Property Decomposition for Network Verification

Sana Mahmood
Johns Hopkins University

Krishan Sabnani
Johns Hopkins University

Soudeh Ghorbani
Johns Hopkins University & Meta

***Abstract*—Verification has become an integral part of today's networks as it helps ensure their reliability and correctness by checking whether the desired network properties hold. The set of properties that existing network verification tools focus on can be broadly categorized into safety and liveness properties. This categorization is important because the complexity required to verify these two properties differs greatly and existing verifiers focus on verifying safety or liveness. In this work, we characterize specifications of networked systems that are not easily distinguishable into the two categories, and instead encompass both safety and liveness elements, making it hard to verify them directly via existing tools. We address this challenge by creating a property decomposition tool that uses an automata-theoric approach to represent and decompose a network property into safety and liveness automata. We also build a basic verifier that uses the decomposed automata for verifying the safety and bounded liveness - evaluating liveness on a finite number of network configurations. The verifier simulates the network events and uses these events to trigger state transitions in the decomposed safety and liveness automata. The final automata states indicate whether safety and (bounded) liveness properties hold. Compared to an existing safety and liveness verification baseline (NICE), our tool takes orders of magnitude lower time to verify the network correctness, taking 0.0035 seconds as opposed to 0.481 seconds in NICE, for a network with 7 nodes in a star topology with a waypoint property requirement.**

I. Introduction

Verification has become a crucial component of today's networks, let it be the expansion of the existing networks, changes in the control plane logic, or checking data plane network functions. Verification plays an important role in ensuring that systems and networks work as expected, deliver the required properties, and meet the service level agreements (SLAs) [1]–[8].

The correctness and reliability of a network rely on ensuring that a set of network specifications and requirements are met. Existing network verifiers usually focus on a particular class of network properties and check whether these properties hold. These network properties vary from reachability and latency requirements of static data planes to the correctness of dynamically changing networks [1]–[3], [5]–[7]. For example, some verifiers are focused on data plane reachability properties on static network snapshots, such as ensuring a direct path exists between two nodes, there are no loops in the path, and there are no packet blackholes, etc [2], [3], [6], [7]. In contrast, other verifiers focus on dynamically changing networks and check that they eventually converge to a correct state [1], [8]. Each of the existing verifiers provides a dedicated model that efficiently and effectively checks for a specific set of properties and handles issues with verifying the particular property. For example, using equivalence classes to avoid state explosion and/or a concise representation of network state [2], [8], or leveraging formal languages such as linear temporal logic (LTL), finite state machines (FSM) or computation tree logic (CTL) to better capture the dynamicity of a network [1], [8]. Each verifier is essentially customized for efficiently handling the properties it focuses on. The wide range of network properties can generally be categorized into safety and liveness properties. Safety properties proscribe a "bad thing", for example, if a reachability property of a network is to ensure that *Host A* and *Host B* should be able to communicate with each other, then *Host A* not being able to reach host *B* is a bad thing which violates the network requirement. Another example can be that of a firewall with a policy to only allow authorized external hosts to communicate with internal hosts, failure to blacklist/block an unauthorized external host *A* if it tries to contact an internal host *B*, is a bad thing. The occurrence of a "bad thing" indicates that the safety property is not met. Liveness properties on the other hand focus on "good things" that eventually happen, in the same reachability example mentioned above, "good thing" would mean that a path from *A* to *B* is eventually found, or in case of the firewall policy, "good thing" is eventually whitelisting/unblocking an external host *A* if an internal host *B* authorizes it by initiating communication with it.. Based on this characterization, existing verifiers can be easily categorized into safety vs. liveness checkers, which we elaborate on in §II. However, network specifications often cannot be discretely categorized as safety vs. liveness and many times comprise both safety and liveness components (Table I), which makes it important to decompose such network specifications into their safety and liveness parts and then apply the appropriate network verification tool to them, respectively.

In this work, we leverage the automata-theoric property decomposition, provided in [9], for network verification. We decompose network properties into their safety and liveness counterparts and separately apply verification on both of them, which allows for reduced complexity and execution time for verification. To perform decomposition, the network property is expressed as reduced Buchi automata - finite-state automata that accept an infinite input sequence and have at least one accepting state that is visited infinitely often [9]. Following the principles provided in [9], the property's automaton is decomposed into its safety and liveness automata. To realize this, we created a tool that allows a user to input a network property in the form of Buchi automata, the tool then breaks the property down into safety and liveness and returns their respective Buchi automata. We also implemented a basic verifier that uses the decomposed automata to verify safety and bounded liveness - checking liveness on a finite set of network snapshots.

To perform verification, our tool takes a network model as input and emulates various network events on it, which are then used to trigger the state transition in the safety or liveness Buchi automata. For example, if we are verifying a Buchi automaton with a state transition from q_0 to q_1 upon packet drop, then our verifier would look out for a packet drop event during the network emulation. If such an event occurs, the verifier would trigger this transition if the automaton's current state is q_0. At the end of the network event emulation, if the Buchi automaton ends up in an accepting state (which was visited "infinitely often") then it indicates the property specified by the automaton holds. The verifier is applied to safety and liveness automata separately. To verify safety, the tool takes one network snapshot and emulates the network events for it. However, a single network snapshot does not suffice to verify liveness. So, to this end, our tool verifies bounded liveness over a set of network snapshots taken over time with a high probability of network converging to the desired state within that timeframe (proof for bounded liveness is provided in §IV). Decomposition of a property and separate verification of safety and liveness (bounded) allows our tool to efficiently evaluate the network for a given property, allowing it to reduce the execution time by orders of magnitude compared to NICE (a baseline verifier for safety and liveness property) [4]. For a home network with 7 nodes and a waypoint property, NICE takes 0.481 seconds while our tool finishes verification in 0.0035 seconds.

The rest of the paper is structured as: §II provides background on existing network verification work and their categorization into safety vs liveness checkers. §III formally defines safety and liveness properties in the context of Buchi automata, and characterizes a set of network properties that encompass safety and liveness. §IV elaborates on the design of our tool that decomposes and verifies the network specifications. Lastly, §V provides an evaluation of property decomposer and verifier.

II. Background

Keeping the aforementioned categorization of network properties into safety and liveness, we provide an overview of different verification tools and categorize them based on the type of properties they verify. We also look into related work that target safety and liveness properties in some capacity.

[3] provides a tool that verifies the correctness of complex networks with inter-dependency among different network stack layers. This multi-layer verifier, called Tiramisu, provides a fast and scalable design to check a wide range of network policies, such as ensuring traffic always flows between a source and a destination (reachability properties), ensuring there are no forwarding loops in the network, making sure there are no blackholes in the network and load balancing. Tiramisu ensures speed and scalability by performing isolated verification for each network layer using symbolic execution and then composing their results together to verify the network as a whole. Tiramisu can be categorized as a safety property checker, because it checks the network to ensure that the desired network policies hold, violation of these properties would cause the verification to fail. For example, finding a case where two hosts are not reachable from each other, or finding a blackhole or a loop in the forwarding rules. By safety definition, a "bad thing" (policy violation) causes the verifier to fail, and any of the aforementioned examples would result policy violation and consequently a failed verification. [2] provides a verification tool, called Veriflow, for checking control plane correctness for SDN controllers which dynamically updates the network as the conditions change. Rather than checking the entire static network configuration, it sits in the shim layer between the SDN controller (control plane) and network elements, it intercepts and verifies all the updates before they are sent out by the controller and applied to the relevant switches. For each update, Veriflow checks to ensure that the network-wide invariants, such as reachability properties, isolation between VLANs, loop freeness, consistency among network entities, etc, hold. Veriflow uses equivalence classes to reduce the search space and improve the efficiency of the tool by only observing the impact of updates on affected nodes. Veriflow can also be categorized as a safety property checker, because it rejects an update if it does not fulfill the desired network-wide invariant, for example, if an update causes two routers that are expected to follow the same forwarding behavior, to contain inconsistent forwarding rules ("bad thing"), it rejects the update. [1] provides a verification tool to check the correctness of stateful network functions. Unlike previous tools that check different flavors of reachability properties, this tool focuses on verifying a much richer set of liveness properties of stateful network functions ensuring that correct behavior is eventually established. For example, a connection established or a connection teardown eventually occurs when the right conditions are met, or a request eventually receives a response. The verifier provides a "one big switch" abstraction to the user to input a network with stateful network functions, where match action rules result in an updated network state and create a dynamically changing network. It uses TCL to specify the properties of the stateful function for their verification. This verifier belongs to the liveness checker category, because unlike safety verifiers which reject an input over a "bad thing", this verifier waits for the desired property "good thing" to eventually happen. For example, for a connection to be eventually established, for a packet to eventually find a path towards the destination, or for a request to be eventually responded to. Waiting for these properties ("good things") to be eventually fulfilled is a characteristic of liveness properties. [5] presents a verifier, called Anvil, that aims to verify the correctness of cluster management controllers, which are responsible for managing all system resources and application lifecycles in large-scale distributed systems. Such large-scale systems are prone to failures (including node and link failures), link delays, and other errors, the correctness of such systems

is verified by ensuring that despite the failures the desired services eventually succeed and do not remain pending forever. Anvil is also categorized as a liveness checker verifier because the controllers adapt to the network changes and failures and make sure the network reaches a desired state (referred to as eventually reconciled state), where the resources are granted, and services are provided eventually (”good thing”).

Verifying safety and liveness via the same tool has been done in a limited capacity by [4], which provides a tool that leverages model checking complemented with symbolic execution of controllers' event handlers, to find bugs in SDN controllers. The bugs found are essentially violations of desired network specifications. These specifications include, there are no packet blackholes, or no forwarding loops in the network (safety), and also extend to a limited set of stateful properties such as after the two-way exchange of a packet between two end hosts, no packet between their communication should go through the controller, this comes under liveness because checking for this property requires information about transitions and changes that occur in the network. NICE [4], provides verification of both safety and liveness properties but it is limited in a few ways, it is only capable of checking for properties relevant to an SDN controller, and it runs into scalability issues when testing for a large-scale network [2], and lastly as it mentions in the paper the liveness properties it verifies are limited.

III. Property Decomposition

This section explains the automata-theoretic decomposition of a property into safety and liveness properties. We then characterize a set of network properties that are a combination of both safety and liveness. We then zoom in on one of those network properties, provide its Buchi automaton, and demonstrate its decomposition into safety and liveness.

A. Automata-Theoretic Decomposition

In traditional distributed systems properties are categorized as safety properties and liveness properties. A safety property ensures that ”bad things” never happen. For example, in the case of a computer program a ”bad thing” would be to produce an incorrect output, or in the case of a multi-threaded program, two threads concurrently accessing the critical section would be a ”bad thing”. The occurrence of a ”bad thing” indicates that a program is not performing as intended. Liveness properties require that ”good things” eventually happen. For example, a computer program should eventually terminate (and not get stuck), or in a multi-threaded program, a thread should eventually gain access to the critical section. A ”good thing” eventually occurring indicates that a program converges to a correct state over time.

Usually, a program's requirements are not easily distinguishable into safety and liveness, which creates the need to separate its safety and liveness elements. In distributed systems, the decomposition of properties into safety and liveness has been solved using an automata-theoretic approach [9]. A property is expressed as a reduced Büchi automaton - a Buchi automaton is a finite-state automaton that accepts an infinite input sequence and contains at least one accepting state that is visited infinitely often. A reduced Buchi automaton is a Buchi automaton with redundancy removed, for example, it does not contain states that do not lead to an accepting state or have no outgoing transition. In [9] a reduced Buchi automaton of a property is decomposed into two automata, one that specifies the safety property of the main property while the other specifies liveness. The decomposition works on the observation that each property is said to proscribe ”bad things” (safety) and prescribe ”good things” (liveness), so identifying what constitutes a ”bad thing” and what constitutes a ”good thing” in a Buchi automaton and isolating the two is the key to providing an automata-theoretic characterization of safety and liveness.

which in reduced Büchi automata translates to having no undefined transitions between states. Each property also prescribes ”good things” (liveness), meaning a Büchi automata eventually enters an accepting state infinitely often.

a) Safety Decomposition:: By definition, a safety property proscribes ”bad things”. So, if a partial execution of a program triggers a ”bad thing”, it is caused by a state following a finite sequence and is irremediable. This means that appending any infinite sequence of states to this finite sequence will not remedy the ”bad thing” [9]. In formal logic, its contrapositive is used to define the safety property P is defined as below, where α is a sequence of program states, S is a set of program states, S^* is a set of finite sequence of states, S^ω is a set of infinite sequence of states.

$\forall \sigma : \sigma \in S^\omega : \sigma \vDash P \Leftrightarrow (\forall i : 0 \leq i : (\exists \beta : \beta \in S^\omega : \sigma[\cdots i]\beta \vDash P))$ [9]

In automata theory, attempting an undefined transition is a ”bad thing”, because an undefined transition fails the automata, and any sequence of input following the undefined transition can not remedy it. Therefore, to verify that an automaton specifies a safety property, we need to check whether or not a sequence of states attempts an undefined transition. This is achieved by taking the closure of the automata [9] - closure of an automaton m, (Cl(m)), is an automaton where each state of m is turned into an accepting state. Cl(m) only rejects an input if it attempts an undefined transition (”bad thing” occurs). So, if m and Cl(m) accept the same input [9], then m specifies a safety property.

b) Liveness Decomposition:: By definition, the property of liveness prescribes that ”good things” eventually happen. This indicates that any partial execution of a program state is not irremediable, because if it were it would be a ”bad thing” and liveness does not proscribe a ”bad thing” [9]. In formal logic, liveness is defined below.

$\forall \alpha : \alpha \in S^* : (\exists \beta : \beta \in S^\omega : \alpha\beta \vDash P)$ [9]

In automata-theory, if an automaton specifies liveness then it should not have any undefined transitions because rejecting an input for causing an undefined transition would not lead to the eventual ”good thing”. So, to check whether an automata m specifies a liveness property, we need to augment m so that it does not reject an input for causing a ”bad thing” (undefined transition). [9] suggests that if an automaton m specifies a composed safety and liveness property then augmenting m by including a trap state - an accepting state with an infinite self-transition, and directing undefined transitions of all other states towards the trap state creates automata that specifies liveness.

B. Network Properties

Now that we have formally defined safety and liveness, we define two umbrella properties composed of both safety and liveness components. Then, we characterize a set of network properties that fall under either of the umbrella properties.

- **Total correctness** for a program means that it generates the correct/expected output (safety), also called partial correctness, and that it eventually terminates (liveness).
- **Mutual exclusion** is a property in distributed/parallel processing stating that no two processes can access a critical piece of code simultaneously (”bad thing” / safety), and if a process requests to access a critical region it should be able to do so in some bounded wait time (”good thing” / liveness).

Table I lists the network properties that comprise both safety and liveness, and also identifies the umbrella property that they fall under. We now discuss one property in detail and present how it can be represented as a Buchi automata, and what its decomposition into safety and liveness look like.

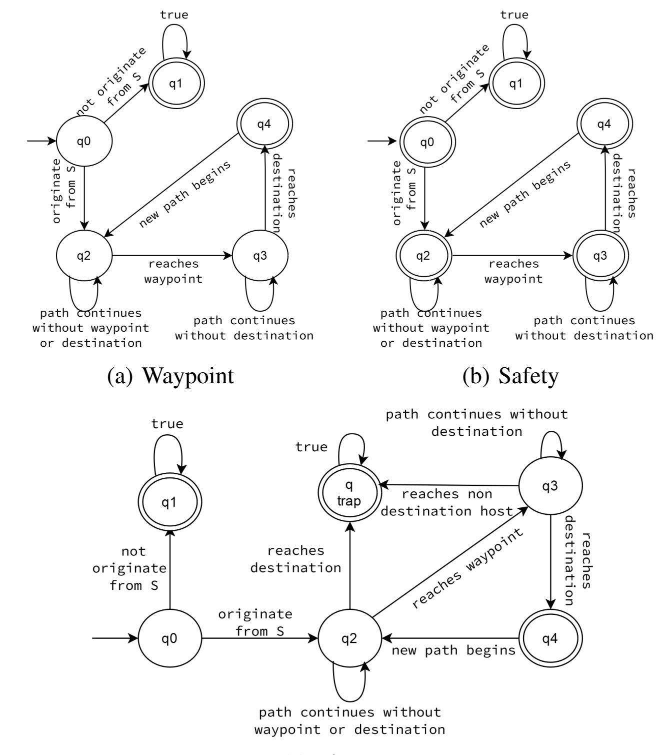
1) Waypoint: In computer networks, a waypoint is a node or entity through which the traffic must traverse before reaching the destination. Waypoint plays an important role in improving security and monitoring of systems. For example, in a local office network it can be used for intrusion detection, where all incoming traffic must pass through the firewall (the waypoint) before reaching internal machines, or in the case of a campus network it can be used to enforce bandwidth limitations, where all traffic originating from dormitories must pass through the firewall (waypoint) which caps the bandwidth. Waypoint falls under total correctness and can be mapped as:

Partial correctness (Safety): A path from S to D always passes through a waypoint W.

Termination (Liveness): Routing eventually establishes a path between S and D, that passes through W.

Fig 1a, provides a Buchi automata that specifies this property. Starting with state $q0$ where traffic originating from S is a Pre condition, which leads to state $q2$. The automata move to the next state $q3$ if traffic passes through waypoint W, from $q3$ it moves to $q4$ upon reaching destination D, which is an accepting state and automata visits this state infinitely often. The automata ensures safety because the accepting state is reached only after passing through q_3 which ensures that traffic traverses the waypoint. The liveness is ensured because any traffic originating from S finds a path to D which is an accepting state and will be visited infinitely by either staying in that state or due to multiple paths fulfilling the waypoint requirement.

Upon decomposing the reduced Buchi automata into safety and liveness, based on the approach presented in [9] we get the Fig 1b and Fig 1c, respectively. Fig 1b specifies safety



(a) Waypoint (b) Safety

(c) Liveness

Fig. 1: Reduced Buchi automata representation of the waypoint property and its decomposition into safety and liveness

because all of its states are turned into accepting states and it will now only reject inputs that result in undefined transitions, for example, if we reach destination D without reaching the waypoint W in state q_2 the input is rejected and therefore safety is not met. Fig 1c specifies liveness because, for all undefined transitions, it defines a transition to the accepting trap state. This ensures the automata specifies liveness because it will not reject undefined transitions and rather wait for the network to eventually converge to the desired state, which is to find a path from S to D with W.

IV. Tool Design

In this section, we provide the design details of our tool, which takes a property specification in the form of a reduced Buchi automaton, and decomposes the property into its safety and liveness components. The tool also takes as input a set of network snapshots, which detail the network topology and the forwarding behavior of all the nodes. The tool separately checks Buchi automata for safety and liveness against the provided network model to verify whether those properties hold. To avoid the complexity of complete liveness verification, we resort to verifying bounded liveness, by looking at a set of network snapshots and deciding whether liveness requirements are eventually met within the provided network states.

Fig 2 provides a high-level overview of the tool. A set of network graphs (with forwarding behavior) and a property to test are provided as input. The property is provided in the form

Property	Type	Safety	Liveness
Waypoint [6], [7]	Total Correctness	All traffic from A to B must pass through W	If traffic reaches from A to B, it eventually passes through W before it reaches B
Resilience [4], [10]	Total Correctness	All paths between A and B are disjoint	If multiple paths exist between A and B, they eventually follow a non-overlapping route
Load Balance [4], [8]	Total Correctness	Load is evenly distributed among multiple paths between A and B	If traffic reaches from A to B, it is eventually distributed evenly across multiple paths leading to B.
Isolated Updates [11]	Mutual Exclusion	Only a single update is performed on a set of dependent entities at a given time	If an update is requested on a set of entities, it is eventually executed.
Resource Allocation [12]	Mutual Exclusion	No two tenants are allocated the same or overlapping resources at the same time.	If a tenant requests a resource, its access is eventually granted to them.
Traffic Isolation	Mutual Exclusion	Two different traffic classes should not share the same priority queue or compete with each other to get forwarded.	If traffic from a particular class arrives at the switch, it is eventually placed in its relevant queue.

TABLE I: Network Specifications consisting of safety and liveness properties

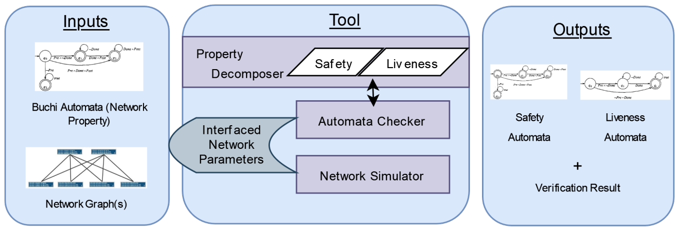


Fig. 2: Tool Overview

Name	Description
CURRENT_NODE	ID of the node being visited
SOURCE	ID of the node from where traversal began
HOP_COUNT	Total hops taken starting from SOURCE node
PATH_ID	ID of the path being taken
PROCESS_ID	ID of the process currently running (such as update ID)
DROPPED	Indicates a packet drop
DROP_NODE	ID of the node where drop occurred

TABLE II: Interfaced Network Parameters

of a reduced Buchi automata that uses Interfaced Network Parameters (INP) to define state transitions, and the network is input in the form of an XML file that contains nodes, their connections, and routing tables. Once the input is received, the tool decomposes the automata into its safety and liveness parts and then emulates network events to verify the respective safety and liveness properties. The tool then outputs the safety and liveness automata and whether these properties hold.

The main components of the tools are:

- Buchi automata Representation
- Property Decomposition
- Basic Network Verification

A. Buchi automata Representation

In our tool, network requirements are expressed in the form of reduced Buchi automata, which essentially means that the user needs to provide the states of an automaton and the transitions between them, similar to Fig 1a for the waypoint property. While it is straightforward to express state transitions using phrases or pseudo-code as done in Fig 1a, such expressions are not interpretable by the tool when it comes to associating transitions with various network events. For our tool to correctly interpret the transitions and associate them to the provided network for verification, the transitions need to be in the ”tool legible language”. We achieve this by taking states and transitions (the reduced Buchi automaton) in code via a Python script. We define a library that allows users to create automata states and write their transitions using various network event identifiers. Each transition is coded as a Boolean function whose outcome dictates whether the transition should occur. We create a library of interfaced network parameters (INP) to enable users to associate transitions to different network events.

*a) **Interfaced Network Parameters (INP)**:* The tool interfaces a set of network parameters via a network library, to allow the users to express state transition functions in terms of network events. Table II provides the list of these interfaced network parameters. These parameters enable the input Buchi automata to peek into the current state of the network and perform relevant transitions because the values of these parameters update as the network changes and the relevant events occur, for example, $PATH\_ID$ is updated whenever a new path is traversed. To give an example of the usage of these parameters, if the user needs to specify the transition between state $q2$ to $q3$ from the waypoint example (in section III-B1) that traffic passes through waypoint ”W”, it can be represented as a function shown below by using one of the interfaced parameters.

```
def pass_waypoint():
    return CURRENT_NODE == "W":
```

We argue that this set of parameters can be used to define a wide range of network properties, for example, the $CURRENT\_NODE$ parameter can be used to check path progression, detect new paths, check reachability, etc. Moreover, this list can be extended to interface more network events.

The INPs also serve as a bridge between the input network model and the Buchi automata. The values of these parameters change as the network progresses (relevant events occur), for example, $CURRENT\_NODE$ updates whenever a new node is visited during the network emulation. A change in

INP values can trigger a potential state transition in the Buchi automata. This is achieved via callback hooks, where the transition functions are hooked to the relevant INPs, and get called if the value of the hooked INP changes. The outcome of the triggered transition function then dictates the state change.

B. Decomposition

Once the input Buchi automaon is received and hooked to relevant INPs, the property is decomposed into its safety and liveness components.

1) Safety Decomposition: As described in section III, the safety property proscribes ”bad things” which translates to undefined transitions in Buchi automata. As long as there are no undefined transitions in the network, the safety property is satisfied. Therefore, transforming the input Buchi automata by turning all of its states into accepting states fulfills the safety decomposition. The new Buchi automaton specifies the safety property because a network that does not trigger any undefined transition would cause this Buchi automaton to visit the accepting state(s) infinitely often.

2) Liveness Decomposition: As per [9], liveness does not proscribe a ”bad thing”, rather it states that ”good things” eventually happen. So, the liveness automata should not reject undefined transitions aka ”bad things”. As explained in section III, liveness decomposition varies based on whether the original automata of the property is deterministic or non-deterministic. To this end, we require the user to inform whether the input Buchi automata is deterministic. However, our future goal is to equip the tool to test for the determinism of an automaton. The key component of liveness decomposition for both deterministic and non-deterministic automata is the same, which is to create a trap state (an accepting state that infinitely transitions to itself) and then create transitions from all other states towards the trap state such that all undefined transitions of a state get directed towards the trap state. To achieve this, the tool traverses all states and creates a transition that is true only when all other transitions of that state are false. This suffices to ensure that all the undefined transitions, and only the undefined transitions, are sent towards the trap state. Because if a transition callback of a state is triggered, it means the value of an INP that the state transitions rely on has changed. If the new value of the network parameter does not satisfy any of the states' transition functions, we have come across an undefined transition and it should be forwarded towards the trap state.

C. Basic Verification Model

Lastly, once the automata is decomposed into its safety and liveness component, our tool also provides a basic verifier to check whether the safety and liveness properties are met.

1) Network Input: The tool takes as input a set of network snapshots, each network snapshot consists of the network graph, nodes, and the connections between them, along with each node's routing tables. The routing table contains match action rules, with match parameters such as source, destination, protocol, priority, metric, in port, etc, actions being forward or drop.

2) verification: The verification is performed by emulating network events for the provided network state and using these events as input to the relevant Buchi automata (safety automata or liveness automata). The network emulation is done based on user requirements, and our tool provides various network traversal prompts to select from. For example, one such prompt is to explore all paths between two given nodes, the tool starts from the source and traverses the network to reach the destination using routing table rules until all paths are covered. As the network traversal takes place, the INP values are updated, triggering the state transition callback functions and potentially changing the states of the relevant Buchi automata. Once the traversal is finished, the state of the Buchi automata is checked to verify whether the property holds or not.

Safety Verification: The tool takes a snapshot of the network (input as mentioned above) and traverses the network based on the user requirement. The changes in the network are reflected in the Buchi automata. If automata face an undefined transition during the network traversal, the verification is stopped because this indicates that the safety property does not hold. If the traversal finishes without triggering an undefined transition, then the safety property holds because the automata would be in an accepting state (all states of a safety automata are accepting).

Bounded Liveness: To avoid the complexity that comes with liveness verification, the tool focuses on checking liveness within bounds by taking as input a set of network snapshots taken over some time, and checking whether the network converges to the desired state within this time bound. The liveness automata is reset at the beginning of each network traversal. At the end of traversing all snapshots, the verifier checks whether the automata reach an accepting state that was not a trap state at least once. If it did then it verifies the liveness property because a good thing eventually happened which is the definition of liveness. However, if it did not arrive at an accepting state during all of the snapshots traversal then the network fails the liveness check. On the other hand, if the network only got accepted via ending up in a trap state, then it shows that the ”good thing” did not eventually occur within these bounds and can potentially occur at a later point. So, for the bounded interval, liveness could not be verified. This makes it important to set the interval of taking network snapshots reasonably large so it guarantees liveness verification with a high probability

3) Proof for Bounded Liveness: We build a premise that a dynamically changing network is bound to converge to a desired state within a time interval T with high probability. We first argue that taking n network snapshots over time $[0-T]$ would suffice in verifying whether the network satisfies a given property with a high probability if T is chosen correctly. Next, provide guidelines for selecting T, and prove that if T is chosen with those guidelines it would capture the network convergence with high probability.

Correctly verifying liveness:

Let's assume that the network is guaranteed to converge to a desired state where a liveness property ϕ holds, within time T with a high probability. If we take a snapshot of the network every Δt, resulting in n snapshots of the network $S_1, S_2, \cdots, S_n$. Then these snapshots capture the convergence of the network to a desired state where ϕ holds.

Let $f(t)$ be a function on network snapshot S taken at time t, such that $f(t) = True$ if the network snapshot satisfies ϕ, and $f(t) = False$ if the network snapshot violates ϕ. Since the network is guaranteed to converge within T, if for a given S_i, $f(i) = False$, then the network transiently violates ϕ. Moreover, if $f(m) = True$, where $0 \leq m \leq T$ then the network has converged to the desired state with a high probability and will continue to remain in this state after T as well. The chances of missing a violation during this interval are low and depend on two factors

1) $P_{snapshot\_miss}$, which is the probability of missing a snapshot where a violation of ϕ occurs after a snapshot satisfied ϕ.
2) $P_{non\_convergence}$, which is the probability of the network not converging to a desired state within T

Both $P_{snapshot\_miss}$ and $P_{non\_convergence}$ can reduced with a sufficiently large n and T, respectively. However, a large value of either of these parameters would increase the computational overhead and complexity. Moreover, a large T would increase n as well, so as long as we set T appropriately enough that it captures network converges with high probability while not being large.

Selecting T:

We should select T such that it can capture network convergence with a high probability while ensuring it does not grow too large in value. The value of T relies on various network factors mentioned below.

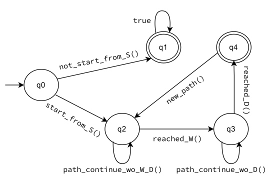
1) Propagation Delay (P): Maximum delay for relaying/triggering information to certain nodes that impact ϕ, we should set $T \geq P$ to ensure the information that can cause network state to converge gets relayed within T.
2) Processing/Computation Time (C): Estimated time it would take for a node to install/update rules, we should set $T \geq C$ to ensure the relevant rules are correctly added within T.
3) Uncertainty (V): While the above two factors seem enough to ensure the network converges within T, uncertain events can occur, a link/node failure or control packet drop for example, which can delay the converges of the network. Setting $T \geq V$ would ensure we account for such uncertainties.

We should set T such that $T \geq P + C + V$. As the value of T approaches this value, the probability of convergence within T, $P_{convergence}$ rises and $P_{non\_convergence}$ decreases.

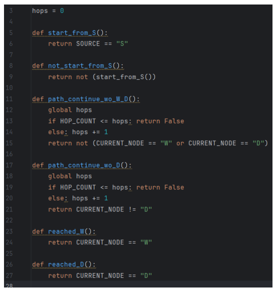
V. Evaluation

We evaluate our tool to test its correctness as well as compare it with existing safety+livenees verifier. We aim to answer following questions from our experiments.

1) Are the interfaced network parameters of our too sufficient to express a wide range of network properties?



(a) Waypoint



```
hops = 0

def start_from_S():
    return SOURCE == "S"

def not_start_from_S():
    return not (start_from_S())

def path_continue_wo_W_D():
    global hops
    if HOP_COUNT <= hops: return False
    else: hops += 1
    return not (CURRENT_NODE == "W" or CURRENT_NODE == "D")

def path_continue_wo_D():
    global hops
    if HOP_COUNT <= hops: return False
    else: hops += 1
    return CURRENT_NODE != "D"

def reached_W():
    return CURRENT_NODE == "W"

def reached_D():
    return CURRENT_NODE == "D"
```

(b) Transition Function Script

Fig. 3: Waypoint Büchi Automata represented in our tool's language

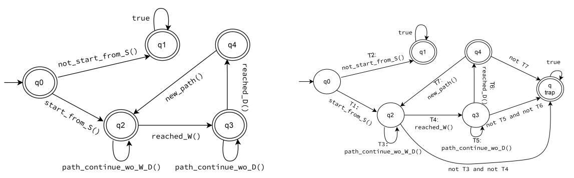
2) Does the decomposition retain the safety and liveness elements of the original property, aka the decomposed safety and liveness automata specify the property correctly?
3) Can our verification model successfully assess whether the safety and bounded liveness properties are met and how long does the verification take?

To this end, we specify a total correctness property using our tool and check its safety and bounded liveness.

A. Waypoint

Fig 3 shows how the waypoint example of Fig 1a can be specified as Buchi automata using the language of our tool. Recall that the property in this example is that any path between S and D should pass through W. The transitions of the input automata are provided in the form of boolean functions, which are defined in the input script provided by the user as shown in Fig 3b. Most transitions for waypoints can be defined by using the CURRENT_NODE parameter. The input script provided by the user also shows how the user can maintain state for certain transitions, by using a global variable *hops*, the user can tell whether the path progresses.

The Buchi automata of waypoint using interfaced network parameters correctly specifies the waypoint property. Where it only accepts inputs that reach D by passing through W. Fig 4 shows the output safety and liveness automata of this property, Fig 4a specifies safety property, and shows that any input that reaches D without passing through W is rejected. Furthermore, it accepts any input that might never reach D, for example, if a drop occurs on the way, the input is still accepted. Because the safety specification of "bad thing" aka "reaching D without passing through W" is not violated. Next, Fig 4b represents the liveness automata, which accepts all inputs that reach D, including the ones that do not pass through W. Because liveness does not proscribe the "bad thing" (reaching D without passing W), and waits for the "good thing" (reaching D after passing through W) to eventually happen. So, if S can reach D, liveness waits for it to eventually pass through W before arriving at D. This shows that our tool can correctly represent and decomposed a waypoint property.



(a) Safe(Waypoint) (b) Live(Waypoint)

Fig. 4: Decomposed safety and liveness automata of the Waypoint property by our tool

1) Small Network Test: : We now look at some real world examples of waypoint and test whether our tool correctly specifies and verifies the safety and liveness properties in small-scale networks such as home networks.

Home Network: We used a small home network consisting of a router, an access point, and three home devices connected to the router/access point in a star topology. The routing is done based on static shortest-path routing. The network aims to ensure that traffic originating from all of the children's devices passes through the access point before getting sent over the internet, this is useful for parental control for monitoring or blocking traffic. The access point serves as the waypoint in this case and the safety and liveness properties of this specification:

- Safety: All traffic originating from children's devices should pass through the access point before entering the internet.
- Liveness: Traffic originating from chilren's devices will eventually pass through the waypoint before getting sent over the internet.

The safety automata will reject any input where traffic from a child's device reaches the internet gateway before passing through the access point. The liveness automata, on the other hand, accepts any input where a child's device traffic passes through the access point before going through the internet gateway or reaches the internet gateway without passing through the access point. This occurs because if the traffic directly reaches the internet gateway, it will eventually pass through the access point before arriving at the gateway.

We now test how long it takes for our tool to verify whether safety and bounded liveness hold for given network configurations and compare it against NICE [4]. We use the same network setup in NICE with the controller adding static rules in the router and the access point. We experimented with a case where the network is safe and also converges to a correct state, therefore expecting no violations of the property. In our tool, we used 4 different network snapshots to test liveness. Our tool successfully verified the network within 0.0035 seconds, while NICE took orders of magnitude higher time, 0.481 seconds, to verify the correctness of the same network. The reduced complexity of bounded liveness and separation of safety and liveness property allowed our tool to reduce the verification time while retaining correctness with a high probability.

2) Large-Scale Network: Next we move to large-scale networks and to test the efficiency of our tool at scale.

Datacenter Topology We tested with k-ary fat tree topology, with the waypoint specification that ensures background traffic is always routed through dedicated core switches. Such a specification is useful in separating different traffic types, ensuring that latency-sensitive traffic does not have to compete with long running traffic. The specification of safety and liveness is:

- Safety: All inter-pod background traffic (identified by priority group) should be routed through the dedicated core switches before arriving at the destination pod.
- Liveness: The inter-pod background traffic is eventually routed only via dedicate core switches.

We tested with different scales of fat tree topology with k=4,8 etc. We tested the waypoint specification by dividing the network into halves and servers from one half communicate with those in the other half. The reason for choosing this communication pattern is to keep the network configuration consistent with NICE for comparison. Furthermore, in our tool, we used 6 synthesized network snapshots for bounded liveness verification, because inter-pod traffic passes through 5 nodes, and keeping the snapshots to 6 would allow us to capture the changing network dynamics and potential convergence. Similar to the prior test, the network was build to satisfy both safety and liveness properties. For the given communication pattern, our tool verifies the safety and bounded liveness within 0.036 seconds and 3.3 seconds for 4-ary and 8-ary fat tree, respectively. NICE on the other hand took over an hour for the same network and was terminated without completion.

VI. Conclusion

In this work, we showed that many network requirements are composed of safety and liveness properties, while the existing verifiers largely focus on verifying either safety or liveness because the level of complexity associated with their verification varies greatly. We leverage an automata-theoric decomposition approach to break down a network requirement into safety property and liveness property. We, then separately verify safety and bounded liveness to evaluate whether a given network violates either of these properties. We show that the decomposition of the property allows for separate verification, allowing for reduced complexity and improved verification time.

References

[1] F. Yousefi, A. Abhashkumar, K. Subramanian, K. Hans, S. Ghorbani, and A. Akella, "Liveness verification of stateful network functions," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 257–272. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/yousefi

[2] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying Network-Wide invariants in real time," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 15–27. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/khurshid

[3] 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)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 201–219. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/abhashkumar

[4] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE way to test OpenFlow applications," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, Apr. 2012, pp. 127–140. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/canini

[5] X. Sun, W. Ma, J. T. Gu, Z. Ma, T. Chajed, J. Howell, A. Lattuada, O. Padon, L. Suresh, A. Szekeres, and T. Xu, "Anvil: Verifying liveness of cluster management controllers," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 649–666. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/sun-xudong

[6] A. Ludwig, M. Rost, D. Foucard, and S. Schmid, "Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XIII. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–7. [Online]. Available: https://doi.org/10.1145/2670518.2673873

[7] K. Subramanian, L. D'Antoni, and A. Akella, "Genesis: synthesizing forwarding tables in multi-tenant networks," ser. POPL '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 572–585. [Online]. Available: https://doi.org/10.1145/3009837.3009845

[8] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable dynamic network control," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 59–72. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/kim

[9] B. Alpern and F. B. Schneider, "Recognizing safety and livenes," *Distributed Computing*, vol. 2, pp. 117–126, 1987.

[10] H. AlMansouri and Z. Hussain, "Real-time fault-tolerance node-to-node disjoint paths algorithm for symmetric networks," 2021. [Online]. Available: https://arxiv.org/abs/2103.09801

[11] J. Lembke, S. Ravi, P.-L. Roman, and P. Eugster, "Secure and reliable network updates," *ACM Trans. Priv. Secur.*, vol. 26, no. 1, Nov. 2022. [Online]. Available: https://doi.org/10.1145/3556542

[12] E. Zahavi, A. Shpiner, O. Rottenstreich, A. Kolodny, and I. Keslassy, "Links as a service (laas): Feeling alone in the shared cloud," 2016. [Online]. Available: https://arxiv.org/abs/1509.07395