Declarative Verifiable SDI Specifications

Rick McGeer
US Ignite
rick.mcgeer@us-ignite.org

ABSTRACT

The point of Software-Defined Infrastructure is an infrastructure that is at once more flexible, controllable, and transparent to user and developer. One important characteristic of this infrastructure is that it is not owned or controlled by the user. At runtime, it is an opaque black box. Thus, it must have guaranteed properties of both performance and function. Infrastructure also has limited visibility and debuggability. It's hard to diagnose network problems, and it's hard to diagnose runtime issues on a remote system. Thus, programs which manipulate the infrastructure (e.g., orchestration systems, SDN applications, etc.) should have their infrastructure manipulations verified, to the extent that this is possible. We need to catch bugs statically to the extent that we can, performance and correctness both.

Infrastructure configurations ought to be inherently verifiable. Verification of state-free systems is in $\mathcal{NP}$; verification of finite-state systems is always in $\mathcal{P}$-Space, and many problems are in $\mathcal{NP}$. It has been shown by a number of authors that OpenFlow rulesets are state-free, and verification is therefore in $\mathcal{NP}$. Similar arguments can be made for configuration layers and workflow engines, depending on precise semantics. These results imply that the underlying model of computation for configuration of software-defined networking and at least some elements of software-defined infrastructure are state-free or, at worst, finite-state, and therefore that verification of these systems is relatively tractable.

The large challenge before the community is then to design configuration models for software-defined infrastructure that preserve the precise and weak semantics of the implementation domain; offer appropriate abstractions of performance characteristics; and nonetheless retain usability and concision.

1. INTRODUCTION

The opening decade of the 21st century was dominated by the Cloud infrastructure, where relatively simple systems could scale automatically to support millions to tens of millions of simultaneous users. This decade and the next will be dominated by truly distributed systems, where the network forms the primary bottleneck on performance. This will be true for both economic and physical reasons. On the economic side, bandwidth price-performance is on a slower doubling curve than compute and storage price-performance. This means that the ratio between our ability to capture, store, and process data and our ability to transmit it is growing and continues to grow; over the past 15 years this ratio has grown by a factor of 100, and it shows no signs of slowing down[28].

As for physics, a nanosecond is, famously, a foot. But this in fact overstates the case; in fiber, light travels only eight inches in a nanosecond, and if a round-trip is required this is cut in half again: four inches. This means that a microsecond is roughly the standard American unit of measure – a football field. Highly interactive applications require a tight user/application interaction loop, which means that applications either require fat-client support or the Cloud node hosting the application must be a relatively short distance from the user. Since users can be anywhere, this means a Cloud POP to the user must be everywhere: hence our argument for a distributed cloud infrastructure, which is variously known as the Distributed Cloud[5], the Fog, or Cloudlets[29].

This is the emerging model of software systems: large-scale deployment of cloud systems connected by programmable, software-defined networks. The latter are required since the programmer must have explicit control of the infrastructure element critical to the performance of his application. Viewed in this light, offering an API for the programmer to control routing in the network is similar to giving a storage-system developer an API to control the layout of blocks on disks. But this model of software system deployment defeats traditional methods of debugging: the system itself will mostly be deployed remotely from the programmer, and its deployment will affect the infrastructure on which it rests. This implies a new scope for verification of such large-scale systems *before* deployment and auditing during deployment. We want to be able to answer questions such as: are my VM's configured correctly? Can VM x ever get into state y? Will packets sent by VM x reach VMs u, v, w? Do packets sent by VM x for VM y arrive

with header bits set ot α? Etc.

The typical means of answering these and similar questions was the usual trinity: simulation, emulation, and in-situ monitoring and alerts. But formal verification – rigorous proofs – can be added to the arsenal if the deployment and configuration of the virtual machines and the configuration of their underlying software-defined network are appropriately described. This paper describes such a description format and gives preliminary indications that it makes verification of networks of VMs feasible and is consistent with the tools used in configuration and deployment today.

The abstraction we will use throughout this paper is that of a *slice*: a virtual network of virtual machines. The term slice was first used to describe a collection of distributed PlanetLab virtual machines[24], and later used as the fundamental architectural unit of GENI. In this paper, we will be investigating rigorous descriptions of slices with strong verification properties.

The rest of this paper is organized as follows. In Section 2, we review the Turing Hierarchy of computational models and explore the verification properties of each of them. In Section 3 we define a mathematical model of switching networks. In Section 4 we examine the tools used for the automatic configuration and deployment of distributed virtual machines and conclude there is a finite-state model of computation which underlies them. In Section 5 we put these together to propose a model for slices, and in Section 6 we draw some final thoughts.

2. THE TURING HIERACHY AND THE VERIFICATION HIERARCHY

The Turing hierarchy is familiar to every advanced undergraduate computer science student: finite-state machines, pushdown automata, linear-bounded automata, and Turing Machines. These are distinguished by the amount of state a program can access. They are also mathematically equivalent to various classes of formal language, and this correspondence is taught in every first class in CS Theory.

For the purposes of computing systems, the only classes of machine that arise naturally are the Turing Machine, with unlimited state, and the Finite-State Machine, with fixed state. Strictly speaking, every extant computing system is a finite-state machine; however, once the number of states grows beyond a relatively small number, for purposes of verification the machine is effectively infinite-state.

To the classic models of computation, we add two at the bottom end, state-free systems and logic-free systems. A *state-free system* is any system where the system's output is dependent only on the value of its inputs, and not on any internal dynamic state. It is most frequently implemented as an acyclic graph of logic gate, and every state-free system is isomorphic to a graph (in fact, many graphs) of logic gates. A logic-free system is any system where the output is independent of the inputs; a good example is a graph. These are only of interest because some properties rely only on topological properties of the system rather than any functional value, and these are easily verified in polynomial time. One must be careful, however; for many years timing properties of logic circuits were verified using topological properties only. However, it was shown in [16] that accurate validation required considering functional properties as well, and this moved the problem from polynomial to $\mathcal{NP}$-complete.

Computational Model	Verification Complexity
Turing Complete	Undecidable
Finite State	$\mathcal{NP}$-complete to $\mathcal{P}$-space complete
State-Free	$\mathcal{NP}$-complete
Logic-Free	Polynomial

Table 1: Computational Class and Verification Complexity

As one moves up the Turing Hierarchy, complexity of verification increases. The most elementary verification of Turing Machine properties – is a Turing Machine guaranteed to halt? – was the first undecidable problem. At the other end of the scale, verification of logic-free systems is polynomial, and verification of state-free systems is in $\mathcal{NP}$. Verification of finite-state systems can range from $\mathcal{NP}$-complete to $\mathcal{P}$-space complete, depending upon the specific property to be proved (the "verification obligation"). Verification properties on finite state systems are typically expressed in a variant of temporal logic, and different temporal logics have different properties.

By far the most common form of finite-state system verification is the *reachable states iteration*, which, given an initial state set $R_0(x)$ and a Transition Relation $T(x, y)$ ($T(x, y) = 1$ iff y is a successor state of x), finds the fixpoint of:

$$R_n(x) = R_{n-1}(x) \bigvee \exists y(R_{n-1}(y) \bigwedge T(y, x))$$

Where the fixpoint is the least n such that $R_n(x) = R_{n-1}(x)$.

Denoting the fixpoint as $R^\star$, it is straightforward to show that the sum-of-products form of $R^\star$ is less than the size of the minimum sum-of-products form of T plus the size of the sum-of-products form of R. Exact complexity of the verification problem for finite-state systems varies with the form of temporal logic used to express the verification problem and obligation. Computation Tree Logic gives a polynomial verification problem; linear temporal logic gives at $\mathcal{P}$-space complete verification problem[3].

This gives us an interesting taxonomy of computa-

tional models and complexity of verification obligations, shown in Table 1. In this table we neglect degenerate special-case classes of easy problems. For example, verification of logic functions with only two variables per term, or verification of monotone logic functions, are trivially polynomial; finite-state machines show a range because the classes of finite-state problem which are easier than $\mathcal{P}$-space complete are nontrivial and cover many common cases.

The most obvious fact that jumps out from Table 1 is that the strength of the computational model and the complexity of the verification calculation are correlated: what this means is that, in practice, the ability to verify a system and the strength of its underlying computational model are *inversely* related. Put simply: **weak models make for strong verification, and strong models make for weak verification**.

This consideration gives pause to the general Computer Science predilection for offering the user and programmer arbitrarily strong computational models. It is easier than not to build Turing-complete computational models, and the power that comes with them is very attractive: "X is a great language; I can say anything I want in X" should perhaps be rephrased: "X is a problematic language; I can make any mistake at all in X, and it will be very difficult to find". In place of the principle of providing an arbitrarily-strong computational model, adopting two complementary principles values simultaneously expressiveness and verifiability:

1. The underlying computational model should not be stronger than the computational model of the underlying physical domain;

2. The underlying computational model should be no stronger than that required to perform the underlying task.

Granted, these principles are hard to achieve in general: they require a sophisticated knowledge of the underlying implementation domain and the programmer's task. There is no one-size-fits-all language or domain. But what one *can* do is look at broad classes of problem, analyze the characteristics of the implementation domain and the complexity of the programmer's task, and devise computational models appropriate to both the implementation domain and the task.

A good, if simple, example is in the area of syntax defintion and parsing of computer programs. The syntax of a programming language is given by a (typically, restricted) context-free grammar. A parser for the language is specified by describing the rules of this context-free grammar, with an action for each grammar rule; a parser generator such as YACC[11] or ANTLR[23] (or many others) can not only generate a parser for the language, it can find errors in the grammar specification such as ambiguities (existence of an expression with two or more distinct parse trees). This verification power comes from the fact that the grammar itself is a logic-free system, and thus its validation properties are polynomially-solvable.

3. A MATHEMATICAL MODEL OF NETWORKS

A network is a graph of *switches* and *routers*, whose principal task is to forward packets through the network according to a set of rules; each router or switch has its own individual ruleset. A rule is typically of the form $(\text{pat}, \text{port}) \rightarrow (\text{newHeader}, \text{outputPorts})$, which indicates that a packet whose header bits match pat arriving on port port should be sent out on the set of ports outputPorts with new header bits newHeader. Ports are physical pins on the router or switch.

For reasons of economy, transparency, and rapid processing, all of the state used in a router/switch's forwarding decisions is encapsulated in the rule set at any given moment; a switch or a router is optimized to forward packets as rapidly as possible, at hardware speeds where feasible. Execution of the rule set is given by the router/switch's *data plane*; updating the ruleset is the function of the *control plane*. Routers and switches are principally distinguished by the protocols they use to update the control plane; since this is beyond the scope of this article henceforward we refer to both routers and switches by the generic term "switch" with the understanding that this refers to both classes of device.

Historically, network verification centered on simultaneous verification of both the control and data planes. This was problematic, for a variety of reasons. First, the control plane implementation on a switch was a Turing-complete collection of programs, so simply verifying the control plane on an individual switch was undecidable. Second, switches change state asynchronously, in response to signals from other network switches and out-of-band control instructions, which means the global state of the network at any time is ill-defined.

The OpenFlow protocol[22, 21] decoupled the network's control and data plane. Rather than using an integrated on-switch control plane, an OpenFlow network conceptually uses a single network-wide controller, which downloads and periodically updates the rulesets on each switch. This opened up the possibility of verifying the network's data plane by itself.

The isomorphism of a single switch's data plane to a logic network had already been established in the context of switch optimization[20]; in fact this was immediate in the case of an OpenFlow switch from the fact that OpenFlow's forwarding rules are independent of any switch state. A network of switches was immediately isomorphic to a cyclic graph of logic circuits. A cyclic graph of logic circuits with delay on the internetwork connections is isomophic to a finite-state ma-

chine, where the inter-circuit connections are the state-holding elements. This in turn indicated that verification of the data plane was no harder than verification of a finite-state machine. This isomorphism was immediately exploited in [12], which modeled each packet as a finite-state machine whose state was changed as it traversed the network. This work used a variant of symbolic simulation[7] to validate network properties, and demonstrated strong results.

In fact, by exploiting a well-known property of switching networks, it was shown that verification of switching networks was isomophic to the simpler problem of verifying logic circuits. A well-known technique in verifying finite-state systems is bounded model checking[4]; this involves unrolling the finite state machine through k iterations into a single logic circuit; the circuit is of size $O(k|T|)$, where T is the size of the transition relation, and is verified using standard techniques of verifying logic circuits. This method is exact when the logic circuit encapsulates all possible states and transitions of the underlying FSM; in this case, the FSM is simply a compact representation of an underlying state-free logic circuit, and verification of the logic circuit verifies the FSM.

Switching networks permits a packet a finite number of network hops; this is the so-called "time-to-live" parameter of the network, and each packet keeps track of its remaining hops. Using this fact, one can unroll the switching network, where circuits at time t connect to circuits at time $t + 1$. The resulting logic circuit is acyclic, and of size nS, where S is the size of the switching network's logic circuits and n the maximum time-to-live. In [18] this was used to show that verification of the data plane of a switching network was in $\mathcal{NP}$.

A large number of network verification papers have appeared in the last few years, all exploiting the relative simplicity of verification of the data plane. Particularly notable in this area is [15, 35, 1, 13]. An excellent review and summary is given in [36].

The isomorphism of switching networks to logic circuits has had synergistic benefit to a number of well-known difficult problems in network management. An excellent example is network update, which can result in transient bad network configurations. The first method which guaranteed correctness appeared in [27], updated in [26]. This method operated both the original and updated network configuration in parallel and used markers on header bits to choose between them. An approach which used fewer network resources but required more latency was given in [17], and one which used verification techniques to find a safe schedule without any overhead was given in [19].

It should be noted that the verification process has limits: only the data plane is verified The control plane remains a Turing-complete artifact which can only be validated heuristically. The use mode that has grown up around this is verification of the data plane rule tables before they are loaded onto switches. Since the effect of the control plane is to configure the switch tables correctly, this amounts to run-time validation of the switch control plane.

The tractability of previously-insoluble problems due to the choice of an appropriate and accurate computation model for the network control plane has given rise to a number of description and specification formats which are carefully tailored to state-free specification of the network. Frenetic[9], Nettle[34] and FML[10] are all examples of declarative, state-free specification systems that preserved the stateless computational model of the network data plane.

4. A MATHEMATICAL MODEL OF VM CONFIGURATION

The mathematics of VM Configuration are much less well-developed than that of networks. This has only become an issue in recent years, with the advent of very large-scale Cloud and distributed systems. Up until this point, the management of remote virtual machines has been done through a combination of shell, Perl, and ssh scripts, ssh overlays such as Fabric, These were largely imperative tools designed to deploy, configure, and manage virtual machines, not describe their desired behavior or reason about it.

More recent tools such as Ansible[2], Chef[31], Puppet[25, 33], and Fabric[8] have addressed similar problems, but again their primary focus is on scalability and expressiveness of management, not on verifiability or semantics of the implementation domain.

A Virtual Machine is of course Turing-complete, and so verification of its complete state is undecidable. However, we can learn from the results on network verification summarized in the preceding section. Rather than focusing on the full behavior of the VM, we focus solely on its configuration, which comprises:

- Which software packages are loaded
- The state of each VM software daemon, to include at a minimum running or stopped, but with a richer state set as declared by the developer
- The state of network ports and flows, and other I/O devices (notably files)

The above-mentioned tools largely incorporate this model, particularly those (Chef, Puppet, Ansible), which incorporate a "Verifier Model" . Largely tuned to the coarsest aspect of system state (whether a software package successfully installed or not, whether a daemon is running or not) these systems check back and report to

the user/developer whether a particular task has succeeded.

In the cases of Ansible, Chef, and Puppet, state is implicit in the description and buried in the automation engine. Users can manage state explicitly through the use of variables in the configuration description languages for each tool. All three tools use declarative configuration languages, effectively compactly specifying the transition relation of a finite-state machine. In the case of Puppet and Chef, these are tool-specific languages; in the case of Ansible, Yet Another Markup Language (YAML) is used.

At a higher level are the workflow engines[30] such as Pegasus[6], XSede[32], or Kepler[14]. These are designed to manage the flow of scientific workflows across distributed grids. These share with the configuration management tools implicit finite-state semantics, though they are more focussed on deploying large-scale workflows across infrastructures. Most of these tools implicitly associate state with workflow elements such as large data sets.

The preponderance of finite-state engines for the configuration, deployment, management, and orchestration of processes and virtual machines strongly suggests that networks of finite-state machines are the most natural model of computation for these engines, and explicit expression of and utilization of their state will yield verification methods for these distributed infrastructures patterned on the verification methods for networks described in the previous section.

5. A MATHEMATICAL MODEL OF SLICES

A Slice is a virtual network of virtual machines, where the network is fully described by a set of forwarding rules on each switch and the VMs are fully described by their configuration, management, and execution. The results of the previous two sections have argued that each of these components alone has a rigorous, verifiable mathematical description: realized in the case of networks, implicit and nascent in the case of virtual machines.

The obvious conclusion to draw from this is to put these together into a single Slice Description framework, which would both provide a firm basis for verification and associated tools such as safe change and update, and from which the slice configuration, management, deployment, orchestration and workflow tools could be derived and from which they would operate.

6. CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK

The conclusion was effectively given in the previous section: we should build a mathematically rigorous configuration and deployment description for slices, based on a weak model of computation – ideally, interacting finite-state machines interconnected by verifiable logic circuits. This can be used as the basis of a mathematically rigorous verification system and as the basis for a reliable, secure generation of tools.

Acknowledgements

The author wishes to thank his colleagues at US Ignite for much support throughout this process, and Glenn Ricart for stimulating conversations. Experience with configuration management largely came from Andy Bavier, the author's long-time collaborator on many projects. Sharad Malik is always willing to talk over verification problems and listen politely to the author's offbeat ideas. The program committee, particularly Sergey Bratus, has been far more patient with the author than anyone should have to be. This work was partially supported by the GENI Project Office.

7. REFERENCES

[1] E. Al-Shaer and S. Al-Haj. Flowchecker: configuration analysis and verification of federated openflow infrastructures. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, SafeConfig '10, pages 37–44, New York, NY, USA, 2010. ACM.

[2] Ansible: Simple it automation. `https://www.ansible.com/`.

[3] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and software verification: model-checking techniques and tools*. Springer Science & Business Media, 2013.

[4] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[5] Y. Coady, O. Hohlfeld, J. Kempf, R. McGeer, and S. Schmid. Distributed cloud computing: Applications, status quo, and challenges. *Computer Communication Review*, 45(2):38–43, 2015.

[6] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, July 2005.

[7] S. Devadas, H.-K. T. Ma, and A. R. Newton. On the verification of sequential machines at differing levels of abstraction. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 7(6):713–722, 1988.

[8] Fabric api documentation. `http://docs.fabfile.org/en/1.8/`.

[9] N. Foster, R. Harrison, M. L. Meola, M. J. Freedman, J. Rexford, and D. Walke. Frenetic: A high-level language for openflow networks. In *ACM PRESTO 2010*, 2010.

[10] T. Hinrichs, N. Gude, M. Casado, J. Mitchell, and S. Shenker. Practical declarative network management. In *Proceedings of ACM SIGCOMM Workshop: Research on Enterprise Networking (WREN)*, 2009.

[11] S. C. Johnson. *Yacc: Yet another compiler-compiler*, volume 32. 1975.

[12] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[13] 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)*, April 2013.

[14] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[15] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, pages 290–301, New York, NY, USA, 2011. ACM.

[16] P. C. McGeer and R. K. Brayton. *Integrating Functional and Temporal Domains in Logic Design: The False Path Problem and Its Implications*. Kluwer, 1991.

[17] R. McGeer. A safe, efficient update protocol for openflow networks. In *Proceedings of Hot SDN*, 2012.

[18] R. McGeer. Verification of switching network properties using satisfiability. In *ICC Workshop on Software-Defined Networks*, June 2012.

[19] R. McGeer. A correct, zero-overhead protocol for network updatesâĂİ. In *Proceedings of Hot SDN*, 2013.

[20] R. McGeer and P. Yalagandula. Minimizing rulesets for tcam implementation. In *Proceedings IEEE Infocom*, 2009.

[21] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[22] The openflow switch specification. `http://OpenFlowSwitch.org`.

[23] T. Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[24] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences building planetlab. In *In Proceedings of the 7th USENIX Symp. on Operating Systems Design and Implementation (OSDI*, 2006.

[25] Puppet. `https://puppet.com/`.

[26] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and W. David. Abstractions for network update. *SIGCOMM Comput. Commun. Rev.*, Aug. 2012.

[27] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Software updates in openflow networks: Change you can believe in. In *Proceedings of HotNets*, 2011.

[28] G. Ricart and R. McGeer. Us ignite and smarter geni cities. In *The GENI Book*, chapter 20. Springer-Verlag, New York, 2016.

[29] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, 8(4):14–23, 2009.

[30] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields. *Workflows for e-Science: Scientific Workflows for Grids*. Springer Publishing Company, Incorporated, 2014.

[31] M. Taylor and S. Vargo. *Learning Chef: A Guide to Configuration Management and Automation*. O'Reilly, 2015.

[32] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkins-Diehr. Xsede: Accelerating scientific discovery. *Computing in Science Engineering*, 16(5):62–74, Sept 2014.

[33] J. Turnbull. *Pulling Strings with Puppet: Configuration Management Made Easy*. Apress, 2008.

[34] A. Voellmy and P. Hudak. Nettle: Functional reactive programming of openflow networks. In *Symposium on Practical Aspects of Declarative Languages (PADL)*, 2011.

[35] S. Zhang and S. Malik. SAT based verification of network data planes. In *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, pages 496–505, 2013.

[36] S. Zhang, S. Malik, and R. McGeer. Verification of computer switching networks: An overview. In *Tenth International Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2012.