Automated Synthesis of Reactive Controllers for Software-Defined Networks

Anduo Wang Salar Moarref Boon Thau Loo
Department of Computer and Information Science
University of Pennsylvania
{anduo,moarref,boonloo}@seas.upenn.edu

Ufuk Topcu
Department of Electrical and Systems Engineering
University of Pennsylvania
utopcu@seas.upenn.edu

Andre Scedrov
Department of Mathematics
University of Pennsylvania
scedrov@math.upenn.edu

***Abstract*—With the tremendous growth of the Internet and the emerging software-defined networks, there is an increasing need for rigorous and scalable network management methods and tool support. This paper proposes a synthesis approach for managing software-defined networks. We formulate the construction of network control logic as a reactive synthesis problem which is solvable with existing synthesis tools. The key idea is to synthesize a strategy that manages control logic in response to network changes while satisfying some network-wide specification. Finally, we investigate network abstractions for scalability. For large networks, instead of synthesizing control logic directly, we use its abstraction—a smaller network that simulates its behavior—for synthesis, and then implement the synthesized control on the original network while preserving the correctness. By using the so-called simulation relations, we also prove the soundness of this abstraction-based synthesis approach.**

I. Introduction

The past two decades witnessed tremendous growth in networking systems such as the Internet, and rapid expansion in applications domains such as clouds, datacenters; and yet ever-increasing demands for more reliable and affordable networking services. On the other hand, the programmable router paradigm and the flourishing software-defined network (SDN) platform [2] have made network management even less tractable.

However the practice of network management has remains a primitive low-level process, forcing the network operators to reason at switch level, worrying about large amount of details [7], [8]. Existing management techniques are largely restricted to management friendly protocols and data-plane primitives, such as specific mechanism for routing, signaling, QoS and virtualization. Moreover, the low-level reasoning is made even less applicable with the sheer size of the networks, thanks to the rapid expanding of new applications such as cloud networks and datacenters [6].

This primitive state of management practice and insufficient tool support have become an increasing larger source for network problems such as misconfigurations and performance degradation [1], [3]. There has been an increasing need for rigorous and scalable management techniques. This paper proposes an abstraction-based reactive synthesis approach towards automatic, provably correct, and scalable network management for software-defined networks.

978-1-4799-1270-4/13/$31.00 ©2013 IEEE

To achieve scalable and rigorous management automation, we need to solve two separate problems. (1) How can we automatically enforce or realize a control logic over a complex data-plane; and (2) How can we automatically construct the right control logic, given a network-wide request (property). Most network management primitives in current platforms concern the first problem, enabling network operators to realize a given control logic in various forms [9]. The second problem of *"control logic construction"* , however, has attracted less attention, and is the problem addressed in this paper.

To address the construction of control logic in a network that is contently changing, we propose a synthesis approach, which seeks to find a strategy that manages the control logic in response to network requests. By utilizing state of the art synthesis tools and model checking researches in managing autonomous systems [19], [18], [12]. We are able to automatically construct control logic management strategy. In addition to automation, added benefit is that, by formal synthesis, we always produce a provably correct management solution. Therefore, unlike existing empirical or heuristic-based management techniques, our approach does not require the evaluation step, this has additional practical value due to the lack of evaluation benchmark in current network management research [3].

Finally, to scale up *provably correct automation*, we investigate abstraction technique. Intuitively, a network's abstraction is a smaller network that simulates its behavior and preserves its properties. In abstraction-based synthesis, a large control construction problem is decomposed into a smaller problem on the abstract network, and the implementation of the synthesized control in the original network. With simulation, we also proved that the abstraction-based synthesis is sound: the network property of the synthesized controller in the abstract network is guaranteed to hold for the original network.

II. Control Logic Construction

In this section, we describe the control logic construction problem, and present a working example, which will be used in the rest of the paper.

A software-defined network is a graph of connected switches installed with operational rules (flow tables). The flow tables constitute the network's data plane. A network operator controls the network by managing the flow tables. In the control logic construction problem, given a set of

requests and invariants, the goal is to construct a flow table that achieves the requests while preserving the invariants. The network request specifies the intended network behavior and network state change, e.g. what flows can transit the network and when a switch leaves or joins the network; The network invariant specifies the constraints imposed by the network, e.g. the bandwidth constraints and switch capacities. We further assume that, over the time, the network is presented with a series of independent requests, but the invariant constraint stays relatively stable.

Informally, we view the construction of control logic a reactive synthesis problem: find a strategy that dynamically manages (e.g. updates) a network's control plane in response to any network request while satisfying the network invariant.

Consider the software-defined network shown in Figure 1 (left) that consists of three switches I, F_1, F_2. There are four types of flows (groups of network traffic): faculty (F), student (S), untrusted (U), and guest (G) flows. The operational rules (flow tables) are depicted alongside each switch. Each flow table rule is of the form of *Type: Action*. The actions includes forwarding ($Forwardto*$) and access-control ($Monitor, Deny, Allow$). All flows enter the network by I, and are then forwarded to $F_{1,2}$ based on the international devices (omitted in the graph) they are destined for. The network requests are routing path re-allocation (e.g. for traffic balancing reasons), given by the forwarding rule (rules with forwarding actions) changes. The invariant is a security policy that denies untrusted flows while allowing others.

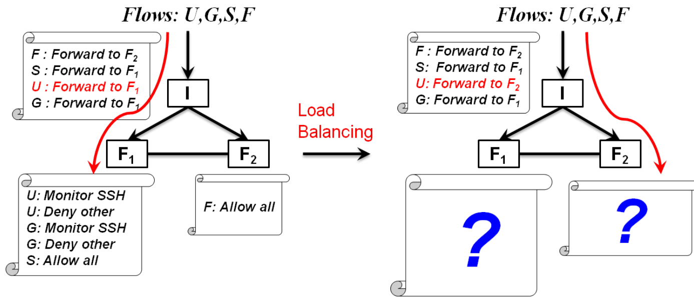

Fig. 1. Example control logic construction: When routing path changes, update the access-control rules.

The control logic construction problem is to find a strategy for updating access-control rules in the flow tables in response to the forwarding rule changes while maintaining the security policy. For example, in Figure 1 (right), when routing path for U flow changes from path 1 to 2, given by the highlighted forwarding rule changes, the strategy shall tell how to update the access-control rules accordingly (question marks).

In Section III, we present an automatic construction of a strategy on this small example through formal synthesis. To scale up this approach, we will discuss abstraction techniques in Section IV.

III. Problem Formulation

In this section we introduce the necessary background and terminology and formalize the problem.

A. Problem Statement

Definition 1 (Software-defined network): A network system includes a set of switches N. Each switch $n \in N$ is associated with a set R_n of rules of the form $\mathrm{type} : \mathrm{action}$.

The set R_n represents the flow table for node n, meaning that for a flow matching *type*, n applies *action*. The installed operational rules for the network Net is defined by $R = \bigcup_{n \in N} R_n$. Let $TYPE = \{t_1, t_2, ..., t_k\}$ be the set of all possible flow types in the network. We denote the action corresponding to the switch n and flow type t_i with a_n^i. These actions form the set of system *variables*. (more on type of actions, like deny and forwarding)

Definition 2 (Network state): A system consists of a set $\mathcal{V}$ of variables. The domain of $\mathcal{V}$, denoted by $dom(\mathcal{V})$ is the set of valuations of $\mathcal{V}$. A state of the system is an element $v \in dom(\mathcal{V})$.

For a switch n, the action corresponding to a flow type t_i may or may not be controllable. Therefore, we partition the variables into two subsets $\mathcal{E}$ and $\mathcal{R}$, representing uncontrollable (environment) and controllable variables respectively. We write $\mathcal{V} = (\mathcal{E}, \mathcal{R})$. The $\mathcal{R}$ variables are the controllable parts of the system through which network management is implemented. We call $\mathcal{E}$ "uncontrollable" variables because they are not part of the control logic we are synthesizing. Note that, though called "uncontrollable", the $\mathcal{E}$ variables may encode network states that change either unexpected or as planned by some logic not handled by a SDN controller.

Definition 3 (Network transition system): The transition system for a network is a tuple $(\mathcal{V}_0, \mathcal{V}, \mathcal{T})$, $\mathcal{V}_0 \subseteq \mathcal{V}$ is the initial state of the network, $\mathcal{T} \subseteq \mathcal{V} \times \mathcal{V}$ is the transition relation.

A network execution (or trace) is a sequences of states $\sigma = v_0 v_1 \cdots$ where $v_0 \in \mathcal{V}_0$, for any $i > 0, v_i \in \mathcal{V}, (v_i, v_{i+1} \in \mathcal{T})$.

Definition 4 (Network property): A network wide property is a linear temporal logic (LTL) statement on the network transition system.

This paper uses linear temporal logic (*LTL*) as the specification language for its expressiveness power and existing game solvers. A LTL statement is either a proposition (predicate) p over network state v, denoted $p(v)$; or a composite statement built from LTL connectives: logic connectives such as negation$\neg$, conjunction $\wedge$; and temporal modal connectives such as next ($\bigcirc$), eventually ($\Diamond$), always ($\Box$).

A network property (LTL formula) is interpreted over a sequence of network states. For example, given a network execution (trace) $\sigma = v_0 v_1 v_2 \cdots$ and a property φ, $\bigcirc\varphi$ holds for v_i in σ, if φ holds for all v_{i+1}; $\Box\varphi$ holds for v_i if it holds for all $v_j, j > i$; and $\Diamond\varphi$ holds for v_i, if there exists a $v_j, j > i$, φ holds for v_j.

The examples presented here are sufficient for the rest of the paper. For a complete description of LTL semantics, interesting readers are referred to [14].

A *two-player deterministic game graph* is a tuple $\mathcal{G} = (Q, Q_0, E)$ where Q can be partitioned into two disjoint sets Q_1 and Q_2. Q_1 and Q_2 are the sets of states of player 1 and 2, respectively. Q_0 is the set of initial states. $E = Q \times Q$ is

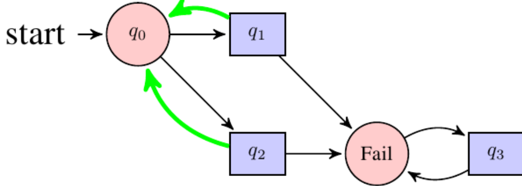

Fig. 2. An example of a game graph and a winning strategy for player 2 (red edges)

the set of edges. Players take turn to play the game. At each step, if the current state belongs to Q_1, player 1 chooses the next state. Otherwise player 2 makes a move. A *play* of the game graph $\mathcal{G}$ is an infinite sequence $\sigma = q_0 q_1 q_2 ...$ of states such that $q_0 \in Q_0$, and $(q_i, q_{i+1}) \in E$ for all $i \geq 0$. We denote the set of all plays by Π. A *strategy* for player $i \in \{1, 2\}$ is a function $\alpha_i : Q^*.Q_i \rightarrow Q$ that chooses the next state given a finite sequence of states which ends at a player i state. Given strategies α_1 and α_2 for players and a state $q \in Q$, the *outcome* is the play starting at q, and evolved according to α_1 and α_2. Formally, $outcome(q, \alpha_1, \alpha_2) = q_0 q_1 q_2 ...$ where $q_0 = q$, and for all $i \geq 0$ we have $q_{i+1} = \alpha_1(q_0 q_1 ... q_i)$ if $q_i \in Q_1$ and $q_{i+1} = \alpha_2(q_0 q_1 ... q_i)$ if $q_i \in Q_2$. An *objective* for a player is a set $\Phi \subseteq \Pi$ of winning plays. A strategy α_1 for player 1 is winning for some state q if for every strategy α_2 of player 2, we have $outcome(q, \alpha_1, \alpha_2) \in \Phi$.

The realizability problem for LTL formulas is known to be 2EXPTIME-complete [16]. However, recently, Piterman et al. show that the realizability and synthesis problems for a fragment of LTL known as Generalized Reactivity(1) (GR(1)) [13] can be solved efficiently in polynomial time.

Figure 2 shows an example of a game graph. Circle (box) nodes represents player 1 (2) states, respectively. The game starts at state q_0 and depending on the transition chosen by player 1, the next state is either q_1 or q_2 which are player 2 states. Assume that the objective for player 2 is to prevent the game from reaching the $Fail$ state. A winning strategy for player 2 is to choose the transition to q_0 from both q_1 and q_2 states (green edges). It is easy to see that using this strategy the game never reaches the state $Fail$, regardless of how player 1 plays.

B. Example synthesis

We now describe the synthesis of control logic for Figure 1, using TuLiP tool [18], a specification tool that interfaces existing game solver. Figure 1 can be viewed as a game between route forwarding and access-control. The forwarding-rule player represents network requirement for newly selected route, and manipulates the uncontrollable variables $\mathcal{E}$ that denotes forwarding rule actions. In response, the system player represents access-control rules, and need to adjusts the system variables $\mathcal{R}$, i.e., updating access-control rule. The task is then to find a winning strategy for access-control to pick the right updates for access control such that security policy is never violated. More specifically:

- **Environment variables** $\mathcal{E} = \{a^U_{I_F}\}$ denote the forwarding rule for untrusted flows (U) at switch I. The domain $dom(\mathcal{E}) = \{I, F_1, F_2\}$ denotes the forwarding action of sending flows through next hop switches I, F_1, F_2 respectively.
- **System variables** $\mathcal{R} = \{a^U_{I_{ac}}, a^U_{F_{1ac}}, a^U_{F_{2ac}}\}$ denote the access control for untrusted flows at switch I, F_1 and F_2. The domain $dom(\mathcal{R}) = \{deny, allow\}$ denotes the two access control actions of deny or allow.

For simplicity, we only consider untrusted flows here, and omit variables (rules) for the other flows. Without confusion, we use a_* for a^U_*. Using these variables, we can specify the initial network state v_0, transition relation $\mathcal{T}$, and the network invariant φ_s, as follows.

In the initial state v_0, we have $a_{I_f} = 1, a_{I_{ac}}, a_{F_{1ac}}, a_{F_{2ac}} = deny$, that is, the flows will take route through path 1, and all access controls are set to deny. The transition relation $\mathcal{T}$ specifies the evolution of the variables by one move of either environment or system player. For system player, a move is given by the concurrent transitions for the three variables $a_{I_{ac}}, a_{F_{1ac}}, a_{F_{2ac}}$. For example, transition for a_{f1ac} are $\{(deny, deny), (deny, allow), (allow, allow), (allow, deny)\}$, that is, at each move, the controller may either flip the access control or keep it unchanged. Similarly, we have transition relations for $a_{I_{ac}}, a_{F_{2ac}}$. For environment player, the transition is given by all possible moves of a_{I_F}. We write the transition of each move by $(vo, v'o')$ where $v = a_{I_F} a_{I_{ac}} a_{F_{1ac}} a_{F_{2ac}}$ and $o = a_{reach}$ a new output variable denoting the reachability of untrusted flows. One possible transition is of the form $(a_{I_F} a_{I_{ac}} a_{F_{1ac}} \cdots, \cdots 0)$ meaning that if access control rules are all set to deny as long the path of the flow, the flow will not be reachable. Using output variable, we define the network invariant φ_s of security policy as $\Box a_{reach} = 0$, that is, untrusted flows are not reachable.

Given the specification of $(\mathcal{E}, \mathcal{R}), \mathcal{T}, \varphi_s$, TuLiP generates a winning strategy for $\mathcal{E}$ that always satisfies φ_s, as shown by the automaton in Figure 3.

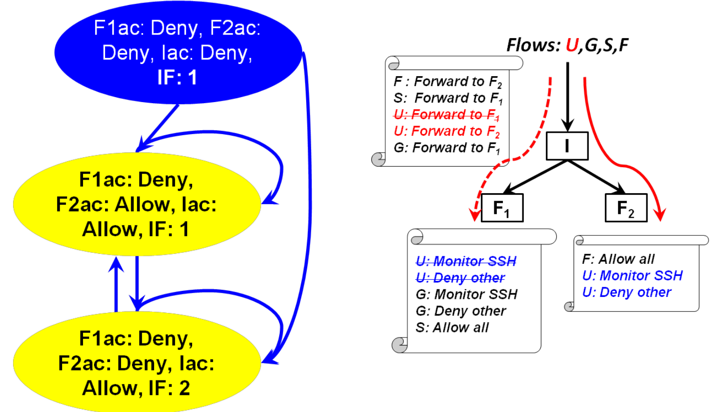

Fig. 3. Example control logic construction: A strategy that updates access-control rules when forwarding changes

In the automaton, each state transition corresponds to a move (v, v'), i.e. $((e, r), (e', r'))$ initiated by forwarding rule change (e, e'), followed by the move of access-control rule (r, r'). For example, from state 1 to 2 says, when forwarding rule changes from $a_{I_F} = F_1$ to $a_{I_F} = F_2$, the access-control player picks the action that sets $a_{F_{2ac}} = deny$. Thus, by simply

reading the automaton, we can set the missing access-control rules in Figure 1.

Finally, given a strategy and the forwarding rule change (e, e'), to install the corresponding access control update (r, r'), we need to implement r' over the network configured with r. The difficulty is that r' may differ from r in multiple nodes, while the rule is inserted at one node at a time. Thus it may take multiple steps to enforce (r, r'), and it is critical to ensure φ_s during all the transient network steps. A possible approach is to find a invariant-preserving ordering of the update steps. Such an ordering can be synthesized by solving a reachability problem using model checker. It is actually a special case of the two-player temporal logic game. Alternative approaches such as heuristics and additional mechanism to enforce invariants are discussed in previous works [5], [15].

IV. Scaling by Abstraction

In networking community, network abstractions have long been investigated for scalability [10], [9], [11], [4]. In control logic synthesis, we also face the state-explosion problem. When network size increases, the system state rapidly grows and the resulting strategy to be synthesized quickly becoming impractical. To mitigate this problem, we propose to use network abstractions. Instead of synthesizing a large strategy directly, we introduce an abstract network, and construct the strategy for this smaller representation instead. The abstract solution is then mapped back to the original network for implementation.

This section first uses an example to introduce network abstraction-based synthesis. Then we formally define network abstraction through simulation. Simulation allows us to transfer the network property to the abstract network.

A. Abstraction example

Consider the software-defined network in Figure 4 (left) consists of 12 switches, where the possible network flows are depicted by the directed arrows. Consider access-control updates problem when the forwarding rule changes. The network invariant is to prevent untrusted flows from traversing the network.

This example is relatively straightforward and constructed for the presentation of the main ideas behind abstraction-based synthesis procedure. Note that it presents a larger but same control logic construction problem as in Figure 1. For example, when the forwarding rule changes and causes flows to route from path 1 to 2, we want to update the access-control rules in the rest of the network accordingly. On the other hand, the network state is larger. The system state of Figure 1 involves 5 (1 forwarding rule, 3 access-control rules, 1 dependent) variables, whereas here, we need $14((1 + 12 + 1))$ variables.

An abstraction for Figure 4 (left) is shown in Figure 4 (middle), where the nodes are grouped into 3 abstract nodes. The control update problem on the smaller abstract representation is solved exactly as in Figure 1. The solution is then implemented on the the original network, as shown in Figure 4 (right). The idea of "implementation" is as follows: when routing path changes from 1 to 2, shown in Figure 4, the strategy synthesized on the abstract network gives us the

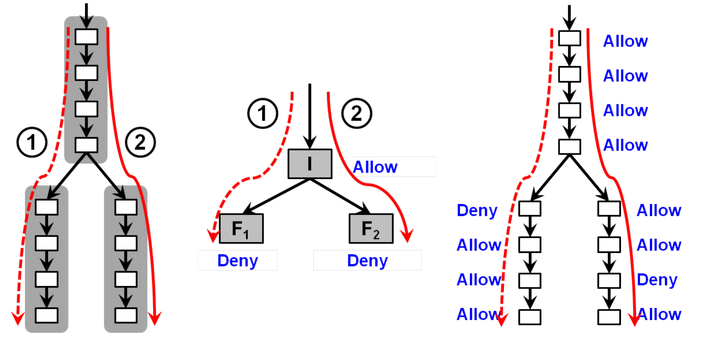

Fig. 4. Abstraction scales up synthesis: synthesize on abstraction representation (middle), implement abstract solution on original network (right)

action to be applied to the access-control rule, that is, the new access-control rules. These "abstract" access-control rule values, shown in the middle, are then mapped back to the original network. For example, the `Deny` access-control rule for F_2 (middle) is mapped to four separate access-control rules whose joint effect is deny (right).

B. Network abstraction

We have constructed an "abstract network" for Figure 4 (left), by grouping nodes for synthesis purpose. We now develop the concept of "abstract network" as a network system that simulates the original network and preserves the property for synthesis. To this end, we introduce a few relevant concepts.

A transition system with observation $(\mathcal{V}_0, \mathcal{V}, \mathcal{T}, \mathcal{O}, H)$ extends the transition system $(\mathcal{V}_0, \mathcal{V}, \mathcal{T})$ (Definition 3) by including a output function $H : \mathcal{V} \to O$ that maps the system states into observable output $\mathcal{O}$. The output variable $\mathcal{O}$ specifies the network behaviors relevant in synthesis, that is, network property is temporal property defined over $\mathcal{O}$. As seen later, it plays a key role in constructing abstractions, different $\mathcal{O}$ on the same network can lead to different abstractions.

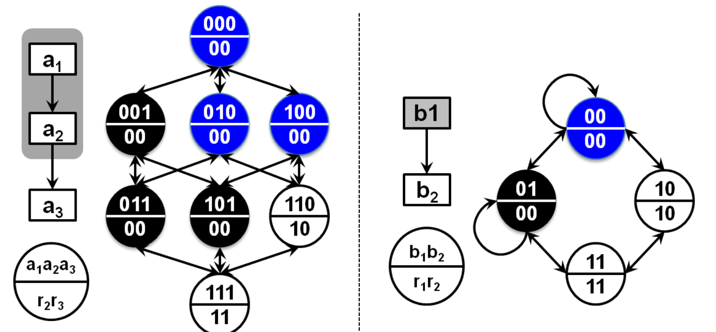

Fig. 5. An abstract network (right) simulates the original network (left).

For example, Figure 5 (left) depicts a transition system with observation for a network of three switches a_1,a_2, and a_3. In the network, we are only interested in the access-control rules, so the system state consists of just three bits, denoted as $a_1a_2a_3$, encoding the access-control rule (0 for deny, 1 for allow). The output variables are $r_{2,3}$, encoding the reachability property of flows for $a_{2,3}$ respectively. We write $a_1a_2a_3r_2r_2$

to denote the state, and in the graphical representation, we separate a_1, a_2, a_3 and r_1, r_2 by a bar. For example, in the state marked with 00100, a_1, a_2 are set to deny and a_3 allow, as a result, neither r_2 nor r_3 can be reached, so they are both 0. Finally, the transition relation says, for any $(v, v') \in \mathcal{T}$. The states v, v' differ by only one bit. That is, we only consider state transition with exactly one access-control rule change. In the rest of the paper, we will simply call such transition system with observation "system".

Definition 5 (Simulation relation): Let $S_a = (\mathcal{V}_{a0}, \mathcal{V}_a, \mathcal{T}_a, \mathcal{O}_a, H_a)$ and $S_b = (\mathcal{V}_{b0}, \mathcal{V}_b, \mathcal{T}_b, \mathcal{O}_b, H_b)$ be two systems. A relation $R \subseteq \mathcal{V}_a \times \mathcal{V}_b$ is a simulation relation from $\mathcal{V}_a$ to $\mathcal{V}_b$ if

- for every $v_{ao} \in \mathcal{V}_{ao}$ there exists a $v_{b0} \in \mathcal{V}_{bo}$,
- for every $(v_a, v_b) \in R$, $H_a(v_a) = H_b(v_b)$, and
- for every $(v_a, v'_a) \in \mathcal{T}_a$, $(v_a, v_b) \in R$, there exists some v'_b satisfying $(v_b, v'_b) \in \mathcal{T}_b$ and $(v'_a, v'_b) \in R$.

If there exists such a simulation relation, we say S_b simulates S_a. It is not hard to check that in Figure 5, the right transition system for a network of switches b_1, b_2 simulates the left transition system of switches a_1, a_2, a_3. Under relation R, states maps to the same one are depicted with the same color. For example, all nodes in black (left) are mapped to state with variables 01 and output variables 00. Note that R on the transition system corresponds to the mapping of a_1, a_2 to b_1 and a_3 to b_2 on the network. We call the network of b_1, b_2 an *abstraction* of the network of a_1, a_2, a_3, since it simulates the observable behavior in terms of output variables.

To better illustrate how the output variables affect network abstractions, Figure 6 depicts more examples. On the left, given the observable variable r_3, denoting reachability for switch a_3, there exists an abstraction by grouping nodes a_1 and a_2. To prove this to be a valid abstraction, we only need to check the two transition systems accordingly to Definition 5. Similarly, we can prove that the network in the middle has an abstraction by just grouping a_2 and a_3. Note that, even though the network is the same as in Figure 5 (left), the abstraction is different due to the difference in observable variables.

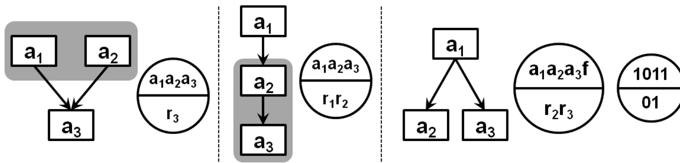

Fig. 6. More abstraction examples: The network properties determine the abstract network (left, middle); A network that has no abstraction (right).

On the other hand, Figure 6 (right) gives an network that has no abstraction, that is, there does not exist an network of few nodes that simulates its behavior. Note that the transition system has a state with output variables 01 meaning that while r_3 is reachable r_2 is not. However, such output variables never occurs in a two-nodes network's transition system. That is, there does not exist a network of fewer than 3 nodes that simulate its behavior. Hence the network has no abstraction.

C. Synthesis through abstraction

The purpose of network abstraction is to reduce to the control synthesis problem of a large network to that of an abstract smaller network. We now prove that this approach is sound. By sound, we mean the property synthesized for the abstract network also holds for the original network. To achieve soundness, We utilize a key property of simulation, proved in [17], as follows.

Proposition 1: Let $S_a = (\mathcal{V}_{a0}, \mathcal{V}_a, \mathcal{T}_a, \mathcal{O}_a, H_a)$ and $S_b = (\mathcal{V}_{b0}, \mathcal{V}_b, \mathcal{T}_b, \mathcal{O}_b, H_b)$ be two systems, S_a is simulated by S_b. Let φ be a LTL property over the output variables $\mathcal{O}_a$. Then, we have S_b satisfies φ implies S_a satisfies φ.

As illustrated in Figure 7, for a large physical network, instead of synthesizing control logic directly on it. We perform synthesis on its smaller network abstraction, which simulates the physical network. Proposition 1 ensures the network property φ synthesized on the abstract network also holds for the original larger one. Hence, our abstraction-based synthesis is sound, and network abstraction enables us to scale up control logic construction for large networks.

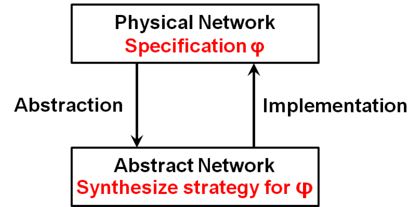

Fig. 7. Synthesis through abstraction: perform formal synthesis on the abstract network; implement the synthesized solution on the physical network

For examples, in Figure 4, the property synthesized on the abstraction (middle) is that the security policy (always block untrusted flows) is always enforced. Proposition 1 guarantees that the winning strategy we computed and implemented on the original network (right) conforms to the same security policy.

V. Acknowledgment

This research is partly supported by NSF grants CNS-0845552, the NSF Expeditions in Computer Augmented Program Engineering (ExCAPE) project ITR-1138996, AFOSR Young Investigator Award FA9550-12-1-0327, and AFOSR grant FA9550-12-1-0302.

References

[1] Al-Shaer, E., Greenberg, A., Kalmanek, C., Maltz, D. A., Ng, T. S. E., and Xie, G. G. New frontiers in internet network management. *SIGCOMM Comput. Commun. Rev. 39*, 5 (Oct. 2009), 37–39.

[2] Casado, M., Freedman, M. J., Pettit, J., Luo, J., McKeown, N., and Shenker, S. Ethane: taking control of the enterprise. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2007), SIGCOMM '07, ACM, pp. 1–12.

[3] Cerf, V., Davie, B., Greenberg, A., Landau, S., and Sincoskie, D. FIND Observer Panel Report. April 2009.

[4] Drutskoy, D., Keller, E., and Rexford, J. Scalable network virtualization in software-defined networks. *IEEE Internet Computing 17*, 2 (2013), 20–27.

[5] GHORBANI, S., AND CAESAR, M. Walk the line: consistent network updates with bandwidth guarantees. In *Proceedings of the first workshop on Hot topics in software defined networks* (New York, NY, USA, 2012), HotSDN '12, ACM, pp. 67–72.

[6] ISARD, M. Autopilot: Automatic Data Center Management. *SIGOPS Oper. Syst. Rev. 41*, 2 (2007), 60–67.

[7] KANG, N., LIU, Z., REXFORD, J., AND WALKER, D. An Efficient Distributed Implementation of One Big Switch. Open Networking Summit (ONS), Research Track, 2013.

[8] KELLER, E., AND REXFORD, J. The "Platform as a service" model for networking. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking* (Berkeley, CA, USA, 2010), INM/WREN'10, USENIX Association, pp. 4–4.

[9] KOPONEN, T., CASADO, M., GUDE, N., STRIBLING, J., POUTIEVSKI, L., ZHU, M., RAMANATHAN, R., IWATA, Y., INOUE, H., HAMA, T., AND SHENKER, S. Onix: a distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–6.

[10] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., AND WALKER, D. Composing Software-Defined Networks. Proc. Networked Systems Design and Implementation, April 2013.

[11] MUDIGONDA, J., YALAGANDULA, P., MOGUL, J., STIEKES, B., AND POUFFARY, Y. Netlord: a scalable multi-tenant network architecture for virtualized datacenters. In *Proceedings of the ACM SIGCOMM 2011 conference* (New York, NY, USA, 2011), SIGCOMM '11, ACM, pp. 62–73.

[12] OZAY, N., TOPCU, U., AND MURRAY, R. Distributed power allocation for vehicle management systems. In *Decision and Control and European Control Conference (CDC-ECC), 2011 50th IEEE Conference on* (2011), pp. 4841 –4848.

[13] PITERMAN, N., PNUELI, A., AND SA'AR, Y. Synthesis of reactive (1) designs. In *Verification, Model Checking, and Abstract Interpretation* (2006), Springer, pp. 364–380.

[14] PNUELI, A. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science* (Washington, DC, USA, 1977), IEEE Computer Society, pp. 46–57.

[15] REITBLATT, M., FOSTER, N., REXFORD, J., SCHLESINGER, C., AND WALKER, D. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 323–334.

[16] ROSNER, R. Modular synthesis of reactive systems. *Ann Arbor* (1991).

[17] TABUADA, P. *Verification and Control of Hybrid Systems: A Symbolic Approach*. Springer, 2009.

[18] WONGPIROMSARN, T., TOPCU, U., OZAY, N., XU, H., AND MURRAY, R. M. Tulip: a software toolbox for receding horizon temporal logic planning. In *Proceedings of the 14th international conference on Hybrid systems: computation and control* (New York, NY, USA, 2011), HSCC '11, ACM, pp. 313–314.

[19] XU, H., TOPCU, U., AND MURRAY, R. A case study on reactive protocols for aircraft electric power distribution. In *Decision and Control (CDC), 2012 IEEE 51st Annual Conference on* (2012), pp. 1124–1129.