Synchronization Synthesis for Network Programs

CAV 2017, Submission #77

Abstract. In software-defined networking (SDN), a controller program updates the forwarding rules of a network in response to events. Such programs are often physically distributed, running on different nodes of the network. Their distributed nature makes these programs difficult to write and debug. Furthermore, bugs in these programs can lead to serious problems such as packet loss and security violations. In this paper, we propose a program synthesis approach that makes it easier to write distributed controller programs. The programmer can specify each sequential process, and add a declarative specification of paths that packets are allowed to take. The synthesizer then inserts enough synchronization between the distributed controller processes such that the declarative specification is satisfied by all packets that traverse the network. Our key technical contribution is a counterexample-guided synthesis algorithm that adds synchronization between controller processes to prevent races that would cause a specification violation. Our programming model is based on Petri nets, and generalizes several models from the networking literature. Importantly, our programs can be implemented in a way that prevents races between updates to individual switches and in-flight packets. To our knowledge, this is the first counterexample-guided technique that automatically adds synchronization constructs to Petri-net based programs. We demonstrate that our prototype implementation can fix realistic concurrency bugs described previously in the literature, and that our tool can handle real network topologies.

1 Introduction

Software-defined networking (SDN) enables programmers to more easily implement important applications such as traffic engineering, distributed firewalls, and network virtualization. These applications react to network events—such as topology changes, shifts in traffic load, or arrival of packets at switches. SDN provides the abstraction of a *controller machine* that manages forwarding rules installed on individual switches. The application programmer can write code which runs on the controller, enabling custom event-driven behavior.

Concurrency in Network Programs. Network control is often physically distributed, with controller processes running on multiple network nodes [28, 14]. The fact that these distributed programs control a network which is *itself* a distributed packet-forwarding system means that these programs can be especially difficult to write and debug. In particular, there are two types of races that can occur, resulting in incorrect behavior. First, there are races between updates of forwarding rules at individual nodes, or between packets that are in-flight during updates. Second, there are races among the different processes of the distributed controller. We call races of the first type *packet races*, and races of the second

type *controller races*. Bugs resulting from these types of races can lead to serious problems such as packet loss and security violations.

Illustrative Example. Let us examine the difficulties of writing distributed controller programs, in regards to the two types of races. Consider the network topology in Figure 1(a). In the initial configuration, packets entering at $H1$ are forwarded through $S1, S5, S2$ to $H2$. There are two controllers (not shown), $C1$ and $C2$—controller $C1$ manages the upper part of the network $H1, S1, S5, S3, H3$, and $C2$ manages $H2, S2, S5, S4, H4$. Now imagine that the network operator wants to take down the forwarding rules that send packets from $H1$ to $H2$, and instead install rules to forward packets from $H3$ to $H4$. Furthermore, the operator wants to ensure that the following property P holds: *all packets entering the network through* $H1$ *must exit at* $H2$. When developing the program to do this, the network operator must keep the following problems in mind:

- Packet race: If $C1$ removes the rule that forwards from $S1$ to $S5$ before removing the rule at $H1$, then a packet entering at $H1$ will be dropped at $S1$, violating specification P.
- Controller race: Suppose $C1$ only removes the rule that forwards from $S3$ to $S5$, and suppose $C2$ adds rules that forward from $S5$ to $S4$, and from $S4$ to $H4$. Then a packet entering at $H1$ will exit at $H4$, violating specification P.

Goal. We present a program synthesis approach that makes it easier to write distributed controller programs. The programmer can specify each sequential process, and add a declarative specification of paths that packets are allowed to take. The synthesizer then inserts enough synchronization between the controller processes such that the declarative specification is satisfied by all packets traversing the network. In effect, our approach allows the programmer to reduce the amount of effort spent on keeping track of possible interleavings of controller processes and inserting low-level synchronization constructs, and instead focus on writing a declarative specification which describes allowed packet paths. In the examples we considered, we found these declarative specifications to be a clear and easy way to write the desired correctness properties.

Network Programming Model. In our approach, similar to network programming languages like Kinetic [26] and OpenState [6], we allow a network program to be described as a set of concurrently-operating finite state machines (FSMs) consisting of event-driven transitions between global network states. We generalize this by allowing the input network program to be a set of *event nets*, which are 1-safe Petri nets where each transition corresponds to a network event, and each place corresponds to a set of forwarding rules. This model extends network event structures [33] to enable modeling programs with loops. An advantage of extending this particular programming model is that its programs can be efficiently implemented without packet races (discussed further in Section 3).

Problem Statement. Our synthesizer has two inputs: (1) a set of event nets that represents sequential processes of the distributed controller, and (2) an LTL specification of paths that packets are allowed to take. For example, the network programmer can specify properties such as "packets from $H1$ should always pass

through Middlebox $S5$ before exiting the network." The output is an event net consisting of the input event nets with added synchronization constructs, such that all packets traversing the network satisfy the LTL specification. The added synchronization eliminates problems caused by controller races. Since we use event nets, which can be implemented without packet races, both types of races are eliminated in the final implementation of the distributed controller.

Algorithm. Our main contribution is a counterexample-guided inductive synthesis (CEGIS) algorithm for event nets. This consists of (1) *repair engine* that synthesizes a candidate event net from the input event nets and a set of counterexample traces, and (2) a *verifier* that checks whether the candidate satisfies the LTL property, producing a counterexample trace if not. The repair engine uses SMT to produce a candidate event net by adding synchronization constructs which ensure that it does not contain the counterexample traces discovered so far. Repairs are chosen from a variety of constructs (barriers, locks, condition variables), and other constructs can be added as needed. Given an event net, the verifier checks whether it is deadlock-free (i.e., there is an execution where all processes can proceed without deadlock), and whether it satisfies the LTL property. We encode this as an LTL model-checking problem—the check fails (and returns a counterexample) if the event net exhibits an incorrect interleaving.

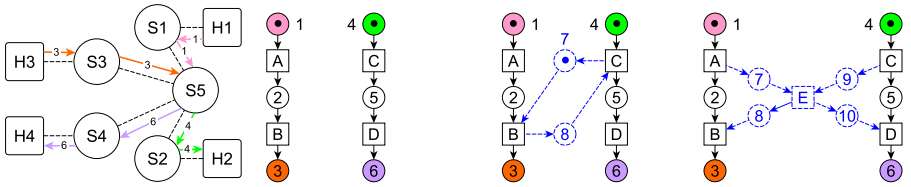
Evaluation. We have implemented our techniques, and evaluated our tool on examples from the SDN literature. We show that our prototype implementation can fix realistic concurrency bugs, and can handle real network topologies.

Contributions. This paper contains the following contributions:

- We describe *event nets*, a new model for representing concurrent network programs, which extends several previous approaches, enables using and reasoning about many synchronization constructs, and admits an efficient distributed implementation (Section 2-3).
- We present *synchronization synthesis for event nets.* To our knowledge, this is the first counterexample-guided technique that automatically adds synchronization constructs to Petri-net based programs. Our solution includes a *model checker for event nets*, and an SMT-based *repair engine for event nets* which can insert a variety of synchronization constructs (Section 4).
- We demonstrate the usefulness and efficiency of our approach through several real-world examples (Section 5).

2 Network Programming using Event Nets

Network programs change global forwarding behavior of the network in response to events. Recently proposed network programming languages such as OpenState [6] and Kinetic [26] allow a network program to be specified as a set of finite state machines, where each state is a static configuration (i.e., a set of forwarding rules at switches), and the transitions are driven by events in the network (packet arrivals, etc.). In this case, support for concurrency is enabled by allowing FSMs to execute in parallel, and any conflicts of the global forwarding state due to concurrency are avoided by either requiring the FSMs to be restricted to *disjoint* types of traffic, or by ignoring conflicts entirely. Neither



(a) Configurations (b) Input Net (c) Iteration 1 (d) Output Net

Fig. 1: Example #1

of these options solves the problem—as we will see here (and in the Evaluation), serious bugs can arise due to unexpected interleavings. Overall, network programming languages typically do not have strong support for handling (and reasoning about) *concurrency*, and this is becoming especially necessary, as SDNs are moving to distributed or multithreaded controllers.

Event Nets for Network Programming. We introduce a new approach which extends the finite-state view of network programming with support for concurrency and synchronization. Our model is called *event nets*, an extension of *1-safe Petri nets*, a well-studied framework for concurrency. An event net is a set of *places* (denoted as circles) which are connected via *directed edges* to *events* (denoted as squares). The current state of the program is indicated by a *marking* which assigns *tokens* to places, and an event can change the current marking by consuming a token from each of its input places and emitting a token to each of its output places. Since event nets model network programs, each place is labeled with a static network configuration, and at any time, the global configuration is taken as the union of the configurations at the marked places.

Figure 1(b) shows an example event net. In this paper, we will use integer IDs (and alternatively, colors) to distinguish static configurations. Figure 1(a) shows the network topology corresponding to this example. In a given topology, the configurations associated with the event net are drawn in the color of the *places* which contain them, and also labeled with the corresponding place IDs. For example, place 3 in Figure 1(b) is orange, and this corresponds to enabling forwarding along the orange path $H3, S3, S5$ (labeled with "3") in the topology (Figure 1(a)). In the initial state of this event net, places $1, 4$ contain a token, meaning forwarding is initially enabled along the red (1) and green (4) paths.

Event Nets and Synchronization. Event nets allow us to specify synchronization easily. In Figure 1(b), we have added places $7, 8$—this makes event C unable to fire initially (since it does not have a token on input place 8), forcing it to wait until event B fires (B consumes a token from places $2, 7$ and emits a token at 8). Ultimately, we will show how these types of *synchronization skeletons* can be produced automatically. In Figure 1(b)-(d), the original event net is shown in black (solid lines), and synchronization produced by our tool is shown in blue (dashed lines). We will now demonstrate by example how our tools works.

Example—Tenant Isolation in a Datacenter. Koponen et al. [27] describe an approach for providing *virtual networks* to *tenants* (users) of a datacenter, allowing them to connect VMs using virtualized networking functionality (middleboxes,

etc.). An important aspect of this is ensuring *isolation* between tenants. One tenant intercepting another tenant's traffic would be a severe security violation.

Let us extend the simple example described in the Introduction. In Figure 1(a), $S5$ is a physical device initially being used as a virtual middlebox processing Tenant X's traffic, which is being sent along the red (1) and green (4) paths. We wish to perform an update in the datacenter which allows Tenant Y to use $S5$, and moves the processing of Tenant X's traffic to a different physical device. Let us assume that for efficiency, two controllers will be used to execute this update—path 1 is taken down and path 3 is brought up by $C1$, and path 4 is taken down and path 6 is brought up by $C2$. The event net for this network program is shown in Figure 1(b). The combinations of configurations $1, 6$ and $4, 3$ both allow traffic to flow between tenants, violating isolation.

We formalize the isolation specification using the following two properties:

1. ϕ_1: *no packet originating at* $H1$ *should arrive at* $H4$, and
2. ϕ_2: *no packet originating at* $H3$ *should arrive at* $H2$.

Properties like these which describe *single-packet traces* can be encoded straightforwardly in linear temporal logic (LTL). Note that instead of LTL, we can use the more user-friendly PDL, or a domain-specific specification language that can be compiled to LTL. Given an LTL specification, we ask a *verifier* whether the event net has any reachable marking whose configuration violates the specification. If so, a *counterexample trace* is provided, i.e., a sequence of events (starting from the initial state) which allows the violation. For example, using the specification $\phi_1 \wedge \phi_2$ and the Figure 1(b) event net, our verifier informs us that the sequence of events C, D leads to a property violation—in particular, when the tokens are at $6, 1$, traffic is allowed along the path $H1, S1, S5, S4, H4$, violating ϕ_1. Next, we ask a *repair engine* to suggest a fix for the event net which disallows the trace C, D, and in this case, our tool produces 1(c). Again, we call the verifier, which now gives us the counterexample trace A, B (when the tokens are at $4, 3$, traffic is allowed along the path $H3, S3, S5, S2, H2$, violating property ϕ_2). When we ask the repair engine to produce a fix which avoids *both* traces C, D and A, B, we obtain the event net shown in 1(d). A final call to the verifier confirms that this event net satisfies both properties.

The synchronization skeleton produced in Figure 1(d) functions as a *barrier*—it prevents tokens from arriving at 6 or 3 until *both* tokens have moved from $4, 1$. This ensures that $1, 4$ must *both* be taken down before bringing up paths $3, 6$.

3 Synchronization Synthesis for Event Nets

Before describing our synthesis algorithm in detail, we first need to formally define the concepts/terminology mentioned so far.

SDN Preliminaries. A *packet pkt* is a record of fields $\{f_1; f_2; \cdots; f_n\}$, where fields f represent properties such as source and destination address, protocol type, etc. The (numeric) values of fields are accessed via the notation $pkt.f$, and field updates are denoted $pkt[f \leftarrow n]$. A *switch sw* is a node in the network with one or more *ports pt*. A *host* is a switch that can be a source or a sink of

packets. A *location* l is a switch-port pair $n{:}m$. Locations may be connected by (bidirectional) physical links (l_1, l_2).

A *located packet* $lp = (pkt, sw, pt)$ is a tuple consisting of a packet and a location $sw{:}pt$. A *packet-trace* h is a non-empty sequence of located packets. Packet forwarding is dictated by a *network configuration* C. We model C as a relation on located packets: if $C(lp, lp')$, then the network maps lp to lp', possibly changing its location and rewriting some of its fields. Since C is a relation, it allows multiple output packets to be generated from a single input. In a real network, the configuration only forwards packets between ports within each individual switch, but for convenience, we assume that our C also captures link behavior (forwarding between switches), i.e. $C((pkt, n_1, m_1), (pkt, n_2, m_2))$ and $C((pkt, n_2, m_2), (pkt, n_1, m_1))$ hold for each link $(n_1{:}m_1, n_2{:}m_2)$. We say that a packet-trace $h = l_0 l_1 l_2 \cdots l_n$ is *allowed by* configuration C if and only if $\forall 1 \leq k \leq n : C(l_{k-1}, l_k)$, and we denote this as $h \in C$.

Petri Net Preliminaries. Our treatment of Petri nets closely follows that of Winskel [46] (Chapter 3). A *Petri net* is a tuple (P, T, F, M_0), where P is a set of *places*, T is a set of *transitions*, $F \subseteq (P \times T) \cup (T \times P)$ is a set of *edges*, and M_0 is multiset of places denoting the *initial marking*. We require that $P \neq \emptyset$, and $\forall x \in P : M_0(x) > 0 \vee (\exists t \in T : (x,t) \in F \vee (t,x) \in F)$, and $\forall t \in T : \exists x, y \in P : (x,t) \in F \wedge (t,y) \in F$. Given a transition t, we define its pre- and post-places as $t^\bullet = \{x \in P : (t,x) \in F\}$ and ${}^\bullet t = \{x \in P : (x,t) \in F\}$ respectively. This can be extended to $T'^\bullet$ and ${}^\bullet T'$, where $T' \subseteq T$.

A marking indicates the number of *tokens* at each place. We say that a transition $t \in T$ is *enabled* by a marking M (denoted $t \subseteq M$) if and only if $\forall x \in P : (x,t) \in F \implies M(x) > 0$. A marking M_i can transition into another marking M_{i+1} as dictated by the *firing rule*: $M_i \xrightarrow{T'} M_{i+1} \iff T' \subseteq M_i \wedge M_{i+1} = M_i - {}^\bullet T' + T'^\bullet$. The *state graph* of a Petri net is a graph where each node is a marking (the initial node is M_0), and an edge $(M_i \xrightarrow{t} M_j)$ is in the graph if and only if $M_i \xrightarrow{\{t\}} M_j$ in the Petri net. A *trace* τ of a Petri net is a sequence $t_0 t_1 \cdots t_n$ starting from the initial node and such that $(t_i, t_{i+1}) \in F$, for all $i < n$. We define *markings*$(t_0 t_1 \cdots t_n)$ to be the sequence $M_0 M_1 \cdots M_{n+1}$, where $M_0 \xrightarrow{\{t_0\}} M_1 \xrightarrow{\{t_1\}} \cdots \xrightarrow{\{t_n\}} M_{n+1}$ in the Petri net. We can *project* a trace onto a Petri net (denoted $\tau \rhd N$) by removing any transitions in τ which are not in N. A *1-safe* Petri net is a Petri net in which for any marking M_j reachable from the initial marking M_0, we have $\forall x \in P : 0 \leq M_j(x) \leq 1$.

Event Nets. An *event net* E is a pair (P, λ), where P is a 1-safe Petri net, and λ labels each place with a network configuration, and each transition with an *event*. An event is a tuple (ψ, l), where l is a location, and ψ can be any predicate over network state, packet locations, etc. For instance, in [33], an event encodes an arrival of a packet with a header matching a given predicate to a given location.

Semantics of Event Nets. Given event net marking M, we denote the *global configuration* of the network $C(M)$, given as $C(M) = \bigcup_{y \in M} \lambda(y)$. Given event net $E = (P, \lambda)$, let $T(E)$ be its set of traces. $T(E)$ is defined as a set of traces of P.

Given trace τ of an event net, we use $configs(\tau)$ to denote $\bigcup_{M \in markings(\tau)} C(M)$, i.e., the set of global configurations reachable along that trace. Given event net E, we define $pktTraces(E)$ to be the set $\{h : \exists \tau \in T(E) : \exists C \in configs(\tau) : h \in C\}$. The set $pktTraces(E)$ is the set of packet traces allowed by E. Note that in this definition, the labeling of transitions by λ does not play a role. We could define a more precise semantics by allowing transitions to execute only if the event occurred (as in [33]), but here we choose the overapproximate semantics in order to be independent of the exact types of events and event occurrences.

Implementability of Event Nets. We define *local event net* to be an event net in which for any two events $e_1 = (\psi_1, l_1)$ and $e_2 = (\psi_2, l_2)$, we have $({}^\bullet e_1 \cap {}^\bullet e_2 \neq \emptyset) \Rightarrow (l_1{=}l_2)$, i.e., any two events sharing a common input place must be handled at the same location. McClurg et al. [33] present an approach for implementing a network program encoded as an event structure, under certain locality conditions similar to the one above. We can extend their construction to our local event nets using the correspondence between Petri nets and event structures presented in [46], which gives us Theorem 1:

Theorem 1 (Implementability). *Each local event net E has a distributed implementation whose single-packet traces are a subset of $pktTraces(E)$.*

The theorem follows from Theorem 1 in [33]. Intuitively, it implies that there are no packet races in the implementation, since the theorem says that each packet is processed by a trace in one of the reachable configurations. In other words, a packet is never processed in a mix of configurations.

Packet-Trace Specifications. Beyond simply freedom from packet races, we wish to rule out *controller races*, i.e., unwanted interleavings of concurrent events in an event net. In particular, we use LTL to specify formulas that should be satisfied by each packet-trace possible in each global configuration. The LTL formulas are over a single packet *pkt*, with special field *pkt.loc* denoting the packet's current location. For example, we can use the property $(pkt.loc{=}H_1 \wedge pkt.dst{=}H_2 \implies \mathbf{F} pkt.loc{=}H_2)$ to mean that any packet located at Host 1 destined for Host 2 will eventually reach Host 2. Given a trace τ of an event net, we use the notation $\tau \models \varphi$ to mean that φ holds in each global configuration $C \in configs(\tau)$.

Deadlock Freedom and 1-Safety. The input to our algorithm is a set of disjoint event nets, which we call *processes*, and we can use simple graph union to represent this as a single event net $E = \bigsqcup\{E_1, E_2, \cdots, E_n\}$. We want to avoid adding synchronization which fully deadlocks any process E_i. Let E' be an event net containing processes $E_1, E_2, \cdots, E_n$, and let P_i, T_i be the places and transitions of each E_i. We say that E' is *deadlock-free* if and only if there exists a trace $\tau \in E'$ such that $\forall 0 \leq i \leq n, M_j \in markings(\tau), t \in T_i : (({}^\bullet t \cap P_i) \subseteq M_j) \Rightarrow (\exists M_k \in markings(\tau) : k \geq j \wedge (t^\bullet \cap P_i) \subseteq M_k)$, i.e. a trace of E' where transitions t of each E_i fire as if they experienced no interference from the rest of E'. We encode this as an LTL formula, obtaining a *progress* constraint φ_{progr} for E'. Similarly, we want to avoid adding synchronization which produces an event net that is not 1-safe. We can also encode this as an LTL constraint φ_{1safe}.

Algorithm 1: Synchronization Synthesis Algorithm

Input: event net $E = \bigsqcup\{E_1, E_2, \cdots, E_n\}$, LTL property φ, upper bound Y on the number of added places, upper bound X on the number of added transitions, upper bound I on the number of synchronization skeletons

Result: event net E' such that E' correctly synchronizes E

```
1  initRepairEngine(E_1, E_2, ..., E_n, X, Y, I) ;   // initialize repair engine (§4.1)
2  E' ← E; (φ_1safe, φ_progr) ← makeProperties(E_1, E_2, ..., E_n);
3  while true do
4      ok ← true; props ← {φ, φ_1safe, φ_progr};
5      for φ' ∈ props do
6          τ_ctex ← verify(E', φ') ;                    // check the property (§4.2)
7          if (τ_ctex = ∅ ∧ φ' = φ_progr) ∨ (τ_ctex ≠ ∅ ∧ φ' = φ_1safe) then
8              differentRepair(); ok ← false ;          // try different repair (§4.1)
9          else if τ_ctex ≠ ∅ ∧ φ' ≠ φ_progr then
10             assertCtex(τ_ctex); ok←false ;           // record counterexample (§4.1)
11     if ok then
12         return E' ;                                  // return correctly-synchronized event net
13     E' ← repair(E') ;                                // generate new candidate
14     if E' = ⊥ then
15         return fail ;                                // cannot repair
```

Synchronization Synthesis Problem. Given event net $E = \bigsqcup\{E_1, E_2, \cdots, E_n\}$ and property φ, produce $E' = \bigsqcup\{E, S\}$ which *correctly synchronizes* E, i.e.,

1. $\forall \tau \in \mathit{traces}(E') : (\tau \rhd E) \in \mathit{traces}(E)$, i.e., each τ of E' (modulo added events) is a trace of E, and
2. $\forall \tau \in \mathit{traces}(E') : \tau \models \varphi$, i.e., all reachable configurations satisfy φ, and
3. $\forall \tau \in \mathit{traces}(E') : \tau \models \varphi_{1safe}$, i.e., E' is 1-safe, and
4. $\exists \tau \in \mathit{traces}(E') : \tau \models \varphi_{progr}$, i.e., E' deadlock-free.

Event net S consists of *synchronization skeletons* (described in Section 4).

4 Fixing and Checking Synchronization in Event Nets

Algorithm 1 describes our solution—an instance of the CEGIS algorithm in [18, 22] which is now set up for problems of the form $\exists E'((\forall \tau \in E' : \phi(E', E, \varphi, \varphi_{1safe})) \wedge \neg(\forall \tau \in E' : \tau \not\models \varphi_{progr}))$, where E, E' are input/output event nets, and ϕ captures 1-3 of the above specification. Our *event net repair engine* (Section 4.1) performs synthesis (producing candidate solutions for $\exists$), and our *event net verifier* (Section 4.2) performs verification (checking $\forall$).

The function *makeProperties* produces the $\varphi_{1safe}, \varphi_{progr}$ formulas as described in Section 3. We will now describe the details of the other functions.

4.1 Repairing Event Nets Using Counterexample Traces

The repair engine uses an SMT solver to perform the search for synchronization constructs to fix a finite set of bugs (given as event-net traces which should not be allowed). Figure 2 shows three types of *synchronization skeletons* which our repair engine can add between the processes of the input event net E'. As

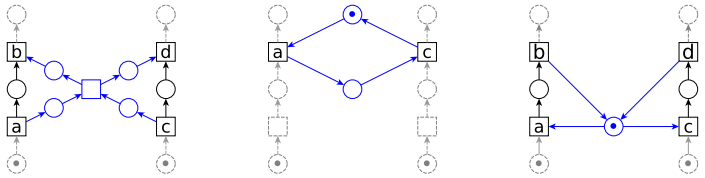


Fig. 2: Synchronization skeletons: (1) Barrier, (2) Condition Variable, (3) Mutex.

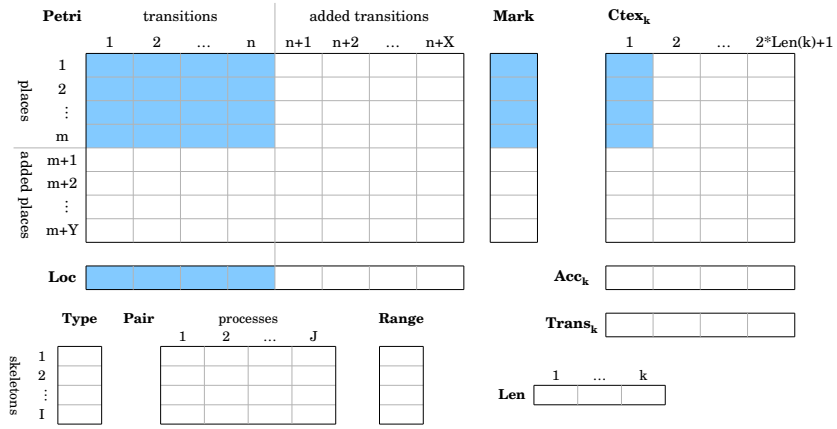


Fig. 3: SMT Function Symbols.

the figure indicates, the *barrier* skeleton does not allow events b, d to fire until *both* a, c have fired. The *condition variable* requires event a to fire before event c can fire. The *mutex* ensures that the events between a and b (inclusive) cannot interleave with the events between c and d (inclusive). Our algorithm explores different combinations of these skeletons, up to the given bounds.

Repair Engine Initialization. Algorithm 1 calls $\mathit{initRepairEngine}(E_1, E_2, \cdots, E_n, X, Y, I)$, which "initializes" the function symbols shown in Figure 3 with the values from the input event nets, and asserts well-formedness constraints. Labels in bold are function symbol names, and cells are the corresponding values. For example, *Petri* is a 2-ary function symbol, and *Mark* is 1-ary. Note that there is a separate *Ctex*, *Acc*, *Trans* for each k. Letting $\mathbb{B}$ denote $\{\mathit{true}, \mathit{false}\}$, the types of the function symbols are: $\mathit{Petri} : \mathbb{N}\times\mathbb{N} \to \mathbb{B}\times\mathbb{B}$, $\mathit{Mark} : \mathbb{N} \to \mathbb{N}$, $\mathit{Loc} : \mathbb{N} \to \mathbb{N}\times\mathbb{N}$, $\mathit{Type} : \mathbb{N} \to \mathbb{N}$, $\mathit{Pair} : \mathbb{N} \to \mathbb{N}\times\mathbb{N}\times\mathbb{N}$, $\mathit{Range} : \mathbb{N} \to \mathbb{N}\times\mathbb{N}\times\mathbb{N}\times\mathbb{N}$, $\mathit{Len} : \mathbb{N} \to \mathbb{N}$, $\mathit{Ctex}_k : \mathbb{N}\times\mathbb{N} \to \mathbb{N}$, $\mathit{Acc}_k : \mathbb{N} \to \mathbb{B}$, $\mathit{Trans}_k : \mathbb{N} \to \mathbb{N}$.

The regions highlighted in Figure 3 are "set" (asserted equal) to values matching the input event nets. For example, $\mathit{Petri}(y, x)$ is of the form (b_1, b_2), where we set b_1 if and only if there is an edge from place y to transition x in E', and similarly set b_2 if and only if there is an edge from transition x to place y. $\mathit{Mark}(y)$ is set to 1 if and only if place y is marked in E'. $\mathit{Loc}(x)$ is set to the location (switch/port pair) of the event at transition x. The bound Y limits how many places can be added, and X limits how many transitions can be added.

The bound I limits how many synchronization skeletons can be used simultaneously. Each row i of $\mathit{Type}, \mathit{Pair}, \mathit{Range}$ represents a single added skeleton.

$Type(i)$ identifies one of the three types of skeletons. Up to J processes can participate in each skeleton (Figure 2 shows the skeletons for 2 processes, but they generalize to $j \geq 2$), and by default, J is set to the number of processes. $Pair(i, j)$ is a tuple (id, fst, snd), where id identifies a process, and fst, snd is a pair of events in that process. $Range(i)$ is a tuple $(pMin, pMax, tMin, tMax)$, where $pMin, pMax$ reserve a range of rows in the *added places* section of Figure 3, and similarly, $tMin, tMax$ reserve a range of columns in the *added transitions*.

We assert a conjunction ϕ_{global} of well-formedness constraints to ensure that proper values are used to fill in the empty (un-highlighted) cells of Figure 3. Primarily, these constraints force the *Petri* cells to be populated as dictated by any synchronization skeletons appearing in the *Type*, *Pair*, *Range* rows.

Asserting Counterexample Traces. Once the repair engine has been initialized, Algorithm 1 can add counterexample traces by calling $assertCtex(\tau_{ctex})$. To add the k-th counterexample trace $\tau_k = t_0 t_1 \cdots t_{n-1}$, we assert the conjunction ϕ_k of the following constraints. These constraints make the columns of $Ctex_k$ correspond to the sequence of markings of the current event net in *Petri* if it were to fire the sequence of transitions τ_k. More specifically, $Ctex_k$ is inductively defined as $Ctex_k(1) = Mark$ and for $x > 1$, $Ctex_k(x)$ is equal to the marking that would be obtained if t_{x-1} were to fire in $Ctex_k(x-1)$. The symbol Acc_k is similarly defined as $Acc_k(1) = true$ and for $x > 1$, $Acc_k(x) \iff (Acc_k(x-1) \wedge (t_{x-1}$ is enabled in $Ctex_k(x-1)))$. We also assert a constraint requiring that Acc_k must become false before the end of the trace.

We must modify the above constraints to handle *general* counterexamples. Specifically, if a trace of the event net in *Petri* is equal to τ_k modulo transitions added by the synchronization skeletons, that trace should be rejected. We do this by instead considering the trace $\tau'_k = 0, t_0, 0, t_1, \cdots, 0, t_{n-1}$, and for the "0" transitions, set $Ctex_k(x)$ as if we fired any enabled *added transitions* in $Ctex_k(x-1)$, and for the t_i transitions, update $Ctex_k(x)$ as described previously. Therefore, the constraints ϕ_k are as follows:

1. The first column of $Ctex_k$ is equal to $Mark$.
2. $Len(k){=}n \wedge Acc_k(1) \wedge \neg Acc_k(2 \cdot Len(k) + 1)$.
3. $Acc_k(x) \iff (Acc_k(x-1) \wedge (Trans_k(x){=}0 \vee (Trans_k(x)$ is enabled in $Ctex_k(x-1))))$.
4. For *odd* indices $x \geq 3$, $Trans_k(x) = t_{(x-3)/2}$, and $Ctex_k(x)$ is set as if $Trans_k(x)$ fired in $Ctex_k(x-1)$.
5. For *even* indices $x \geq 2$, $Trans_k(x) = 0$, and $Ctex_k(x)$ is set as if all enabled *added transitions* fired in $Ctex_k(x-1)$.

The last constraint works because for our synchronization skeletons, any added transitions that occur immediately after each other in a trace can also occur in parallel. The constraint $\neg Acc_k(2 \cdot Len(k) + 1)$ makes sure that any synchronization generated by the SMT solver will not allow the full trace τ to be accepted.

Trying a Different Repair. The *differentRepair*() function in Algorithm 1 makes sure the repair engine does not propose the current candidate again. When this is called, we prevent the current set of synchronization skeletons from appearing again by taking the conjunction of the *Type* and *Pair* values, as well as the

Algorithm 2: Event Net Verifier (PROMELA Model)

```
1  marked ← initMarking(); run singlePacket, transitions;
2  Process singlePacket:
3      lock(); status ← 1; pkt ← pickPacket(); n ← pickHost();
4      do
5          pkt ← movePacket(pkt, marked);
6      while pkt.loc ≠ drop ∧ ¬isHost(pkt.loc);
7      status ← 2; unlock();
8  Process transitions:
9      while true do
10         lock();
11         t ← pickTransition(marked); marked ← updateMarking(t, marked);
12         unlock();
```

values of *Mark* corresponding to the places reserved in *Range*, and asserting the negation. We denote the current set of all such assertions ϕ_{skip}.

Obtaining an Event Net. When the synthesizer calls $repair(E')$, we query the SMT solver for satisfiability of the current constraints. If satisfiable, values of *Petri*, *Mark* in the model can be used to add synchronization skeletons to E'.

Note that formulas $\phi_{global}, \phi_{skip}, \phi_1, \cdots$ have polynomial size in terms of the input event net size and bounds Y, X, I, J, and are expressed in the decidable fragment `QF_UFLIA` (quantifier-free uninterpreted function symbols and linear integer arithmetic). We found this to scale well with modern SMT solvers (§5).

Lemma 1 (Correctness of the Repair Engine). *If the SMT solver finds that $\phi = \phi_{global} \wedge \phi_{skip} \wedge \phi_1 \wedge \cdots \wedge \phi_k$ is satisfiable, then the event net represented by the model does not contain any of the seen counterexample traces $\tau_1, \cdots, \tau_k$. If the SMT solver finds that ϕ is unsatisfiable, then all synchronization skeletons within the bounds fail to prevent some counterexample trace.*

4.2 Checking Event Nets

This section describes the $verify(E', \varphi')$ function in Algorithm 1. From event net E', we produce a PROMELA model which we provide to an off-the-shelf LTL model checker. Algorithm 2 shows the model pseudocode, which is an efficient implementation of the semantics described in Section 3. Global variable *marked* is a list of boolean flags, indicating which places currently contain a token. The *initMarking* macro sets the initial values based on the initial marking of E'. The *singlePacket* process randomly selects a packet *pkt* and puts it at a random host, and then moves *pkt* until it either reaches another host, or is dropped ($pkt.loc = drop$). The *movePacket* macro modifies/moves *pkt* according to the current marking's configuration. The *pickTransition* macro randomly selects a transition $t \in E'$, and *updateMarking* updates the marking to reflect t firing.

We ask the model checker for a *counterexample trace* demonstrating a violation of φ' in this model. If found, we extract the sequence of events e chosen by *pickEvent*. We *generalize* this sequence by removing any events which are not in the original input event nets. This sequence is returned as τ_{ctex} to Algorithm 1.

Lemma 2 (Correctness of the Verifier). *If the verifier returns counterexample τ, then E' violates φ in one of the global configurations in configs(τ). If the verifier does not return a counterexample, then all traces of E' satisfy φ.*

Overall Correctness Results. We now characterize the correctness of our synchronization synthesis approach (the proofs use Lemma 1-2, and Theorem 1).

Theorem 2 (Soundness of Algorithm 1). *Given E and φ, if Algorithm 1 returns an event net E', then E' correctly synchronizes E with respect to φ.*

Theorem 3 (Completeness of Algorithm 1). *If there exists an $E' = \bigsqcup\{E, S\}$, where $|S| \leq I$ and synchronization skeletons S have fewer than X total transitions and fewer than Y total places, and E' correctly synchronizes E, then our algorithm will return such an E'. Otherwise, the algorithm returns "fail."*

5 Implementation and Evaluation

We have implemented a prototype of our synchronization synthesis tool. The repair engine described in Section 4.1 utilizes the Z3 SMT solver, and the verifier described in Section 4.2 utilizes the SPIN LTL model checker. In this section, we evaluate our system by answering the following questions:

1. Can we use our approach to model a variety of real-world network programs?
2. Is our tool able to fix realistic concurrency-related bugs?
3. Is the performance of our tool reasonable on real networks?

We address 1-2 via case studies based on real concurrency bugs described in the networking literature, and address 3 by choosing one of these examples and trying different topologies. Figure 5 shows performance results and quantitative metrics. The first group of columns denote the number of switches (*switch*), CEGIS iterations (*iter*), SPIN counterexamples (*ctex*), event nets "skipped" due to a deadlock-freedom or 1-safety violation (*skip*), and formulas asserted to the SMT solver (*smt*). The second group of columns report the runtime of the SPIN verifier generation/compilation (*build*), SPIN verification (*verify*), repair engine (*synth*), various auxiliary (*misc*), and overall execution (*total*).[1]

Example #1—Tenant Isolation in a Datacenter. We formalize the isolation property from Section 2 using the LTL properties $G(loc{=}H1 \implies G(loc{\neq}H4))$ and $G(loc{=}H3 \implies G(loc{\neq}H2))$. Our tool finds the *barrier* shown in Figure 1(b), which properly synchronizes the event net to avoid isolation violations.

Example #2—Conflicting Controller Modules. In a real bug (El-Hassany et al. [17]) encountered using the POX SDN controller, the concurrent modules *Discovery* and *Forwarding* made conflicting assumptions about which forwarding rules should be deleted, resulting in packet loss. Figure 4(a) shows a simplified version of the scenario, where the left side $(1, A, 2, B)$ corresponds to the Discovery module, and the right side $(4, C, 3, D)$ corresponds to the Forwarding module. In this example, Discovery is responsible for ensuring that packets can be forwarded to H1 (i.e., that the configuration labeled with 2 is active), and

[1] We ran these on a machine w/ 20GB RAM and 3.2 GHz 4-core Intel i5-4570 CPU.

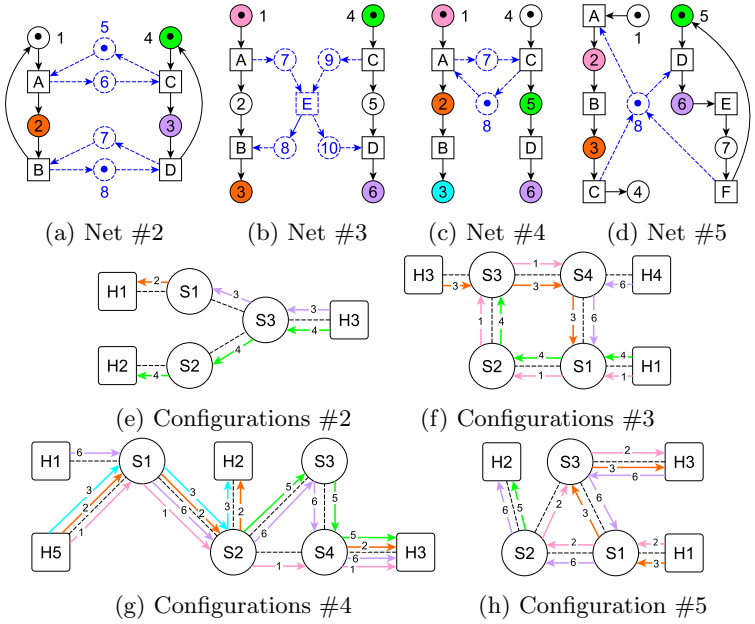


Fig. 4: Experiments—Event Nets and Configurations.

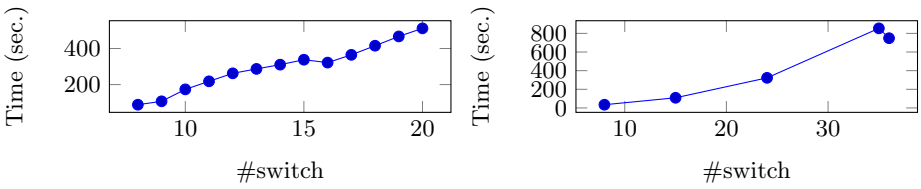
Forwarding is responsible for choosing a path for traffic from H3 (either the path labeled 3 or 4). In all cases, we require that all traffic from H3 is not dropped.

We formalize this requirement using the LTL property $G(loc{=}H3 \implies G(loc{\neq}drop))$, and our tool finds the *two condition variables* which properly synchronize the event net to avoid packet loss. As shown in Figure 4(a), the path corresponding to place 2 must be brought up *before* the path corresponding to place 3 (i.e., event C can only occur after A), and can only be taken down *after* the path 3 is moved back to path 4 (i.e., event B can only occur after D).

Example #3—Discovery Forwarding Loop. In a real bug scenario (Scott et al. [41]) the NOX SDN controller's discovery functionality attempted to learn the network topology, but an unexpected interleaving of packets caused a small forwarding loop to be created. We show how a forwarding loop can arise due to an unexpected interleaving of controller modules. In Figure 4(b), the *Forwarding*/*Discovery* modules are the left/right sides respectively. Initially, *Forwarding* knows about the red (1) path in Figure 4(f), but will delete these rules, and later set up the orange (3) path. On the other hand, *Discovery* first learns that the green (4) path is going down, and then later learns about the violet (6) path. Since these modules both modify the same forwarding rules, they can create a forwarding loop when configurations $1, 6$ or $4, 3$ are active simultaneously.

We wish to disallow such forwarding loops, formalizing this using $G(status{=}1 \implies F(status{=}2))$, where, as discussed in Section 4.2, *status* is set to 1 when the packet is injected into the network, and set to 2 when/if the packet subsequently exits or is dropped. Our tool enforces this requirement by inserting a *barrier*, as in Figure 4(b), preventing the unwanted combinations of configurations.

benchmark	#number					time (sec.)				
	switch	iter	ctex	skip	SMT	build	verify	synth	misc	total
ex01-isolation	5	2	2	0	320	10.56	10.23	0.04	0.55	21.38
ex02-conflict	3	14	3	10	388	20.50	5.23	1.14	1.70	28.57
ex03-loop	4	2	1	0	259	5.07	4.78	0.01	0.48	10.34
ex04-compose	5	2	1	0	307	25.28	21.00	0.03	0.50	46.81
ex05-exclusive	3	4	2	2	522	28.26	8.09	0.07	0.92	37.35
Ans	18	2	1	0	259	92.53	131.10	0.01	0.48	224.13
Cesnet1993	10	2	1	0	259	20.81	26.08	0.02	0.49	47.40
Claranet	15	2	1	0	259	69.84	85.03	0.02	0.51	155.40
Compuserve	14	2	1	0	259	61.19	69.20	0.02	0.51	130.91
Fatman	17	2	1	0	259	90.59	119.34	0.02	0.45	210.40
Gblnet	8	2	1	0	259	17.65	16.78	0.01	0.47	34.92
HostwayIntl	16	2	1	0	259	82.54	108.71	0.02	0.52	191.80
Itnet	11	2	1	0	259	24.62	31.61	0.02	0.47	56.71
JanetExtern	12	2	1	0	259	26.38	28.52	0.02	0.48	55.40
Layer42	6	2	1	0	259	18.22	17.10	0.01	0.45	35.79
Netrail	7	2	1	0	259	28.15	27.02	0.02	0.48	55.67
Nsfcnet	9	2	1	0	259	27.60	27.59	0.02	0.48	55.69
Nsfnet	13	2	1	0	259	42.82	57.34	0.02	0.50	100.68
Renam	5	2	1	0	259	10.59	10.23	0.02	0.50	21.34
Savvis	19	2	1	0	259	86.26	103.90	0.02	0.49	190.67
sw-07-4	7	2	1	0	259	33.99	37.17	0.01	0.45	71.63



(a) small-world benchmarks (b) FatTree benchmarks

Fig. 5: Performance of Examples. Scalability results: (a) Topology Zoo, (b) FatTree.

Example #4—Policy Composition. In an update scenario (Canini et al. [10]) involving *overlapping* policies, one policy enforces HTTP traffic monitoring and the other requires traffic from a particular hosts(s) to *waypoint* through a device. Problems arise for traffic processed by the *intersection* of these policies (e.g., HTTP packets from a particular host), causing a policy violation.

Figure 4(g) shows such a conflict. The left process of 4(c) is traffic monitoring, and the right process is waypoint enforcement. HTTP traffic is initially enabled along the red (1) path. Traffic monitoring then intercepts this traffic and diverts it to $H2$ by setting up the orange (2) path and subsequently bringing it down to form the blue path (3). Waypoint enforcement initially sets up the green path (5) through the waypoint $S3$, and finally allows traffic to enter by setting up the violet (6) path from $H1$. For *HTTP traffic from* $H1$, if traffic monitoring is not set up *before* the waypoint enforcement enables the path from $H1$, then this traffic can circumvent the waypoint (on the $S2 \rightarrow S4$ path), violating the policy.

Properties $G(pkt.type{=}HTTP \wedge pkt.loc{=}H5 \implies F(pkt.loc{=}H2 \vee pkt.loc{=}H3))$, and $(\neg(pkt.src{=}H1 \wedge pkt.loc{=}H3)U(pkt.src{=}H1 \wedge pkt.loc{=}S3))$ form the specification (where U is *until*). Our tool finds Figure 4(c), which forces traffic monitoring to divert traffic *before* waypoint enforcement proceeds.

Example #5—Topology Changes during Update. Peresíni et al. [38] describe a scenario in which a controller attempts to set up forwarding rules, and concurrently the topology changes, resulting in a forwarding loop being installed.

Figure 4(h), examines a similar situation where the processes in Figure 4(d) interleave improperly, resulting in a forwarding loop. The left process updates from the red (2) to the orange (3) path, and the right process extends the green (5) to the violet (6) path (potential forwarding loops: $S1, S3$ and $S1, S2, S3$).

We use the Example #3 loop-freedom property. Our tool finds the *mutex* synchronization skeleton shown in Figure 4(d). Note that both places $2, 3$ are protected by the mutex, since either would interact with place 6 to form a loop.

Scalability Experiments. Recall Example #1 (Figure 1(a)). Instead of the short paths between the pairs of hosts $H1, H2$ and $H3, H4$, we picked a random set of real wide-area network topologies from the Topology Zoo dataset, as well as highly-connected ("small-world") graphs, and datacenter FatTree topologies, and randomly selected long host-to-host paths corresponding to Example #1. We note in all of the experiments that the SMT component scales much more readily than building/running SPIN verifiers.

6 Related Work

Synthesis for Network Programs. Yuan et al. [47] present NetEgg, pioneering the approach of using examples to write network programs. In contrast, we focus on distributed programs and use specifications instead of examples. Additionally, different from our SMT-based strategy, NetEgg uses a backtracking search which may limit scalability. Padon et al. [37] "decentralize" a network program to work properly on distributed switches. Our work on the other hand takes a buggy decentralized program and inserts the necessary synchronization to make it correct. Saha et al. [40] and Hojjat et al. [19] present approaches for repairing a buggy network configuration using SMT and a Horn-clause-based synthesis algorithm respectively. Instead of repairing a static configuration, our event net repair engine repairs a network program. A *network update* is a simple network program—a situation where the global forwarding state of the network must change once. Many approaches solve the problem with respect to different consistency properties [20, 25, 30, 24, 32, 48]. In contrast, we provide a new model (event nets) for succinctly describing how multiple updates can be composed, as well as an approach for synthesizing synchronization for this composition.

Concurrent Programming for Networks. Some well-known network programming languages (e.g., NetKAT [1, 43]) only allow defining static configurations, and they do not support stateful programs and concurrency constructs. Many languages [36, 35, 26], provide support for stateful network programming (often with finite-state control), but lack direct support for synchronization. There are two recently proposed exceptions: SNAP [2], which provides atomic blocks,

and the approach by Canini et al. [10], which provides transactions. Both of these mechanisms are expensive and difficult to implement without damage to performance. In contrast, our solution is based on locality and synchronization synthesis. Our method is more fine-grained and efficiently implementable than previous approaches. It builds on and extends network event structures (NES) [33], which addresses the problem of rigorously defining correct event-driven behavior. From the *systems* side, basic support for stateful concurrent programming is provided by switch-level mechanisms [8, 6, 42] and hypervisors [23], but global coordination still must be handled carefully at the language/compiler level.

Petri Net Synthesis. Ehrenfeucht et al. [16] introduce the "net synthesis" problem, i.e., producing a net whose state graph is *isomorphic to a given DFA*, and present the "regions" construction on which Petri net synthesis algorithms are based. Many researchers continued this theoretical line of work [13, 12, 3, 21] and developed foundational (complexity-theoretic) results. Synthesis from examples for Petri nets was also considered [5, 9], and examined in the slightly different setting of *process mining* [15, 39]. Neither of these approaches is directly applicable to our problem of program repair by inserting synchronization to eliminate bugs. More closely related is *process enhancement* for Petri nets [31, 4] but these works either modify the semantics of systems in arbitrary ways, whereas we only restrict behaviors by adding synchronization, or they rely on other abstractions (such as *timed* Petri nets) which are unsuitable for network programming.

Synthesis/Repair for Synchronization. There are many approaches for fixing concurrency bugs which use constraint (SAT/SMT) solving. The application areas include weak memory models [34, 29], and repair of concurrency bugs [11, 45, 7, 44]. The key difference is that while these works focus on shared-memory programs, we focus on message-passing Petri-net based programs. Our Petri net model is a general framework for synthesis of synchronization where many different types of synchronization constructs can be readily described and synthesized.

7 Conclusion

Summary. We have presented an approach for synthesis of synchronization to produce network programs which satisfy correctness properties. We allow the network programmer to specify a network program as a set of concurrent behaviors, in addition to high-level temporal correctness properties, and our tool inserts synchronization constructs necessary to remove unwanted interleavings. The advantages over previous work are that (a) we provide a language which leverages Petri nets' natural support for concurrency, and (b) we provide an efficient algorithm for synthesizing synchronization for programs in this language.

Future Work. This paper suggests several directions for future research. First, one could investigate a repair engine which would add *arbitrary* sets of places, transitions, and edges, rather than choosing from our limited (yet customizable) set. Second, it is possible to improve the integration between the repair engine and the verifier, e.g., by taking advantage of the incremental capabilities of the SMT solver, and building a verifier that is *incremental* in the sense that it only re-checks the parts of the event net that changed since the last check.

References

[1] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. "NetKAT: Semantic Foundations for Networks". In *POPL* (2014).

[2] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. "SNAP: Stateful Network-Wide Abstractions for Packet Processing". In *SIGCOMM* (2016).

[3] Eric Badouel, Luca Bernardinello, and Philippe Darondeau. "The Synthesis Problem for Elementary Net Systems is NP-Complete". In *Theor. Comput. Sci.* 186.1-2 (1997), pp. 107–134.

[4] F. Basile, P. Chiacchio, and J. Coppola. "Model repair of Time Petri Nets with temporal anomalies". In *IFAC-PapersOnLine* 48.7 (2015). 5th {IFAC} International Workshop on Dependable Control of Discrete SystemsDCDS 2015, pp. 85–90. ISSN: 2405-8963.

[5] Robin Bergenthum, Jörg Desel, Robert Lorenz, and Sebastian Mauser. "Synthesis of Petri Nets from Finite Partial Languages". In *Fundam. Inform.* 88.4 (2008), pp. 437–468.

[6] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. "OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch". In *ACM SIGCOMM CCR* (2014).

[7] Roderick Bloem, Georg Hofferek, Bettina Könighofer, Robert Könighofer, Simon Ausserlechner, and Raphael Spork. "Synthesis of synchronization using uninterpreted functions". In *FMCAD*. IEEE, 2014, pp. 35–42.

[8] Pat Bosshart et al. "P4: Programming Protocol-independent Packet Processors". In *ACM SIGCOMM CCR* (2014).

[9] Maria Paola Cabasino, Alessandro Giua, and Carla Seatzu. "Identification of Petri Nets from Knowledge of Their Language". In *Discrete Event Dynamic Systems* 17.4 (2007), pp. 447–474.

[10] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. "Software transactional networking: concurrent and consistent policy composition". In *HotSDN*. ACM, 2013, pp. 1–6.

[11] Pavol Černý, Thomas A Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. "Efficient Synthesis for Concurrency by Semantics-preserving Transformations". In *CAV* (2013).

[12] Jordi Cortadella, Michael Kishinevsky, Luciano Lavagno, and Alexandre Yakovlev. "Synthesizing Petri nets from state-based models". In *ICCAD*. IEEE, 1995, pp. 164–171.

[13] Jörg Desel and Wolfgang Reisig. "The Synthesis Problem of Petri Nets". In *Acta Inf.* 33.4 (1996), pp. 297–315.

[14] Advait Abhay Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman, and Ramana Kompella. "ElastiCon: An Elastic Distributed Sdn Controller". In *ANCS*. Los Angeles, California, USA, 2014.

[15] Marlon Dumas and Luciano García-Bañuelos. "Process Mining Reloaded: Event Structures as a Unified Representation of Process Models and Event

Logs". In *Petri Nets*. Vol. 9115. Lecture Notes in Computer Science. Springer, 2015, pp. 33–48.

[16] Andrzej Ehrenfeucht and Grzegorz Rozenberg. "Partial (Set) 2-Structures. Part II: State Spaces of Concurrent Systems". In *Acta Inf.* 27.4 (1990), pp. 343–368.

[17] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin T. Vechev. "SDNRacer: concurrency analysis for software-defined networks". In *PLDI*. ACM, 2016, pp. 402–415.

[18] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. "Synthesis of Loop-free Programs". In *PLDI* (2011).

[19] Hossein Hojjat, Philipp Ruemmer, Jedidiah McClurg, Pavol Cerny, and Nate Foster. "Optimizing Horn Solvers for Network Repair". In *FMCAD* (2016).

[20] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. "Achieving High Utilization with Software-driven WAN". In *SIGCOMM* (2013).

[21] Richard P. Hopkins. "Distributable nets". In *Applications and Theory of Petri Nets*. Vol. 524. Lecture Notes in Computer Science. Springer, 1990, pp. 161–187.

[22] Susmit Jha, Sumit Gulwani, Sanjit Seshia, Ashish Tiwari, et al. "Oracle-guided Component-based Program Synthesis". In *ICSE* (2010).

[23] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. "CoVisor: A Compositional Hypervisor for Software-Defined Networks". In *NSDI* (2015).

[24] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. "Dynamic Scheduling of Network Updates". In *SIGCOMM* (2014).

[25] Naga Praveen Katta, Jennifer Rexford, and David Walker. "Incremental Consistent Updates". In *HotSDN*. ACM. 2013, pp. 49–54.

[26] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. "Kinetic: Verifiable Dynamic Network Control". In *NSDI* (2015).

[27] Teemu Koponen et al. "Network Virtualization in Multi-tenant Datacenters". In *NSDI* (2014).

[28] Teemu Koponen et al. "Onix: A Distributed Control Platform for Large-scale Production Networks". In *OSDI*. USENIX Association, 2010, pp. 351–364.

[29] Michael Kuperstein, Martin T. Vechev, and Eran Yahav. "Automatic inference of memory fences". In *FMCAD*. IEEE, 2010, pp. 111–119.

[30] A. Ludwig, M. Rost, D. Foucard, and S. Schmid. "Good Network Updates for Bad Packets: Waypoint Enforcement Beyond Destination-Based Routing Policies". In *HotNets*. 2014.

[31] Ulises Martínez-Araiza and Ernesto López-Mellado. "{CTL} Model Repair for Bounded and Deadlock Free Petri Nets". In *IFAC-PapersOnLine* 48.7

(2015). 5th {IFAC} International Workshop on Dependable Control of Discrete SystemsDCDS 2015, pp. 154–160. ISSN: 2405-8963.

[32] Jedidiah McClurg, Hossein Hojjat, Pavol Cerny, and Nate Foster. "Efficient Synthesis of Network Updates". In *PLDI* (2015).

[33] Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Cerny. "Event-driven Network Programming". In *PLDI* (2016).

[34] Yuri Meshman, Noam Rinetzky, and Eran Yahav. "Pattern-based Synthesis of Synchronization for the C++ Memory Model". In *FMCAD*. IEEE, 2015, pp. 120–127.

[35] Masoud Moshref, Apoorv Bhargava, Adhip Gupta, Minlan Yu, and Ramesh Govindan. "Flow-level State Transition as a New Switch Primitive for SDN". In *HotSDN*. 2014.

[36] Tim Nelson, Andrew D Ferguson, MJ Scheer, and Shriram Krishnamurthi. "Tierless Programming and Reasoning for Software-Defined Networks". In *NSDI* (2014).

[37] Oded Padon, Neil Immerman, Aleksandr Karbyshev, Ori Lahav, Mooly Sagiv, and Sharon Shoham. "Decentralizing SDN Policies". In *POPL* (2015).

[38] Peter Peresíni, Maciej Kuzniar, Nedeljko Vasic, Marco Canini, and Dejan Kostic. "OF.CPP: consistent packet processing for openflow". In *HotSDN*. ACM, 2013, pp. 97–102.

[39] Hernán Ponce de León, César Rodríguez, Josep Carmona, Keijo Heljanko, and Stefan Haar. "Unfolding-Based Process Discovery". In *ATVA*. Vol. 9364. Lecture Notes in Computer Science. Springer, 2015, pp. 31–47.

[40] Shambwaditya Saha, Santhosh Prabhu, and P. Madhusudan. "NetGen: synthesizing data-plane configurations for network policies". In *SOSR*. ACM, 2015, 17:1–17:6.

[41] Colin Scott et al. "Troubleshooting blackbox SDN control software with minimal causal sequences". In *SIGCOMM*. ACM, 2014, pp. 395–406.

[42] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. "Packet Transactions: High-Level Programming for Line-Rate Switches". In *SIGCOMM* (2016).

[43] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. "A Fast Compiler for NetKAT". In *ICFP* (2015).

[44] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. "Sketching concurrent data structures". In *PLDI*. ACM, 2008.

[45] Martin Vechev, Eran Yahav, and Greta Yorsh. "Abstraction-guided Synthesis of Synchronization". In *POPL* (2010).

[46] Glynn Winskel. *Event Structures*. Springer, 1987.

[47] Yifei Yuan, Dong Lin, Rajeev Alur, and Boon Thau Loo. "Scenario-based Programming for SDN Policies". In *CoNEXT* (2015).

[48] Wenxuan Zhou, Dong Jin, Jason Croft, Matthew Caesar, and P. Brighten Godfrey. "Enforcing Generalized Consistency Properties in Software-Defined Networks". In *NSDI* (2015).