Toward Systematic Detection and Resolution of Network Control Conflicts*

Dennis Volpano
Department of Computer Science
Naval Postgraduate School
volpano@nps.edu

Xin Sun
School of Computing & Information Sciences
Florida International University
xinsun@cs.fiu.edu

Geoffrey G. Xie
Department of Computer Science
Naval Postgraduate School
xie@nps.edu

ABSTRACT

The problem of detecting and resolving control conflicts has started to receive attention from the networking community. Corybantic [16] is an example of recent work in this area. We argue that it is too coarse grain in that it does not model the combined operational objectives of multiple controller functions. This paper proposes a finer grain approach where a network control function is represented as a deterministic finite-state transducer. The machine runs on inputs provided by an SDN controller and outputs instructions that update the network as needed to meet objectives. Standard proof techniques and algorithms can be leveraged to analyze properties of these machines. Specifically, their intersection describes precisely the *stable operating region* of a network when the machines operate in parallel. The stable region comprises conditions under which no control function is in the process of updating the network.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—network management; D.2.4 [**Software Engineering**]: Software/Program Verification—formal methods, correctness proofs

Keywords

transducers, SDN, controller function interaction

1. INTRODUCTION

By making the control plane programmable and centralizing it, software defined networking (SDN) builds upon well-established software development principles such as modular design and open APIs in an attempt to contain the growing complexity and cost of network management. In the last five years, a number of network controller functions have been conceptualized, each of which targets a single or narrow set of operational objectives, such as power manager [9], load balancer [1], bandwidth allocator [2, 17], application control manager [5], and virtual machine (VM) migrator [7, 15].

The proliferation of single-objective controller functions is both encouraging and daunting. On the one hand, it has successfully elevated the level of abstractions for network operation from device-level configuration to service-level requirements. On the other hand, it fosters independent development of functions that can interact in unpredictable ways and de-stabilize a network. For example, consider the network in Figure 1. Suppose the operator runs two controller functions: one aims to balance load per destination across links v and w, and the other aims to turn off E if it's under utilized (assuming that E consumes more power than F). The power-save function is at odds with the load-balancing function. The latter's objective is to balance load even if traffic is light at each router. So both routers must remain on. The power-save function will see this as an opportunity to shift flows to F and turn off E, which can trigger load balancing to turn it back on. So even under constant operating conditions, power to E can flipflop indefinitely with these two functions. This is an example of *oscillation* caused by multiple functions. Oscillation can also occur within a single function under constant conditions. This may be due to poor design or be intentional. For instance, the function may be implementing a circular scheduling algorithm for access to a shared medium like a software-defined wireless access network. We consider here oscillation caused by multiple functions. This kind of oscillation is also a challenge for network functions virtualization (see pg. 11 of [4]).

The problem of detecting and resolving control conflicts has started to receive attention from the networking community. Corybantic [16] is an example of recent work in this area. However, Corybantic's approach is too coarse grain in that it avoids modeling combined operational objectives of competing controller modules. For example, when multiple modules propose actions in response to an event such

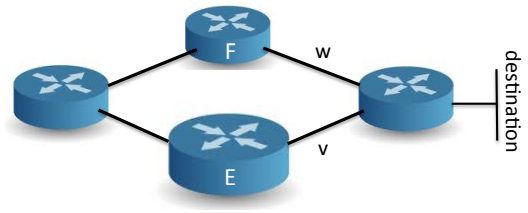

Figure 1: Unequal cost (power) load balancing

*Approved for public release; distribution is unlimited.

HotSDN'14, August 22, 2014, Chicago, IL, USA.

http://dx.doi.org/10.1145/2620728.2620745.

as a significant load increase over a link, Corybantic uses a policy-based approach to select only one of the proposals to implement. This approach is suboptimal from the perspective of meeting operational requirements because a) it discards possible solutions where multiple proposals without conflicts may be implemented at the same time; and b) it requires each controller module to determine the acceptability of the winning proposal (from another module) with an abstract currency that does not accurately measure how well the module will be able to meet its own objective.

A finer grain approach to detecting and resolving control conflicts is needed. There are two important requirements. First, a uniform representation of the operational objectives is needed in order to reason about the joint effect of multiple controller functions (Corybantic introduces an abstract currency for this purpose). Second, a controller function should be represented in a way that facilitates formal reasoning and makes key problems decidable. To this end, we propose a new programming model for SDN controllers that is quite different from current practice. The controller executes only code that is produced algorithmically from operational objectives expressed independently of one another. The code continuously observes the network and adjusts it in real time as needed to satisfy the requirements. Properties of the code are proved under the assumption that only the code can change the network. For instance, we might be able to show that there will never be an attempt to shift load to a router that is off. But if other controller code can power down devices then there's no guarantee. Therefore, no manually-written controller code using APIs like POX [12] is allowed, nor is application-level access to network resources like that provided by PANE [5].

In the next section, we describe a finer grain approach to treating conflicts among operational objectives. Examples are given for controlling power and load balancing. We show how the approach can also be used to control a single device, namely a MAC-learning switch. Then we discuss tradeoffs of the approach in Section 5, followed by related and future work in Sections 6 and 7.

2. A FINER GRAIN STRATEGY

We propose a controller function be modeled as a deterministic finite-state transducer (DFT) [8]. A DFT is a primitive computing machine that runs on inputs, computed by an SDN controller, and outputs a sequence of instructions to change the network as needed to meet the function's objective. The inputs are measurements and other parameters of a running network suitably discretized for the machine. They include link utilization, end-to-end bandwidth requirements, flow-to-path assignments, etc. Inputs are supplied once per "cycle" to a machine. The machine transitions accordingly and may produce output that changes network configuration in some way. The minimum period of a cycle is determined by the SDN controller's ability to refresh the measurements during that time. With one transition per cycle, the machine has a crude form of clock that is useful for implementing timeouts.

The advantage DFTs is that one can analyze properties of them and their intersection to understand the scope of interaction between controller functions. Important properties are decidable for these machines such as whether two machines have a nonempty or finite intersection. It is possible to precisely describe the scope of interaction for a given

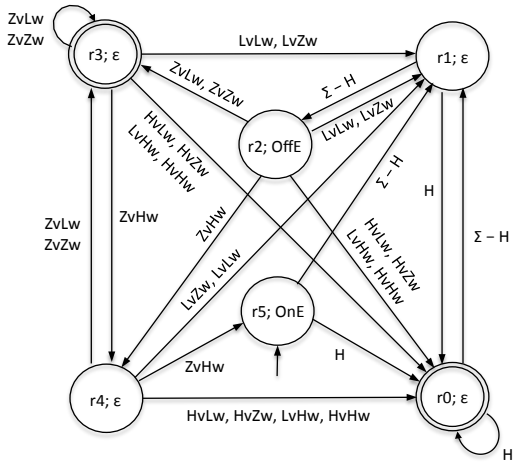

Figure 2: Link-local power-save transducer

network in advance to give an operator a sense of how their network can behave. This comes from analyzing the intersection of DFTs. With proper handling of their output functions, they can be intersected through a standard product construction. An operator then gets a picture of what we call the *stable operating region* of the network under the intersection. This is the set of conditions under which no controller function is attempting to alter the network. Depending on how narrow the region is, a decision can be made about whether to deploy the functions simultaneously. One can experiment with different combinations of controller functions to see the stable regions they produce. Resolving conflicts translates into choosing the right mix of machines or even tuning them in order to achieve an acceptable stable region. Moreover, useful properties of an intersection are obtained for "free" from properties of the constituent machines. As we shall see, the invariant for a state (p, q) of an intersection is obtained by merely conjoining the invariants proven separately for states p and q in their respective machines.

2.1 Modeling a power-save function

Consider the network in Figure 1. Assume each link exhibits a utilization rate that can vary over time. This rate can be sampled on each cycle and is either high (H), low (L) or zero (Z). Suppose a power-save machine tries to power down E when the link utilization of links v and w is low or zero for two consecutive cycles.[1] Any new flows are then routed to F. A DFT runs on a sequence of input symbols. Here each symbol reflects the utilization rates of both links v and w at the end of a cycle. For instance, HvLw is a symbol that denotes a high and low utilization of v and w respectively. Therefore the input alphabet of the machine is $\Sigma = \{\mathsf{Hv}, \mathsf{Lv}, \mathsf{Zv}\} \times \{\mathsf{Hw}, \mathsf{Lw}, \mathsf{Zw}\}$.

The power save transducer for the network in Figure 1 is given in Figure 2; in a slight abuse of notation, H stands for all symbols of Σ containing H. The output function produces the empty output ϵ for all states except r2 and r5 where it outputs OffE and OnE. The machine waits for two consecutive cycles where link w is experiencing heavy load before restoring power to E. The final states r0 and r3 represent

[1]The choice of two cycles is completely arbitrary. It can be whatever an operator wants.

stable operating states for the network where the machine is not in the process of possibly powering down E. For r3, E is powered down with low use of w, and for r0, both routers are powered up with one or both links experiencing high usage. Precisely, the following invariants can be shown for these states by a straightforward mutual induction on the length of input w:

1. the power-save transducer is in r0 on input w with output y only if w ends with a member of H and y ends with OnE, and
2. the power-save transducer is in r3 on input w with output y only if w ends with ZvLw or ZvZw and y ends with OffE.

It may seem curious that transitions are possible out of r2, r3 and r4 with heavy load over v when E is supposedly powered off in those states. The machine has been designed this way because the attempt to turn E off may have failed or been overridden. Thus it treats OffE (and OnE) more as a request than a command. Alternatively, it could be treated as a command and re-tried until it succeeds, a retry maximum is reached, or conditions have changed making it unnecessary. Power status would have to be added to the input stream in this case. So the power-save function can be modeled in many different ways. In fact, the decision to disable a router can push downstream link utilization rates higher. So a link-local approach such as the one described here may be unacceptable. A path-based approach is considered in Section 3.

2.2 Modeling a load-balancing function

Suppose we also want to balance load per destination across links v and w of the network in Figure 1. The load balancing transducer for this network is given in Figure 3. It has the same input alphabet as the power-save transducer; B stands for symbols representing a load-balanced network:

$$\{\mathsf{HvHw}, \mathsf{ZvLw}, \mathsf{LvLw}, \mathsf{LvZw}, \mathsf{ZvZw}\}$$

The only stable state is q0 for which it can be shown that the load balancing transducer is in this state on input w with output y only if w ends with a member of B and y is empty or ends with F to E or E to F; the former means move flows from F to E and the latter from E to F. This machine remains in states q3 and q4 waiting indefinitely for the balance to occur when in practice, a machine would wait until there is a balance or a timeout occurs.

2.3 The intersection transducer

We want to know the stable operating region for the network in Figure 1 under the power-save and load-balancing machines. How narrow is this region? The intersection, or more precisely the product, of the machines is computed for this purpose. A portion of the product is shown in Figure 4. It has a total of 13 reachable states with two final (stable) states, namely r0q0 and r3q0 (not all transitions from r0q0 and r3q0 are shown). From the invariants above, we know that the product is in state r0q0 on input w only if w ends with a member of H ∩ B which is just {HvHw}. This tells us that the product will be in this stable state only when the utilization rates of both v and w are high. It will be in stable state r3q0 only if w ends with ZvLw or ZvZw, that is, when link v is not being used, perhaps because E is off,

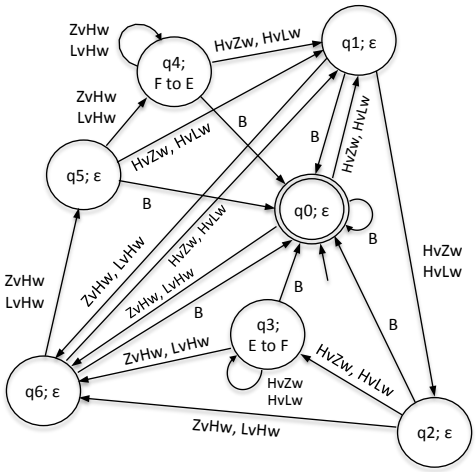

Figure 3: Load balancing transducer

and w is not experiencing high usage. So the stable region for the network is the disjunction of the conditions for these two states.

Note that low utilization of both links can cause oscillation since low, yet nonzero, usage of both links is not part of the stable operating region produced by the product. To see this, suppose powering off router E causes high utilization of link w. Then starting from the start state r5q0, state r1q0 is revisited on input

$$\mathsf{LvLw\ LvLw\ ZvHw\ ZvHw\ ZvHw\ LvLw}$$

All but the first symbol of this input will repeat indefinitely if both links have low usage whenever both routers are on (see red path in Figure 4). It is a subtle consequence of the machines interacting of which an operator should be aware.

Now suppose an operator is later asked to provide redundancy in the network by ensuring both routers are always on. On the surface, this certainly appears to be at odds with power saving but by how much? Will the network be completely inoperable because of these competing concerns? It may not because the answer depends on how the network normally operates. All we can safely say is that the stable operating region will likely be narrowed. To illustrate, consider the redundancy transducer in Figure 5. There's only

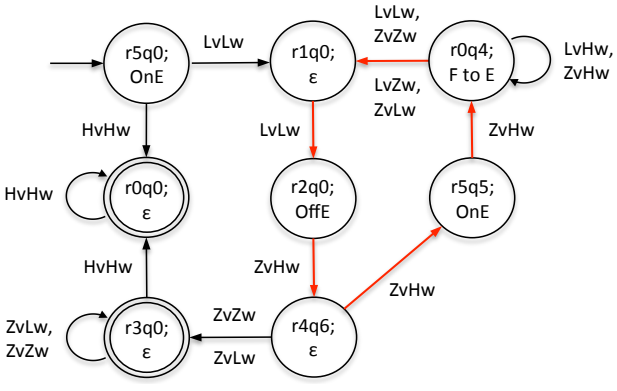

Figure 4: A portion of the product machine for the power save and load balancing transducers

one stable state, s0, and it requires nonzero load on both links to remain in it. So the product of all three machines has only one stable state, namely r0s0q0, and it requires high utilization of links v and w to stay there. This condition may be too narrow to be a useful stable region. However, the point here is that we discovered there is a network condition under which all three machines can co-exist without causing any oscillation.

3. PATH-BASED POWER SAVING

The power-save transducer in the preceding section works at the granularity of links. Alternatively, we can use transducers to model controller functions at the granularity of forwarding paths, for instance, if the network deploys an MPLS like traffic engineering solution. We sketch here one approach to a path-based transducer for power saving. Like ElastricTree [9], it treats path capacities and flows in its decision to alter power.

Consider the network in Figure 6. There are three possible paths to the destination, P1, P2 and P3. Every flow in the network is assigned to a path and consumes a portion of the available bandwidth for that path. As long as all flows can be serviced by P1 without forcing any flow's consumption below what it expects, router E can go dormant.

Inputs to the path-based transducer contain three elements: the current flow-path assignments, the bandwidth each flow is currently experiencing and the utilization of each path. To represent the latter two, we assume the network is instrumented to produce totally-ordered symbols Z, L and H. In the case of bandwidth, each is a range in which a flow's bandwidth lies during a cycle. The ranges may vary across flows because what might be a high range for one may be low for another depending on their service-level agreements. There is another set of ranges for utilization which reflect level of path usage in a cycle. For example, below is an input symbol for the network in Figure 6 with two flows:

$$(\mathsf{P3}, \mathsf{P1},\ \ \mathsf{L}, \mathsf{H},\ \ \mathsf{Z}, \mathsf{Z}, \mathsf{L})$$

The first two elements convey flow-path assignments, the next two convey current bandwidths experienced by each flow, and finally the utilization of each path.

A portion of the transducer is shown in Figure 7 for two flows f1 and f2. The transition to powering up E is made when both flows occupy P1, f1 is experiencing bandwidth in its lowest range and there's insufficient capacity on P1 to improve it (utilization there is H). The transducer moves flow f2 to P3 after powering up E (a choice that in practice would be based on P3's capacity and f2's expected service level). Powering down E is triggered by P3 meeting f2's expected service level (f2's bandwidth is high) without using significant capacity (P3's utilization is Z). Both flows then

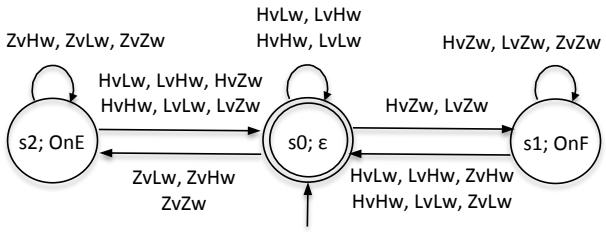

Figure 5: Redundancy transducer

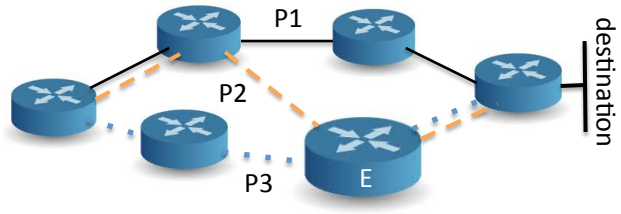

Figure 6: Router E consumes more power

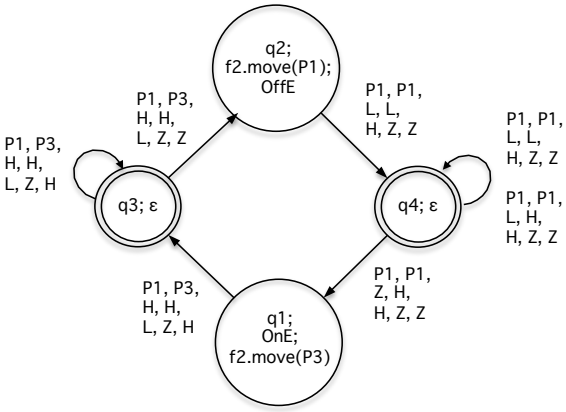

Figure 7: Path-based power-save transducer

occupy P1 again. Note that unlike our previous examples, there is no hysteresis implemented by this transducer. A more practical machine would implement a more elaborate backoff scheme before touching E.

4. A MAC-LEARNING SWITCH

DFTs can also be used as building blocks for device-specific control functions. Their product then gives the desired control function and also reveals a stable region for the device with respect to that function.

For example, a MAC-learning switch handler is given in [3]. It learns the input port for each non-broadcast source MAC address. If the destination port is known, the handler installs a forwarding rule and instructs the switch to send the packet by that rule, otherwise flood it. The control of flooding can be designed completely apart from that of forwarding. The former is about when to flood and the latter about when to update the location of a given MAC address.

Suppose that a given destination MAC address M can be located at port 2 or 3 of a switch. We define an input alphabet $\{\mathsf{nil}, 2, 3\}$ where on a given cycle, nil is signaled by the switch if M was not detected as the source address of any frame received at either port, 2 is signaled if M was detected at port 2, and so on. The transducer to control forwarding is given in Figure 8. In state q_1, instruction FWD 2 is output, indicating that frames destined for M should be forwarded to port 2. There are three stable states, two for when M continues to reside behind one of the two ports, and q_0 for when M's destination port remains unknown. The transducer to control flooding is given in Figure 9. It waits three consecutive cycles without seeing traffic from M before issuing a flood instruction directing the switch to flood frames destined for M. Their product is given in Figure 10.

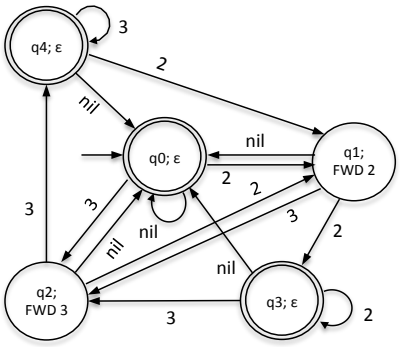

Figure 8: Forwarding transducer

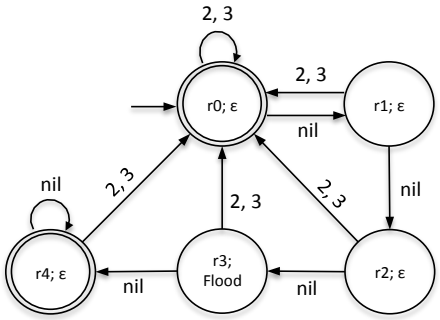

Figure 9: Flooding transducer

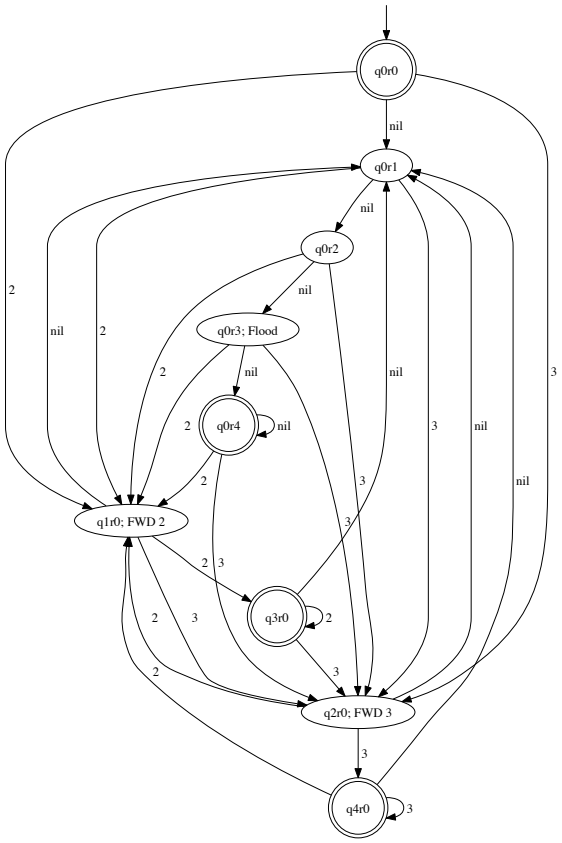

Figure 10: Transducer for MAC-learning switch

5. DISCUSSION

The SDN controller programming paradigm proposed in this paper is a major departure from current practice. Transducers replace controller applications written in say Python or C++. A controller platform provides a measurement API and runtime system for transducers. The API provides the input streams for the transducers, which in turn use another API to issue instructions for updating the network. There are no event handlers, other than the transducers, sending instructions. So what are the pros and cons of using DFTs?

Some expressiveness is obviously lost when controlling a network using a DFT versus a Python application. A DFT is limited to recognizing network conditions that can be defined as regular sets. This rules out conditions that require unbounded counting. For example, the condition that the total number m of ingress frames at a switch equals its egress count n at the end of every cycle is not regular, as m and n are unbounded. Bounds would make it regular but even practical bounds would likely make a DFT impractical here since it would be exponentially large in the length of its binary inputs. More experience is needed to determine how extensive regular conditions are in practice.

The biggest advantage of DFTs over Python and C++ code is the ability to decide problems like emptiness of intersection (empty implies no stable region). This problem isn't even semi-decidable for Python or C++ since one can express in them a parser for any context-free set. Another benefit of DFTs is that it's much easier to prove properties about them as they admit straightforward inductive proofs. Reasoning about a Python program is far more complicated. Canini et al. [3] describe a system for model checking applications written in Python or C++ for the NOX controller platform [6]. These programs can perform arbitrary computation and maintain state so techniques like symbolic execution of them are used to reveal relevant inputs in an attempt to reduce the size of the state space.

Scalability of DFTs will depend on the measurement API and the instruction API which will have to provide support for high-level operations like graceful switch shutdown and transfers of flows from one path to another. Product machines should scale well. The product of two DFTs with state sets Q_1 and Q_2 and alphabet Σ can be constructed in worst-case time $|Q_1| \times |Q_2| \times |\Sigma|$ and will not have more than $|Q_1| \times |Q_2|$ states. While more experience is needed, we suspect that in practice the product will usually have far fewer reachable states. For example, the product of the power save and load balancing transducers (a portion of which is shown in Figure 4) has only 13 reachable states out of 42 (6×7) potentially reachable states.

6. RELATED WORK

Corybantic [16] is perhaps the most closely related work as was discussed earlier. Operational objectives in Corybantic are not formulated in a way that permits their interaction to be analyzed in any rigorous way. Other efforts have formulated objectives more formally, thus facilitating detection and resolution of conflicts between them automatically. Examples include FML [10], VeriFlow [14] and NetPlumber [13] among others. FML is a relational language for specifying flow policy in which policy conflicts are resolved automatically. Veriflow and NetPlumber check in real time for com-

pliance of flow rule updates. These efforts concern flow-level objectives based on policy or properties like absence of loops and black-holes, reachability, and avoiding/enforcing waypoints. None of them though can detect oscillation caused by satisfying multiple objectives since oscillation is not a violation of flow policy.

PANE [5] provides an API for end users to request special network treatment for certain flows, including access control, bandwidth reservation, path control and rate-limits. PANE resolves conflicts in different users' requests by consulting pre-defined policies regarding their privileges. Expressing operational objectives, like power conservation and load balancing, and their interaction as discussed in this paper, are outside the scope of PANE.

7. FUTURE WORK

A key design question is what events transducers will be allowed to observe and how often? A measurement API is needed that allows a transducer to register for events of interest. Model checking may help in designing transducer input alphabets. Relevant inputs, derived by symbolic execution of existing Python controller code, exercise different control paths through the code [3]. That would seem to make them good candidates for input symbols, especially if the transducer mirrors the logic of the code. An input symbol might be associated with many concrete inputs, each of which could be used by the transducer in outputs only.

Initially we expect transducers will continue to be designed by hand per network, as was done in this paper, using logic distilled from today's controller modules as a guide. The open source python-automata project [11] has provided a good code base for tools to intersect and analyze deterministic transducers. But more automated support is needed. Ideally, there would be a compiler for each control function that would take network information relevant to meeting the control function's objective and output a transducer for that objective and the given network.

8. CONCLUSION

Network stability in the face of diverse control and optimization mechanisms is becoming a challenge. More precise techniques are needed to take the guesswork out of determining how these mechanisms impact stability. This paper introduces one such technique. It gives operators a better basis for customizing control functions per network. As was shown in Section 2.3, the stable operating region may turn out to be too narrow to be useful when combining controller functions. It is imperative that an operator be able to see how mixing different functions can impact their network, as the impact can vary under the same set of functions.

9. ACKNOWLEDGMENTS

Early discussions with Franck Le about network oscillation helped inspire this work. We would like to thank him and also the reviewers for their helpful comments.

10. REFERENCES

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proceedings of USENIX NSDI*, 2010.

[2] H. Ballani, P. Costa, T. Karagiannis, and A. I. Rowstron. Towards Predictable Datacenter Networks. In *Proceedings of ACM SIGCOMM*, 2011.

[3] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *Proceedings of USENIX NSDI*, 2012.

[4] M. Chiosi, D. Clarke, and et al. Network Functions Virtualisation – Introductory White Paper. In *SDN and OpenFlow World Congress*, 2012.

[5] A. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *Proceedings of ACM SIGCOMM*, 2013.

[6] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: Towards an Operating System for Networks. *SIGCOMM Computer Comm Rev*, 38:105–110, 2008.

[7] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *Proceedings of ACM CoNEXT*, 2010.

[8] E. Gurari. *An Introduction to the Theory of Computation*. Computer Science Press, 1989.

[9] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. Elastictree: Saving Energy in Data Center Networks. In *Proceedings of USENIX NSDI*, 2010.

[10] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical Declarative Network Management. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, Barcelona, Spain, 2009.

[11] https://code.google.com/p/python-automata.

[12] http://www.noxrepo.org/pox/about-pox/.

[13] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proceedings of USENIX NSDI*, 2013.

[14] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Proceedings of USENIX NSDI*, pages 15–27, 2013.

[15] L. Liu, H. Wang, X. Liu, X. Jin, W. B. He, Q. B. Wang, and Y. Chen. Greencloud: A New Architecture for Green Data Center. In *Proc. 6th ACM Int'l Conf on Autonomic Computing and Communications Industry Session*, 2009.

[16] J. A. Mogul, A. AuYoung, S. Banerjee, L. Popa, J. Lee, J. Muidgonda, P. Sharma, and Y. Turner. Corybantic: Towards the Modular Composition of SDN Control Programs. In *Proc. of ACM Workshop on Hot Topics in Networking*, 2013.

[17] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes. Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. In *Proceedings of USENIX WIOV*, 2011.