A Verification Platform for SDN-Enabled Applications

Richard Skowyra rskowyra@bu.edu
Andrei Lapets lapets@bu.edu
Azer Bestavros best@bu.edu
Assaf Kfoury kfoury@bu.edu

Computer Science Department
Boston University

***Abstract*—Recent work on integration of SDNs with application-layer systems like Hadoop has created a class of system, SDN-Enabled Applications, which implement application-specific functionality on the network layer by exposing network monitoring and control semantics to application developers. This requires domain-specific knowledge to correctly reason about network behavior and properties, as the SDN is now tightly coupled to the larger system. Existing tools for SDN verification and analysis are insufficiently expressive to capture this composition of network and domain models. Unfortunately, it is exactly this kind of automated reasoning and verification that is necessary to develop robust SDN-enabled applications for real-world systems.**

In this paper, we present ongoing work on Verificare, a verification platform being built to enable formal verification of SDNs as components of a larger domain-specific system. SLA, safety, and security requirements can selected from a variety of formal libraries and automatically verified using a variety of off-the-shelf tools. This approach not only extends the flexibility of existing SDN verification systems, but can actually provide more fine-grained analysis of possible network states due to extra information supplied by the domain model.

I. Introduction

The emerging field of Software-Defined Networking (SDN) has enabled programmatic, adaptive networks to be deployed across a variety of application domains. Google has deployed inter-datacenter traffic shaping and engineering networks which use the OpenFlow SDN architecture, and ongoing research efforts have contributed to its use in datacenter networking, Hadoop-based High-Performance Computing [17], and high-availability MTCP networks [19].Despite this breadth of applications that SDNs have serviced, they have largely remained constrained to semi-automated management and control of the network layer.

Yet many of these management functions are motivated by optimizing the performance of application-layer services such as high-performance computing or IaaS software stacks. This optimization is largely indirect: many SDNs control network conditions either based on a network-centric heuristic about how those conditions impact the application layer (*e.g.* latency or loss thresholds), or by inferring future application-layer traffic based on current network conditions. Google's traffic shaping system, for example, is designed to rapidly recover from link failures in order to support loss-intolerant applications. MicroTE [3], an OpenFlow-based datacenter networking architecture, attempts to minimize congestion by predicting application layer traffic via statistical inference over network traffic matrices. While this indirect optimization has certainly proved useful, we posit that much more fine-grained control is possible by exposing applications to network monitoring and control semantics directly via an API-like interface. Some APIs of this type have in fact recently been developed [6]. These SDN-Enabled applications implement application-specific functionality on the network layer (e.g. in switch routing data structures), and can reconfigure the network with full knowledge of their behavior, resource needs, and performance constraints.

This tighter coupling of the application and network layers does not come without cost. The rich suite of tools which have developed to support SDN development and analysis, which range from network debuggers [9] to programming languages [7], [20], [13] and model checkers [4], are insufficiently expressive to reason about the composition of the SDN with an application. The *ndb* tool, for example, is a debugging system for SDNs which implements standard debugger semantics (breakpoints, backtrace, etc.) over SDN controllers [9]. This is useful, but if the controller exports an API to SDN-enabled programs then significant network control state remains outside of its diagnostic view. Similarly, the NICE model checker is designed to check for certain common network errors arising from, among other problems, concurrency issues [4]. NICE uses SDN controller source code to construct a state space of 'interesting' packets designed to exercise all portions of controller code. It assumes that all controller events occur as a result of packets received from the network switches, and has no facility for encoding out-of-band communications like RPCs from SDN-enabled applications.

Unfortunately, these kinds of tools are exactly what developers of SDN-enabled applications need. The composed system is complex, and automated reasoning tools are required to verify that the application meets the designer's requirements while not violating any network-specific requirements. Evaluation of each component in isolation of the other does not suffice: many of these requirements may be defined over the application (*e.g.*

frames per second) but are impacted by network behaviors (*e.g.* latency, jitter) in complex ways.

In this paper we present ongoing work on Verificare, a verification platform for SDN-enabled applications. Verificare addresses the lack of expressive power in existing SDN-related tools by providing a general component-based modeling environment in which domain- and application-specific modules can be written by a developer and composed with pre-defined network mechanisms like OpenFlow switches and controllers, using an API-like syntax. A library of common requirements and assumptions is available to the user, and can be augmented by user-specified requirements. These properties are formally verified by existing off-the-shelf verification tools, and counter-examples are returned as traces leading up to the violation.

The remainder of this paper is laid out as follows. §II provides a motivating example for the problem. §III presents the Verificare Modeling Language (VML), and §IV discusses how high-level models can be compiled into formal structures. Reusable requirements are described in §V, and §VI discusses how off-the-shelf tools can be used to verify system properties.

II. Motivation and Related Work

Among networked systems, those supporting interactive sessions or communications-bound processing are particularly impacted by changing network conditions. These include streaming media services, VoIP and video conferencing sessions, online gaming servers, and HPC and data analytics applications like Hadoop. This sensitivity to network conditions motivates their development as SDN-Enabled applications which utilize a network monitoring and control API, enabling the application itself to optimize the network for its use.

This paradigm works well as long as the application in question is the only process manipulating the network control plane, or the network is dedicated to servicing that particular application. In multi-tenant environments, however, this is rarely the case.

As a motivating example, consider a network of online gaming servers deployed in a cloud setting. The quality of service (frame rate and responsiveness, for example) experienced by users playing the hosted game is directly impacted by network conditions like latency and congestion. If these server applications are SDN-Enabled, they can easily distribute incoming connections amongst themselves depending on their current application load. Rather than relying on an application-layer solution like a proxy, dynamically updated load balancing rules can be directly installed in network switches in order to transparently redirect traffic to the least-utilized server [21].

Of course, servers which are reachable from the the outside world (as gaming servers must be) have to consider issues of software security. A recent SDN-based defense to worms, port scanners, and other vulnerability detection systems is OpenFlow Random Host Mutation (OF-RHM) [12]. This moving target defense uses OpenFlow's packet header rewriting facility and switch-based flow rules to maintain a rapidly mutating set of IP addresses per host, ensuring that any network reconnaissance against a particular IP are only valid for a short time. As an OpenFlow controller it could be easily deployed on the gaming server network in order to reinforce security without requiring any modifications to end-hosts.

It is unclear how these two SDN applications compose, however. Will OF-RHM impair application-level QoS? Are the semantics of user connection and servicing compatible with the assumptions OF-RHM makes about incoming connections?

A. Existing Approaches

The authors are aware of two ways to address these questions using existing tools. First, the suite of existing SDN analysis and verification tools can be employed.

NICE [4] is a model checker designed specifically for OpenFlow controllers, and can perform rapid verification with respect to network-centric properties like the presence of forwarding loops. While a powerful tool for reasoning about complex network states, NICE focuses solely OpenFlow-based SDN controllers. This makes modeling of OF-RHM straightforward, but there is no facility for including a model of another process interacting with the controller. Furthermore, the set of requirements that can be checked by NICE is limited to those which can be expressed as Python code snippets defined over controller state. If critical system state resides in the application instead, it is not clear how this could be checked.

The ndb [9] debugger is equally network-centric. It provides a gdb-like facility for setting breakpoints in SDN controller code and allowing manual inspection of network state leading up to an exception (anomalous packet). While useful, it cannot capture state which it outside of the controller on the application layer.

Mininet [16] is a network emulator which allows rapid prototyping of SDNs. In this environment, end-hosts which emulate realistic workloads can be deployed and measured. SDN-Enabled applications can be translated with less difficulty into a Mininet-like system, but Mininet suffers from the same issue that any simulation or emulation based approach does: it tends to capture average-case behavior and offers no guarantee that edge-cases have been explored during its run.

A second approach is to use existing formal verification tools like Spin [10], PRISM [15], SAL [5], or Alloy [11]. All of these are general-purpose verification tools which perform a form of statespace search in order to find reachable model states that correspond to counterexamples of assertions or requirements. They are quite expressive, in that almost any system of communicating processes can be modeled. However, each tool is limited to verifying requirements expressible in specific logics (*e.g.* LTL, FOL, PCTL*, etc.). Since real world systems rarely have requirements that fit nicely into a single logical specification language, it is often necessary to write a new version of the model for each tool needed to check system requirements.

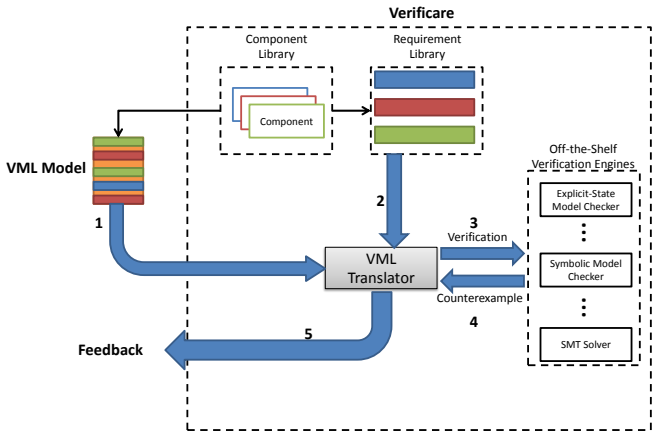

Fig. 1. Verificare Components and Workflow

B. *Verificare*

The Verificare system (depicted schematically in Figure 1) offers a third approach, designed to encapsulate domain-specific knowledge and automatically compose individual components into a complete model. Under this paradigm, an application developer would write a model of their SDN-enabled gaming server in VML (the tool's modeling language) which manipulates the network using an API exported by the SDN controller component (Step 1 above). Concurrently, a network administrator would create a model of the network topology and of the modified OF-RHM OpenFlow controller. Verificare can compose each component into a single model of the resultant network. Important QoS and safety properties that the application developer requires can then be either selected from on-board, parameterized libraries of common requirements, or defined by the developer (Step 2). Verificare will formally verify these properties over the model (Step 3) using a variety of existing tools (like Spin, PRISM, and Alloy), and translate any counter-examples found by these tools back into statements about the VML model (Step 4). These are then output to the user (Step 5).

Verificare has three primary components, illustrated in Figure 1 and described in detail in the following sections: a simple modeling language, VML; a library of parameterized, reusable formal requirements which can be automatically checked over any model built using its associated abstraction; and a series of translators to and from off-the-shelf formal verification tools.

III. Verificare Modeling Language

The Verificare modeling language is a simple imperative language designed to easily capture the behavior of event-based systems composed of communicating processes. Models written in VML have two parts: a list of component definitions describing system dynamics, and a configuration which describes the static structure of the system.

Component definitions introduce a new composable module (which may have multiple instances) into the system. This module, once defined a single time, can be re-used in any other model which includes the API methods (if any) that the module uses. Verificare already includes definitions for common networking primitives (e.g. queues, message-passing networks) and SDN constructs (e.g. OpenFlow switches and basic controllers).

```
agent end_host:
...
loop:
  select:
    pkts_in_flight < 5:
      pkt_out.payload=msg
      q_out.put(pkt_out)
      pkts_in_flight = pkts_in_flight + 1
    not q_in.empty():
      pkt_in = in.get()
      if in.payload==’ack:
        pkts_in_flight = pkts_in_flight-1
      if in.payload == ’msg’:
        pkt_out.payload=ack
        q_out.put(pkt_out)
```

Fig. 2. Simple VML agent definition

Components in VML can be one of two types: *agents*, active asynchronous processes which initiate component interactions (e.g. clients and servers); and *environments*, passive collections of state and methods which mediate interactions between agents (e.g. queues and networks). Agents interact by invoking API methods on a shared environment (e.g. writing and reading a shared 'memory' environment). The binding of agents to environments establishes the data- and control-flow between system components, and is assigned during system configuration. Note that since agents only communicate over environments, changing the implementation of one agent prompts no changes to other components as long as it uses the same set of method invocations as before. Similarly, environment implementations can be changed as long as they export the same set of methods to agents.

As an ongoing example, Figure III presents the VML agent definition for a simple end-host that sends packets as long as fewer than five packets are un-acknowledged, and acknowledges packets that it receives. Packets are sent and received over separate queue environments, the `q_in` and `q_out` objects. Each queue has a `put` and `get` method that the end-host agents use to send and receive packets, respectively. For brevity, variable declarations are omitted.

Once all components have been defined, the system configuration is used to specify how they compose. A configuration consists of one cardinality constraint on components, and zero or more formulas of first-order logic which constrain the binding of agents to environments. Figure III is the configuration for a simple system in which two of the end-hosts defined above communicate with one another via shared queue environments. In this case, the queue components used by each end-host are constrained such that the `in` queue instance of one end host will be the same queue as the `out` queue of the other.

Verificare uses a SAT solver to find an static system

```
configuration:
  for 2 end_host, 2 queue
  end_host[0].q_in == end_host[1].q_out
  end_host[0].q_.out == end_host[1].q_in
```

Fig. 3. Simple VML configuration

```
agent foo:
 int x = [0..2]
 loop:
   select:
     x < 2: x = x+1
     x == 2:x = 0
     x > 0: x = x-1
```

Fig. 4. Sample VML model for conversion to an LTS

state which meets all of the constraints. This allows under-specification in cases where a single initial state isn't important or must only be from some class of configurations (e.g. cycle-free single-homed networks of any topology). If no satisfying initial state can be found, (e.g. due to error or over-specification) Verificare informs the user and aborts the verification run.

IV. Automatic Model Composition

Once all components have been defined and an initial state has been found, Verificare compiles the system model to a Labeled Transition System (LTS) $M = (S, A, T, I)$, where:

- S is a set of states;
- A is a set of actions;
- $T \subseteq S \times A \times S$ is a ternary transition relation;
- I is a set of initial states.

Labeled Transition Systems are a well-supported formalism for verification of concurrent systems and communicating processes [8], [15], [10] with a variety of well-developed techniques for composition and exploration [2]. Verificare creates a distinct LTS for every component in the model. The state space per component represents all possible states of local variables and the instruction pointer for a component. Transitions denote control flow branching or method invocation, and are labeled with one of four *composition types*, discussed below.

As brief an example of translation from VML to an LTS, consider the extremely simple VML model depicted in Figure 4. In this model, a bounded integer, `x`, is initialized to 0 and then nondeterministically modified. The associated LTS, which is generated automatically from the model code, is shown in Figure 5. Edge labels correspond to control flow statements and to modifications to local variable state. States correspond to unique configurations of the component with respect to the value of its local variables and its position in its control flow. Once constructed, the LTS can be formally verified with respect to a variety of logical properties, as described in Section VI.

Once every component has been compiled to a Labeled Transition System, it is necessary to compose these into a single LTS which captures all valid system states, and which can be verified with respect to a set of requirements. Recall that VML distinguishes between active agents and passive

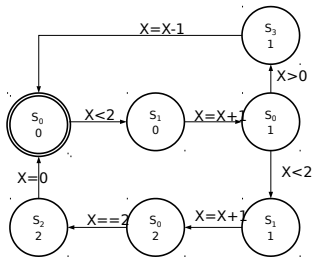

Fig. 5. Sample Model LTS

environments, however. Any sound composition operation must enforce these semantics (e.g. a queue should not have a packet unless an agent invoked `put`). To enforce this constraint, Verificare computes the *asymmetric product* of the *asynchronous product* of the environment and agents LTS, as shown in 1.

$$\prod_{0}^{n}(A_0, ...A_n) \oslash \prod_{0}^{m}(E_0, ...E_m) \tag{1}$$

The asynchronous product $M = \prod(N_0, ..N_n)$ of a set of LTS is a well-known, generic (e.g. label-agnostic) composition operator which computes the full interleaving of all possible orderings of the component LTS transitions. It is defined as the LTS M such that $S = N_0.S \times ... \times N_n.S$, $I = N_0.I \times ... \times N_n.I$, $A = N_0.A \times ... \times N_n.A$, and $M.T = \{(x_i, a, y_i) | a \in M.A \wedge \exists i (0 \leq i \leq n), \forall j (0 \leq j \leq n), j \neq i \Rightarrow x_j \equiv y_j\}$. That is, the LTS in which each state transition represents a single state transition in one of the component LTS.

The asymmetric product is an operator unique to Verificare. It computes the LTS created by composing the single asynchronous agent LTS A with the single asynchronous environment LTS E, such that either some agent transitions internally (i.e control flow branching) or an agent and an environment bound to it transition simultaneously (i.e. method invocation). The operator makes use of composition type labels written during compilation from VML to an LTS, which are one of four types: $L_0 : \phi$, which denotes a logical formula acting as a pre-condition to that transition; $L_1 : (id, c_0, ...c_n)$, which denotes invocation of the method id with constant values $c_0 ... c_n$; $L_2 : (id, \phi(v_0, ..., v_n))$, which denotes an invokable method id with pre-condition ϕ that has free variables $v_0, ...v_n$; and $L_3 : [v_0 \Rightarrow c_0, ..., v_n \Rightarrow c_n]\phi$, which denotes substitution of the free variables $v_0, ..v_n$ in ϕ with the constant values $c_0, ...c_n$.

Using these composition types, the asymmetric product $M = A \oslash E$ is defined as:

- $M.S = A.S \times E.S$
- $M.I = A.I \times E.I$
- $M.A = A.A \cup E.A \cup sig(A.A, E.A)$
- $M.T = \{(a_h e_i, l, a_j e_k) | e_i \equiv e_k \wedge l : L_0 \vee$
  $(\exists m : L_1(a_h, m, a_j) \in A.T \wedge$
  $\exists n : L_2(e_i, n, e_k) \in E.T \wedge$
  $match(m, n) \wedge l = sig(m, n))\}$

where $match(L_1, L_2) \rightarrow \{true, false\}$ compares the id fields of the two arguments and $sig(L_1, L_2) \rightarrow L_3$ consumes an invoker and invokable type label, and returns the substitution label with all constants from the first argument substituted into the free variables of the second. This operation constrains environments to transition only when an agent invokes a method in that environment, which can significantly reduce the number of edges in the composed LTS.

V. Specifying Requirements

Once all components of a system have been defined, requirements to be checked can be selected. To check a requirement, in this case, is defined as verifying that all requirements hold (or not) over all reachable, valid system states. In order to perform this verification automatically, requirements must be translated to one or more formulas in an appropriate formal logic.

Unfortunately, in most formal settings the resultant requirement formula is inextricably coupled to model syntax (*e.g.*, `local_state` $==$ `3` $\wedge$ `some_array`[`5`] $==$ `true`). This is especially unfortunate for high-level properties which are broadly applicable to a domain, such as a statement about network forwarding loops or packet losses. Furthermore, small changes to the same model often necessitate re-writing of requirements (consider the above example, if the array is re-ordered or resized). In fact we have observed certain properties in real-world models, generally based on statements about combinations of model elements (*e.g.*, $(p_1 \wedge p_2) \vee (p_1 \wedge p_3) \vee (p_2 \wedge p_3)$, which require quadratic amounts of re-writing when key model parameters change [18], [14]. This re-writing is mechanical, but quite tedious to do manually. Needing to re-define the same high-level property repeatedly, both over different related models and over different versions of the same model, also creates the possibility of errors in requirement specification.

Ideally, the model of a system and the requirements to be checked should be completely independent, *i.e.* definable in isolation, in any order, with no knowledge of one another. This would eliminate the troubling coupling of model and requirement seen above and enable the re-use of requirements across models and over different versions of the same model.

Verificare provides this independence by defining *reusable requirements* over agents and environments. Since components communicate only method invocations, component source code can be written once and used in many other system models. Requirements specified over this source code are similarly portable, as models should by definition be agnostic to the internal implementation of a component. This enables researchers to define and check new requirements over old models, import properties defined by others, and cleanly partition modeling a system from verifying its requirements.

In order to maximize the utility of reusable requirements, we are compiling libraries of pre-defined requirements and properties for common components used in SDNs that are already provided by Verificare, such as queues and OpenFlow switches. Each entry in these libraries consists of a logical formula and an English-language version of the requirement. Users need only to select requirements of interest to have them verified against their model. Existing requirements include reachability, packet dropping and duplication, network black-holes, structural and logical forwarding loops. These libraries are in active development, however, and are being updated regularly.

VI. Verification

Once a VML model has been compiled to a LTS and a set of requirements has been chosen, formal verification can be used to search for counter-examples: model states in which requirements are not met or some failure scenario is possible. The precise mechanism used to perform this search varies somewhat depending on the underlying formalism used (*e.g.*, automata, binary decision diagrams, or logical resolution), but in general formal verification amounts to an algorithmic search through a finite, bounded state space of reachable model configurations.

This approach is lightweight compared to other formal methods like theorem-proving and proof assistants, in that it requires little initial work and verification is accomplished quickly, but it sacrifices completeness due to its bounded nature. While incomplete verification is problematic when used to prove that a property holds in all cases, it is quite useful at the design phase of a system in order to quickly discard designs where a requirement demonstrably does not hold.[1]

The majority of existing formal verification systems utilize a single underlying formalism optimized for a particular verification technique, and generally support a requirement specification logic that is compatible with that formalism. SPIN, for example, utilizes composed Buchi automata and supports Linear Temporal Logic (LTL). Formulas in LTL can be directly translated to a Buchi automaton, which can be automatically composed with the user's model to create a state space for verification. Unfortunately, real-world systems have requirements that span many domains of interest (*e.g.*, safety, performance, security), and can rarely be fully specified using a single formal language. This often requires developers to re-implement the same model multiple times in different modeling languages, drastically increasing the effort of using such tools.

Verificare overcomes this limitation by the model LTS and a subset of its requirements to check into multiple formalisms and uses different off-the-shelf verification tools to check system requirements over the model. In general, any verification tool whose underlying formalism is at least as expressive as an LTS can host a translator. New tools can be added simply by writing a translation plugin, making Verificare an extensible interface to a variety of formal tools. We have currently

[1]The small-scope hypothesis [1] posits that this incompleteness is less problematic than it would first appear. Significant empirical evidence indicates that most real-world bugs (as opposed to maliciously designed flaws) which are detected in large-scale, time-intensive verification, are also present at much smaller, more easily verified scales.

implemented translators to SPIN, PRISM, and Alloy, and are developing translators for NICE and Mininet. If any of these tools find a requirement violation, their output is automatically translated back to statements about the high-level VML model.

Translation to multiple formalisms is clearly necessary because no one verifier can handle all formal logics. This introduces complexities, however, as the subset of requirements that any one tool can check together define a less restricted state space than the complete set of requirements does. If at least one violation of requirements is reachable in one verifier (that is, there is at least one actual bug), then false positives may manifest in other verifiers due to exploration of state space coordinates reachable only due to the (undetected in that verifier) bug.

VII. Conclusion

In this paper, we argued that SDN-Enabled applications, which have access to network monitoring and control semantics through an API-like interface, require new methods to assure their safety, security, and performance. The Verificare tool is designed to provide this assurance via formal verification of composed application designs and SDN system models. Unlike SDN-verification tools which focus on the network in isolation, Verificare enables compositional modeling of both the SDN and the applications which depend on it. Requirements spanning dimensions of safety, security, and reliability (among others) can selected from a variety of formal libraries, and are automatically verified using a variety of off-the-shelf tools.

Acknowledgment: This work was supported by NSF CISE CNS Award #1239021 and #1012798

References

[1] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov. Evaluating the "small scope hypothesis". *Unpublished*, 2003.

[2] C. Baier, J.-P. Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[3] T. Benson, A. Anand, A. Akella, and M. Zhang. Microte: fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, page 8. ACM, 2011.

[4] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE way to test OpenFlow applications. In *NSDI*, 2012.

[5] L. De Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. Sal 2. In *Computer Aided Verification*, pages 496–500. Springer, 2004.

[6] A. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *Proc. ACM SIGCOMM '13*, Hong Kong, China, August 2013.

[7] N. Foster, R. Harrison, and M. Freedman. Frenetic: A network programming language. *ACM SIGPLAN Notices*, 46(9):279–291, 2011.

[8] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. Cadp 2010: a toolbox for the construction and analysis of distributed processes. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 372–387, 2011.

[9] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. Where is the debugger for my software-defined network? In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 55–60. ACM, 2012.

[10] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, 2005.

[11] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, Apr. 2002.

[12] J. H. Jafarian, E. Al-Shaer, and Q. Duan. Openflow random host mutation: transparent moving target defense using software defined networking. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 127–132. ACM, 2012.

[13] N. P. Katta, J. Rexford, and D. Walker. Logic Programming for Software-Defined Networks. In *XLDI*, 2012.

[14] M. Kwiatkowska, G. Norman, and D. Parker. Analysis of a gossip protocol in prism. *ACM SIGMETRICS Performance Evaluation Review*, 36(3):17–22, 2008.

[15] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *23rd International Conference on Computer Aided Verification*, pages 585–591. Springer, 2011.

[16] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[17] S. Narayan, S. Bailey, A. Daga, M. Greenway, R. Grossman, A. Heath, and R. Powell. Openflow enabled hadoop over local and wide area clusters. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1625–1628. IEEE, 2012.

[18] PRISM. Randomised mutual exclusion: Pnueli and zuck.

[19] R. van der Pol, S. Boele, F. Dijkstra, A. Barczyk, G. van Malenstein, J. H. Chen, and J. Mambretti. Multipathing with mptcp and openflow. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1617–1624. IEEE, 2012.

[20] A. Voellmy, H. Kim, and N. Feamster. Procera: a language for high-level reactive network control. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 43–48. ACM, 2012.

[21] R. Wang, D. Butnariu, and J. Rexford. Openflow-based server load balancing gone wild. In *Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services*, pages 12–12. USENIX Association, 2011.