for B174, and $i = 2, 4, 8, 25, 75, 754$ for B754.

Fig. 14 and Fig. 15 show how the number of fragments affects SMT solve time and NV time, respectively. Like other benchmarks, larger cuts lead to greater reductions in SMT solve time, while NV time is lowest for non-full cuts ($i = 4$ for B174 and $i = 25$ for B754).

[12] We included verification times for both connected and disconnected fragments: we did not see significant differences between the two.

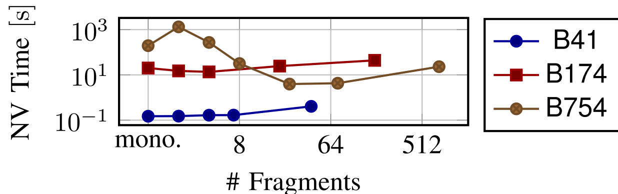


Fig. 15: NV times for TopologyZoo networks.

VIII. Checking Multi-Solution Fragments

Theorems IV.1 and IV.2 respectively ensure that, given solutions to our fragments, we have a solution to the monolithic SRP, and that a solution to the monolithic SRP is also a solution to our fragments. Practically speaking, it is easier to reason over a network's behavior when the network converges to a single solution, and multiple solutions are hence usually a sign that something is *wrong* in a network. That being said, there is theoretical interest in understanding how to design a verification procedure to check solutions of multi-solution networks. Such a procedure, like our Algorithm 1, should be able to check that a user's interfaces are correct and that their properties of interest hold.

We remarked in §V that Algorithm 1 is designed for networks with at most one solution for every output node, and may report spurious guarantee violations for multi-solution networks. This is because the algorithm checks if there exists a mapping that satisfies the assumptions and network semantics, but violates its guarantees. In a fragment with a single solution, this is sufficient to determine that the interface is incorrect, as the counterexample mapping will correspond to the sole solution of the fragment. But if there are multiple solutions satisfying a fragment's network semantics, there will *always* exist an alternative solution that violates the guarantees.

To resolve this problem, we can modify our algorithm to let users provide *multiple* interfaces. Given a fragment where a particular choice of assumptions can satisfy multiple guarantees, the algorithm should ask if there exists *another* mapping that violates *all* of the guarantees specified so far, given this choice of assumptions. When the solver finds that another mapping (*i.e.,* a counterexample) exists, the user can examine it to decide if this counterexample represents (i) an unexpected behavior (suggesting a bug), or (ii) another intended solution the network converges to, but which none of the guarantees considered. In the first case, we can debug the network as before; in the latter case, the user could rerun verification with an additional interface representing this intended solution. This process resembles an All-SMT enumeration of the fragments' solutions, which the user can terminate at any point, once they have considered all the solutions of interest.

1) Fragment matrices: To check a choice of assumptions against multiple guarantees at once, we need to group all the fragments of our network by their assumptions ass. Different fragments' assumptions may differ, either in terms of (a) their domain $\mathsf{dom}(\mathsf{ass})$, determined by which nodes are in the fragment; and (b) their image, determined by the routes assigned by the interfaces. We first group the fragments by their assumptions' domains, and then by their image. The first grouping is straightforward from how the network is cut. If we provide m interfaces that all cut the same set of edges of a monolithic network N, after performing the m cuts there will be $m \times n$ fragments in total. We call the entire group of $m \times n$ fragments a *fragment matrix*. Each of the m rows of the matrix represent all the fragments produced by one interface, while each of the n columns represent all the annotations over one "sub-topology" (Figure 16 shows four fragments that would all belong in the same column). We refer to each column of the fragment matrix as *equal modulo interfaces*: since we used different interfaces over the same set of edges, by the definition of Cut, all the fragments in the column have equal components apart from their ass and guar functions.[13]

Definition VIII.1 (Fragment Equality Modulo Interfaces)**.** *Let T_1 and T_2 be fragments. T_1 is* equal modulo interfaces *to T_2 if*

$$G_1 = G_2 \wedge R_1 = R_2 \wedge \oplus_1 = \oplus_2 \wedge \\ \mathsf{trans}_1 = \mathsf{trans}_2 \wedge \mathsf{init}_1 = \mathsf{init}_2$$

For example, given three interfaces I_1, I_2 and I_3 which cut an SRP S into a "left" and "right" fragment, we cut S with the interfaces to produce a 3-by-2 *matrix* $\mathbf{T}$ of *six* fragments $T_{1,1}, T_{1,2}, T_{2,1}, T_{2,2}, T_{3,1}, T_{3,2}$. The columns $\mathbf{T}_{*,1} = (T_{1,1}, T_{2,1}, T_{3,1})$ and $\mathbf{T}_{*,2} = (T_{1,2}, T_{2,2}, T_{3,2})$ of this matrix then group together the "left" fragments and "right" fragments respectively, such that every fragment in the column is equal modulo interfaces I_1, I_2, I_3.

Even though each column is equal modulo interfaces, individual interfaces may assign different assumptions and guarantees to the input and output nodes, so that the images of ass and guar differ across fragments. Different assumptions ass can produce different routing behaviors: as such, to check multiple guarantees, we want to identify each group of identical assumptions that applies to one or more possibly-different guarantees. To do so, we subdivide each column of fragments into *assumption groups*, such that all fragments in the assumption group also have equal ass. This ensures that we verify all output routes resulting from each choice of input assumptions. We can then check that each choice of assumptions produces one of the provided guarantees in a single SMT query.

For example, we have three fragments that are equal modulo interfaces in column $\mathbf{T}_{*,1} = (T_{1,1}, T_{2,1}, T_{3,1})$. Suppose that $\mathsf{ass}_{1,1} = \mathsf{ass}_{2,1}$, but $\mathsf{ass}_{1,1} \neq \mathsf{ass}_{3,1}$ and $\mathsf{ass}_{2,1} \neq \mathsf{ass}_{3,1}$. Then the column subdivides into two assumption groups: $T_1^{\mathsf{ass}} = \{T_{1,1}, T_{2,1}\}$ and $T_2^{\mathsf{ass}} = \{T_{3,1}\}$.

The multi-solution checking algorithm is shown in Algorithm 2. Line 2.8 shows our Check procedure from before, which now takes a variable number m of interfaces $I_1, \dots, I_m$ as input. Check requires that all interfaces have the same domain, meaning they cut the same edges: $\mathsf{dom}(I_1) = \mathsf{dom}(I_2) = \dots = \mathsf{dom}(I_m)$. It then loops over the interfaces to construct the fragment matrix $\mathbf{T}$ on line 2.10. Next,

[13]Under the usual extensional meaning of function equality, *i.e.,* two functions $f : A \to B$ and $g : C \to D$ are equal *iff* $A = C$, $B = D$ and $\forall x \in A, f(x) = g(x)$.

Algorithm 2 The multi-solution fragment checking algorithm.

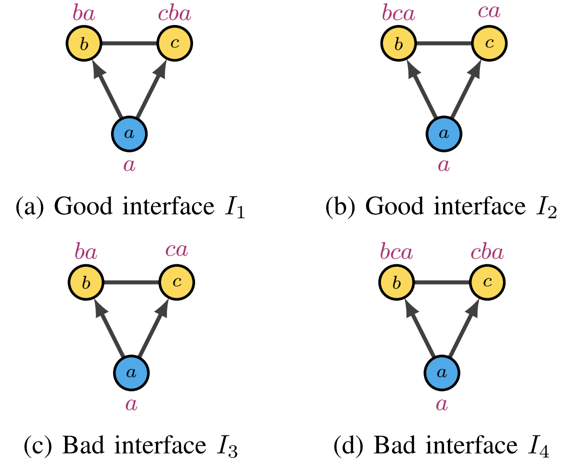
Require: $\forall T_1, T_2 \in T^a.\ \mathsf{ass}_1 = \mathsf{ass}_2$
1: **proc** SOLVE(assumption group T^a, property P)
2: $\quad T \leftarrow$ *choose representative* $T \in T^a$
3: $\quad M \leftarrow$ ENCODE(T) ▷ (6)
4: $\quad A \leftarrow \bigwedge_{u \in V_T^{in}} \mathcal{L}_T(u) = \mathsf{ass}_T(u)$ ▷ (7)
5: $\quad G \leftarrow \bigvee_{T_i \in T^a} \bigwedge_{u \in V_T^{out}} \mathcal{L}_T(u) = \mathsf{guar}_{T_i}(u)$ ▷ (8)
6: $\quad Q \leftarrow \bigwedge_{u \in V_T^{base} \cup V_T^{out}} P(u, \mathcal{L}_T(u))$ ▷ property check
7: $\quad$ **return** ASKSAT($A \wedge M \wedge \neg(G \wedge Q)$)

Require: $\mathsf{dom}(I_1) = \mathsf{dom}(I_2) = \ldots = \mathsf{dom}(I_m)$
8: **proc** CHECK(SRP S, property P, interfaces $I_1, I_2, \ldots, I_m$)
9: $\quad$ **for** $i \leftarrow 1, m$ **do in parallel**
10: $\quad\quad T_{i,1}, \ldots, T_{i,n} \leftarrow$ CUT(S, I_i) ▷ fragment matrix
11: $\quad$ **for** $j \leftarrow 1, n$ **do in parallel**
12: $\quad\quad T_1^a, T_2^a, \ldots, T_o^a \leftarrow$ ASSUMPTIONGROUPS($\mathbf{T}_{*,j}$)
13: $\quad\quad$ **for** $k \leftarrow 1, o$ **do in parallel**
14: $\quad\quad\quad r \leftarrow$ SOLVE(T_k^a, P)
15: $\quad\quad\quad$ **if** $r \neq$ UNSAT **then**
16: $\quad\quad\quad\quad$ **return** r
17: $\quad$ **return** UNSAT

each column $\mathbf{T}_{*,j}$ (corresponding to all fragments over the same subgraph) is subdivided by the ASSUMPTIONGROUPS procedure on line 2.12, which partitions the column into o assumption groups $T_1^a, T_2^a, \ldots, T_o^a$. We use a modified SOLVE procedure to check each assumption group on line 2.1. Since all the fragments in the assumption group are equal apart from their guarantees guar, SOLVE arbitrarily picks a "representative" fragment T from the group to encode the network semantics as a formula M (line 2.3) assumptions as a formula A (line 2.4) and property check as a formula Q (line 2.6) exactly as in Algorithm 1. The change is on line 2.5: we modify the guarantee formula G to take the disjunction of *all* fragments' guarantees guar_i for all $T_i \in T^a$. We can then check each group T^a with a single SMT query (line 2.7): given the network semantics encoding M and the assumptions A, does there exist a solution $\mathcal{L}_T$ which satisfies M and A but violates *all* fragments' guarantees or the property P? Our total number of SMT queries is the number of fragments times the number of *distinct* (non-equal) assumptions given per fragment, $|\{\mathsf{ass}_T \mid T \in \text{CUT}(S, I_j), 1 \leq j \leq m\}|$, which is upper bounded by $n \times m$.

2) Checking Guarantees with Algorithm 2: Let's explore how our algorithm works by using the DISAGREE network presented earlier in §IV-D. Figure 16 presents the T_{bc} fragment of the network (a is an input node, b and c are output nodes), annotated with four different interfaces: two good interfaces I_1 and I_2, corresponding to solutions of the monolithic network; and two bad interfaces I_3 and I_4, which do not correspond to monolithic solutions.

a) One good interface: Suppose we cut the network using interface I_1 shown in Figure 16a. This interface represents one of the two possible solutions to the monolithic network, but it is not the only one. We may ask the solver to find us



(a) Good interface I_1 (b) Good interface I_2 (c) Bad interface I_3 (d) Bad interface I_4

Fig. 16: The T_{bc} fragment of the DISAGREE SRP from [35], annotated with four different interfaces. Interfaces I_1 and I_2 are "good" interfaces which capture solutions of the monolithic network, while I_3 and I_4 are "bad" interfaces which do not correspond to solutions of the monolithic network.

another solution to T_{bc}, such that $\mathcal{L}(b) \neq ba$ or $\mathcal{L}(c) \neq cba$. In this case, it returns the counterexample:

$$\mathcal{L}(a) = a \qquad \mathcal{L}(b) = bca \qquad \mathcal{L}(c) = ca$$

which corresponds to interface I_2 shown in Figure 16b. As shown in §IV-D, this is the other monolithic solution!

b) All good interfaces: We can rerun verification using *both* interfaces I_1 and I_2, asking the solver to find a solution after blocking the earlier solutions, *i.e.,*

$$(\mathcal{L}(b) \neq ba \vee \mathcal{L}(c) \neq cba) \wedge (\mathcal{L}(b) \neq bca \vee \mathcal{L}(c) \neq ca)$$

This query is unsatisfiable, so the solver returns UNSAT, allowing us to conclude that there does not exist any other solution to the fragment.

c) Bad interfaces: Suppose instead that we called CHECK with interface I_3 (Figure 16c). This interface does *not* correspond to a solution. When we provide it to the solver, we once again receive a counterexample corresponding to one of the two previous interfaces I_1 or I_2. Calling CHECK with I_3 and the additional interface I_4 (Figure 16d) produces a similar response. We ask the solver for a solution to T_{bc} such that $(\mathcal{L}(b) \neq ba \vee \mathcal{L}(c) \neq ca) \wedge (\mathcal{L}(b) \neq bca \vee \mathcal{L}(c) \neq cba)$: the solutions captured by I_1 and by I_2 both satisfy this formula, so the algorithm again returns a counterexample.

d) Non-minimal interface sets: Observe that a limitation of this algorithm is it does not identify whether any of the given interfaces correspond to non-solutions, or suggest whether the given set of interfaces can be "trimmed" to remove non-solutions. For example, if we provided interfaces I_2, I_3, I_4 to the algorithm, it will return the same counterexample

$$\mathcal{L}(a) = a \qquad \mathcal{L}(b) = ba \qquad \mathcal{L}(c) = cba$$

just as if we'd given only I_2, or I_2 and I_3, or I_2 and I_4 as potential interfaces. We therefore should interpret a result of UNSAT as stating that the given interfaces capture all solutions of the monolithic network, and that the property holds on

them — but not that the given interfaces are the minimal such set. Indeed, if a fragment has *no* solution, then, no matter the property we provide, our procedure will return UNSAT— much like any other verification procedure that searches for property *violations* or behaviors that should *not* occur.

3) Checking Properties with Algorithm 2: Checking properties via Algorithm 2 works similarly to how it did in Algorithm 1, but with the caveat that interfaces may potentially constrain the solutions of the fragment relative to the monolithic network, as discussed in §IV-D. For instance, in the example here, the M and A constraints in SOLVE allow for both a ba and a bca route at node b. Therefore, if we supplied a property P to check that node b has a route ba *or* bca — $P(b, \mathcal{L}(b)) \equiv \mathcal{L}(b) = ba \vee \mathcal{L}(b) = bca$, our SMT query in SOLVE would ask if there exists a solution to our fragment such that b does *not* have such a route. This property holds regardless of the given interface — no interface could constrain $\mathcal{L}(b)$ to be any other value such that $\neg Q$ is satisfiable — and we will hence be able to conclude that P is valid.

If on the other hand we supplied an incorrect property P' where $P'(b, \mathcal{L}(b)) \equiv \mathcal{L}(b) = ba$, then our algorithm will ask the solver if there exists a solution where b does not have the route ba. In this case, the solver can find a property counterexample where b's route is bca. The exact counterexample returned depends in part on the interfaces provided. As line 2.6 shows, the solver may search for a guarantee violation *or* a property violation, so bad interfaces may lead to only guarantee violations, but can also produce guarantee and property violations, *e.g.,* calling CHECK with interface I_4 violates the guarantees and the property $\mathcal{L}(b) = ba$.

If we cut the T_{bc} fragment again to produce two fragments T_b and T_c, T_b and T_c will have only one solution each, depending on the choice of interface. For instance, if T_b assumes a route $\mathcal{L}(c) = bca$, $\mathcal{L}(b)$ is guaranteed to be ba. Checking an interface akin to I_2 will hence not reveal if the property P' holds of all solutions of b in the monolithic network. We must provide another interface where $\mathsf{ass}(c) = ca$ and $\mathsf{guar}(b) = bca$ to cover b's other monolithic solution.

IX. RELATED WORK

A. Data Plane Analysis

Much prior work has analyzed properties of the network data plane [6]–[13]. These tools operate on snapshots of the data plane — representing the global forwarding state at a single point in time — and verify that forwarding properties are satisfied.

Our approach most closely resembles the work of Jayaraman *et al.* on SECGURU and RCDC [13]. SECGURU verifies reachability using invariants it infers from specific data center topologies: our work develops a formal theory to verify arbitrary properties and invariants as specified by a user's interface, provides a framework for doing so automatically and instead focuses on the control plane.

Another relevant work is that of Plotkin *et al.* [12]. They demonstrate the use of bisimulations to relate simpler networks and formulas to more complex ones, improving verification scalability. Modular verification is recognized as a viable direction but left as future work; we focus on using modular verification in the control plane.

B. Control Plane Analysis

Our open SRP model builds on work on formal models of control planes, in particular, Bonsai's SRP model [19]. Other prior work [35], [40], [41] considered properties of routing algebras that are necessary to ensure networks converge to a unique solution; we instead focus on checking safety properties of these networks' solutions, irrespective of how many solutions exist.

Two efforts closest to our own in modular control plane verification are Lightyear [53] and Timepiece [54]. Lightyear verifies safety properties for BGP networks using local invariant checks on a network's nodes and edges. Invariants in Lightyear are formulas rather than concrete routes, reducing the annotation burden; however, it cannot verify reachability properties and is restricted to BGP. Timepiece likewise chooses different tradeoffs to Kirigami: while it also allows users to specify arbitrary invariants, the user must provide specific times for their invariants and reason over when routes arrive, which may be difficult to do for complex networks.

Other control plane verification tools scale by abstracting routing behaviors, rather than modularizing the network. Bonsai [19] and Origami [20] *compress* concrete networks to smaller abstract networks which soundly approximate the original. Compression requires similar forwarding behavior between nodes; our approach avoids this restriction.

Our SMT encoding is inspired by Minesweeper [15], although we do not consider packet forwarding (only routing) and Minesweeper cannot perform modular verification. Several other tools [21]–[25], [28] use simulation-based techniques to scale verification up to large networks, but make pragmatic choices as to what arbitrary behaviors they can represent or properties they can verify. For instance, Plankton [24] uses explicit-state model checking to check a comparable set of properties to Minesweeper. Plankton can analyze networks with symbolic packets by using equivalence classes, but appears to need additional support to model other routing characteristics symbolically. None of these tools modularize the network: one could potentially extend them with modular techniques to improve their scalability. Other analyses also do not consider modularizing the network, and many are more restrictive than our approach: either limited to specific network properties [16], [17] or to specific protocols [18].

C. Modular Verification

Our work borrows from the compositional verification technique of *assume-guarantee reasoning* [30], [55], [56]. Such reasoning has been widely used in software, hardware and reactive systems [29], [56], [57]. While [58] applies assume-guarantee in network congestion control, it appears to be unexplored in analyzing routing. Instead of modeling processes, we model network fragments, whose shared environment is their input and output nodes. By requiring a partition's assumptions and guarantees to be equal, our reasoning avoids the common pitfall of circularity by relying on the stability of an open

SRP's solution. Work exists on improving SMT solver performance by heuristically partitioning an SMT instance into independent instances with distinct search spaces [59]. Our approach is specifically for network fragments, where we focus on *generating* already-partitioned SMT formulas. For large formulas, we could additionally apply formula partitioning techniques.

X. Conclusion

Networks are growing faster than SMT-based verification can scale. Scalable and modular verification techniques can harness the fact that operators build networks bottom-up using local policies at each node. By providing interfaces describing a network's local invariants, operators can make their networks more robust and easier to understand. We present a formal model representing a network as a collection of fragments, and show how our modular verification procedure uses this model to catch bugs and check the correctness of users' interfaces. We prove our fragments capture the monolithic network's behavior and demonstrate our procedure with Kirigami, which dramatically speeds up network verification.

References

[1] P. Lapukhov, A. Premji, and J. Mitchell, "Use of BGP for routing in large-scale data centers," Internet draft, 2015.

[2] S. Sharwood, "Facebook rendered spineless by buggy audit code that missed catastrophic network config error," https://www.theregister.com/2021/10/06/facebook_outage_explained_in_detail/, 2021.

[3] K. McCarthy, "BGP super-blunder: How verizon today sparked a 'cascading catastrophic failure' that knackered cloudflare, amazon, etc," https://www.theregister.com/2019/06/24/verizon_bgp_misconfiguration_cloudflare/, 2019.

[4] Y. Sverdlik, "Microsoft: misconfigured network device led to azure outage," http://www.datacenterdynamics.com/content-tracks/servers-storage/microsoft-misconfigured-network-device-led-to-azure-outage/68312.fullarticle, 2012.

[5] T. Strickx and J. Hartman, "Cloudflare outage on June 21, 2022," https://blog.cloudflare.com/cloudflare-outage-on-june-21-2022/, 2022.

[6] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *SIGCOMM*, 2011.

[7] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *NSDI*, April 2012, https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final8.pdf.

[8] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *NSDI*, April 2013, https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final100.pdf.

[9] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *NSDI*, April 2013, pp. 99–112, https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final8.pdf.

[10] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "NetKAT: Semantic foundations for networks," in *POPL*, 2014.

[11] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *NSDI*, 2015, https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-lopes.pdf.

[12] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese, "Scaling network verification using symmetry and surgery," in *POPL*, January 2016.

[13] K. Jayaraman, N. Bjørner, J. Padhye, A. Agrawal, A. Bhargava, P.-A. C. Bissonnette, S. Foster, A. Helwer, M. Kasten, I. Lee, A. Namdhari, H. Niaz, A. Parkhi, H. Pinnamraju, A. Power, N. M. Raje, and P. Sharma, "Validating datacenters at scale," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 200–213.

[14] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, "A general approach to network configuration analysis," in *NSDI*, October 2015, https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-fogel.pdf.

[15] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *SIGCOMM*, August 2017.

[16] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *SIGCOMM*, August 2016.

[17] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese, "Efficient network reachability analysis using a succinct control plane representation," in *OSDI*, 2016, https://www.usenix.org/system/files/conference/osdi16/osdi16-fayaz.pdf.

[18] K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock, "Formal semantics and automated verification for the border gateway protocol," in *NetPL*, March 2016, https://www.dougwoos.com/papers/bagpipe-netpl16.pdf.

[19] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "Control plane compression," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: ACM, 2018, pp. 476–489.

[20] N. Giannarakis, R. Beckett, R. Mahajan, and D. Walker, "Efficient verification of network fault tolerance via counterexample-guided refinement," in *International Conference on Computer Aided Verification*. Springer, 2019, pp. 305–323.

[21] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "Abstract interpretation of distributed network control planes," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–27, 2019.

[22] A. Abhashkumar, A. Gember-Jacobson, and A. Akella, "Tiramisu: Fast multilayer network verification," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 201–219, https://www.usenix.org/system/files/nsdi20-paper-abhashkumar.pdf.

[23] N. Giannarakis, D. Loehr, R. Beckett, and D. Walker, "NV: An intermediate language for verification of network control planes," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 958–973.

[24] S. Prabhu, A. Kheradmand, B. Godfrey, and M. Caesar, "Predicting network futures with plankton," in *Proceedings of the First Asia-Pacific Workshop on Networking*, ser. APNet'17, August 2017, pp. 92–98.

[25] F. Ye, D. Yu, E. Zhai, H. H. Liu, B. Tian, Q. Ye, C. Wang, X. Wu, T. Guo, C. Jin, D. She, Q. Ma, B. Cheng, H. Xu, M. Zhang, Z. Wang, and R. Fonseca, "Accuracy, scalability, coverage: A practical configuration verifier on a global WAN," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 599–614, https://doi.org/10.1145/3387514.3406217. [Online]. Available: https://doi.org/10.1145/3387514.3406217

[26] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of model checking*. Springer, 2018, pp. 305–343.

[27] A. Gember-Jacobson, A. Akella, R. Mahajan, and H. H. Liu, "Automatically repairing network control planes using an abstract representation," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 359–373.

[28] N. P. Lopes and A. Rybalchenko, "Fast BGP simulation of large datacenters," in *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2019, pp. 386–408, https://web.ist.utl.pt/nuno.lopes/pubs/fastplane-vmcai19.pdf.

[29] T. A. Henzinger, S. Qadeer, and S. K. Rajamani, "You assume, we guarantee: Methodology and case studies," in *International Conference on Computer Aided Verification*. Springer, 1998, pp. 440–451.

[30] D. Giannakopoulou, K. S. Namjoshi, and C. S. Păsăreanu, "Compositional reasoning," in *Handbook of Model Checking*. Springer, 2018, pp. 345–383.

[31] A. Abhashkumar, K. Subramanian, A. Andreyev, H. Kim, N. K. Salem, J. Yang, P. Lapukhov, A. Akella, and H. Zeng, "Running BGP in data centers at scale," in *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI*. USENIX Association, 2021, pp. 65–81.

[32] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM*, 2008.

[33] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "Dcell: A scalable and fault-tolerant network structure for data centers," in *SIGCOMM*, 2008.

[34] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, “BCube: A high performance, server-centric network architecture for modular data centers,” in *SIGCOMM*, 2009.

[35] T. G. Griffin, F. B. Shepherd, and G. Wilfong, “The stable paths problem and interdomain routing,” *IEEE/ACM Trans. Networking*, vol. 10, no. 2, 2002, https://ieeexplore.ieee.org/abstract/document/993304.

[36] T. Alberdingk Thijm, R. Beckett, A. Gupta, and D. Walker, “Kirigami, the verifiable art of network cutting,” in *2022 IEEE 30th International Conference on Network Protocols (ICNP)*. IEEE, 2022, pp. 1–12.

[37] R. Chandra, P. Traina, and T. Li, “BGP communities attribute,” Internet Requests for Comments, RFC Editor, rfc 1997, 1996, https://www.rfc-editor.org/rfc/rfc1997.txt. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1997.txt

[38] Y. Rekhter, T. Li, S. Hares *et al.*, “A border gateway protocol 4 (BGP-4),” Internet Requests for Comments, RFC Editor, RFC 4271, 2006, https://www.rfc-editor.org/rfc/rfc4271.txt. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4271.txt

[39] J. a. L. Sobrinho, “An algebraic theory of dynamic network routing,” *IEEE/ACM Trans. Netw.*, vol. 13, no. 5, pp. 1160–1173, October 2005, https://ieeexplore.ieee.org/abstract/document/1528502.

[40] T. G. Griffin and J. L. Sobrinho, “Metarouting,” in *SIGCOMM*, August 2005, pp. 1–12.

[41] M. L. Daggitt, A. J. Gurney, and T. G. Griffin, “Asynchronous convergence of policy-rich distributed Bellman-Ford routing protocols,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 103–116.

[42] K. R. M. Leino, “Dafny: An automatic program verifier for functional correctness,” in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 2010, pp. 348–370.

[43] M. Danelutto and R. Di Cosmo, “A “minimal disruption” skeleton experiment: Seamless map & reduce embedding in ocaml,” *Procedia Computer Science*, vol. 9, pp. 1837–1846, 2012, proceedings of the International Conference on Computational Science, ICCS 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050912003237

[44] L. De Moura and N. Bjørner, “Z3: An efficient SMT solver,” in *TACAS*, March 2008.

[45] L. Gao, “On inferring autonomous system relationships in the internet,” *IEEE/ACM Transactions on networking*, vol. 9, no. 6, pp. 733–745, 2001, https://ieeexplore.ieee.org/abstract/document/974527.

[46] I. Pepelnjak, “Valley-free routing in data center fabrics,” https://blog.ipspace.net/2018/09/valley-free-routing-in-data-center.html, ipSpace.net, 2018.

[47] J. Y. Yen, “Finding the k shortest loopless paths in a network,” *Management Science*, vol. 17, no. 11, pp. 712–716, 1971.

[48] P. Erdös and A. Rényi, “On random graphs i,” *Publicationes Mathematicae (Debrecen)*, vol. 6, pp. 290–297, 1959, https://www.renyi.hu/~p_erdos/1959-11.pdf.

[49] E. N. Gilbert, “Random graphs,” *The Annals of Mathematical Statistics*, vol. 30, no. 4, pp. 1141–1144, 1959, https://www.jstor.org/stable/2237458.

[50] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, “Multilevel hypergraph partitioning: applications in VLSI domain,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 1, pp. 69–79, 1999, https://ieeexplore.ieee.org/abstract/document/748202.

[51] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, “The internet topology zoo,” *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011, https://ieeexplore.ieee.org/abstract/document/6027859.

[52] M. Luckie, B. Huffaker, k. claffy, A. Dhamdhere, and V. Giotsas, “As relationships, customer cones, and validation,” in *ACM Internet Measurement Conference (IMC)*, 2013-10, pp. 243–256.

[53] A. Tang, R. Beckett, K. Jayaraman, T. Millstein, and G. Varghese, “LIGHTYEAR: Using modularity to scale BGP control plane verification,” 2022, https://arxiv.org/abs/2204.09635.

[54] T. Alberdingk Thijm, R. Beckett, A. Gupta, and D. Walker, “Modular control plane verification via temporal invariants,” 2022, https://arxiv.org/abs/2204.10303.

[55] R. Alur and T. A. Henzinger, “Reactive modules,” *Formal methods in system design*, vol. 15, no. 1, pp. 7–48, 1999.

[56] C. Flanagan and S. Qadeer, “Thread-modular model checking,” in *International SPIN Workshop on Model Checking of Software*. Springer, 2003, pp. 213–224.

[57] O. Grumberg and D. E. Long, “Model checking and modular verification,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 843–871, 1994.

[58] A. Lomuscio, B. Strulo, N. Walker, and P. Wu, “Assume-guarantee reasoning with local specifications,” in *International conference on formal engineering methods*. Springer, 2010, pp. 204–219.

[59] A. E. Hyvärinen, M. Marescotti, and N. Sharygina, “Search-space partitioning for parallelizing smt solvers,” in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2015, pp. 369–386.