Synchronization Synthesis for Network Programs

Jedidiah McClurg[1], Hossein Hojjat[2], and Pavol Černý[1]

[1] University of Colorado Boulder, USA
[2] Rochester Institute of Technology, USA

Abstract. In software-defined networking (SDN), a controller program updates the forwarding rules installed on network packet-processing devices in response to events. Such programs are often physically distributed, running on several nodes of the network, and this distributed setting makes programming and debugging especially difficult. 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 among the distributed controller processes such that the declarative specification will be satisfied by all packets traversing the network. Our key technical contribution is a counterexample-guided synthesis algorithm that furnishes network controller processes with the synchronization constructs required to prevent any races causing specification violations. 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 readily scale to network topologies with 1000+ nodes.

1 Introduction

Software-defined networking (SDN) enables programmers or *network operators* to more easily implement important applications such as traffic engineering, distributed firewalls, network virtualization, etc. These applications are typically *event-driven*, in the sense that the packet-processing behavior can change in response to network events such as topology changes, shifts in traffic load, or arrival of packets at various network nodes. SDN enables this type of event-driven behavior via a *controller machine* that manages the network *configuration*, i.e., the set of *forwarding rules* installed on the network *switches*. The programmer can write code which runs on the controller, as well as instruct the switches to install custom forwarding rules, which inspect incoming packets and move them to other switches or send them to the controller for custom processing.

Concurrency in Network Programs. Although SDN provides the abstraction of a *centralized* controller machine, in reality, network control is often physically distributed, with controller processes running on multiple network nodes [13]. The fact that these distributed programs control a network which is *itself* a distributed packet-forwarding system means that event-driven network applications 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 switches, or between packets that are in-flight during updates. Second, there are races among the different processes of the distributed controller. We call the former *packet races*, and the latter *controller races*. Bugs resulting from either of 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 1a. 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 the lower part ($H2, S2, S5, S4, H4$). 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 ϕ holds at all times: *all packets entering the network from* $H1$ *must exit at* $H2$. When developing the program to do this, the network operator must consider the following:

- Packet race: If $C1$ removes the rule that forwards from $S1$ to $S5$ before removing the rule that forwards from $H1$ to $S5$, then a packet entering at $H1$ will be dropped at $S1$, violating specification ϕ.
- Controller race: Suppose $C1$ makes no changes, and $C2$ adds rules that forward from $S5$ to $S4$, and from $S4$ to $H4$. In the resulting configuration, a packet entering at $H1$ can be forwarded to $H4$, again violating ϕ.

Our Approach. We present a program synthesis approach that makes it easier to write distributed controller programs. The programmer can specify each sequential process (e.g., $C1$ and $C2$ in the previous example), and add a declarative specification of paths that packets are allowed to take (e.g., ϕ in the previous example). The synthesizer then inserts *synchronization constructs* that constrain the interactions among the controller processes to ensure that the specification is always satisfied by any packets traversing the network. Effectively, this 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 have considered, we find these specifications to be a clear and easy way to write desired correctness properties.

Network Programming Model. In our approach, similar to network programming languages like OpenState [6], and Kinetic [20], we allow a network program to be

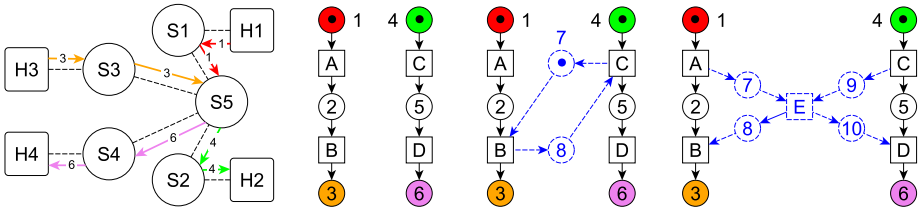
described as a set of concurrently-operating finite state machines (FSMs) consisting of event-driven transitions between global network states. We generalize this by allowing the input network program to be a set of *event nets*, which are 1-safe Petri nets where each transition corresponds to a network event, and each place corresponds to a set of forwarding rules. This model extends network event structures [25] to enable straightforward modeling of programs with loops. An advantage of extending this particular programming model is that its programs can be efficiently implemented without packet races (see Section 3 for details).

Problem Statement. Our synthesizer has two inputs: (1) a set of event nets representing sequential processes of the distributed controller, and (2) a linear temporal logic (LTL) specification of paths that packets are allowed to take. For example, the programmer can specify properties such as "packets from $H1$ must pass through Middlebox $S5$ before exiting the network." The output is an event net consisting of the input event nets and added synchronization constructs, such that all packets traversing the network satisfy the specification. In other words, 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) a *repair engine* that synthesizes a candidate event net from the input event nets and a finite set of known 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). Given a candidate event net, the verifier checks whether it is deadlock-free (i.e., there is an execution where all processes can proceed without deadlock), whether it is 1-safe, 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.

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 (Sections 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 show the usefulness and efficiency of our prototype implementation, using several examples featuring network topologies of 1000+ switches (Section 5).



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

Fig. 1: Example #1

2 Network Programming using Event Nets

Network programs change the network's global forwarding behavior in response to events. Recently proposed approaches such as OpenState [6] and Kinetic [20] 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 network events (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 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 increasingly important, 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 at most one *token* to each place, 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 1b shows an example event net. We will use integer IDs (and alternatively, colors) to distinguish static configurations. Figure 1a 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 1b is orange, and this corresponds to enabling forwarding along the orange path $H3, S3, S5$ (labeled with "3") in the topology shown in Figure 1a. 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 1c, 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 constructs produced by our tool are shown in blue (dashed lines). We will now demonstrate by example how our tools works.

Example—Tenant Isolation in a Datacenter. Koponen et al. [21] describe an approach for providing *virtual networks* to *tenants* (users) of a datacenter, allowing them to connect virtual machines (VMs) using virtualized networking functionality (middleboxes, etc.). An important aspect is *isolation* between tenants—one tenant intercepting another tenant's traffic would be a severe security violation.

Let us extend the example described in the Introduction. In Figure 1a, $S5$ 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. For efficiency, let us use two controllers 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 program is shown in Figure 1b. The combinations of configurations $1, 6$ and $4, 3$ both allow traffic to flow between tenants, violating isolation. We can formalize the isolation specification as follows:

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 could use the more user-friendly PDL). 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 1b 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 places $6, 1$, 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 1c. 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 1d. A final call to the verifier confirms that this event net satisfies both properties.

The synchronization skeleton produced in Figure 1d 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 $3, 6$. The following sections detail this synchronization synthesis approach.

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 $pkt[f \leftarrow n]$, where n is a numeric value. 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). The graph formed using the locations as nodes and links as edges is referred to as the *topology*. We fix the topology for the remainder of this section.

A *located packet* $lp = (pkt, sw, pt)$ is a packet and a location $sw{:}pt$. A *packet-trace* (*history*) 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 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 $(n_1{:}m_1, n_2{:}m_2)$. Consider a packet-trace $h = lp_0 lp_1 lp_2 \cdots lp_n$. We say that h is *allowed by* configuration C if and only if $\forall 1 \leq k \leq n.\, C(lp_{k-1}, lp_k)$, and we denote this as $h \in C$.

Petri Net Preliminaries. Our treatment of Petri nets closely follows that of Winskel [35] (Chapter 3). A *Petri net* N is a tuple (P, T, F, M_0), where P is a set of *places* (shown as circles), T is a set of *transitions* (shown as squares), $F \subseteq (P \times T) \cup (T \times P)$ is a set of *directed edges*, and M_0 is multiset of places denoting the *initial marking* (shown as dots on places). For notational convenience, we can view a multiset as a mapping from places to integers, i.e., $M(p)$ denotes the number of times place p appears in multiset M. We require that $P \neq \emptyset$, and $\forall p \in P.\, (M_0(p) > 0 \vee (\exists t \in T.\, ((p,t) \in F \vee (t,p) \in F)))$, and $\forall t \in T.\, \exists p_1, p_2 \in P.\, ((p_1, t) \in F \wedge (t, p_2) \in F)$. Given a transition t, we define its post- and pre-places as $t^\bullet = \{p \in P : (t,p) \in F\}$ and ${}^\bullet t = \{p \in P : (p,t) \in F\}$ respectively. This can be extended in the obvious way to $T'^\bullet$ and ${}^\bullet T'$, for subsets T' of 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 if and only if $\forall p \in P.\, ((p,t) \in F \implies M(p) > 0)$, and we use the notation $T' \subseteq M$ to mean that all $t \in T'$ are enabled by M. A marking M_i can transition into another marking M_{i+1} as dictated by the *firing rule*: $M_i \xrightarrow{T'} M_{i+1} \iff T' \subseteq M_i \wedge M_{i+1} = M_i - {}^\bullet T' + T'^\bullet$, where the $-/+$ operators denote multiset difference/union respectively. 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 we have $M_i \xrightarrow{\{t\}} M_j$ in the Petri net. A *trace* τ of a Petri net is a sequence $t_0 t_1 \cdots t_n$ such that there exist $M_i \xrightarrow{t_i} M_{i+1}$ in the Petri net's state graph, for all $0 \leq i \leq n$. We define $\mathit{markings}(t_0 t_1 \cdots t_n)$ to be the sequence $M_0 M_1 \cdots M_{n+1}$, where $M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \cdots \xrightarrow{t_n} M_{n+1}$ is in the state graph. 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 $\forall x \in N.\, (0 \leq M_j(x) \leq 1)$, i.e., there is no more than 1 token at each place.

Event Nets. 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 [25], an event encodes an arrival of a packet with a header matching a given predicate to a given location. A *labeled net* L is a pair (N, λ), where N is a Petri net, and λ labels each place with a network configuration, and each transition with an event. An *event net* is a labeled net (N, λ) where N is 1-safe.

Semantics of Event Nets. Given event net marking M, we denote the *global configuration* of the network $C(M)$, given as $C(M) = \bigcup_{p \in M} \lambda(p)$. Given event net $E = (N, \lambda)$, let $\mathit{Tr}(E)$ be its set of traces (the set of traces of the underlying N). Given trace τ of an event net, we use $\mathit{Configs}(\tau)$ to denote $\{C(M) : M \in \mathit{markings}(\tau)\}$, i.e., the set of global configurations reachable along that trace.

Given event net E and trace τ in $\mathit{Tr}(E)$, we define $\mathit{Traces}(E, \tau)$, the packet traces allowed by τ and E, i.e., $\mathit{Traces}(E, \tau) = \{h : \exists C \in \mathit{Configs}(\tau).\, (h \in C)\}$. Note that labeling λ is not used here—we could define a more precise semantics by specifying consistency guarantees on how information about event occurrences propagates (as in [25]), but we instead choose an overapproximate semantics, to be independent of the precise definition of events and consistency guarantees.

Distributed Implementations of Event Nets. In general, an implementation of a network program specifies the initial network configuration, and dictates how the configuration changes (e.g., in response to events). We abstract away the details, defining the semantics of an *implemented network program* Pr as the set $W(Pr)$ of *program traces*, each of which is a set of packet traces. A program trace models a full execution, captured as the packet traces exhibited by the network as the program runs. We do not model packet trace interleavings, as this is not needed for the correctness notion we define. We say that Pr *implements* event net E if $\forall tr \in W(Pr).\, \exists \tau \in \mathit{Tr}(E).\, (tr \subseteq \mathit{Traces}(E, \tau))$. Intuitively, this means that each program trace can be explained by a trace of the event net E.

We now sketch a *distributed* implementation of event nets, i.e., one in which decisions and state changes are made locally at switches (and not, e.g., at a centralized controller). In order to produce a (distributed) implementation of event net E, we need to solve two issues (both related to the definition of $\mathit{Traces}(E, \tau)$).

First, we must ensure that each packet is processed by a single configuration (and not a mixture of several). This is solved by *edge* switches—those where packets enter the network from a host. An edge switch fixes the configuration in which a packet *pkt* will be processed, and attaches a corresponding tag to *pkt*.

Second, we must ensure that for each program trace, there exists a trace of E that explains it. The difficulty here stems from the possibility of *distributed conflicts* when the global state changes due to events. For example, in an application where two different switches listen for the same event, and *only the first* switch to detect the event should update the state, we can encounter a conflict where both switches think they are first, and both attempt to update the state. One way to resolve this is by using expensive coordination to reach agreement on which was "first." Another way is to use the following constraint. 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., events sharing a common input place must be handled at the same location (*local labeled net* can be defined similarly). A local event net can be implemented without expensive coordination [25].

Theorem 1 (Implementability). *Given a local event net E, there exists a (distributed) implemented network program that implements E.*

The theorem implies that there are no packet races in the implementation, since it guarantees that each 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. We use LTL because it is a very natural language for constructing formulas that describe *traces*. For example, if we want to describe traces for which some condition φ *eventually* holds, we can construct the LTL formula $\mathbf{F}\ \varphi$, and if we want to describe traces where φ holds *at each step (globally)*, we can construct the LTL formula $\mathbf{G}\ \varphi$.

Our LTL formulas are over a single packet *pkt*, which has a special field *pkt.loc* denoting its current location. For example, the property ($pkt.loc{=}H_1 \wedge pkt.dst{=}H_2 \implies \mathbf{F}\ pkt.loc{=}H_2$) means that any packet located at Host 1 destined for Host 2 should eventually reach Host 2. Given a trace τ of an event net, we use $\tau \models \varphi$ to mean that φ holds in each configuration $C \in \mathit{Configs}(\tau)$.

For efficiency, we forbid the next operator. We have found this restricted form of LTL (usually referred to as *stutter-invariant* LTL) to be sufficient for expressing many properties about network configurations.

Processes and Synchronization Skeletons. The input to our algorithm is a set of disjoint local event nets, which we call *processes*—we can use simple pointwise-union of the tuples (denoted as $\bigsqcup$) to represent this as a single local event net $E = \bigsqcup\{E_1, E_2, \cdots, E_n\}$. Given an event net $E = ((P, T, F, M_0), \lambda)$, a *synchronization skeleton* S for E is a tuple (P', T', F', M_0'), where $P \cap P' = \emptyset$, $T \cap T' = \emptyset$, $F \cap F' = \emptyset$, and $M_0 \cap M_0' = \emptyset$, and where $((P \cup P', T \cup T', F \cup F', M_0 \cup M_0'), \lambda)$ is a labeled net, which we denote $\bigsqcup\{E, S\}$.

Deadlock Freedom and 1-Safety. We want to avoid adding synchronization which fully deadlocks any process E_i. Let $L = \bigsqcup\{E, S\}$ be a labeled net where $E = \bigsqcup\{E_1, E_2, \cdots, E_n\}$, and let P_i, T_i be the places and transitions of each E_i. We

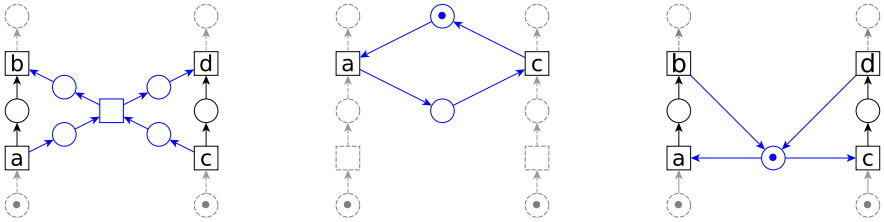


Fig. 2: Synchronization skeletons: (1) Barrier, (2) Condition Variable, (3) Mutex.

say that L is *deadlock-free* if and only if there exists a trace $\tau \in L$ such that $\forall 0 \leq i \leq n, M_j \in \mathit{markings}(\tau), t \in T_i.\, (((^\bullet t \cap P_i) \subseteq M_j) \Rightarrow (\exists M_k \in \mathit{markings}(\tau).\, (k \geq j \wedge (t^\bullet \cap P_i) \subseteq M_k)))$, i.e. a trace of L where transitions t of each E_i fire as if they experienced no interference from the rest of L. We encode this as an LTL formula, obtaining a *progress* constraint φ_{progr} for E. Similarly, we want to avoid adding synchronization which produces a labeled net that is not 1-safe. We can also encode this as an LTL constraint φ_{1safe}.

Synchronization Synthesis Problem. Given φ and local event net $E = \bigsqcup\{E_1, E_2, \cdots, E_n\}$, find a local labeled net $L = \bigsqcup\{E, S\}$ which *correctly synchronizes* E:

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

4 Fixing and Checking Synchronization in Event Nets

Our approach is an instance of the CEGIS algorithm in [17], set up to solve problems of the form $\exists L.\, ((\forall \tau \in L.\, (\phi(\tau, E, \varphi, \varphi_{1safe}))) \wedge \neg(\forall \tau \in L.\, (\tau \not\models \varphi_{progr})))$, where E, L are input/output event nets, and ϕ captures 1-3 of the above specification. Our *event net repair engine* (§4.1) performs synthesis (producing candidates for $\exists$), and our *event net verifier* (§4.2) performs verification (checking $\forall$). Algorithm 1 shows the pseudocode of our synthesizer. The function *makeProperties* produces the $\varphi_{1safe}, \varphi_{progr}$ formulas discussed in §3. The following sections describe the other components of the algorithm.

4.1 Repairing Event Nets Using Counterexample Traces

We use SMT to find synchronization constructs to fix a finite set of bugs (given as unwanted event-net traces). Figure 2 shows *synchronization skeletons* which our repair engine adds between processes of the input event net. The *barrier* prevents events b, d from firing until *both* a, c have fired, *condition variable* requires a to fire before c can fire, and *mutex* ensures that 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 a given set of bounds.

Algorithm 1: Synchronization Synthesis Algorithm

Input: local 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: local labeled net L which correctly synchronizes E

```
1  initRepairEngine(E_1, E_2, ..., E_n, X, Y, I)      // initialize repair engine (§4.1)
2  L ← 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(L, 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 L                                        // return correctly-synchronized event net
13     L ← repair(L)                                       // generate new candidate
14     if L = ⊥ then
15         return fail                                     // cannot repair
```

Repair Engine Initialization. Algorithm 1 calls *initRepairEngine*, which initializes the function symbols shown in Figure 3 and asserts well-formedness constraints. Labels in bold/blue are function symbol names, and cells are the corresponding values. For example, *Petri* is a 2-ary function symbol, and *Loc* is a 1-ary function symbol. Note that there is a separate *Ctex*, *Acc*, *Trans* for each k (where k is a counterexample index, as will be described shortly). The return type (i.e., the type of each cell) is indicated in parentheses after the name of each function symbol. For example, letting $\mathbb{B}$ denote the Boolean type $\{true, false\}$, the types of the function symbols are: $Petri : \mathbb{N}\times\mathbb{N} \to \mathbb{B}\times\mathbb{B}$, $Mark : \mathbb{N} \to \mathbb{N}$, $Loc : \mathbb{N} \to \mathbb{N}\times\mathbb{N}$, $Type : \mathbb{N} \to \mathbb{N}$, $Pair : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, $Range : \mathbb{N} \to \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, $Ctex_k : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, $Acc_k : \mathbb{N} \to \mathbb{B}$, $Trans_k : \mathbb{N} \to \mathbb{N}$, $Len : \mathbb{N} \to \mathbb{N}$ (note that *Len* is not shown in the figure).

The regions highlighted in Figure 3 are "set" (asserted equal) to values matching the input event net. In particular, $Petri(y, x)$ 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. $Mark(y)$ is set to 1 if and only if place y is marked in E. $Loc(x)$ 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.

Bound I limits how many skeletons can be used simultaneously. Each "row" i of the *Type*, *Pair*, *Range* symbols represents a single added skeleton. More specifically, $Type(i)$ 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,

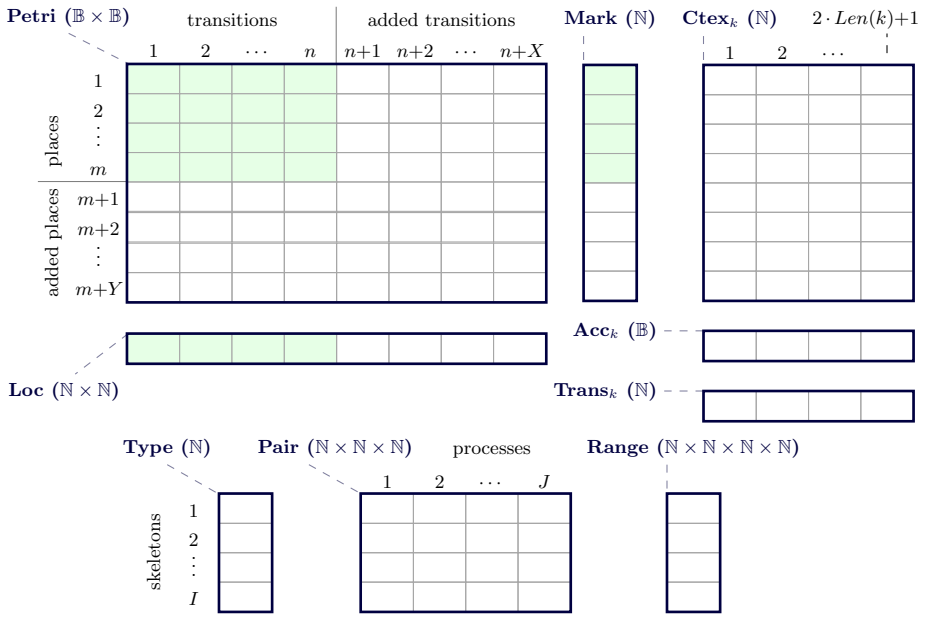


Fig. 3: SMT function symbols.

but they generalize to $j \geq 2$), and by default, J is set to the number of processes. Thus, $Pair(i, j)$ 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. The primary constraint forces the *Petri* cells to be populated as dictated by any synchronization skeletons appearing in the *Type*, *Pair*, *Range* rows. For example, given a row i where $Type(i) = 1$ (*barrier* synchronization skeleton), we would require that $Range(i) = (y_1, y_2, x_1, x_2)$, where $(y_2 - y_1) + 1 = 4$ and $(x_2 - x_1) + 1 = 1$, meaning 4 new places and 1 new transition would be reserved. Additionally, the values of *Petri* for rows y_1 through y_2 and columns x_1 through x_2 would be set to match the edges for the *barrier* construct in Figure 2.

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. In essence, 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. Let $Ctex_k(*, x)$ denote the x-th "column" of $Ctex_k$. We define $Ctex_k$ inductively as $Ctex_k(*, 1) = Mark$ and for $x > 1$, $Ctex_k(*, x)$ is equal to the marking that would be obtained if t_{x-2} were to fire in $Ctex_k(*, x-1)$. The symbol Acc_k is similarly defined as $Acc_k(1) = true$

and for $x > 1$, $Acc_k(x) \iff (Acc_k(x-1) \wedge (t_{x-2}$ is enabled in $Ctex_k(*, x-1)))$. We also assert a constraint requiring that Acc_k must become false at some point.

An important adjustment must be made 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 just as τ_k would be. We do this by instead considering the trace $\tau'_k = \epsilon\ t_0\ \epsilon\ t_1\ \cdots\ \epsilon\ t_{n-1}$ (where ϵ is a placeholder transition used only for notational convenience), and for the ϵ transitions, we set $Ctex_k(*, x)$ as if we fired any enabled *added transitions* in $Ctex_k(*, x-1)$, and for the t transitions, we update $Ctex_k(*, x)$ as described previously. More specifically, the adjusted constraints ϕ_k are as follows:

1. $Ctex_k(*, 1) = Mark$.
2. $Len(k){=}n \wedge Acc_k(1) \wedge \neg Acc_k(2 \cdot Len(k) + 1)$.
3. For $x \geq 2$, $Acc_k(x) \iff (Acc_k(x-1) \wedge (Trans_k(x){=}\epsilon \vee (Trans_k(x)$ 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 $x \geq 2$, $Trans_k(x) = \epsilon$, 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 negated acceptance constraint $\neg Acc_k(2 \cdot Len(k) + 1)$ makes sure that any synchronization generated by the SMT solver will not allow the counterexample trace τ_k 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 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*(L), 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 L. We can use *optimizing* functionality of the SMT solver (or a simple loop which asserts progressively smaller bounds for an objective function) to produce a minimal number of synchronization skeletons.

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.*

Algorithm 2: Event Net Verifier (PROMELA Model)

```
1  marked ← initMarking()                          // initial marking from input event net
2  run singlePacket, transitions                   // start both processes
3  Process singlePacket:
4      lock(); status ← 1; pkt ← pickPacket(); n ← pickHost()
5      do
6          pkt ← movePacket(pkt, marked)           // move according to current config.
7      while pkt.loc ≠ drop ∧ ¬isHost(pkt.loc)
8      status ← 2; unlock()
9  Process transitions:
10     while true do
11         lock()
12         t ← pickTransition(marked); marked ← updateMarking(t, marked)
13         unlock()
```

4.2 Checking Event Nets

We now describe $verify(L, \varphi')$ in Algorithm 1. From L, we produce a PROMELA model for LTL model checking. Algorithm 2 shows the model pseudocode, which is an efficient implementation of the semantics described in Section 3. 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 L. 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 $t \in L$, and *updateMarking* updates the marking to reflect t firing.

We ask the model checker for a *counterexample trace* demonstrating a violation of φ'. This gives the sequence of transitions t chosen by *pickTransition*. We *generalize* this sequence by removing any transitions 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 L violates φ in one of the global configurations in $Configs(\tau)$. If the verifier does not return a counterexample, then all traces of L satisfy φ.*

4.3 Overall Correctness Results

The proofs of the following theorems use Lemmas 1, 2, and Theorem 1.

Theorem 2 (Soundness of Algorithm 1). *Given E, φ, if an L is returned, then it is a local labeled net which correctly synchronizes E with respect to φ.*

Theorem 3 (Completeness of Algorithm 1). *If there exists a local labeled net $L = \bigsqcup\{E, S\}$, where $|S| \leq I$, and synchronization skeletons in S are each of the form shown in Figure 2, and S has fewer than X total transitions and fewer than Y total places, and L correctly synchronizes E, then our algorithm will return such an L. Otherwise, the algorithm returns "fail."*

benchmark	#number					time (sec.)				
	switch	iter	ctex	skip	SMT	build	verify	synth	misc	total
ex01-isolation	5	2	2	0	318	0.48	0.43	0.04	0.52	1.47
ex02-conflict	3	10	3	6	349	0.28	0.94	0.61	1.14	2.98
ex03-loop	4	2	1	0	257	0.48	0.43	0.01	0.45	1.37
ex04-composition	4	2	1	0	305	0.48	0.74	0.03	0.50	1.75
ex05-exclusive	3	5	3	3	583	5.17	4.48	0.10	1.00	10.74

Fig. 4: Performance of Examples 1-5.

5 Implementation and Evaluation

We have implemented a prototype of our synthesizer. The repair engine (§4.1) utilizes the Z3 SMT solver, and the verifier (§4.2) utilizes the SPIN LTL model checker. In this section, we evaluate our system by addressing the following:

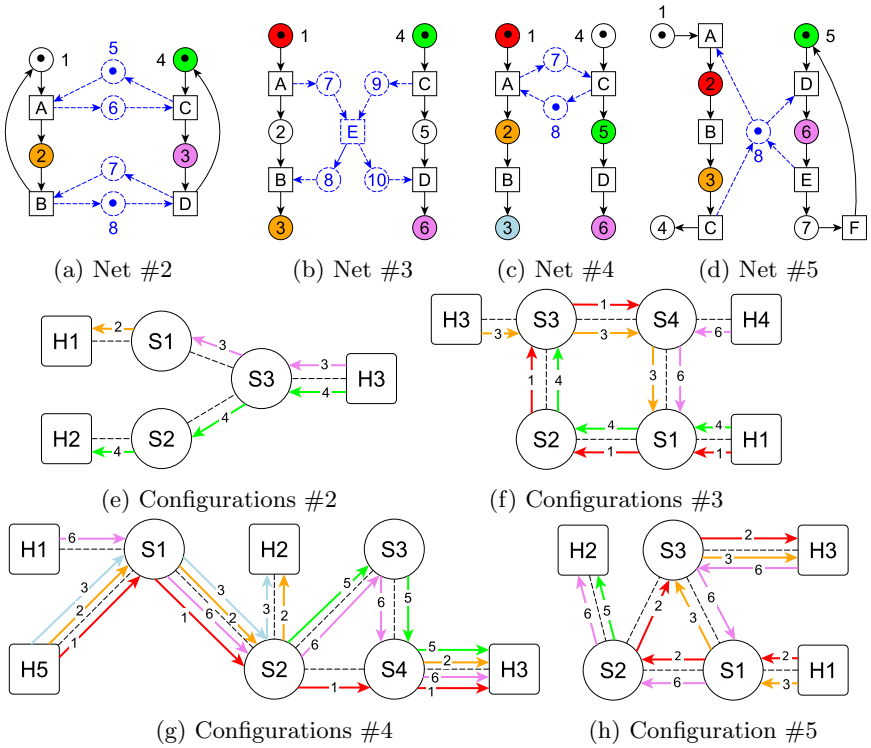
1. Can we use our approach to model a variety of real-world network programs?
2. Is our tool able to fix realistic concurrency-related bugs?
3. Is the performance of our tool reasonable when applied to real networks?

We address #1 and #2 via case studies based on real concurrency bugs described in the networking literature, and #3 by trying increasingly-large topologies for one of the studies. Figure 4 shows quantitative results for the case studies. The first group of columns denote 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 remaining columns report runtime of the SPIN verifier generation/-compilation (*build*), SPIN verification (*verify*), repair engine (*synth*), various auxiliary/initialization functionality (*misc*), and overall execution (*total*). Our experimental platform had 20GB RAM and a 3.2 GHz 4-core Intel i5-4570 CPU.

Example #1—Tenant Isolation in a Datacenter. We used our tool on the example described in Section 2. We formalize the isolation property using the following LTL properties: $\phi_1 \triangleq \mathbf{G}(loc{=}H1 \implies \mathbf{G}(loc{\neq}H4))$ and $\phi_2 \triangleq \mathbf{G}(loc{=}H3 \implies \mathbf{G}(loc{\neq}H2))$. Our tool finds the *barrier* in Figure 1d, which properly synchronizes the event net to avoid isolation violations, as described in Section 2.

Example #2—Conflicting Controller Modules. In a real bug (El-Hassany et al. [16]) encountered using the POX SDN controller, two concurrent controller modules *Discovery* and *Forwarding* made conflicting assumptions about which forwarding rules should be deleted, resulting in packet loss. Figure 5a shows a simplified version of such a 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 Forwarding is responsible for choosing a path for traffic from H3 (either the path labeled 3 or 4). In all cases, we require that traffic from H3 is not dropped.

We formalize this requirement using the LTL property $\phi_3 \triangleq \mathbf{G}(loc{=}H3 \implies \mathbf{G}(loc{\neq}drop))$. Our tool finds the *two condition variables* which properly synchronize the event net. As shown in Figure 5a, this requires the path corresponding to place 2 to be brought up *before* the path corresponding to place 3



(a) Net #2 (b) Net #3 (c) Net #4 (d) Net #5

(e) Configurations #2 (f) Configurations #3

(g) Configurations #4 (h) Configuration #5

Fig. 5: Experiments—Event Nets and Configurations.

(i.e., event C can only occur after A), and only allows it to 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. [32]), 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 such a forwarding loop can arise due to an unexpected interleaving of controller modules. In Figure 5b, the *Forwarding*/*Discovery* modules are the left/right sides respectively. Initially, *Forwarding* knows about the red (1) path in Figure 5f, 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 loops, formalizing this using the following property: $\phi_4 \triangleq \mathbf{G}(status{=}1 \implies \mathbf{F}(status{=}2))$. 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 by inserting a *barrier* (Figure 5b), preventing the unwanted combinations of configurations.

Example #4—Policy Composition. In an update scenario (Canini et al. [9]) involving *overlapping* policies, one policy enforces HTTP traffic monitoring and the other requires traffic from a particular hosts(s) to *waypoint* through a device (e.g., an intrusion detection system or firewall). Problems arise for traffic processed by the *intersection* of these policies (e.g., HTTP packets from a particular host), causing a policy violation.

Figure 5g shows such a scenario. The left process of 5c is traffic monitoring, and the right is waypoint enforcement. HTTP traffic is initially enabled along the red (1) path. Traffic monitoring 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* $H1$ *destined for* $H3$, if traffic monitoring is not set up *before* waypoint enforcement enables the path from $H1$, this traffic can circumvent the waypoint (on the $S2 \rightarrow S4$ path), violating the policy.

We can encode this specification using the following LTL properties: $\phi_6 \triangleq \mathbf{G}((pkt.type{=}HTTP \wedge pkt.loc{=}H5) \Rightarrow \mathbf{F}(pkt.loc{=}H2 \vee pkt.loc{=}H3))$ and $\phi_7 \triangleq (\neg(pkt.src{=}H1 \wedge pkt.dst{=}H3 \wedge pkt.loc{=}H3)\ \mathbf{W}\ (pkt.src{=}H1 \wedge pkt.dst{=}H3 \wedge pkt.loc{=}S3))$, where $\mathbf{W}$ is *weak until*. Our tool finds Figure 5c, which forces traffic monitoring to divert traffic *before* waypoint enforcement proceeds.

Example #5—Topology Changes during Update. Peresíni et al. [29] 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 5h, examines a similar situation where the processes in Figure 5d 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 loop-freedom property ϕ_4 from Example #3. Our tool finds a *mutex* synchronization skeleton (Figure 5d). 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 1a). Instead of the short paths between the pairs of hosts $H1, H2$ and $H3, H4$, we gathered a large set of real network topologies, and randomly selected long host-to-host paths with a single-switch intersection, corresponding to Example #1. We used datacenter FatTree topologies (e.g., Figure 7a), scaling up the *depth* (number of layers) and *fanout* (number of links per switch) to achieve a maximum size of 1088 switches, which would support a datacenter with 4096 hosts. We also used highly-connected ("small-world") graphs, such as the one shown in Figure 7b, and we scaled up the number of switches (*ring size* in the Watts-Strogatz model) to 1000. Additionally, we used 240 wide-area network topologies from the Topology Zoo dataset—as an example, Figure 7c shows the *NSFNET* topology, featuring physical nodes across the United States. The results of these experiments are shown in Figure 6, 8a, and 8b.

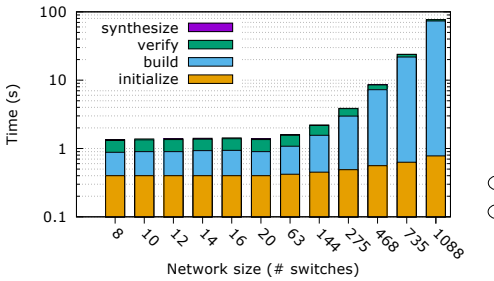
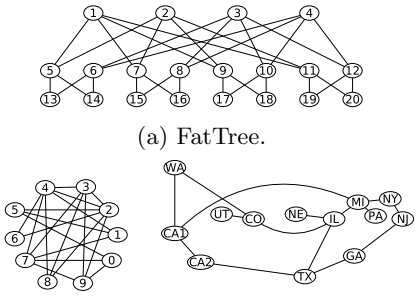
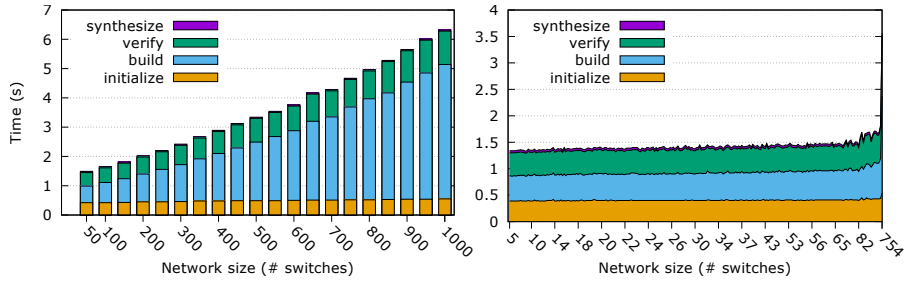


Fig. 6: Performance results: scalability of Example #1 using Fat Tree topology.



(a) FatTree. (b) Small World. (c) Topology Zoo.

Fig. 7: Example network topologies.



(a) using Small World topologies. (b) using Topology Zoo topologies.

Fig. 8: Performance results: scalability of Example #1 (continued).

6 Related Work

Synthesis for Network Programs. Yuan et al. [36] 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. [28] "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. [31] and Hojjat et al. [18] 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 [23, 37]. 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]) only allow defining static configurations, and they do not support stateful programs and concurrency constructs. Many languages [27, 20], 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. [9], which provides transactions. Both of these mechanisms are difficult to implement without damage to performance. In contrast, our solution is based on locality and synchronization synthesis, and is more fine-grained and efficiently implementable than previous approaches. It builds on and extends network event structures (NES) [25], 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], but global coordination still must be handled carefully at the language/compiler level.

Petri Net Synthesis. Ehrenfeucht et al. [15] 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 [12, 11, 3, 19] and developed foundational (complexity-theoretic) results. Synthesis from examples for Petri nets was also considered [5], and examined in the slightly different setting of *process mining* [14, 30]. 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 [24, 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. Application areas include weak memory models [26, 22], and repair of concurrency bugs [10, 34, 7, 33]. The key difference is that while these works focus on shared-memory programs, we focus on message-passing Petri-net based programs. Our model is a general framework for synthesis of synchronization where many different types of synchronization constructs can be readily described and synthesized.

7 Conclusion

We have presented an approach for synthesis of synchronization to produce network programs which satisfy correctness properties. We allow the 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 needed to remove unwanted interleavings. The advantages over previous work are that we provide (a) a language which leverages Petri nets' natural

support for concurrency, and (b) an efficient counterexample-guided algorithm for synthesizing synchronization for programs in this language.

Acknowledgments. We would like to thank Nate Foster and P. Madhusudan for fruitful discussions. This research was supported in part by the NSF under award CCF 1421752, and by DARPA under agreement FA8750-14-2-0263.

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* (2015).

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

[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.

[8] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. "P4: Programming Protocol-independent Packet Processors". In *ACM SIGCOMM CCR* (2014).

[9] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. "Software Transactional Networking: Concurrent and Consistent Policy Composition". In *HotSDN*. 2013.

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

[11] Jordi Cortadella, Michael Kishinevsky, Luciano Lavagno, and Alexandre Yakovlev. "Synthesizing Petri Nets from State-based Models". In *ICCAD*. 1995.

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

[13] Advait Abhay Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman, and Ramana Kompella. "ElastiCon: An Elastic Distributed SDN Controller". In *ANCS*. 2014.

[14] 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. LNCS. Springer, 2015, pp. 33–48.

[15] 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.

[16] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin T. Vechev. "SDNRacer: Concurrency Analysis for Software-defined Networks". In *PLDI*. 2016.

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

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

[19] Richard P. Hopkins. "Distributable Nets". In *Applications and Theory of Petri Nets*. Vol. 524. LNCS. Springer, 1990, pp. 161–187.

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

[21] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Natasha Gude, Paul Ingram, et al. "Network Virtualization in Multi-tenant Datacenters". In *NSDI* (2014).

[22] Michael Kuperstein, Martin T. Vechev, and Eran Yahav. "Automatic Inference of Memory Fences". In *FMCAD*. 2010.

[23] 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.

[24] Ulises Martínez-Araiza and Ernesto López-Mellado. "CTL Model Repair for Bounded and Deadlock Free Petri Nets". In *IFAC* (2015).

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

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

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

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

[29] Peter Peresíni, Maciej Kuzniar, Nedeljko Vasic, Marco Canini, and Dejan Kostic. "OF.CPP: Consistent Packet Processing for OpenFlow". In *HotSDN*. 2013.

[30] Hernán Ponce de León, César Rodríguez, Josep Carmona, Keijo Heljanko, and Stefan Haar. "Unfolding-Based Process Discovery". In *ATVA*. 2015.

[31] Shambwaditya Saha, Santhosh Prabhu, and P. Madhusudan. "NetGen: Synthesizing Data-plane Configurations for Network Policies". In *SOSR*. 2015.

[32] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, Hrishikesh B. Acharya, Kyriakos Zarifis, and Scott Shenker. "Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences". In *SIGCOMM*. 2014.

[33] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. "Sketching Concurrent Data Structures". In *PLDI*. 2008.

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

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

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

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