Witnessing Network Transformations

Chaoqiang Deng[1] and Kedar S. Namjoshi[2](✉)

[1] New York University, New York, USA
deng@cs.nyu.edu
[2] Bell Laboratories, Nokia, Murray Hill, USA
kedar.namjoshi@nokia-bell-labs.com

Abstract. Software-defined networking (SDN) is transforming the way networks are managed, as fixed distributed protocols give way to flexible route calculation software. The shift brings to the forefront the issue of software errors, which may produce wrong routes, and cause significant network disruption. We propose a run-time certification mechanism that rejects any wrongly calculated route before it is installed in the network. Certification is done through a strategy called *witnessing*, where a witness (i.e., a justification) is generated by the software for each routing decision. The witness provided for a route change is validated against the original user request, using a formal network model, before the change is installed on the real network. Witnessing shifts trust away from the complex system software to a relatively simple witness checker. We define a formal language to specify connection-based user requests ("intents"), witnesses for each type of intent, and the checking algorithm. We also formulate a notion of refinement between networks, and show that it preserves the realizability of intents across abstraction levels.

1 Introduction

Computer networks have long been managed with standardized, distributed protocols. The advent of software-defined networking (SDN) (cf. [9]) is radically transforming this view to one where flexible, programmable routing engines operate on a formal network model. This makes it possible to apply sophisticated route selection algorithms and to experiment with variations.

Such flexibility, however, comes with potential dangers. Increasing algorithmic sophistication increases the likelihood of errors in their implementation. A miscalculated route may fail to meet the original request or, worse still, disrupt traffic on existing routes by over-committing available bandwidth. In this work, we design a run-time certification mechanism to detect wrongly calculated routes and prevent them from being installed on the real network. The central concept is to require all route selection programs to produce a formal justification – which we call a "witness" – for each routing decision. A valid witness guarantees that the associated routing changes do not adversely affect active routes.

We show how to instantiate this design for an emerging class of *network operating systems* (NOS's) – examples include ONOS [1] and OpenDaylight [2]

S. Lahiri and G. Reger (Eds.): RV 2017, LNCS 10548, pp. 155–171, 2017.
DOI: 10.1007/978-3-319-67531-2_10

– which use SDN principles to unify the management and operation of a collection of networks built with different technologies (e.g., IP, optical, and wireless), each with its own protocols and management software. To facilitate this goal, all networks, regardless of the underlying technology, are represented uniformly as graphs with capacitated connections between nodes. In response to a connectivity request, a global route is calculated by the NOS on the graph model; individual route segments are later configured locally in technology-specific ways. Existing NOS's do not guard against errors in global route calculation, nor do they have a principled mechanism for defining connectivity patterns. Our work makes a number of contributions beyond efficient certification.

– We define a formal language of connectivity patterns on graphs, called *intents*. This includes common patterns such as paths, chains, and trees, with constraints on bandwidth and delay, and allowance for backup paths
– We propose an augmented architecture of a *certifying* NOS. Route selection programs are required to generate a witness (a set of paths) as justification for each routing decision
– We show that witness checking has worst-case linear complexity in the size of the witnesses and in the number of intents. An incremental checking algorithm further reduces the complexity in the common case. Experimental results on a family of synthetic networks support the theoretical analysis
– We define a formal notion of refinement between networks which preserves the realizability of intents: i.e., any intent satisfied on the abstract network can be realized in the concrete network. This allows route selection algorithms to operate on smaller abstract networks, reducing complexity.

The architecture of a certifying network operating system ensures that new route selection algorithms can be implemented and tested quickly, with the "safety-net" guarantee that certification makes it impossible to install erroneous routes that may disrupt network operation. The refinement notions ensure that solutions computed on an abstract network remain realizable at more concrete levels, which makes it possible to chain route selection algorithms operating at different levels of granularity. Taken together, these mechanisms significantly increase the robustness and the safety of a network operating system.

1.1 Overview

A network operating system computes and installs routes on the fly in response to a stream of incoming user requests. The ideal is a formally verified OS, whose output is guaranteed to be correct. Constructing a formally verified network operating system is, however, an enormously difficult undertaking. We propose an alternative solution based on the run-time certification of the results computed by the operating system.

A schematic view of a network operating system (NOS, for short) is shown in Fig. 1(a). The operating system reacts to explicit external requests for routes (referred to as "intents"), and implicitly to changes in the underlying network,

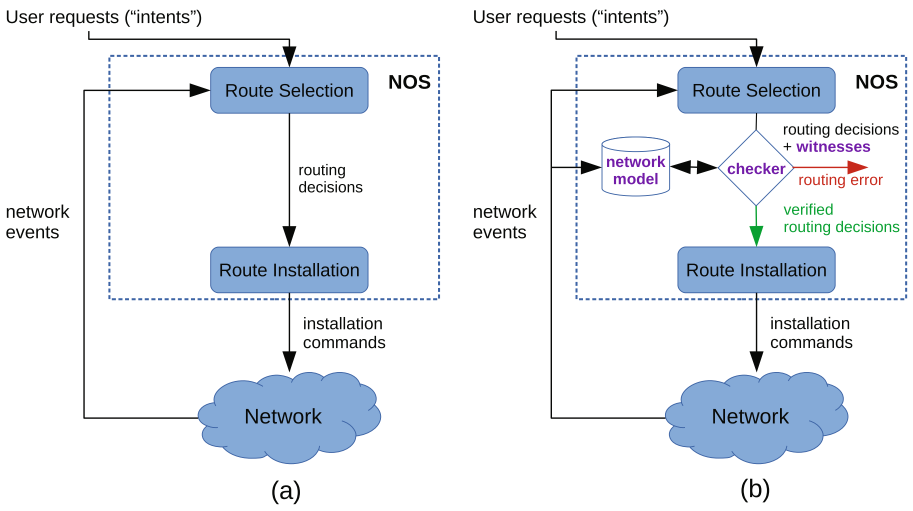


Fig. 1. Network operating system structure: (a) current, (b) with formal certification. (Color figure online)

such as failure or degradation of nodes or links (referred to as "events"). In response, the system uses route selection algorithms to make decisions to set up new routes and, possibly, to move old routes, aiming to preserve all active intents. Those decisions are then configured and installed on the real network. A network operating system is, in essence, a network transformer; it maps a network with allocated routes and a request to a network with modified routes. The standard architecture in Fig. 1(a) leaves no room for error: one must trust the correctness of the route selection algorithms and their implementations. A mistake in either may result in a routing decision that disrupts network traffic.

Our proposal is shown in Fig. 1(b). Two key aspects are the use of formal network models and the generation and checking of witnesses (i.e., justifications) for each intent. The new architecture requires the route selection algorithm to provide a witness, a collection of paths in the network, for every decision. The checker has to perform two tasks: (1) ensure that the route decision (which could be presented in a different format, e.g., as a sequence of commands) is consistent with the supplied witness paths; and (2) check that the supplied paths prove that the network meets the new intent *and* continues to meet all earlier intents. Only if these checks succeed are the changes instantiated on the real network.

The certification step shifts trust away from the complex operating software to a relatively simple checker. There is no need to verify the implementation of the routing algorithm: if a mistake results in a wrong route or an incorrect witness, the error is detected by the checker. The route installation process (lower blue rectangle in the NOS figure) relies on standardized mechanisms such as NETCONF [3] and is a trusted component.

In this paper, we address task (2), defining a formal language of intents, their witnesses, and an algorithm for witness checking. We do not address task (1), as it is specific to the format used by the NOS to represent its routing decision. A general strategy for task (1) is to simulate the route-change commands on the network model and verify that network links are reconfigured exactly as stated in the witness paths.

Real networks have an immense amount of detail, not all of which is relevant for route selection. In the second part of this paper, we formulate a network abstraction notion, and show that it preserves realizability: i.e., every intent realizable on the abstract model is also realizable on the concrete network.

Detailed proofs and additional explanations are provided in the extended version of this paper [7].

2 Networks and Intents

We define the formal network model, the intent language, and intent satisfaction.

2.1 Network Model

A *network* is a hierarchical system of graphs. It is defined by a vector of graphs, say $(G_0, G_1, \ldots, G_n)$, for $n \geq 0$. A graph G_i is either a primitive graph with a single node, or a non-primitive graph where each node is a reference to a copy of a graph G_j, where $j < i$, giving the entire network a hierarchical structure. The graph G_n is the root of the hierarchy.

A network *attribute* is a quantity such as bandwidth, bit-error rate (BER), cost, or delay, which takes values from the appropriate domain. An *attribute vector* is a map from the set of attributes to their domains. E.g., "(bandwidth=1.0, BER=1.0E-5, cost=20, delay=2.5)" is an example vector. For concreteness, we focus on two important attributes: bandwidth and delay, so the vector is written as (bandwidth, delay). Attribute vectors are ordered by a partial relation, $\succeq$ (read as "better than"), defined appropriately. For bandwidth and delay, the relation $(b, d) \succeq (b', d')$ is defined as $(b \geq b') \wedge (d \leq d')$. I.e., (b, d) is better than (b', d') if b represents more bandwidth than b', and d represents a smaller delay than d'. The inverse relation, $\preceq$, is read as "worse than".

A *primitive* graph has only one node whose ports are all *external.* It represents an atomic building block of the network. There can be zero or more internal *links* between each pair of ports. Each link is associated with a *capability*, which is an attribute vector. The implicit understanding is that all links in a primitive graph represent *independent* connections. The capability of the i'th link from port p to port q of node n (if defined) is denoted $\mathsf{cap}(n, p, q, i)$.

Examples of primitive graphs are channels and mux/demux elements. A *channel* has one input port and one output port. A multiplexer (*mux*) has one output port, say q, and multiple input ports; a link is defined only for pairs (x, q), where $x \neq q$. A demultiplexer has one input port, say p, and multiple output ports; a link is defined only for pairs (p, x), where $p \neq x$.

A non-primitive graph, G_i, has internal structure that is given by a pair (N, C), where N is a set of *nodes*, and C is a set of *connections.* Every node has an associated set of *ports*. A connection is a pair of the form $((n, p), (n', p'))$, indicating that port p of node n is to be identified with port p' of node n'. The *external* ports of a graph are those ports that are not part of any connection. Every node of G_i contains a reference to a graph G_j, where $j < i$, along with an isomorphism between the ports of the node and the external ports of G_j. Nodes may have *region* labels, used to state routing constraints that require paths to stay within a certain geographic or network region.

A flat (i.e., non-hierarchical) network can be obtained by starting from G_n and recursively expanding each node into a copy of the graph to which it refers, if that graph is non-primitive. The satisfaction of intents is defined over the flattened graph, which may be exponentially larger than the network description. For convenience, by the links of a node we mean the links of its primitive graph.

Paths. The tuple $(p'_i, n_i, l_i, w_i, p_{i+1})$ represents the l_i'th link between input port p'_i and output port p_{i+1} on node n_i, with an associated attribute weight vector w_i. A *path* from port p of node n to port q of node m, represented as $(p'_0, n_0, l_0, w_0, p_1), (p'_1, n_1, l_1, w_1, p_2), \ldots, (p'_k, n_k, l_k, w_k, p_{k+1})$, is a sequence of such links, with $k \geq 0$, $(p'_0, n_0) = (p, n)$, and $(n_k, p_{k+1}) = (m, q)$. A path should meet the following conditions.

(a) p'_i and p_{i+1} are ports of n_i for all i, and l_i is a valid link between those ports
(b) w_i represents an allocation that is worse than the capability of its link, i.e., $w_i \preceq \mathsf{cap}(n_i, p'_i, p_{i+1}, l_i)$ for all i (I.e., w_i allocates less bandwidth and assumes a higher delay than the actual capability of the link), and
(c) For all i such that $i < k$, the pair $((n_i, p_{i+1}), (n_{i+1}, p'_{i+1}))$ is a connection.

The *allocated weight* of a path π is an attribute vector (b, d) such that b is the least bandwidth entry and d is the sum of all the delay entries in the set of weights $\{w_i\}$. The *capability* of π is the attribute vector (b', d') such that b' is the least bandwidth entry and d' is the sum of all the delay entries in the set of capabilities $\{\mathsf{cap}(n_i, p'_i, p_{i+1}, l_i)\}$. Requirement (b) ensures that the capability of a path is better than its allocated weight.

2.2 Network Intents: Syntax

An *intent* is a connectivity pattern between a set of ports. The pattern includes constraints on minimum bandwidth, or maximum delay. A *region* constraint is defined by a requirement to either *avoid* or to stay *within* the region. We define three common types of intents, and show later how these can be considered as examples of a quite general class of polynomially-checkable intents.

(Basic Segment). A *basic segment* specifies a connection between port p of node n and port q of node m, with constraints on attributes and regions.

(Protected Segment). A *protected segment* specifies a connection between port p of node n and port q of node m that has a degree of failure protection. The protection is defined as a set of basic segments between (n, p) and (m, q). For simplicity, in this paper we suppose that there are only two such segments, one referred to as the primary, and the other as the backup. This is commonly referred to as $1+1$ protection. Each basic segment has its own constraints on attributes and regions.

(Chain). A *chain* is specified as a sequence of segments where the end point of each segment in the chain is connected to the start point of its successor segment (if any). Each segment is specified independently, i.e., some may be protected, while others are basic. A chain may also have end-to-end attribute constraints (i.e., between its endpoints), and globally applicable region constraints. Chains are used to represent paths that must pass through a series of so-called middle-boxes in the network where packet processing occurs.

2.3 Network Intents: Semantics

Consider path $\pi = (p'_0, n_0, l_0, w_0, p_1), (p'_1, n_1, l_1, w_1, p_2), \ldots, (p'_k, n_k, l_k, w_k, p_{k+1})$. It satisfies a minimum bandwidth B if the bandwidth entry in each of the weights $\{w_i\}$ is at least B. It satisfies a maximum delay D if the sum of all the delay entries in the set of weights $\{w_i\}$ is at most D. It satisfies an $\mathsf{avoids}(R)$ constraint, for region R, if none of the nodes on the path is labeled with R, and a $\mathsf{within}(R)$ constraint if all of the nodes on the path are labeled with R. We can now define what it means for an intent to be satisfied.

(Basic Segment). A basic segment between port p of node n and port q of node m is satisfied if there exists a path π from (n, p) to (m, q) such that π satisfies all the attribute and region constraints for the segment.

(Protected Segment). A protected segment between (n, p) and (m, q) with two basic segments x_0, x_1 is satisfied if there are two paths, π_0, π_1 from (n, p) to (m, q) such that for each i, path π_i satisfies the requirements of the segment x_i and, moreover, π_0 and π_1 have no node-port combination in common except the two end points. i.e., the paths are node and port disjoint. Operationally, this implies that a single node or port failure cannot affect both paths, unless it is at the originating or terminating end.

(Chain). A chain from (n, p) to (m, q) is satisfied if there exist path(s) associated with each segment of the chain such that (i) the constraints for each segment are satisfied by its associated path(s), (ii) the end point (i.e., (node, port)) of the path witnessing a segment has a connection to the start point of the path witnessing the next segment, and (iii) the end-to-end constraints and global region constraints for the chain are satisfied on all end-to-end paths that can be constructed from the per-segment paths.

2.4 Witnesses and Satisfaction

For each satisfied intent, there is a network path (or paths) that explain *why* the intent is satisfied. That set of paths is called the *witness* for that intent. Figure 2 illustrates the three types of intents, corresponding witnesses, and how to check that a witness meets its intent. For instance, the witness for the protected segment is a pair of paths connecting Los Angeles to New York: green for

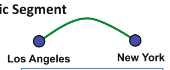
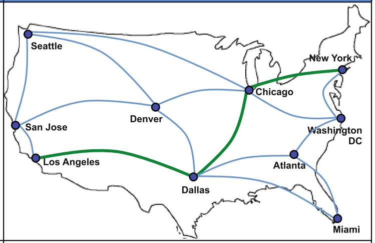
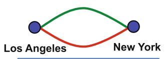
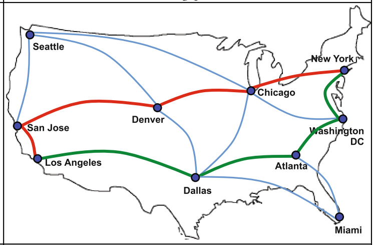
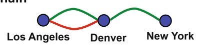
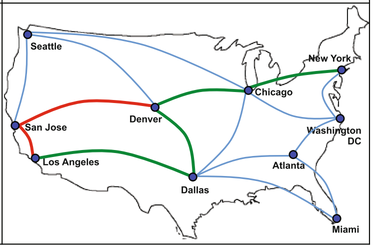
Intent	Witness
Basic Segment  Bandwidth ≥ b Delay ≤ d Avoids "Seattle" Within "USA" For the witness path, check: ❑ Validity ❑ Attribute constraints ❑ Region constraints	 Seattle New York Chicago San Jose Denver Washington DC Los Angeles Atlanta Dallas Miami
Protected Segment  Bandwidth ≥ b Delay ≤ d Avoids "Seattle" Within "USA" For each witness path, check: ❑ Validity ❑ Attribute constraints ❑ Region constraints ❑ Disjointness	 Seattle New York Chicago San Jose Denver Washington DC Los Angeles Atlanta Dallas Miami
Chain  Bandwidth ≥ b Delay ≤ d Avoids "Seattle" Within "USA" Check: ❑ Each (protected) segment ❑ Connection between segments ❑ End-to-end constraints and global region constraints for the complete chain	 Seattle New York Chicago San Jose Denver Washington DC Los Angeles Atlanta Dallas Miami

Fig. 2. Example: three types of intents and corresponding witnesses. (Color figure online)

the primary and red for the backup segment. To determine if this witness is correct, one checks that the witness paths are valid in the network, the constraints on attribute and regions are be satisfied, and that the paths are disjoint.

Joint Satisfaction. A collection of intents is *jointly satisfied* if there are witnesses for each intent such that the witness paths *together* do not over-subscribe the bandwidth on any common link. Given a set of intents, if all the intents can be jointly satisfied, then each individual intent can be satisfied. However, the converse is *not* necessarily true. A trivial counter-example is a graph with a single channel of bandwidth 2. It is possible to individually satisfy intents with min. bandwidth 1 and with bandwidth 1.5, but joint satisfaction is impossible.

The inability to decompose the satisfaction of intents is one reason why route selection algorithms have high complexity. In a graph where all links have bandwidth 1, two basic segments between the same endpoints with bandwidth at least 1 require disjoint paths, which is an NP-complete problem on directed graphs. Another source of complexity is that intents must be satisfied in an *online* fashion, which may lead to sub-optimal decisions. E.g., consider two points connected by disjoint paths π and π', with resp. bandwidths 1 and 2. A request for bandwidth 1 can be satisfied by either path; say it is assigned to π'. A following request for bandwidth 2 cannot then be satisfied, unless the first is re-assigned to π.

Witness Generation. For these reasons, the actual route selection algorithm may be quite complex. However, its natural output is the set of paths that form the witness. With standard algorithmic schemes, no additional work is needed. Such algorithms allocate new routes on a residual capacity network, where the capacity of a link is the amount that remains after satisfying previous intents. If a new request is met on the residual network, its witness does not interfere with those for previous intents, so the algorithm merely reports previously stored witnesses. However, the algorithm may need to backtrack to recover from sub-optimal decisions. In that case, the set of witnesses it needs to report are for the intents that are re-assigned paths. In either case, witness generation does not require additional work. As the validation procedure checks joint satisfaction, witness paths for *all* intents are provided with each routing decision.

3 Witness Checking

The algorithm to check whether a witness matches a basic intent type on a flat network model is shown in Fig. 3. The algorithm follows quite directly from the definitions, as formalized in Sect. 2, and is easy to implement. For each intent type, the algorithm checks that the witness paths provided are (a) valid paths in the network, and (b) satisfy the attribute and region constraints specified for the intent. The capacity of the network is reduced by the bandwidth consumed by the witness paths; the algorithm outputs a network with the remaining capacity. The algorithm operates in linear time in the size of witness. (The disjointness check in Case 2 can be done in linear time on average using hashing.)

The algorithm works on a fully flattened network, which is obtained by flattening the hierarchical network before a NOS is deployed to receive intents. An optimization is to retain the hierarchical form, and flatten only those sections of the network which are traversed by the witness paths. It is an open question whether the check can be performed in polynomial time without flattening, we conjecture that this may not be possible. As the check removes bandwidth from the components through which a witness path passes, copies of the same component may, over time, diverge in the set of feasible paths. This is not the case for pure reachability queries, which can be checked without flattening [4].

General Forms of Intents. The intent types discussed so far fit the following general form, which is inspired by Fagin's beautiful characterization of NP in terms of existential second order formulae on graphs [8]. An intent specifies a sub-graph over a set of points, H, such that there exist sub-graphs $X_0, \ldots, X_n$ for which $\varphi(H, X_0, \ldots, X_n)$ holds, where φ is a polynomial-time checkable property. As an illustration, for a protected segment, the two endpoints (defining H)

```
Function wcheck($i$ : intent, $w$ : witness, $M$ : flat network) : flat network
    Check that each witness path in $w$ is a valid path in network $M$
    if $i$ is a basic segment then
        Check that the path defined by $w$ satisfies the attribute and region
          constraints in $i$ as defined in Section 2
        Let $M'$ be obtained from $M$ by reducing the bandwidth on each link by
          the amount reserved for that link on $w$
        return $M'$
    else if $i$ is a protected segment of intents $i_0, i_1$ with witnesses $w_0, w_1$ then
        Check that the paths $w_0, w_1$ are node and port disjoint
        $M_0 := \mathsf{wcheck}(i_0, w_0, M)$
        $M_1 := \mathsf{wcheck}(i_1, w_1, M_0)$
        return $M_1$
    else $i$ is a chain of intents $i_0, \ldots, i_n$ with witnesses $w_0, \ldots, w_n$, end-to-end
      constraints delay $D$ and bandwidth $B$, and global region constraints
        $D_n := D$
        for $k$ from $n$ down to $0$ do
            if $k > 0$ then
                Check that start point of $w_k$ is connected to end point of $w_{k-1}$
            end
            Let $i'_k$ be $i_k$ with additional constraints of min. bandwidth $B$, max.
              delay $D_k$ and global region constraints.
            $M := \mathsf{wcheck}(i'_k, w_k, M)$
            $D_{k-1} := D_k - \mathsf{maxdelay}(w_k)$
        end
        return $M$
    end
```

Fig. 3. Witness checking algorithm

are connected by path-shaped sub-graphs X_0 and X_1, with φ asserting that the paths are disjoint and satisfy the attribute and region constraints. The witness for an intent in general form is the instantiation given to $X_0, \ldots, X_n$, while witness checking is the evaluation of φ on this instantiation. A number of practically useful connectivity patterns can be specified in this manner. Examples include broadcast and multicast trees, possibly with disjoint backup paths; virtual networks that interconnect several ports; and grid topologies.

3.1 Incremental Checking

Starting from the un-allocated network model, the algorithm above is used to check each witness in succession. This takes time linear in the number of active intents. We describe an efficient incremental algorithm, which checks only those intents whose witnesses have changed.

The key underlying observation is that the order in which a set of witnesses are checked does not matter. Consider witnesses w and w' provided for intents a and a', respectively. Starting from a network M, if the check succeeds in the order $w; w'$, it must also succeed in the order $w'; w$. This is because the check can be split into a step which determines the connectivity of witness paths, ignoring capacity; and another that reduces network capacity along the witness paths, while ensuring that the residual capacity on each link is non-negative. If no link has negative capacity when witness paths are allocated in the order $w; w'$, that is also true for the reverse order $w'; w$.

The algorithm stores the residual capacity network, M, and the list of active intent-witness pairs, W, with the invariant that M represents the residual capacity after processing W on the un-allocated network N. Route selection produces a list of intent-witness pairs, W', listing only the intents that have new witnesses. The incremental algorithm proceeds as follows.

1. For each (i, w') in W', if there is an entry for intent i, say (i, w), in W, undo the capacity reduction effect of checking w by adding back the capacity used by links w to M. Remove the (i, w) entry from W
2. Add into M the effects of any network change that *reduces* the capacity of a link l; if the new capacity of l is negative, *stop with error*
3. Add into M the effects of any network change that adds new links or *increases* link capacity. We suppose that such links are disjoint from those whose capacity has been reduced
4. Check the intent-witness entries in W' on M with the wcheck algorithm, updating the residual capacity in M
5. Append W' to W to obtain the new active list

Incremental algorithms usually trade off increased state (e.g., storage for partial results) for speed. It is interesting that this algorithm uses no additional space. We show the following correctness theorem.

Theorem 1. *The incremental and basic algorithms produce the same result.*

4 Experiments

This section presents an experimental evaluation of our witness checking implementation. We do not have access to real network designs, so the experiments are on a synthetic network, a parameterized grid of size n, shown in Fig. 6, where each link has bandwidth and delay 1. The parameterization makes it simple to scale up network size to assess its influence on witness checking.

For the experiments, a grid network is set up for a particular value of n. Then endpoints and intents connecting them are generated at random. The type of intent (basic or protected) is also chosen at random. Corresponding witnesses paths are calculated via depth first search (DFS) while keeping track of residual capacity. The search is prioritized to prefer links closer to the destination node. The DFS algorithm approximates the work of actual route selection algorithms used in networks. It suffices for our purpose, which is to measure the performance of witness checking, not the quality of the chosen routes.

The implementation is in Java, it includes network creation, intents generation, witness calculation and checking. The checker is about 300 lines of Java code. All of the experiments are performed on a MacBook Pro machine with a 2.4 GHz Intel Core i7, and 8GB 1600 MHz DDR3, running on Mac OS X 10.10.5.

In the experiment, we simulate networks of size from 10 to 1000; accordingly, the number of nodes varies from one hundred to one million. In each network, 500 intents are randomly generated, and corresponding witnesses are calculated and checked by our algorithm in Fig. 3. The results are shown in Fig. 4. The x-axis shows the network size n (there are n^2 network nodes). The left-hand y-axis shows the average time cost of checking a witness for a single intent, and the right-hand y-axis shows the average size of a witness. It is clear that the average time cost of checking is negligible (e.g. for a large network of one million nodes, checking a witness takes only about 1 ms, in the meanwhile, according to our experiment log which is not presented here, witness generation by DFS takes about 20 ms). The graph shows also that the cost of checking is proportional to the witness size, both of which scale as $O(n)$, on average. A second experiment fixes the network size to 1000 but varies the number of intent requests. The results support the theoretical analysis, showing that the cost of the incremental algorithm is essentially constant, while that of the basic algorithm increases linearly with the number of requests.

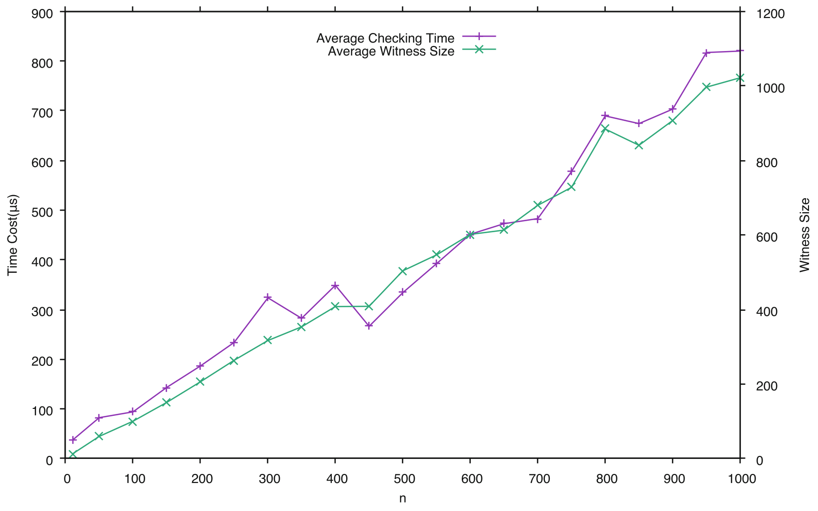


Fig. 4. Time cost of witness checking on networks of n^2 nodes.

5 Network Abstraction

The witness checking algorithm introduced in previous sections works on the complete network. It is, however, often the case that only a small part of a network needs to be examined to select routes. E.g., for an intent requesting a connection between two cities in the east coast, say New York City and Washington DC, it would be superfluous to examine networks in the west coast, as well as tedious to use detailed information about networks inside a single city. Thus, we propose to operate algorithms on abstracted networks. It is vital, however, that the routes discovered at an abstract level are realizable as routes at the concrete level; otherwise, there is no benefit to perform the abstraction.

In this section we introduce *network abstraction*. The general idea is to get an abstract network by collapsing a specified sub-network of a concrete real network into a single node. We define a notion of refinement from the concrete to the abstract level, and show that this preserves the realizability of routes.

5.1 Abstraction and Refinement with Single Nodes

We consider the case where a graph G is abstracted to a new primitive graph H whose external ports are isomorphic to the external ports of G. The key question is to define a relation between paths and capabilities in G with those in H, so that routes in H can be realized as routes in G. As H is primitive, routes in H are links between ports; routes in G are paths through the graph G.

Refinement. A *refinement* map R from H to G is a function such that the following properties hold:

(a) Each link (n, p, q, i) in H (i.e., the i'th link between port p and q of node n) is mapped by R to a path π between ports p and q in G, where the capability of π in G is better than the capability of link (n, p, q, i) in H, and
(b) The set of paths $\{R(n, p, q, i) \mid (n, p, q, i) \textit{ is a link in } H\}$ are node and port disjoint in G, and
(c) Node n and all nodes of G have the same abstract region labels.

The refinement map constrains the capabilities, not the weights of the corresponding paths. Hence, it is possible that a different algorithm can be applied to G to arrange the weights.

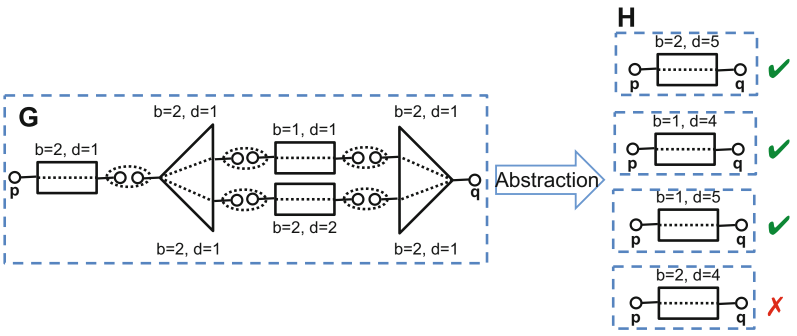


Fig. 5. Collapsing a sub-network into a single node via refinement.

Example. A simple example of refinement is shown in Fig. 5. For the sake of clarity, we do not use the formal notion of graph references, but rather show the details directly. Ports are shown as circles, a long rectangle is a channel, and a triangle is a mux/demux. A dashed line between two ports is a link, and a dashed ellipse with two ports inside shows that those ports are part of a connection. (E.g. in G, the right port of left channel is connected to the left port of demux.) The capability of the single link for each pair of ports is shown near the host node. (E.g. in G, "$b = 2, d = 1$" above the left triangle means that, for the upper link inside the demux, the bandwidth is 2 and delay is 1.) Between ports p and q in G, there are two non-disjoint paths: one path goes through the upper channel in the middle, and has capability "$b = 1, d = 4$"; the other path goes through the lower channel in the middle, and has capability "$b = 2, d = 5$". We show four possible abstractions; the upper three are correct (i.e. there is a refinement connecting H to G). The first two represent the capabilities of the paths in G described above; the third is a manufactured capability representing the worst of the two paths. The bottom abstraction is incorrect, however, as there is no path in G from p to q with capability better than "$b = 2, d = 4$".

5.2 Abstraction and Refinement for Networks

We say that network A is an *abstraction* of network $C = (G_0, G_1, \ldots, G_n)$ if there is a chosen subset GS of $\{G_0, G_1, \ldots, G_n\}$, and A is gained from C by replacing each graph G_i in GS with a primitive graph H_i such that there is a refinement R_i from primitive graph H_i to graph G_i. The *size of abstraction* is defined as the cardinality of GS.

Example. Figure 7 illustrates the process of network abstraction. The concrete network C has two graphs (G_0, G_1), where G_1 contains two connected nodes referring to G_0. There is a refinement relation from the primitive graph H_0 to G_0, thus by replacing G_0 with H_0 we obtain an abstract network (H_0, G_1') where $G_1' = G_1[G_0 := H_0]$ (the brackets indicate substitution of references to G_0 by references to H_0). The size of this abstraction is 1. Furthermore, another abstraction of size 1 can be performed on (H_0, G_1') by replacing it with the primitive graph H_1, since there is an abstraction refinement from H_1 to G_1'. Now an ultimately abstract network (H_0, H_1) is obtained, and no more abstraction can be applied. Furthermore, H_0 can be removed since it is not referred by any network. It is not difficult to find that for any set of intents that can be jointly satisfied in the abstract network (H_1), it can be jointly satisfied in the original network (G_0, G_1) too.

Theorem 2. *Let network A be an abstraction of network C. Every set I of intents that can be jointly satisfied in A can also be jointly satisfied in C.*

Proof Sketch: Suppose the size of abstraction from C to A is k. We generate a series of networks $N_1 = A, N_2, N_3, \ldots, N_k = C$ such that N_{i+1} is a refinement of N_{i+1} with an abstraction of size 1. We show that any set of intents that can

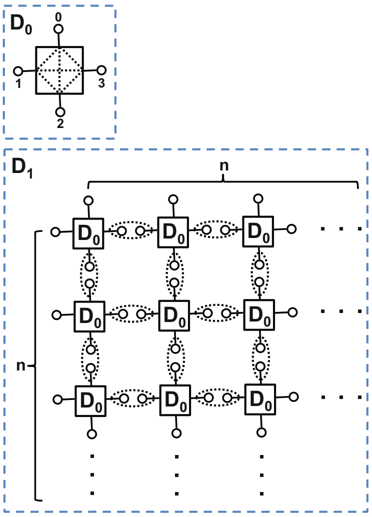


Fig. 6. Virtual network of size n.

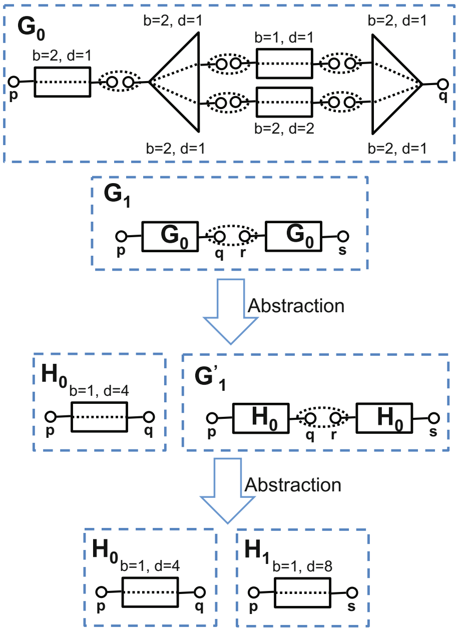


Fig. 7. Network abstraction

be jointly satisfied in N_i, can also be jointly satisfied in N_{i+1}. By induction, it follows that any set I of intents that is jointly satisfied in A is also jointly satisfied in C. **EndProof.**

As illustrated in Fig. 7, realizability is preserved across multiple abstract levels, i.e. if network A abstracts B and B abstracts C, then any set of intents that can be jointly satisfied in A can also jointly satisfied in C.

6 Related Work and Conclusions

The certification strategy is inspired by research on methods to verify compiler transformations. Run-time compiler verification, generally referred to as Translation Validation, uses heuristics to determine whether the resulting program refines the behavior of the original (cf. [20,22,27]). Our recent proposal [19], building on the idea of proof certificates [21,23], suggests having the compiler itself generate candidate refinements; valid refinements are called witnesses. We adopt this general scheme and terminology.

There are, however, fundamental differences between Translation Validation and network validation. For compiler optimizations, correctness is established by showing that the optimized program refines the behavior of the original. A routing decision, however, may change routes arbitrarily so long as the intent is met; thus, correctness does not correspond to a natural refinement on networks. Instead, the criterion adopted here is that the transformed network should satisfy all active intents with the particular route witnesses supplied by the network transformation algorithm. This differs from the model-checking question “Does the transformed network satisfy all intents?”, which implicitly checks for the existence of satisfying routes.

Emerging network operating systems based on SDN principles (cf. [24]), such as ONOS [1] and OpenDaylight [2], make it easy to replace route selection methods. These NOS’s do not, however, guard against potential network disruption caused by miscalculated routes. The lack of error-checking is a significant omission, which this work aims to fill.

There is a growing body of work on formalization of various aspects of SDN at the IP level: reasoning frameworks such as NetKAT [5], verified compilers [12] for OpenFlow [18] and model checkers for network invariants (cf. [6,14,17]). Run-time checking has been investigated at the IP level: the Veriflow [15] system checks routing table modifications against fixed network properties such as the absence of a forwarding loop. Reachability properties can be checked off-line by the system in [16]. As discussed in the introduction, our work applies to NOS’s that work at a different (higher) level of abstraction, managing combinations of networks with diverse technologies. Thus, the existing techniques do not apply.

The network model in this paper is inspired by NetML [10,25], which was designed to describe connectivity in multi-layered networks. Our model expands on NetML to include link attributes such as bandwidth and delay. In turn, this requires new forms of abstraction to preserve the realizability of intents. Work on abstraction in the IP model includes [26], which describes an IP network as a virtual “big switch” (cf. [13]); routes programmed at the virtual level are then refined into routes on a physical topology. This refinement notion preserves reachability but may not preserve path disjointness.

Network management is clearly moving towards increasing levels of abstraction and programmability. With increasing sophistication, however, comes the danger that software errors may result in significant disruption in large area networks. This work has presented a run-time certification method which acts as a safety net, preventing incorrect routing decisions from affecting network operations. The checking process is efficient, and naturally handles a variety of user-defined specifications and dynamic network changes. A promising direction is to explore witnessing for IP networks, particularly where model checking is difficult (e.g., checking reachability in the presence of packet filters is NP-hard [17]).

Acknowledgements. We wish to thank colleagues at Bell Labs for many helpful comments. Kedar Namjoshi was supported, in part, by NSF grant CCF-1563393 during the preparation of this paper.

References

1. ONOS: Open Network Operating System. http://onosproject.org/
2. Open Daylight. https://www.opendaylight.org/
3. RFC 6241 - Network Configuration Protocol (NETCONF). https://tools.ietf.org/html/rfc6241

4. Alur, R., Yannakakis, M.: Model checking of hierarchical state machines. ACM Trans. Program. Lang. Syst. **23**(3), 273–303 (2001). doi:10.1145/503502.503503
5. Anderson, C.J., Foster, N., Guha, A., Jeannin, J., Kozen, D., Schlesinger, C., Walker, D.: NetKAT: semantic foundations for networks. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, January 20–21, 2014, pp. 113–126. ACM (2014). doi:10.1145/2535838.2535862
6. Canini, M., Venzano, D., Peresíni, P., Kostic, D., Rexford, J.: A NICE way to test openflow applications. In: Gribble and Katabi [11], pp. 127–140. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/canini
7. Deng, C., Namjoshi, K.S.: Witnessing network transformations (2017). Extended version of this paper, at http://cs.nyu.edu/~deng/
8. Fagin, R.: Generalized first-order spectra and polynomial-time recognizable sets. In: Karp, R. (ed.) Complexity of Computation, SIAM-AMS Proc., pp. 27–41 (1974)
9. Feamster, N., Rexford, J., Zegura, E.W.: The road to SDN: an intellectual history of programmable networks. Comput. Commun. Rev. **44**(2), 87–98 (2014). doi:10.1145/2602204.2602219
10. Fortune, S.: Equivalence and generalization in a layered network model. J. Comput. Syst. Sci. **81**(8), 1698–1714 (2015). doi:10.1016/j.jcss.2015.06.004
11. Gribble, S.D., Katabi, D. (eds.) Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25–27, 2012. USENIX Association (2012). https://www.usenix.org/publications/proceedings/?f[0]=im_group_audience%3A279
12. Guha, A., Reitblatt, M., Foster, N.: Machine-verified network controllers. In: Boehm, H., Flanagan, C. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle, WA, USA, June 16–19, 2013, pp. 483–494. ACM (2013). doi:10.1145/2462156.2462178
13. Kang, N., Liu, Z., Rexford, J., Walker, D.: Optimizing the "one big switch" abstraction in software-defined networks. In: Almeroth, K.C., Mathy, L., Papagiannaki, K., Misra, V. (eds.) Conference on emerging Networking Experiments and Technologies, CoNEXT 2013, Santa Barbara, CA, USA, December 9–12, 2013, pp. 13–24. ACM (2013). doi:10.1145/2535372.2535373
14. Kazemian, P., Varghese, G., McKeown, N.: Header space analysis: static checking for networks. In: Gribble and Katabi[11], pp. 113–126. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian
15. Khurshid, A., Zou, X., Zhou, W., Caesar, M., Godfrey, P.B.: Veriflow: Verifying network-wide invariants in real time. In: Feamster, N., Mogul, J.C. (eds.) Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2–5, 2013, pp. 15–27. USENIX Association (2013). https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/khurshid
16. Lopes, N.P., Bjørner, N., Godefroid, P., Jayaraman, K., Varghese, G.: Checking beliefs in dynamic networks. In: 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4–6, 2015, pp. 499–512. USENIX Association (2015). https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/lopes

17. Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, B., King, S.T.: Debugging the data plane with Anteater. In: Keshav, S., Liebeherr, J., Byers, J.W., Mogul, J.C. (eds.) Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15–19, 2011, pp. 290–301. ACM (2011). doi:10.1145/2018436.2018470
18. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G.M., Peterson, L.L., Rexford, J., Shenker, S., Turner, J.S.: Openflow: enabling innovation in campus networks. Comput. Commun. Rev. **38**(2), 69–74 (2008). doi:10.1145/1355734.1355746
19. Namjoshi, K.S., Zuck, L.D.: Witnessing program transformations. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 304–323. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38856-9_17
20. Necula, G.: Translation validation of an optimizing compiler. In: Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation (PLDI) 2000, pp. 83–95 (2000)
21. Necula, G., Lee, P.: Safe kernel extensions without run-time checking. In: OSDI (1996)
22. Pnueli, A., Shtrichman, O., Siegel, M.: The code validation tool (CVT) - automatic verification of a compilation process. Softw. Tools Technol. Transf. **2**(2), 192–201 (1998)
23. Rinard, M.C., Marinov, D.: Credible compilation with pointers. In: FLoC Workshop on Run-Time Result Verification (1999)
24. Shenker, S., Casado, M., Koponen, T., McKeown, N.: The future of networking and the past of protocols. Open Networking Summit (2011)
25. Simsarian, J.E., Choi, N., Kim, Y.J., Fortune, S., Thottan, M.K.: Netgraph data model applied to multilayer carrier networks. In: OFC (2016). doi:10.1364/OFC.2016.Th4G.2
26. Smolka, S., Eliopoulos, S.A., Foster, N., Guha, A.: A fast compiler for netkat. In: Fisher, K., Reppy, J.H. (eds.) Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1–3, 2015, pp. 328–341. ACM (2015). doi:10.1145/2784731.2784761
27. Zuck, L.D., Pnueli, A., Goldberg, B.: VOC: a methodology for the translation validation of optimizing compilers. J. UCS **9**(3), 223–247 (2003)