Synchronization Synthesis for Network Programs

JEDIDIAH MCCLURG, University of Colorado Boulder, USA
HOSSEIN HOJJAT, Rochester Institute of Technology, USA
PAVOL ČERNÝ, University of Colorado Boulder, USA

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 real 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 various network *switches*. The application 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 either 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 (Dixit et al. 2014; Koponen et al. 2010). 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 races of the first type *packet races*, and races of the second type *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 1(a). In the initial configuration, packets entering

2017. 2475-1421/2017/7-ART77 $?
DOI: ?

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 be sure 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 keep the following problems in mind:

- 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 to the initial forwarding rules, and suppose $C2$ 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 specification 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 declarative specification is always satisfied by any 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 have considered, we find 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 OpenState (Bianchi et al. 2014), and Kinetic (Kim et al. 2015), 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 (McClurg et al. 2016) 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 (the details are 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) a linear temporal logic (LTL) specification of paths that packets are allowed to take. For example, the network programmer can specify custom 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. 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), 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), 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.

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 readily scale to problems featuring real network topologies of 1000+ switches.

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 demonstrate the usefulness and efficiency of our approach through several real-world examples featuring real network topologies (Section 5).

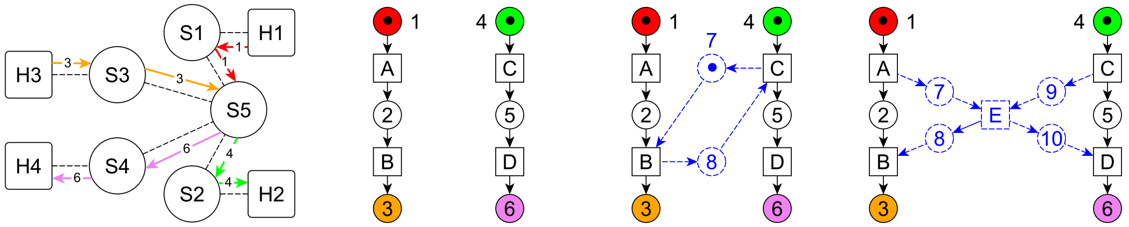
2 NETWORK PROGRAMMING USING EVENT NETS

Network programs change global forwarding behavior of the network in response to events. Recently proposed network programming languages such as OpenState (Bianchi et al. 2014) and Kinetic (Kim et al. 2015) 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 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 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 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 shown in 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(c), 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),



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

Fig. 1. Example #1

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. (2014) 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 of this is ensuring *isolation* between tenants. For example, 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 can 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 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 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 $3, 6$. The following sections describe this synchronization synthesis approach in detail.

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.

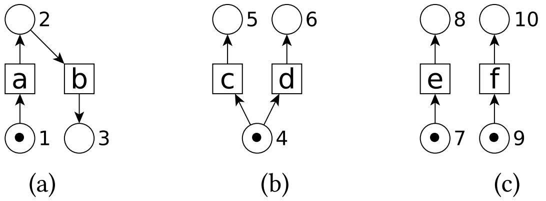
SDN Preliminaries. A *packet pkt* is a record of fields $\{f_1; f_2; \cdots; f_n\}$, where fields f represent properties such as source and destination address, protocol type, etc. The (numeric) values of fields are accessed via the notation $pkt.f$, and field updates are denoted $pkt[f \leftarrow n]$, 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).

A *located packet* $lp = (pkt, sw, pt)$ is a tuple consisting of 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 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 $(n_1{:}m_1, n_2{:}m_2)$. 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 $h \in C$. We use $h(i)$ to denote lp_i, i.e., the i-th packet in the trace, and h^i to denote the corresponding suffix of the trace, i.e., $lp_i lp_{i+1} \cdots lp_n$.

Petri Net Preliminaries. As we have seen, a Petri net is a transition system where one or more tokens can move between *places*, as dictated by *transitions*. Petri nets provide a flexible framework for concurrency that we can utilize. For example, the Petri net in Figure 2(a) shows how *sequencing* can be modeled—transition a must fire first (moving the token to place 2), before transition b can fire. Figure 2(b) shows how *conflict* can be modeled—either c can fire (moving the token to place 5), or d can fire, but not both. Figure 2(c) shows how *concurrency* can be modeled—transition e can fire (moving the token from place 7 to place 8), and f can fire independently.

Our treatment of Petri nets closely follows that of Winskel (1987) (Chapter 3). A *Petri net* 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). 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 post- and pre-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 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 (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 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



(a) (b) (c)

Fig. 2. Petri nets: (a) sequencing, (b) conflict, (c) concurrency.

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 $markings(t_0 t_1 \cdots t_n)$ 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 $\tau \triangleright N$) 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 (McClurg et al. 2016), 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 $Traces(E)$ to be the set $\{h : \exists \tau \in T(E) \exists C \in configs(\tau).(h \in C)\}$. The set $Traces(E)$ 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 (McClurg et al. 2016)), 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. Producing a *distributed* implementation for an event-driven program can be difficult. One primary difficulty is 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 could easily encounter a conflict where both switches think they are the first to detect the event, and thus both attempt to update the state. One way to resolve this is by using expensive coordination between the two switches to agree on which is "first." McClurg et al. (2016) present a different solution, which guarantees *efficient* implementations (i.e., without expensive coordination) for network programs encoded as *event structures*, when a certain *locality condition* is satisfied.

We can also make use of such a locality condition. 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. Winskel (1987) (Chapter 3) shows that a 1-safe Petri net can be "unfolded" into an *occurrence net*, which represents a restricted form of event structure. Thus, we can produce a local event structure from a local event net, which gives us the following Theorem 3.1. The details are omitted due to space constraints, but in essence, the theorem follows from Theorem 1 in (McClurg et al. 2016). 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.

THEOREM 3.1 (IMPLEMENTABILITY). *Each local event net E has an efficient distributed implementation whose single-packet traces are a subset of $Traces(E)$.*

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 linear temporal logic (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*, 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 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

$$
\begin{array}{rcll}
f & \in & \mathit{Field} & \textit{(packet field name)} \\
n & \in & \mathbb{N} & \textit{(numeric value)} \\
x, y & ::= & \mathit{pkt}.f \mid \mathit{pkt}.\mathbf{loc} \mid n \mid x + y \mid x - y \mid x * y \mid x \div y & \textit{(numeric expression)} \\
a & ::= & \mathbf{true} \mid \mathbf{false} \mid x = y \mid x > y \mid x < y \mid x \geq y \mid x \leq y & \textit{(atomic proposition)} \\
\varphi, \psi & ::= & a \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \Rightarrow \psi \mid \varphi \Leftrightarrow \psi \mid \mathbf{X}\,\varphi \mid \varphi\,\mathbf{U}\,\psi \mid \varphi\,\mathbf{R}\,\psi \mid \mathbf{G}\,\varphi \mid \mathbf{F}\,\varphi & \textit{(formula)}
\end{array}
$$

Fig. 3. LTL syntax.

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 $C \in \mathit{configs}(\tau)$.

LTL syntax is shown in Figure 3. The basic formula is an *atomic proposition*, which is either **true**, **false**, or a comparison between *numeric expressions* over the variable *pkt*. Formulas can be extended using the standard logical operators negation ($\neg\varphi$), conjunction ($\varphi \wedge \psi$), disjunction ($\varphi \vee \psi$), implication ($\varphi \Rightarrow \psi$), and equality ($\varphi \Leftrightarrow \psi$). Additionally, LTL provides the *next* operator $\mathbf{X}\,\varphi$, the *until* operator $\varphi\,\mathbf{U}\,\psi$, the *release* operator $\varphi\,\mathbf{R}\,\psi$, the *always (globally)* operator $\mathbf{G}\,\varphi$, and the *eventually (future)* operator $\mathbf{F}\,\varphi$. Given a packet-trace h and an LTL formula φ we define the notion of h satisfying the formula (denoted $h \models \varphi$) using the following recursive definition. We can extend this to a configurations by letting $C \models \varphi$ mean that all packet-traces $h \in C$ satisfy φ.

$h \models a$	$\triangleq$	$h(0) \models a$	atomic proposition a holds in the first step
$h \models \neg\varphi$	$\triangleq$	$\neg(h \models \varphi)$	φ does not hold
$h \models (\varphi \wedge \psi)$	$\triangleq$	$(h \models \varphi) \wedge (h \models \psi)$	both φ and ψ hold
$h \models (\varphi \vee \psi)$	$\triangleq$	$(h \models \varphi) \vee (h \models \psi)$	either φ or ψ holds
$h \models (\varphi \Rightarrow \psi)$	$\triangleq$	$h \models (\neg\varphi \vee \psi)$	if φ holds, then ψ holds
$h \models (\varphi \Leftrightarrow \psi)$	$\triangleq$	$h \models ((\varphi \Rightarrow \psi) \vee (\psi \Rightarrow \varphi))$	φ holds if and only if ψ holds
$h \models \mathbf{X}\,\varphi$	$\triangleq$	$h^1 \models \varphi$	φ holds at the next step
$h \models \varphi\,\mathbf{U}\,\psi$	$\triangleq$	$\exists i \geq 0 : (h^i \models \psi \wedge (\forall 0 \leq j < i : h^j \models \varphi))$	eventually ψ holds, and φ holds until ψ holds
$h \models \varphi\,\mathbf{R}\,\psi$	$\triangleq$	$\neg(\neg\varphi\,\mathbf{U}\,\neg\psi)$	ψ holds until both φ and ψ hold
$h \models \mathbf{G}\,\varphi$	$\triangleq$	$\mathbf{false}\,\mathbf{R}\,\varphi$	φ always holds
$h \models \mathbf{F}\,\varphi$	$\triangleq$	$\mathbf{true}\,\mathbf{U}\,\varphi$	eventually φ holds

Note that the above packet-traces are assumed to be *infinite*, so for the purposes of the definition, we simply consider a finite trace to be an infinite one where the last step repeats indefinitely. For efficiency purposes, we forbid the next operator (resulting in what is known as *stutter-invariant* LTL). We have found this restricted form of LTL to be sufficient for expressing real-world properties about network configurations.

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

Deadlock Freedom and 1-Safety. 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 $\forall 0 \leq i \leq n, M_j \in \mathit{markings}(\tau), t \in T_i : (({}^\bullet t \cap P_i) \subseteq M_j) \Rightarrow (\exists M_k \in \mathit{markings}(\tau) : k \geq j \wedge (t^\bullet \cap P_i) \subseteq M_k)$, i.e. a trace of 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}.

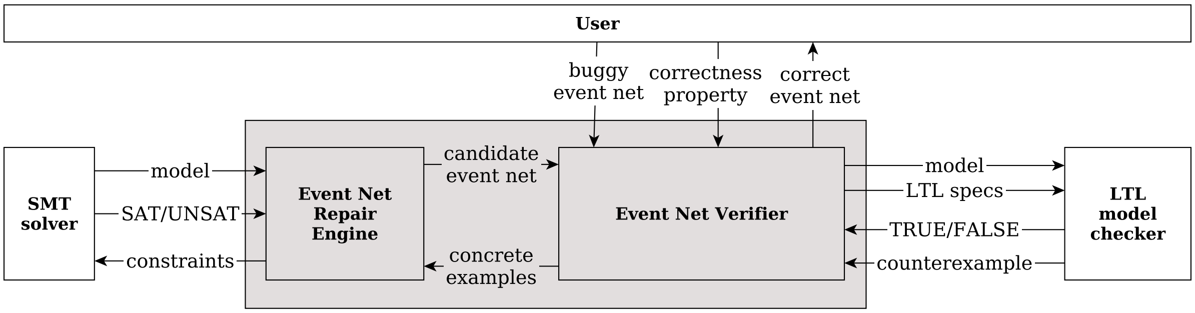


Fig. 4. Synchronization Synthesis—System Architecture

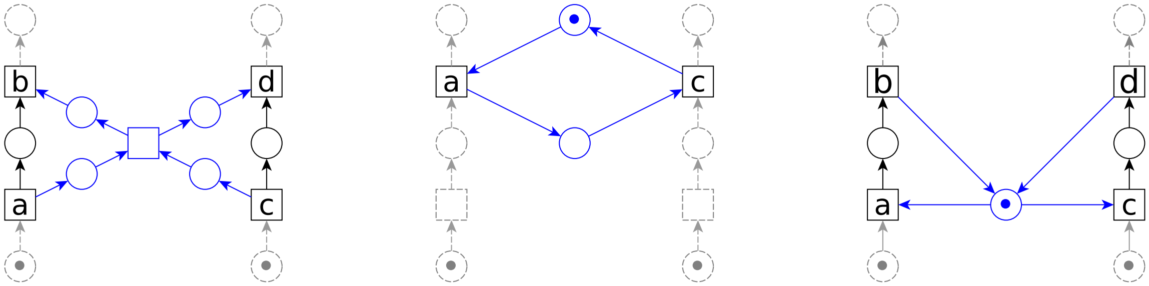


Fig. 5. Synchronization skeletons: (1) Barrier, (2) Condition Variable, (3) Mutex.

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

4 FIXING AND CHECKING SYNCHRONIZATION IN EVENT NETS

Figure 4 shows the architecture of our solution—an instance of the CEGIS algorithm in (Gulwani et al. 2011; Jha et al. 2010) 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 $\forall$).

Algorithm 1 shows the pseudocode. 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 5 shows three types of *synchronization skeletons* which our repair engine can add between the processes of the input event net E'. As 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 synthesis algorithm explores different combinations of these skeletons, up to a given set of bounds.

Algorithm 1: Synchronization Synthesis Algorithm

```
Input: event net $E = \bigsqcup\{E_1, E_2, \cdots, E_n\}$, LTL property $\varphi$, 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 $E$
initRepairEngine($E_1, E_2, \cdots, E_n, X, Y, I$)                // initialize repair engine (§4.1)
$E' \leftarrow E$
$(\varphi_{1safe}, \varphi_{progr}) \leftarrow$ makeProperties($E_1, E_2, \cdots, E_n$)
while true do
    $ok \leftarrow true$
    $props \leftarrow \{\varphi, \varphi_{1safe}, \varphi_{progr}\}$
    for $\varphi' \in props$ do
        $\tau_{ctex} \leftarrow verify(E', \varphi')$                       // check the property (§4.2)
        if $(\tau_{ctex} = \emptyset \wedge \varphi' = \varphi_{progr}) \vee (\tau_{ctex} \neq \emptyset \wedge \varphi' = \varphi_{1safe})$ then
            differentRepair()                                  // try different repair (§4.1)
            $ok \leftarrow false$
        else if $\tau_{ctex} \neq \emptyset \wedge \varphi' \neq \varphi_{progr}$ then
            assertCtex($\tau_{ctex}$)                              // record counterexample (§4.1)
            $ok \leftarrow false$
    if ok then
        return $E'$                             // return correctly-synchronized event net
    $E' \leftarrow repair(E')$                                     // generate new candidate
    if $E' = \bot$ then
        return fail                                                 // cannot repair
```

Repair Engine Initialization. Algorithm 1 calls *initRepairEngine*$(E_1, E_2, \cdots, E_n, X, Y, I)$, which "initializes" the function symbols shown in Figures 6 and 7 with the values from the input event nets, 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 *Mark* is a 1-ary function symbol. Note that there is a separate *Ctex*, *Acc*, *Trans* for each k (where k is a counterexample index). The return value type 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: $Petri : \mathbb{N} \times \mathbb{N} \to \mathbb{B} \times \mathbb{B}$, $Mark : \mathbb{N} \to \mathbb{N}$, $Loc : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$, $Type : \mathbb{N} \to \mathbb{N}$, $Pair : \mathbb{N} \to \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, $Range : \mathbb{N} \to \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$, $Len : \mathbb{N} \to \mathbb{N}$, $Ctex_k : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, $Acc_k : \mathbb{N} \to \mathbb{B}$, $Trans_k : \mathbb{N} \to \mathbb{N}$.

The regions highlighted in Figure 6 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.

The bound I limits how many synchronization 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 5 shows the skeletons for 2 processes, 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

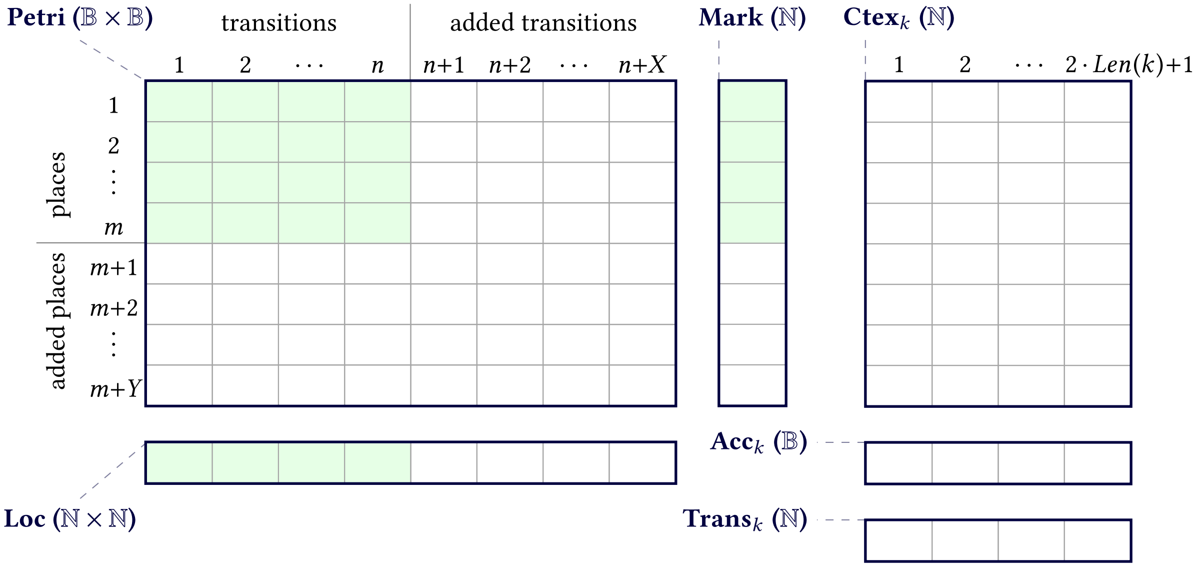

Petri ($\mathbb{B} \times \mathbb{B}$)
transitions
added transitions
1
2
···
n
n+1
n+2
···
n+X
places
1
2
⋮
m
added places
m+1
m+2
⋮
m+Y
Mark ($\mathbb{N}$)
Ctex$_k$ ($\mathbb{N}$)
1
2
···
2·Len(k)+1
Acc$_k$ ($\mathbb{B}$)
Loc ($\mathbb{N} \times \mathbb{N}$)
Trans$_k$ ($\mathbb{N}$)

Fig. 6. SMT function symbols—event net encoding.

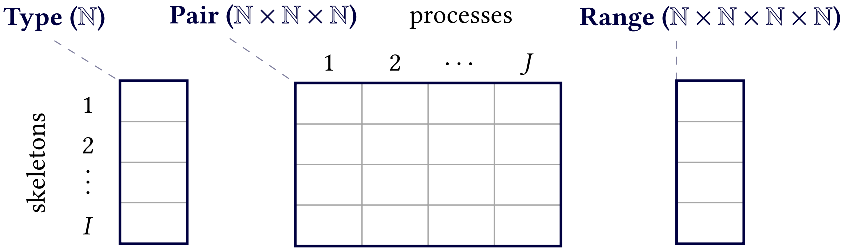

Type ($\mathbb{N}$)
Pair ($\mathbb{N} \times \mathbb{N} \times \mathbb{N}$)
processes
Range ($\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$)
1
2
···
J
skeletons
1
2
⋮
I

Fig. 7. SMT function symbols—synchronization skeletons encoding.

(*pMin*, *pMax*, *tMin*, *tMax*), where *pMin*, *pMax* reserve a range of rows in the *added places* section of Figure 6, 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 6. 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 $Range(i) = (y_1, y_2, t_1, t_2)$, where $(y_2 - y_1) + 1 = 4$ and $(t_2 - t_1) + 1 = 1$, i.e., 4 new places and 1 new transition would be reserved. Additionally, the values of *Petri* for rows y_1 through y_2 and columns t_1 through t_2 would be set to match the edges for the *barrier* construct in Figure 5. Several other constraints are captured by ϕ_{global}—due to space limitations, we will not present the full details, but the following list summarizes the high-level descriptions of the ϕ_{global} constraints:

1. For each active cell (*id*, *fst*, *snd*) in *Pair*, we require that *fst*, *snd* are from the same input process, and the events between *fst* and *snd* (inclusive) in E form a simple chain (i.e., no branching behavior). Additionally, different cells on the same row of *Pair* are from different processes, i.e., they have different *id* values.
2. Cells are in decreasing order of *id* in each row of *Pair*.
3. No two active rows of *Pair* are equal.
4. No two intervals represented in the *Range* cells are overlapping.
5. Each interval in the *Range* cells stays within the *added places/transitions* area of *Petri*.

(6) Each row of *Type* is between 0 and 3 (*no skeleton* (inactive row), or one of the 3 skeleton types respectively).
(7) Un-used places/transitions in the *added places/transitions* area of *Petri* are set to zero.
(8) As described above, interval lengths in *Range* and corresponding *Petri*/*Mark* cells are set based on *Type*.
(9) *Mark* values are between 0 and 1 (enforcing 1-safety).
(10) Two transitions having a common input place have equal corresponding values of *Loc* (enforcing locality).
(11) Each *Loc* value is a valid location in the network topology.

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. 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 $x > 1$, $Acc_k(x) \iff (Acc_k(x-1) \land (t_{x-1}$ 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.

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 $\tau_k' = 0, t_0, 0, t_1, \cdots, 0, t_{n-1}$, and for the "0" transitions, we set $Ctex_k(x)$ as if we fired any enabled *added transitions* in $Ctex_k(x-1)$, and for the t_i transitions, we update $Ctex_k(x)$ as described previously. Therefore, the adjusted constraints ϕ_k are as follows:

(1) The first column of $Ctex_k$ is equal to *Mark*.
(2) $Len(k)=n \land Acc_k(1) \land \neg Acc_k(2 \cdot Len(k) + 1)$.
(3) $Acc_k(x) \iff (Acc_k(x-1) \land (Trans_k(x)=0 \lor (Trans_k(x)$ 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 trace 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 $repair(E')$, 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'. 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 4.1 (CORRECTNESS OF THE REPAIR ENGINE). *If the SMT solver finds that* $\phi = \phi_{global} \land \phi_{skip} \land \phi_1 \land \cdots \land \phi_k$ *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)

```
$marked \leftarrow initMarking()$                          // initial marking from input event net
run $singlePacket, transitions$                            // start both processes
Process $singlePacket$:
    $lock()$                                               // acquire lock
    $status \leftarrow 1$
    $pkt \leftarrow pickPacket(); n \leftarrow pickHost()$ // nondeterministically choose packet / host
    do
        $pkt \leftarrow movePacket(pkt, marked)$  // move packet according to current configuration
    while $pkt.loc \neq drop \land \neg isHost(pkt.loc)$
    $status \leftarrow 2$                                  // processing of the packet is finished
    $unlock()$                                             // release lock
Process $transitions$:
    while $true$ do
        $lock()$                                           // acquire lock
        $t \leftarrow pickTransition(marked)$     // nondeterministically choose enabled transition
        $marked \leftarrow updateMarking(t, marked)$       // fire transition
        $unlock()$                                         // release lock
```

4.2 Checking Event Nets

This section describes the *verify*(E', φ') 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 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 4.2 (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.*

4.3 Overall Correctness Results

Finally, we characterize the correctness of our synchronization synthesis approach The proofs of the following theorems use Lemmas 4.1 and 4.2, as well as Theorem 3.1.

Theorem 4.3 (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 4.4 (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."*

benchmark	#number					time (sec.)				
	switch	iter	ctex	skip	SMT	build	verify	synth	misc	total
ex01-isolation	5	2	2	0	318	0.48	0.45	0.04	0.50	1.47
ex02-conflict	3	20	3	16	359	0.29	1.86	0.68	1.98	4.81
ex03-loop	4	3	2	0	318	0.50	0.72	0.06	0.60	1.88
ex04-composition	4	2	1	0	305	0.49	0.80	0.03	0.50	1.82
ex05-exclusive	3	2	1	0	448	5.00	0.60	0.03	0.56	6.20

Fig. 8. Performance of Examples 1-5.

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 when applied to real networks?

We address #1 and #2 via case studies based on real concurrency bugs described in the networking literature, and we address #3 by choosing one of these examples and trying different increasingly-large topologies. Figure 8 shows performance results and quantitative metrics for the case studies. 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/initialization functionality (*misc*), and overall execution time (*total*). We ran all experiments on a machine with 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 ϕ_1 and ϕ_2.

$$\phi_1 \triangleq \mathbf{G}(loc{=}H1 \implies \mathbf{G}(loc{\neq}H4))$$

$$\phi_2 \triangleq \mathbf{G}(loc{=}H3 \implies \mathbf{G}(loc{\neq}H2))$$

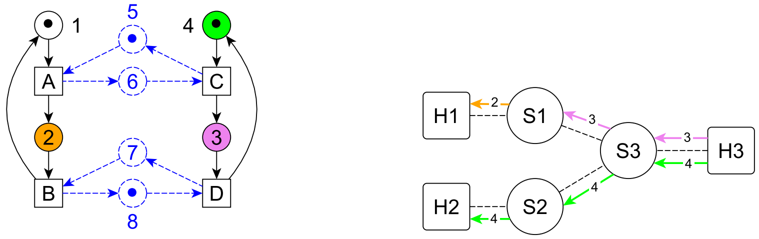
Our tool finds the *barrier* shown in Figure 1(d), which properly synchronizes the event net to avoid isolation violations, as described in Section 2.

Example #2—Conflicting Controller Modules. In a real bug (El-Hassany et al. (2016)) 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 9(a) 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 no traffic from H3 is dropped.

We formalize this requirement using the following LTL property ϕ_3.

$$\phi_3 \triangleq \mathbf{G}(loc{=}H3 \implies \mathbf{G}(loc{\neq}drop))$$

Our tool finds the *two condition variables* which properly synchronize the event net to avoid packet loss. As shown in Figure 9(a), this requires the path corresponding to place 2 to be brought up *before* the path corresponding to

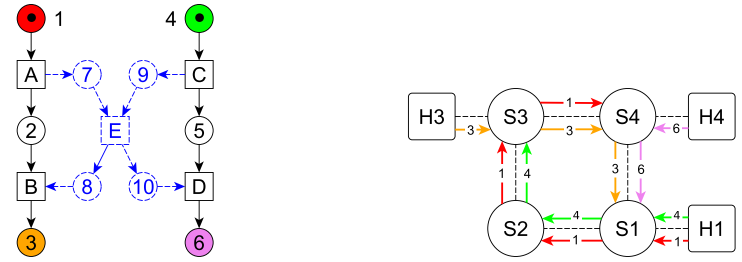


(a) Event net. (b) Configurations.

Fig. 9. Inputs for Example #2.

place 3 (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. (2014)) 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 10(a), the *Forwarding*/*Discovery* modules are the left/right sides respectively. Initially, *Forwarding* knows about the red (1) path in Figure 10(b), 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.



(a) Event net. (b) Configurations.

Fig. 10. Inputs for Example #3.

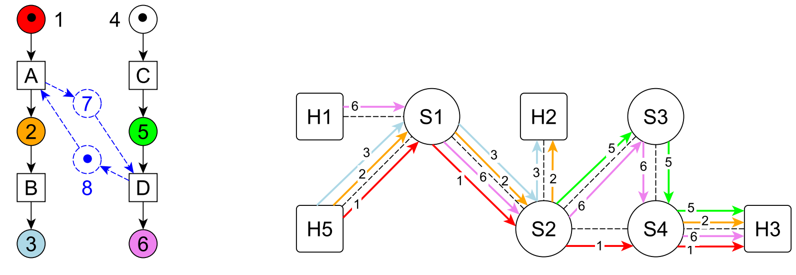
We wish to disallow such forwarding loops, formalizing this requirement using the following LTL property ϕ_4.

$$\phi_4 \triangleq \mathbf{G}(status{=}1 \implies \mathbf{F}(status{=}2))$$

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 requirement by inserting a *barrier*, as in Figure 10(a), preventing the unwanted combinations of configurations.

Example #4—Policy Composition. In an update scenario (Canini et al. (2013)) 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 11(b) shows such a conflict. The left process of 11(a) is traffic monitoring, and the right process is waypoint enforcement. HTTP traffic is initially enabled along the red (1) path. Traffic monitoring then intercepts



(a) Event net. (b) Configurations.

Fig. 11. Inputs for Example #4.

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* the waypoint enforcement enables the path from $H1$, then this traffic can circumvent the waypoint (on the $S2 \rightarrow S4$ path), violating the policy.

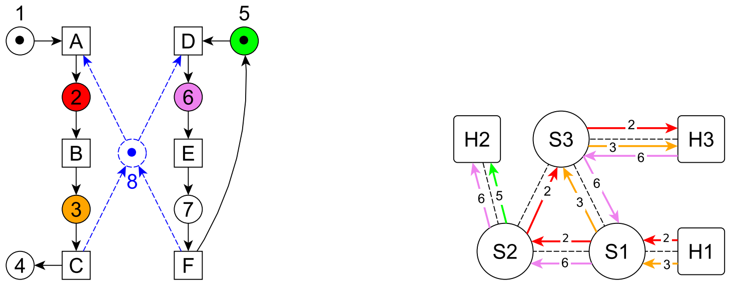
We can encode this specification using the following LTL properties ϕ_6 and ϕ_7.

$$\phi_6 \triangleq \mathbf{G}((pkt.type{=}HTTP \land pkt.loc{=}H5) \Rightarrow \mathbf{F}(pkt.loc{=}H2 \lor pkt.loc{=}H3))$$

$$\phi_7 \triangleq (\neg(pkt.src{=}H1 \land pkt.dst{=}H3 \land pkt.loc{=}H3)\ \mathbf{W}\ (pkt.src{=}H1 \land pkt.dst{=}H3 \land pkt.loc{=}S3))$$

Our tool finds Figure 11(a), which forces traffic monitoring to divert traffic *before* waypoint enforcement proceeds.

Example #5—Topology Changes during Update. Peresíni et al. (2013) 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 12(b), examines a similar situation where the processes in Figure 12(a) 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 $S1, S2, S3$).



(a) Event net. (b) Configurations.

Fig. 12. Inputs for Example #5.

We use the loop-freedom property ϕ_4 discussed in Example #3. Our tool finds the *mutex* synchronization skeleton shown in Figure 12(a). 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 gathered a large set of real network topologies, and randomly selected long host-to-host

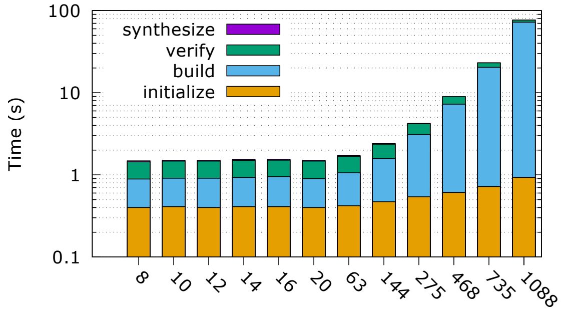
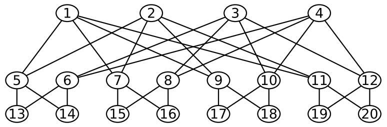
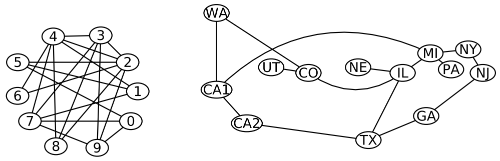


Fig. 13. Performance results: scalability of Example #1 using Fat Tree topology.

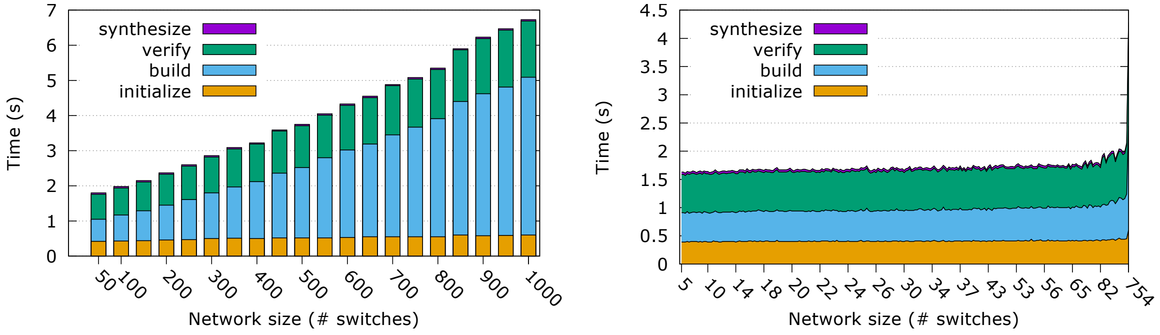


(a) FatTree.



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

Fig. 14. Example network topologies.



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

Fig. 15. Performance results: scalability of Example #1 (continued).

paths with a single-switch intersection, corresponding to Example #1. We used datacenter FatTree topologies (e.g., Figure 14a), 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 14b, and we scaled up the number of switches (*ring size* in the Watts-Strogatz small-world model) to 1000. Additionally, we used 240 wide-area network topologies from the Topology Zoo dataset—as an example, Figure 14c shows the *NSFNET* topology, featuring physical nodes across the United States. The results of these experiments are shown in Figure 13, 15a, and 15b. We note in all of the experiments that the SMT component scales much more readily than building/running SPIN verifiers.

6 RELATED WORK

Network Repair and Network Update Synthesis. Saha et al. (2015) and Hojjat et al. (2016) 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 must repair a network *program.*

A *network update* is a simple network program—a situation where the global forwarding state of the network must change. In the networking community, there are several proposals for packet- and flow-level consistency properties that should be preserved during an update. For example, per-packet and per-flow consistency (Mahajan and Wattenhofer 2013; Reitblatt et al. 2012), and inter-flow consistency (Liu et al. 2015). Many approaches solve the problem with respect to different variants of these consistency properties (Hong et al. 2013; Jin et al. 2014; Katta et al. 2013; Ludwig et al. 2014; McClurg et al. 2015; Zhou et al. 2015). In contrast, we provide a new language for succinctly describing how multiple updates can be composed, as well as an approach for synthesizing a composition which respects customizable LTL properties over packet traces.

Concurrent Programming for Networks. Dudycz et al. (2016) present an algorithm to *compose* network updates correctly with respect to loop freedom, and show that the problem of *optimally* doing so is NP-hard. Beyond network updates, there has been work on composing network programs. For example, Pyretic has a programming language which allows sequential/parallel composition of static policies—dynamic behavior can be obtained via a sequence of policies (Monsanto et al. 2013). NetKAT is a mathematical formalism and compiler which also allows composition of static policies (Anderson et al. 2014; Smolka et al. 2015). CoVisor is a hypervisor that allows multiple controllers to run concurrently (sequential or parallel composition). It can incrementally update the configuration based on intercepted messages from controllers, and does not need to recompile the full composed policy (Jin et al. 2015). The PGA system addresses the issue of how to handle distributed conflicts, via customizable constraints between different portions of the policies, allowing them to be composed correctly (Prakash et al. 2015). Bonatti et al. (2000) present an algebra for properly composing access-control policies. Canini et al. (2013) use an approach based on software transactional networking to handle conflicts. We deal with conflicts automatically, by producing *local event nets*.

Handling persistent *state* properly in network programming is a difficult problem. Although basic support is provided by switch-level mechanisms for stateful behavior (Bianchi et al. 2014; Bosshart et al. 2014; Sivaraman et al. 2016), global coordination still needs to be handled carefully at the language/compiler level. FAST (Moshref et al. 2014), OpenState (Bianchi et al. 2014), and Kinetic (Kim et al. 2015) provide a finite-state-machine-based approach to stateful network programming. Arashloo et al. (2016) present SNAP, a high-level language for writing network programs. SNAP has a language with support for sequential/parallel composition of stateful policies, as well as built-in features beyond what we provide (such as atomic blocks). However, none of these approaches examine how to avoid/handle (or even analyze) distributed conflicts. McClurg et al. (2016) present an approach which formalizes event-driven network programs using event structures, and show how to deal with distributed conflicts. We extend this to a flexible model which has a more natural notion of loops, while retaining the ability to utilize the consistency properties presented there. We also present a synchronization synthesis framework that helps users properly compose several such structures into a single correct network program.

Synthesis/Verification of Concurrent Network Programs. Padon et al. (2015) show how to "decentralize" a network program to work properly on distributed switches. Our work on the other hand takes an improperly-decentralized program and inserts the necessary synchronization to make it correct. El-Hassany et al. (2016) present SDNracer, a tool for discovering concurrency bugs in network programs. Our work instead seeks to repair a buggy concurrent network program to make it satisfy a high-level correctness property. Yuan et al. (2015) present NetEgg, pioneering the approach of using examples to write network programs. Similar to our event net repair engine, they produce a policy compatible with a set of finite traces. However, NetEgg does not support negative examples, limiting its ability to rule out incorrect interleavings. Additionally, in contrast with our SMT-based strategy, NetEgg uses a backtracking search which may limit its scalability when applied to large real-world networks.

Petri Net Synthesis. Ehrenfeucht and Rozenberg (1990) 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. Desel and Reisig (1996) present an algorithm for synthesizing *all* nets isomorphic to a given DFA, in order to find "small" ones. Cortadella et al. (1995) produce elementary nets, minimize the number of places, and use label splitting when the region-based synthesis method fails. Badouel et al. (1997) show that synthesizing *elementary nets* (essentially 1-safe Petri nets without self-loops) is NP-complete.

For *general* Petri nets, the synthesis problem is polynomial-time solvable. Badouel et al. (1995) present a polynomial algorithm (based on linear programming) for pure (no self-loops) bounded nets. Badouel et al. (2002)

present a polynomial-time linear-algebra-based algorithm for synthesizing *distributable nets* (Hopkins 1990). Distributable nets are *local* like our event nets, but not necessarily 1-safe.

The above work is not directly applicable in our context because the definition of "net synthesis" is very different than what is needed for our repair engine. A more closely-related type of synthesis is presented by Bergenthum et al. (2008) and Cabasino et al. (2007), who synthesize minimal Petri nets consistent with positive and positive/negative examples respectively. Our programming model relies on 1-safe Petri nets, so we cannot directly apply these approaches either.

Process Mining. Process mining looks at an *event log* and produces an event structure which generalizes the traces in the log (Dumas and García-Bañuelos 2015). This approach can also synthesize a Petri net—Ponce de León et al. (2015) use the log to produce an event structure, and then generalize the event structure by "folding" it into an equivalent bounded net via SMT, using negative traces to constrain the amount of generalization. This is different than our approach in that (1) their generalization adds potentially *more* behaviors not seen in the positive traces, meaning it would not work for synthesis of synchronization constructs, and (2) they have a strong well-formedness assumption on negative examples, while we allow arbitrary traces.

A related area is *process enhancement* (repair) (Fahland and van der Aalst 2012). This computes a minimal number of changes to the original Petri net such that certain properties are satisfied (such as agreement with the event log). Quality metrics are used to maintain closeness to the original model, and the degree of conformance with the event log. For example, Martínez-Araiza and López-Mellado (2015) use a backtracking algorithm that modifies the Petri net while checking a CTL property, producing a repaired Petri net which satisfies the property. This changes the semantics of the Petri net, while we want *semantics-preserving transformations*. In other words, we do not generate arbitrary repairs—we *restrict* behaviors by adding new events/places (synchronization skeletons). Basile et al. (2015) preserve the semantics of the original (buggy) Petri net, but they are restricted to the context of *time petri nets* (they modify the timing, not the net structure). These do not correspond well to network programs, because careful timing can require expensive synchronization/buffering in the network.

Automata Learning. Our approach is essentially an abstract learning framework (Löding et al. 2016), where our event net repair engine is the learner. Automata learning is conceptually similar, producing a DFA instead of a Petri net, and has been used for verification/synthesis (Vardhan et al. 2004). *Offline* approaches to automata learning (such as RPNI (Oncina and García 1992)) produce an automaton which agrees with a set of labeled (positive/negative) example traces. *Online* approaches such as L^* (Angluin 1987) actively pose queries to the user asking whether certain traces are contained in the target language. For our purposes, an offline approach is desirable, since we wish to provide a fully automatic tool. Learning a minimal DFA from positive/negative examples is known to be an NP-complete problem (Gold 1978), but under various restrictions on the example traces, a polynomial algorithm can be obtained (Dupont 1996; Dupont et al. 1994). It would be interesting to investigate an RPNI-style formalization for learning Petri nets, although (1) in our case, we would need to *modify a given Petri net in a minimal way, such that a set of negative traces are rejected*, rather than producing a *general* Petri net from a set of positive/negative examples, and (2) it is possible that there is not an efficient solution, due to the NP-completeness of both DFA learning and elementary net synthesis. Additionally, it would be interesting to examine the usefulness of an online approach for learning Petri nets (e.g. (Esparza et al. 2010)) in our context, but both of these directions are left for future work.

Synthesis/Repair for Synchronization. Emerson and Clarke (1982) use a decision procedure for satisfiability of CTL to synthesize "synchronization skeletons." The processes themselves are specified using CTL, and the synchronization skeleton is extracted from the model. Chatterjee et al. (2013) present complexity results for distributed LTL synthesis, i.e., synthesizing a set of processes such that their behavior satisfies an LTL specification. Our approach is a similar idea, but we exploit the speed of SMT solvers on quantifier-free linear integer arithmetic.

PSketch (Solar-Lezama et al. 2008) extends Sketch to synthesize concurrent programs. They add constructs for statement reordering, as well as concurrency primitives for forking threads, atomic sections, etc. The SAT-based synthesis component produces a candidate program which avoids a finite set of buggy traces, and a Spin verifier checks that all interleavings of the candidate are correct, and if not, a counterexample representing a new buggy trace is returned. This is conceptually similar to our approach, but PSketch encodes all possible programs as a SAT formula, while we utilize the SMT solver to *add a repair* to the original program.

There are various other SAT/SMT-based approaches, such as synthesis of memory-order (Meshman et al. 2015) and insertion of fences (Kuperstein et al. 2010) in a relaxed memory model, instruction reordering (Černý et al. 2013; Cerný et al. 2014), atomic section insertion (Bloem et al. 2014; Vechev et al. 2010), etc. Additionally, there are program-analysis-based approaches which look at a bug report, and perform semantics-preserving reorderings, thread join/lock, etc., producing a *patch* to fix the bug (Jin et al. 2011, 2012; Liu et al. 2016). Raychev et al. (2013) present an approach to "determinize" a concurrent program by synthesizing order relationships between statements. In our approach, we model programs as Petri nets, resulting in a general framework for synthesis of synchronization where many different types of synchronization constructs can be readily described and synthesized.

Synthesis from Examples. Programming by examples is an active research area, and has been applied in many contexts where individual behaviors are easier to specify than full programs (Gulwani 2011; Polozov and Gulwani 2015). Our approach uses a new example-based Petri repair engine to synthesize programs which respect certain high-level properties.

Transit (Alur et al. 2015; Udupa et al. 2013) allows programmers to synthesize a distributed program (protocol) using both concrete and symbolic partial examples of the program's desired behavior (concolic snippets). This approach uses CEGIS—the synthesizer enumeratively "fills in" program expressions, and uses an SMT solver to check that the resulting candidate protocol agrees with the concolic examples (and invariants). If not, a counterexample is provided as a new concrete example. Our approach is similar, except that rather than enumeration, we use an SMT solver (guided by negative traces) to produce a candidate, and our candidates are Petri nets rather than program expressions.

7 DISCUSSION AND FUTURE WORK

In this section, we briefly describe some limitations of our approach, and suggest several interesting directions for future research.

- In our algorithm, we do not acquire a counterexample for 1-safety or deadlock-freedom violations, meaning that in these cases, we instruct the repair engine to assert a clause that prevents it from revisiting the particular set of synchronization skeletons. Although this works well in our experiments, it could potentially be problematic in situations where many such violations are encountered by the CEGIS loop, since the SMT solver would build up a large set of assertions which do not "guide" the search as effectively as an actual counterexample trace would. We plan to investigate ways to avoid such issues.
- In this paper, we consider *single-packet properties*, meaning we cannot precisely reason about behavior arising from several different interacting packets, such as a stateful firewall. Future research is needed to determine (1) what is the right formalism (such as LTL) for specifying multi-packet properties, and (2) how the verifier can efficiently check such properties when events become "parameterized" over packets.
- Although different from our intended use case, we may be able to use our approach as a solution to the *update synthesis* problem, by modeling each switch/rule update as a two-place event net. Instead of producing a total order on switch/rule updates (as in (McClurg et al. 2015)), we could essentially synthesize a partial order, by inserting only the synchronization skeletons needed to avoid violations of the LTL property. Investigating the usability of our synthesizer in this context is left for future work.

- As seen in the experiments, the verification component is the most expensive piece of the system. We expect that this can be mitigated by improving 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.
- It would be interesting to investigate the efficiency of our synchronization synthesis approach if the repair engine were allowed to add *arbitrary* sets of places, transitions, and edges, rather than choosing from our limited (yet customizable) set of synchronization skeletons, but this is left for future work.
- We are interested in determining whether there is a clean RPNI-like algorithm specialized to event nets, although we expect that such an algorithm would have a high theoretical complexity, making it impractical in the context of a CEGIS algorithm like the one described in this paper.

8 CONCLUSION

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 counterexample-guided algorithm for synthesizing synchronization for programs in this language.

REFERENCES

Rajeev Alur, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. 2015. Automatic Completion of Distributed Protocols with Symmetry. *CAV* (2015).

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

D. Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75, 2 (1987), 87–106.

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

Eric Badouel, Luca Bernardinello, and Philippe Darondeau. 1995. Polynomial Algorithms for the Synthesis of Bounded Nets. In *TAPSOFT (Lecture Notes in Computer Science)*, Vol. 915. Springer, 364–378.

Eric Badouel, Luca Bernardinello, and Philippe Darondeau. 1997. The Synthesis Problem for Elementary Net Systems is NP-Complete. *Theor. Comput. Sci.* 186, 1-2 (1997), 107–134.

Eric Badouel, Benoît Caillaud, and Philippe Darondeau. 2002. Distributing Finite Automata Through Petri Net Synthesis. *Formal Asp. Comput.* 13, 6 (2002), 447–470.

F. Basile, P. Chiacchio, and J. Coppola. 2015. Model repair of Time Petri Nets with temporal anomalies. *IFAC-PapersOnLine* 48, 7 (2015), 85 – 90. 5th {IFAC} International Workshop on Dependable Control of Discrete SystemsDCDS 2015.

Robin Bergenthum, Jörg Desel, Robert Lorenz, and Sebastian Mauser. 2008. Synthesis of Petri Nets from Finite Partial Languages. *Fundam. Inform.* 88, 4 (2008), 437–468.

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

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

Piero A. Bonatti, Sabrina De Capitani di Vimercati, and Pierangela Samarati. 2000. A modular approach to composing access control policies. In *ACM Conference on Computer and Communications Security*. ACM, 164–173.

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

Maria Paola Cabasino, Alessandro Giua, and Carla Seatzu. 2007. Identification of Petri Nets from Knowledge of Their Language. *Discrete Event Dynamic Systems* 17, 4 (2007), 447–474.

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

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

Pavol Cerný, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, and Thorsten Tarrach. 2014. Regression-Free Synthesis for Concurrency. In *CAV (Lecture Notes in Computer Science)*, Vol. 8559. Springer, 568–584.

Krishnendu Chatterjee, Thomas A Henzinger, Jan Otop, and Andreas Pavlogiannis. 2013. Distributed Synthesis for LTL Fragments. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*. IEEE, 18–25.

Jordi Cortadella, Michael Kishinevsky, Luciano Lavagno, and Alexandre Yakovlev. 1995. Synthesizing Petri nets from state-based models. In *ICCAD*. IEEE Computer Society / ACM, 164–171.

Jörg Desel and Wolfgang Reisig. 1996. The Synthesis Problem of Petri Nets. *Acta Inf.* 33, 4 (1996), 297–315.

Advait Abhay Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman, and Ramana Kompella. 2014. ElastiCon: An Elastic Distributed Sdn Controller. In *ANCS*. 12.

Szymon Dudycz, Arne Ludwig, and Stefan Schmid. 2016. Can't Touch This: Consistent Network Updates for Multiple Policies. In *DSN*. IEEE Computer Society, 133–143.

Marlon Dumas and Luciano García-Bañuelos. 2015. Process Mining Reloaded: Event Structures as a Unified Representation of Process Models and Event Logs. In *Petri Nets (Lecture Notes in Computer Science)*, Vol. 9115. Springer, 33–48.

Pierre Dupont. 1996. Incremental Regular Inference. In *Grammatical Interference: Learning Syntax from Sentences*. Springer, 222–237.

Pierre Dupont, Laurent Miclet, and Enrique Vidal. 1994. What is the search space of the regular inference? In *Grammatical Inference and Applications*. Springer, 25–37.

Andrzej Ehrenfeucht and Grzegorz Rozenberg. 1990. Partial (Set) 2-Structures. Part II: State Spaces of Concurrent Systems. *Acta Inf.* 27, 4 (1990), 343–368.

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

E. Allen Emerson and Edmund M. Clarke. 1982. Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. *Sci. Comput. Program.* 2, 3 (1982), 241–266.

Javier Esparza, Martin Leucker, and Maximilian Schlund. 2010. Learning Workflow Petri Nets. In *Petri Nets (Lecture Notes in Computer Science)*, Vol. 6128. Springer, 206–225.

Dirk Fahland and Wil M. P. van der Aalst. 2012. Repairing Process Models to Reflect Reality. In *BPM (Lecture Notes in Computer Science)*, Vol. 7481. Springer, 229–245.

E. Mark Gold. 1978. Complexity of Automaton Identification from Given Data. *Information and Control* 37, 3 (1978), 302–320.

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. *POPL* (2011).

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

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

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

Richard P. Hopkins. 1990. Distributable nets. In *Applications and Theory of Petri Nets (Lecture Notes in Computer Science)*, Vol. 524. Springer, 161–187.

Susmit Jha, Sumit Gulwani, Sanjit Seshia, Ashish Tiwari, and others. 2010. Oracle-guided Component-based Program Synthesis. *ICSE* (2010).

Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated atomicity-violation fixing. In *PLDI*. ACM, 389–400.

Guoliang Jin, Wei Zhang, and Dongdong Deng. 2012. Automated Concurrency-Bug Fixing. In *OSDI*. USENIX Association, 221–236.

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

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

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

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

Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Natasha Gude, Paul Ingram, and others. 2014. Network Virtualization in Multi-tenant Datacenters. *NSDI* (2014).

Teemu Koponen, Martín Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. 2010. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*. USENIX Association, 351–364.

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

Haopeng Liu, Yuxi Chen, and Shan Lu. 2016. Understanding and generating high quality patches for concurrency bugs. In *SIGSOFT FSE*. ACM, 715–726.

Weijie Liu, Rakesh B Bobba, Sibin Mohan, and Roy H Campbell. 2015. Inter-Flow Consistency: Novel SDN Update Abstraction for Supporting Inter-Flow Constraints. *NDSS* (2015).

Christof Löding, P. Madhusudan, and Daniel Neider. 2016. Abstract Learning Frameworks for Synthesis. In *TACAS (Lecture Notes in Computer Science)*, Vol. 9636. Springer, 167–185.

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

Ratul Mahajan and Roger Wattenhofer. 2013. On Consistent Updates in Software Defined Networks. In *SIGCOMM*.

Ulises Martínez-Araiza and Ernesto López-Mellado. 2015. {CTL} Model Repair for Bounded and Deadlock Free Petri Nets. *IFAC-PapersOnLine* 48, 7 (2015), 154 – 160. 5th {IFAC} International Workshop on Dependable Control of Discrete SystemsDCDS 2015.

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

Jedidiah McClurg, Hossein Hojjat, Nate Foster, and Pavol Cerny. 2016. Event-driven Network Programming. http://arxiv.org/abs/1507.07049, *PLDI* (2016).

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

Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. 2013. Composing Software Defined Networks. *NSDI* (2013).

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

José Oncina and Pedro García. 1992. Identifying Regular Languages in Polynomial Time. *Advances in Structural and Syntactic Pattern Recognition* (1992).

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

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

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. *OOPSLA* (2015).

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

Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. 2015. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *SIGCOMM*. ACM, 29–42.

Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2013. Automatic Synthesis of Deterministic Concurrency. In *SAS (Lecture Notes in Computer Science)*, Vol. 7935. Springer, 283–303.

Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for Network Update. *SIGCOMM* (2012).

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

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. 2014. Troubleshooting blackbox SDN control software with minimal causal sequences. In *SIGCOMM*. ACM, 395–406.

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

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

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

Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. 2013. Transit: Specifying Protocols with Concolic Snippets. *PLDI* (2013).

Abhay Vardhan, Koushik Sen, Mahesh Viswanathan, and Gul Agha. 2004. Learning to Verify Safety Properties. In *ICFEM (Lecture Notes in Computer Science)*, Vol. 3308. Springer, 274–289.

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

Glynn Winskel. 1987. *Event Structures*. Springer.

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

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