Network Design Automation --- Treating Networks like Programs and Chips

These Professors! Always wanting the moon! -- Bianca Castafiore in Herge's "The Calculus Affair"

Why are my packets being dropped? Why does my Internet connection keep resetting? Programmers have access to a panoply of tools including compilers, debuggers, static and dynamic checkers. Hardware designers have synthesis, place-and-route, and timing verification tools. Unfortunately, network operators still use crude tools such as TraceRoute and Ping that have stayed unchanged for decades. Meanwhile, networks have grown increasingly complicated: load balancers spread traffic using unknown hash functions, and firewalls interdict messages based on cryptic rules that human operators enter, often incorrectly.

Trends: In a world of services accessed from tablets and mobile devices, network failures are a debilitating source of operational expenditures. Gartner estimates the public cloud market to be $139B dollars, and TBR estimates the private cloud market at $10B. Companies such as Microsoft which traditionally sold software on disks are transitioning to XBOX and Office Online. McKeown [1] suggests the potential market opportunity for network verification by noting that in both software and chip worlds a $10B tool industry supports a $250B (software) and $300B (chip) industry. While money is no guarantee of research importance, it suggests this is a problem industry cares about. A survey we conducted on the NANOG (the organization of network operators) mailing list revealed that 35% of respondents had at least 25 tickets per month that took more than 1 hour to resolve [2].

At the same time, the router market is transitioning to software defined (SDN) routers that can be programmed by users instead of the closed, vertically-integrated devices from Cisco and Juniper that permeate the market today. Merchant silicon vendors such as Broadcom provide cheap ASICs that can be assembled into cheap routers. A language called P4 [3] promises to go beyond the first generation of SDN routers and allow even the *forwarding* process of routers to be redefined at run-time.

While SDNs enable simple and efficient routers that can be tailor-made to an organization's needs, SDNs can amplify the opportunity for errors. While one can carp about the software quality from Cisco, their code has been field-tested for years in real networks. Is it not wise to be a little apprehensive of user-level software that changes often and writes bits in tables that routers blindly execute?

Research Philosophy: On the surface, networks with their collections of routers, Network Interface Cards, and cabling do not resemble code. But one can view the entire network as a "program" that takes packets from the input edges of the network and outputs packets to the output edge after possibly rewriting. The forwarding table at a router can, for instance, be viewed as a large Switch statement across destination addresses.

One can also view a network as a *circuit* using an EDA (Electronic Design Automation) lens [1]. If design rule checking is analogous to static checking, what is the analog of synthesis? At first glance, hardware specific tests such as Layout Versus Schematic (LVS) seem hard to map to networks. However, bugs in data centers can arise from wiring errors that make the physical topology differ from the logical topology. As in LVS tools, could these errors be detected using graph isomorphism?

These analogies have led networking researchers to frame a new research agenda, made compelling by the ubiquity of cloud services, called Network Design Automation (NDA, see Figure 1). We ask: *what are the equivalents of compilers/synthesis tools, debuggers, and static checkers for networks*? Early work has already produced static reachability checkers of impressive speed and scale [4, 5], algorithms to synthesize network firewalls from policy specifications [6], debuggers [7, 8], and automatic network testers [2]. While networking research has a long history of producing tools such as for network tomography, tools inspired by EDA and Software were rarer (see [9] for an early example).

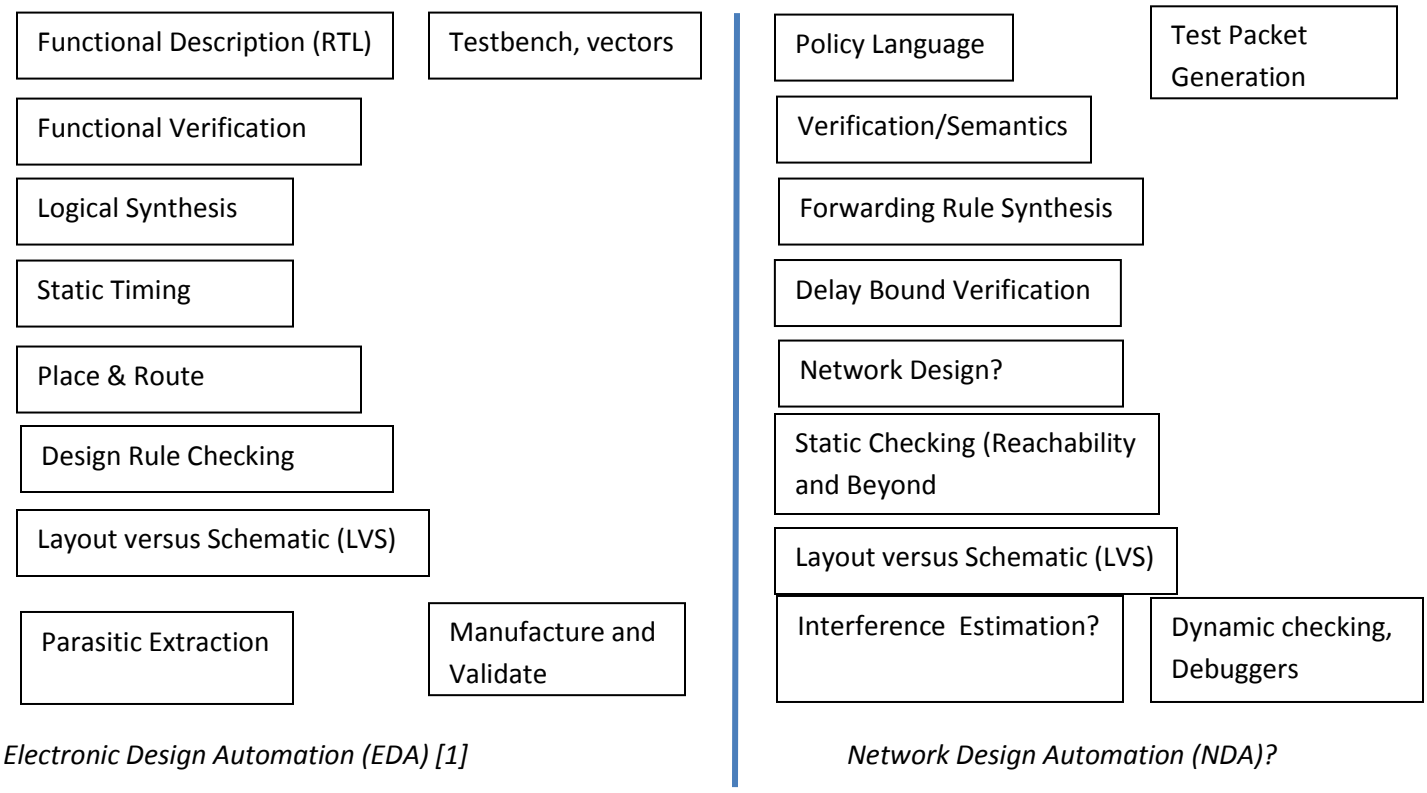


Figure 1: A vision for Network Design Automation

Instead of merely reusing existing ideas from programming languages and EDA, Network Design Automation has already produced *new problems and ideas*. For example, Network Verification often seeks all headers [10] that reach a destination which is more analogous to finding all satisfying assignments instead of the 1 assignment produced by a SAT solver. Network testing extends notions in software testing of code coverage to covering link and router queues [2]. Network debugging finesses the problem of global time in debugging using notions of causality [7]. I now discuss aspects of the vision in Figure 1 in more detail.

Static Verification: Early work has focused on verifying simple reachability predicates such as "Can customer VMs access fabric controllers?" After pioneering work by Anteater [11] that harnessed a SAT Solver, we [7] abstracted router forwarding as mapping a points in a *header space*, regardless of the networking device (e.g., routers, load balancers) or vendor (e.g., Cisco, Arista). Unlike Anteater, we find *all packets* that can or cannot travel between a source and destination. Crucial to ameliorating the cost of

exploring a state space of 2^{80} packet headers was a compression technique called *difference of cubes*, conceptually different from the Binary Decision Diagrams (BDDs) commonly used in symbolic model checking [12] and hardware verification [13].

To handle fast VM migration, we then asked whether reachability checks could be done in microseconds by exploiting *incremental computation*: the addition of a single rule should not substantially affect the set of forwarding equivalence classes. Our tool Net Plumber [5] did so by maintaining a dependency graph between router rules. NetPlumber and a concurrently invented tool called Veriflow [4] are the basis of startups, Forward Networks and Veriflow Networks respectively.

How do these tools relate to years of work in circuit verification and program verification? Network verification has many similarities to extended finite state machine verification. However, there is domain-specific structure we can exploit such as *limited negation, limited rewriting* instead of arbitrary relations, and *loop-freedom* in the error-free case. Further, we often wish to verify a different set of properties.

When trying to reuse some modern such as model checkers and SAT solvers, we found that many had been tuned for finding 1-solution and for other domains such as circuits. We found that Datalog solvers produce all solutions, but we had to rebuild the Datalog implementation engine to scale to large header spaces by defining, for instance, a *fused Select-Project* operator. The resulting Network Optimized Datalog or NoD [14] offers the benefits of a *higher level* verification language: a way to express higher level predicates via Datalog definitions, and support for network dynamism for P4 routers without modifying tool internals as in earlier tools. NoD is being used to write tools to check Azure clusters as they are rapidly rolled out to meet demand.

All existing static checkers for networks have been for reachability and for Boolean predicates (i.e., reachable/unreachable). We [15] are working on generalizing static checking from Booleans to *quantities* such as bandwidth, rates, and even probabilities. With a product team, we are attempting to do *traffic engineering under uncertainty* for the Microsoft backbone. The current SDN controller measures traffic from services and uses it to provision bandwidth, but the measurements are based on 90^{th} percentile measurements. But it is unlikely that many services will simultaneously reach their 90^{th} percentiles concurrently, wasting reserved bandwidth. We seek to pack more traffic in return for a small probability of overflow that must be computed automatically by our tool.

Other intriguing directions abound. First, are there network equivalents of work [16, 17] on *discovering specifications*? For example, if most customer VMs cannot access a service but a few can, the inconsistency may indicate a possible bug without *a priori* knowledge of which services a VM can access. Second, we must move beyond modeling the data plane to modelling the routing process [18] that computes forwarding entries to answer questions such as "How is link load affected by single link failures?" Third, while current tools verify a model of a router's code, a recent paper broke new ground by modelling the actual code of a Click Software Router [19]. Can this be done for router hardware specified in Verilog? Finally, there has been recent work in Data Center Networks that provide deadlines. Despite much prior work in embedded systems, the challenge is to make real-time verification scale to large header spaces.

Semantics: Assigning programs mathematical meaning enables mathematical proofs of correctness and can suggest optimizations in a principled way. With Gordon Plotkin and others at MSR, we have been exploring a new operational semantics for networks that enables modularity using parameterization and renaming. By contrast, NetKat [20] provides a denotational and axiomatic semantics for reasoning about networks but does not support modularity although most data centers such as Azure tend to reuse "pieces" tied together by core routers

Further, the current verification of all-pairs reachability of a large Azure network of 1000 routers and 100,000 rules takes over a day. However, the Azure network exhibits considerable symmetry. As in group theory, a topology exhibits symmetry if we can interchange a set of routers and links in a specified way and the behavior is unchanged. Intuitively, these symmetries arise because data center networks are designed for redundancy and performance using many backup routers and links.

Analogous in earlier work on symmetry reduction for model checking [21], one can "reduce" a symmetric fat-tree topology to an equivalent but simpler "thin tree" and show via a Milner-style *simulation* that any assertion or Hoare-triple that holds in the former holds in the latter. The potential payoff, which we plan to demonstrate experimentally [22], is that the verification cost of the simpler network is significantly smaller. Unlike earlier work [21] the symmetries being exploited are on the *physical* wiring graph and not on a *logical* graph of the state space.

Language Design and Synthesis: Designing either software or chips today is inconceivable without languages such as C++ or Verilog. For networks, there is a need for languages at two very different layers of abstraction. At the router level, there are already languages for reprogramming routers such as Openflow [23] and P4 [3]. However, they operate at a very low level of abstraction akin to machine code without support for modularity, types, or other higher level features [22]. We need a richer set of language features (analogous to those that System C adds over say Verilog) while retaining the ability to implement the language specification at Terabit speeds. We take a first stab at the problem of compiling such languages to hardware pipelines in [27].

With the coming of P4, the microcosm *within* a router is becoming similar to the macrocosm *between* routers in a network. This analogy suggests that debugging and verification work in networks can be useful within a router. We are building a tool [28] to automatically translate P4 specifications to Datalog to allow static checking within a router, and to enable automatic verification of network invariants even in the face of changing router forwarding behaviors.

On the other hand, there is a need for a higher level language to express the *network* policy in terms of access control, quality-of-service at an explicit, *declarative* level. Instead, today policy must be inferred via the *prescriptive* approach of programming configuration files at each router. Such a network policy language could also enable synthesis or compilation to individual forwarding rules at routers; it would have great advantages compared to manual entry of rules, with or without automatic verification.

One way to do this is to regard the network as "one big switch" [6]. Managers specify connectivity among "switch" endpoints, and this "policy" is compiled into forwarding rules at routers while respecting resource constraints. While this is a very good start, in practice our networks are too large to specify every pair of addresses that can communicate.

What seems to be required is some notion of *types* for endpoints (e.g., customer VMs, internal employees, controllers, router ports) and a set of *type restrictions* (e.g., all types can access DNS, but only controllers can access router ports) that can be specified more succinctly. Further, there are a number of grubby details today that complicate synthesis such as internal translation of addresses via NAT, rules added to routers for protection against broadcast storms, and aspects of policy that go beyond connectivity such as Quality of Service.

Dynamic Verification: Inspired by software testing where inputs are selected to maximize code coverage, our first foray in this space was ATPG [2] where the inputs were packets selected to maximize coverage of every link, queue, or forwarding rule. Unlike software testing, ATPG was used for performance testing and isolating links or queues that were causing end-to-end performance issues. When attempting to deploy performance testing within Microsoft, we found a number of issues such as the use of load balancing with unknown hash functions. This led us to devise *probabilistic covers*, a set of packets that cover paths with high probability; aspects of our NetSonar system are deployed in most of Microsoft's data centers as part of Autopilot. While such testing is adequate for smaller networks, perhaps analogs of white or black box fuzz testing [27] are needed for larger networks.

Going forward, inspired by the notion of *performance assertion checking* introduced by Perl [28] for single computers, we [29] are pursuing a more general approach to allow managers to write performance queries on a variety of underlying network raw data such as NetSonar readings, Syslog events, and SNMP counters. Such a query system can generalize the ad hoc alarm systems deployed today in all of Microsoft's properties as well as many special-purpose tools used to verify customer Service Level Agreements (SLAs). While such queries can be written using modern streaming databases, the research challenge is to exploit the domain requirements to build simpler abstractions for managers and a system that can more easily scale out than a general streaming query system. We have an opportunity to use our system to replace the inflexible and expensive tools currently deployed.

Debugging: While preventing failures using static or dynamic verification is the primary goal, speedy debugging after a network malfunction is also vital. The ticket system which logs all major failures in our networks records many failures that took hours to debug. While many research proposals exist including XTRACE [7], NDB [8], and Sherlock [30], they can be regarded as *batch debuggers* that log as many events/outcomes as possible and then sift through the logs to determine the cause of failure.

Just as program debugging has moved beyond print statements, the possibility of *interactive network debugging* is enticing. What are the equivalents of Stepping through and into functions, setting watch and break points, and doing Suspend and Resume, while not incurring excessive network or router overhead? For example, imagine setting a Watch Point that is triggered when end-to-end latency crosses a threshold, at which point the operator can "step into" the network and follow packets to each router in the network where one can examine counters and rule settings. While NDB provides related abstractions, it requires the (large) overhead of logging a "postcard" for every message.

While interactive debugging seems inherently expensive, one can exploit *lazy evaluation* to produce realizable solutions. For example, consider implementing a packet Suspend-Resume function [31] in a modern router pipeline where the packet can be paused at any stage and sent to a controller, and then reinserted from a specified stage. This *appears* to require wires from each stage to the processor, a daunting constraint. But it can be implemented with two extra header bits for Suspend and Resume, and a

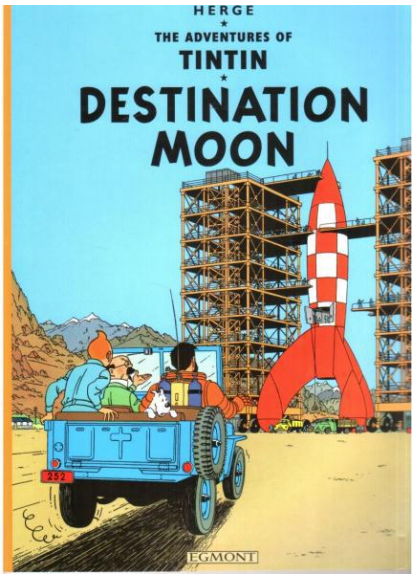
Stage Number **S:** if the Suspend bit is set, for example, all Stages beyond **S** merely pass the packet through without processing to the egress where it is passed to the controller without extra wires.

Similarly, 27 years after its initial proposal, Traceroute is showing its age. Originally, a ruse conceived to trick routers into providing path and latency information without explicit router support, Traceroute does not provide real-time latency information, and fares poorly with load balancing. Instead, imagine a "Trace" bit set in a packet that causes information to be recorded about arrival and departure times and next hops. The information can then be queried after the fact with little overhead using SNMP or new interfaces inspired by standards such as JTAG (used for sending and receiving test data from chips.)

Such additional router features and header bits may seem infeasible but are, in fact, enabled by next generation SDN proposals such as P4 [3]. Router vendors struggling to avoid commoditization have incentives to add such features. Even with vendor support, finding the appropriate user abstractions and putting all the pieces together to build and integrated debugger is a challenge.

Physical Layer Tools: The tools described so far for networks have analogs in both software and hardware settings. However, networks share with chips the property that the lowest layers of abstraction are *physical* – e.g., optics and switching chips. What might the equivalent be of hardware tools such as Layout Versus Schematic [32] or Parasitic Extraction? One way to counter wiring errors in data centers would be to have bar codes in connectors that can be scanned to provide a physical layer wiring model that can be checked for isomorphism with the *logical* connectivity graph. While graph isomorphism is a hard problem in general, it is feasible for *structured networks* like fat-trees. Similarly, analogs of parasitic extraction could be useful for wireless networks where interference is a major issue.

Conclusion: Impressive strides have been made in improving the quality of large complex code including bug-finding for Linux [15] and Microsoft systems software [27], and formal verification of a real-time OS kernel [33]. Similarly, the EDA industry has enabled chips to scale to billions of transistors and still work. Why not networks next? While the ideas executed so far can be regarded as experimental rocket launches, they give us hope that someday we will reach the moon.



REFERENCES

1. N. McKeown, Mind the Gap. SIGCOMM Keynote, 2014, Pages 46 and 47 have a suggestive depiction of the current state of ASIC and Software design processes respectively
2. H. Zeng, P. Kazemian, G, Varghese and N. McKeown. *Automatic Test Packet Generation*, CoNEXT 2012.
3. P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco. A. Vahdat, G. Varghese, and D. Walker, *P4: programming protocol-independent packet processors*. Computer Communication Review 44(3): 87-95 (2014)
4. A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. *VeriFlow: verifying network-wide invariants in real time*. NSDI, 2013.
5. P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. *Real time network policy using Header Space Analysis*, NSDI 2013.
6. N. Kang, Z. Liu, J. Rexford, and D. Walker. *Optimizing the "One Big Switch" Abstraction in Software-defined Networks*, CoNEXT 2013
7. R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. *X-trace: A pervasive network tracing framework*, NSDI 2007
8. N. Handigol, B. Heller, V. Jeyakumar, D. Mazieres, and N. McKeown. *I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks*, NSDI 2014
9. N. Feamster and H. Balakrishnan. *Detecting BGP Configurations with Static Analysis.* NSDI, 2005
10. P. Kazemian, G. Varghese and N. McKeown. *Header space analysis: static checking for networks.* NSDI, 2012.
11. H. Mai, A. Khurshid, R. Agarwal, M. Caesar, B. Godfrey, and S. King. *Debugging the data plane with Anteater*. In SIGCOMM, 2011.
12. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. LICS, 1990.
13. Randal E. Bryant. *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE Transactions on Computers, C-35(8):677–691, 1986
14. N. Lopes, N. Bjorner, P. Godefroid, K. Jayaraman and G. Varghese. *Checking Beliefs in Dynamic Networks*, NSDI 2015, to appear.
15. G. Juniwal, N. Bjorner, U. Krishnaswamy, S. Seshia and G. Varghese. *Traffic Engineering under Uncertainty using Quantitative Network Verification*, work in progress
16. D. Engler, D. Chen, and A. Chou. *Bugs as deviant behavior: a general approach to inferring errors in systems code*. SOSP 2001.
17. W. Li, A. Forin, S. Seshia, *Scalable Specification Mining for Verification and Diagnosis*, 47th Design Automation Conference
18. A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. *A general approach to static analysis of network configurations*. NSDI 2015, to appear
19. M. Dobrescu and K. Argyraki . *Software Dataplane Verification*, NSDI 2014
20. C. Anderson, N. Foster, A. Guha J. Jeannin, D. Kozen, and D. Walker. *NetKAT: Semantic foundations for networks*, POPL 2014
21. E. Clarke, A. Emerson, S. Jha, and A. Sistla. *Symmetry reductions in model checking*, CAV 98
22. G. Plotkin, N. Bjorner, N. Lopes and G. Varghese. *Exploiting topological symmetry to reduce the cost of Network Verification*, work in progress
23. *OpenFlow Switch Specification Version 1.4.0*, https://www.opennetworking.org

24. N. Lopes, N. Bjorner, A. Rybalchenko, N. McKeown, D. Talyaco and G. Varghese. *Compiling P4 to Datalog to enable Automatic Network Verification,* work in progress
25. M. Budiu. *A Brief Critique of P4,* unpublished note
26. L. Yan, L. Jose, G. Varghese, and N. McKeown. *Compiling Packet Programs to Reconfigurable Switches,* NSDI 2015, to appear.
27. P. Godefroid, M. Levin and D. Molnar. *SAGE: Whitebox fuzzing for security testing*, CACM 2012
28. S. Perl. *Performance Assertion Checking*, Ph.D. Thesis, MIT, 1992
29. P. Gao, I. Stoica, S. Shenker, G. Varghese and M. Zhang. *Performance Assertion Checking for Networks*, work in progress
30. P. Bahl, R. Chandra, A. Greenberg, S, Kandula, D. Maltz and M. Zhang. *Towards highly reliable enterprise network services via inference of multi-level dependencies*, SIGCOMM 2007
31. M. Alizadeh, T. Edsall and G. Varghese. *Primitives and Abstractions for Network Debugging*, Ongoing discussions
32. F. Somenzi and A. Kuehlmann. *Equivalence Checking*, in Electronic Design Automation For Integrated Circuits Handbook, by Lavagno, Martin, and Scheffer
33. G. Klein et al. *seL4: Formal Verification of an OS Kernel*, SOSP 2009