From Schedules to Programs — Reimagining Networking Infrastructure for Future Cyber-Physical Systems

MD KOWSAR HOSSAIN, University of Iowa, USA
RYAN BRUMMET, University of Iowa, USA
OCTAV CHIPARA, University of Iowa, USA
TED HERMAN, University of Iowa, USA
STEVE GODDARD, University of Iowa, USA

Future cyber-physical systems will require higher capacity, meet more stringent real-time requirements, and adapt quickly to a broader range of network dynamics. However, the traditional approach of using fixed schedules to drive the operation of wireless networks has inherent limitations that make it unsuitable for these systems. As an alternative, we propose to replace schedules with domain-specific programs that coordinate the operation of the network. Our idea is that nodes in the network will run automatically generated programs that make informed decisions about flows at run time rather than using an a priori fixed schedule. We will sketch a domain-specific language that uses this additional flexibility to increase network capacity significantly. Furthermore, the constructed programs are also sufficiently simple to efficiently analyze key performance metrics such as flow response time and reliability. We conclude with future research directions.

CCS Concepts: • **Networks** → **Network dynamics**; **Network reliability**; **Cyber-physical networks**.

Additional Key Words and Phrases: Wireless networks, real-time wireless networks, reliability, software synthesis

ACM Reference Format:
Md Kowsar Hossain, Ryan Brummet, Octav Chipara, Ted Herman, and Steve Goddard. 2021. From Schedules to Programs — Reimagining Networking Infrastructure for Future Cyber-Physical Systems. In *8th International Conference on Networking, Systems and Security (8th NSysS 2021), December 21–23, 2021, Cox's Bazar, Bangladesh.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3491371.3491387

1 INTRODUCTION

Wireless networks provide critical infrastructure for the next generation of cyber-physical systems (CPS). Examples of such systems include smart factories [8], automated warehouses [10], and clinical healthcare decision-support systems [9]. The basic requirement of these applications is the need to support communication between sensors, controllers, and actuators as part of feedback control loops typical of CPS. Unlike traditional wireless networks, CPS infrastructure must provide real-time and reliable performance. Meeting this requirement is challenging due to the limited wireless resources available and significant network dynamics common to industrial settings. Network dynamics include fluctuations in the quality of wireless links, variations in the workload of applications, and topology changes induced by node failures or mobility [7, 12].

A predictable wireless network is one where it is possible to check its safety and performance requirements under explicit assumptions about variations in network dynamics. A range of tools such as model checkers [16] and satisfiability solvers [21] aim to verify whether a system's requirements are met. Such tools effectively analyze classical safety

Manuscript submitted to ACM

properties, though computational complexity is a concern even for moderate-size networks. Moreover, it can be challenging to express network performance properties (e.g., latency, throughput, reliability, buffer utilization, and energy consumption) in the form needed by the analysis tools. One contribution of recent research and standardization efforts is to accommodate the limitations of tools by constraining the network architecture and its programming model (e.g., WirelessHART, ISA100, TSCH [2]). Simplifying network architecture and its primitives helps us reason about the behavior of the protocols; such simplification might also enable us to leverage existing analysis tools to analyze even performance properties.

As an illustrative example, industrial standards such as WirelessHART and ISA100 use a centralized control plane managed by a network manager. An application expresses its communication needs using *flows* whose parameters (i.e., source, destination, period, the maximum number of packets) are fixed and known at design time. Based on this information and the statistics about the quality of network links, the network manager constructs schedules that specify the time slot and frequency of each packet transmission. The schedule determines the data plane behavior (the timing, retries, and acknowledgments of data packets). The schedule is executed synchronously and cyclically by all nodes. Reliability can be ensured by overprovisioning retransmissions performed over different channels and, potentially, over multiple paths. The relatively simple design of these industrial protocols enables us to guarantee, with high probability, the reliable and timely delivery of flows' packets.

The current state-of-the-art approach has intrinsic limitations, which make it unsuitable for the next-generation CPS applications. New applications are thought to need more flexibility, greater capacity, and more stringent real-time requirements. New hardware offers increased memory on nodes, more in-network computing resources, and timer improvements: the hope is that newer hardware will suffice for new CPS application needs. Higher bandwidth needs are motivated by the deployment of data-intensive sensors like cameras, microphones, and LIDAR. Inherent in the CPS vision is feedback ("closing the loop") to actuators, hence bounded latency is essential. In the short term, improvements in wireless technology may be enough to satisfy new application demands; however, in the long run, it will be difficult for such improvements alone to keep up with the growth and variety of applications. Consequently, we look to how we can redesign and engineer network software with CPS applications in mind.

As a step in this direction, we consider a novel perspective on how industrial networks are designed. Specifically, we propose to dispense with the use of schedules in favor of using *node-communication programs*. Our idea is that nodes in the network will run automatically synthesized programs that make informed decisions about flows at run time rather than having an *a priori* fixed schedule of packet transmissions, retries, queuing, etc. While it is true that fixed schedules, which specify what is transmitted in a given time slot, are simpler to analyze, this simplicity sacrifices some bandwidth that otherwise might be used for higher throughput and greater network capacity. Thanks to more powerful node hardware, some degree of dynamic decision-making is possible in the nodes, based on local conditions. Our idea transforms the design task from determining schedules for nodes into the task of determining the network programs which will run in the nodes. The price we pay for this more powerful architecture is that flow reliability is more difficult to analyze. Yet, we developed analysis techniques that make it tractable for moderate-size networks using automated analysis tools. To facilitate tractability of analysis, node programs will be composed of primitives from a specialized language, which we think of as a minimal domain-specific language (DSL).

The fertile research territory for the network program approach lies between two extremes, fixed schedules, and nodes running programs specified with a general-purpose language. These extremes reflect trade-offs in analysis and expressiveness. On the one hand, network behaviors are quite constrained when specified by fixed schedules, which are less expressive than programs: all the actions in a schedule are executed unconditionally (and schedules do not depend

on state information). Such constrained behavior facilitates traditional static analysis. On the other hand, a program can express many potential behaviors. Tasks of verification and synthesis of node programs become complex due to such expressiveness. Fundamental challenges are limiting the scope and scalability of analysis if node programs are too general. In this paper, we sketch a limited DSL to balance these considerations.

In our design, each node maintains some state information and performs conditional actions which depend on that information. The combination of state and conditional actions leads to programs having multiple possible execution paths. This opens new opportunities for optimization unavailable in the case of schedules that have only a single execution path. We exploit techniques from symbolic execution [3] to analyze execution paths, which automate reasoning about the program properties.

Ongoing trends in networking support the strategy of using node programs rather than fixed schedules, e.g., networks are becoming increasingly programmable. Such trends have precedents in older efforts such as active networks [1] and, more recently, software-defined networks [11]. Furthermore, our proposal opens wireless networks to modern tools that efficiently synthesize software, optimize it, and formally verify its properties. In the remainder of this paper, we demonstrate techniques showing how node programs respond to packet losses, transmit packets opportunistically, and provide guarantees of flow reliability given a prescribed number of packet losses. Node programs can be effectively synthesized and analyzed for the specific problem of packet loss. The paper concludes with pointers for further investigation.

2 MODELING ASSUMPTIONS

Network Model: CPS designers start by formally specifying their assumptions about the system's organization, workload, and reliability. A network is comprised of a *network manager* and up to a hundred *nodes*. Larger networks can be built hierarchically. A *network manager* is a resource-rich machine that coordinates the control plane, constructs programs or schedules, and acts as a bridge node for hierarchically-built larger networks. At the physical layer, we assume that the network supports multiple channels and tight time synchronization. While here we focus on IEEE 802.15.4 or the more recent IEEE 802.15.4a, our general techniques apply to other types of wireless networks such as 5G cellular networks.

Workload Model: A common model adopted in real-time and cyber-physical networks is to model an application's workload as a set of *real-time flows* that is known a priori. A *real-time flow* F_i has a fixed route that packets traverse through the network. A flow's packets usually carry sensor data or actuation commands. In the broader network community, flows are streams of packets; for real-time contexts, flows are sequences of *independent* packets generated periodically at the source and must be delivered to the destination by the deadline. For each flow F_i, a packet is generated on the source node with a phase ϕ_i, period of P_i, and deadline of D_i. The kth instance of flow F_i is released at time $r_{i,k} = \phi_i + k * P_i$ and has an absolute deadline $d_{i,k} = r_{i,k} + D_i$. To simplify notation and the presentation, we assume that the deadlines are less or equal to the periods ($D_i \leq P_i$). Since at most one instance per flow is present in the network, the discussion will be in terms of flows rather than instances.

Entries, Pushes, and Pulls: Programs use both time division and multiple channels. We refer to a slot and channel pair as *an entry*. Though a program may include several conditional pushes or pulls, only one such action may be executed in an entry during the program's run-time execution. A push(F_i) involves the sender transmitting F_i's data and the receiver replying with an acknowledgment (on the same channel). A push fails if the acknowledgment is not received by the end of the slot. A pull(F_i) involves the receiver requesting F_i's data and the sender replying with the flow's data. A pull fails if the data is not received by the end of the slot. A program may handle failed pushes or pulls by repeating

the action several times to achieve the desired reliability. We will refer to the node initiating the communication of a push or a pull as the *coordinator* and the responding node as the *follower*.

Reliability Model: In this paper, we adopt a simple (and unrealistic) reliability model where the pushes and pulls of a coordinator experience at most R failures during the execution of a program whose length is W slots. We call this an *(R,W)-failure* reliability model. In the included examples, there is a single failure during the operation of the network (i.e., $R = 1$ and $W = \infty$). Other reliability models that capture the lossy nature of wireless links better can be considered. However, the (R,W)-failure model is sufficient to illustrate the challenges of synthesizing and analyzing programs that tolerate packet losses.

Constraints: A program must meet the following constraints:

- *Transmission constraints:* At run-time, a node is involved in at most one push or pull in a slot.
- *Channel constraints:* At most, one transmission is performed in an entry to avoid intra-network interference. Additionally, a node must perform consecutive transmissions on different channels.
- *Forwarding constraints:* Each flow has at most one link that is active in any slot. The first link on the flow's route is activated at release time; subsequent links are activated as the previous ones complete relaying the packets.
- *Real-time and reliability constraint:* The packets of a flow reach their destination before their deadline while tolerating at most R failures within a program's W slots at each coordinator.

3 NODE COMMUNICATION PROGRAMS

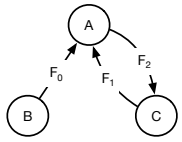
CPS networks must include sufficient transmissions to deliver a flow's data despite a prescribed number of failures. This problem becomes challenging when we require a solution to satisfy the following two requirements. First, the solution must minimize the overhead of retransmissions. Otherwise, it is unlikely the network will support the high data rates required by future CPS applications. Second, since link quality can vary from slot to slot significantly, the network must respond to packet losses quickly without the involvement of the control plane (which is designed to handle slower dynamics such as workload changes). Therefore, a node can only adapt its behavior based on local information, and its adaptation decisions cannot conflict with the transmissions of other nodes. Avoiding conflicting decisions is challenging as nodes have incomplete and inconsistent views of the outcome of transmissions and the state of other nodes.

We first introduce the concept of node communication programs and illustrate their advantages over static schedules using a simple star topology network with one hop flows. Then, we will relax this restriction by allowing multihop topologies and flows.

3.1 Programs on Star Topologies

Consider the star topology shown in Figure 1a and a workload consisting of three *real-time flows* — F_0, F_1, and F_2. The flows are listed in decreasing order of priority. All flows have a period and deadline of six slots and a phase of zero. Flows F_0 and F_1 originate at B and C and are destined for A; F_2 transmits data from A to C.

Schedules: A real-time scheduler (e.g., [14, 18, 20, 22] and surveys [17, 19]) constructs schedules that enforce prioritization and incorporate retransmissions as follows. In slot 0, F_0, F_1, and F_2 are released. The highest priority flow, F_0, is scheduled to transmit first. Using the actions defined in Section 2, node B is scheduled to transmit F_0's data using a push(F_0) command. In the same slot, A waits on the same channel and acknowledges the reception if it receives the push. However, since this transmission may be lost, a retransmission is scheduled in the next slot. During slots 0 and 1, C can sleep because it is not communicating. The remainder of the static schedule is built similarly and included in Figure 1b. Observe that each entry in the schedule consists of a single instruction that either has a node transmit a



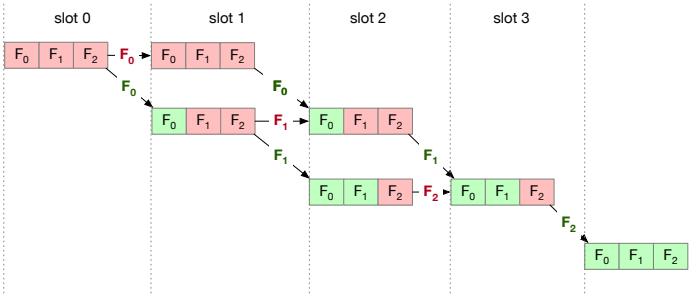
(a) Topology

Slot	Node A	Node B	Node C
0	wait	push(F_0)	sleep
1	wait	push(F_0)	sleep
2	wait	sleep	push(F_1)
3	wait	sleep	push(F_1)
4	push(F_2)	sleep	wait
5	push(F_2)	sleep	wait

(b) Schedule

Slot	Node A	Node B	Node C
0	start(F_0) pull(F_0)	wait	sleep
1	start(F_1) if !has(F_0) then pull(F_0) else pull(F_1) complete(F_0)	wait	wait
2	start(F_2) if !has(F_1) then pull(F_1) else push(F_2) complete(F_1)	sleep	wait
3	if !has(F_2) then push(F_2) complete(F_2)	sleep	wait

(c) Node program



(d) Execution DAG

Fig. 1: Star topology example — We build a `node program` for the topology and flows shown in Figure 1a. Figure 1b shows a static schedule for this workload. The generated `node program` and its associated execution DAG are provided in Figures 1c and 1d, respectively.

packet, wait for a transmission request from another node on a specified channel, or sleep. These actions are performed independent of the outcome of transmissions observed at run-time.

An advantage of static schedules is that they are easy to analyze, especially in terms of each flow's latency and reliability. However, due to their rigidity, networks that use schedules cannot reclaim resources that are not needed when errors do not occur because they do not include state information or support conditional commands. For example, consider the schedule listed in Figure 1b. It is desirable to cancel the push(F_0) scheduled in slot 1 if the push(F_0) scheduled in slot 0 has succeeded, which frees up this slot for another instruction if needed. This observation motivated us to consider transforming schedules of unconditional actions to programs by additing of state information and conditional statements, as shown next.

Programs: To write a program that can tolerate failures efficiently, we make the following two observations. First, observe that node A is in a unique position to determine whether the packets of any flow have reached its destination successfully. Since A is F_2's source, pushes can be used to relay the flow's data. A will know the outcome of F_2's transmission because C responds with an acknowledgment to a push(F_2). Node A is the destination of F_0 and F_1. For such flows, pulls can be used to retrieve the data of those flows. A can determine the outcome of those transmissions since the recipient of a pull responds with a flow's data. Therefore, using the right combination of pushes and pulls, the

central node of a star topology knows precisely the outcome of the flow's transmissions at run-time. In contrast, the other nodes in the star topology have a partial and inconsistent view of these outcomes.

The consequence of the above observation is that we can construct a consistent view of the failures in the star topology by maintaining state at node A. Specifically, let node A maintain a dictionary M that associates a single bit with each flow, indicating a flow's successful (S) or failed (F) delivery. Then, A can conditionally execute different actions depending on its state and according to the packet losses observed at run-time.

Armed with these two insights, a program that delivers the flows when a single packet loss may occur can be written and analyzed. In slot 0, when the flows are released, A marks each flow as not received by initializing $M = \{F_0 : \mathsf{F}, F_1 : \mathsf{F}, F_2 : \mathsf{F}\}$. In this slot, A executes a pull(F_0) resulting in A having two possible states at run-time. Either A received F_0 and its state is $M = \{F_0 : \mathsf{S}, F_1 : \mathsf{F}, F_2 : \mathsf{F}\}$, or A's state is $M = \{F_0 : \mathsf{F}, F_1 : \mathsf{F}, F_2 : \mathsf{F}\}$. Scheduling approaches ignore this information and simply assume the worst-case behavior. In contrast, programs consider both states (and the execution paths that led to the states) when determining future actions.

Consider the case when A's state is $M = \{F_0 : \mathsf{F}, F_1 : \mathsf{F}, F_2 : \mathsf{F}\}$ at the beginning of slot 1. In this case, A has not received F_0 yet (indicated by !has(F_0) being true), so it should execute pull(F_0). Since there can be at most one failure under the assumed fault model with $R = 1$ and this failure already occurred in slot 0, then the pull(F_0) executed in slot 1 must be successful, and A's state becomes $M = \{F_0 : \mathsf{S}, F_1 : \mathsf{F}, F_2 : \mathsf{F}\}$.

The other option is that at the beginning of slot 1, A's state is $M = \{F_0 : \mathsf{S}, F_1 : \mathsf{F}, F_2 : \mathsf{F}\}$. Since F_0 was already delivered successfully, A can execute a pull(F_1). On this execution path, a packet loss has not yet been observed. Thus, the pull can either succeed reaching state $M = \{F_0 : \mathsf{S}, F_1 : \mathsf{S}, F_2 : \mathsf{F}\}$ or fail reaching state $M = \{F_0 : \mathsf{S}, F_1 : \mathsf{F}, F_2 : \mathsf{F}\}$. Figure 1c shows the complete program. The different execution paths through the program can be captured as an execution DAG shown in Figure 1d. The red and green backgrounds in the figure indicate whether a flow's delivery failed (F) or succeeded (S), respectively. The arrows are labeled with the flow that was executed in a given state. The label's color indicates the outcome of each action.

Program Properties: A feature of programs is that a single slot may be shared by multiple transmissions, though only one transmission is executed per slot. For example, in slot 1, either F_0 or F_1 may be executed depending on the outcome of the previous transmission. It is important to note that sharing is possible without having to resort to contention-based techniques. The ability to share slots among multiple flows depending on the execution path is why the program is two slots shorter than the schedule. As the number of flows and allowable failures increases, these improvements will also increase, resulting in significant gains in capacity and reductions in latency.

Formalizing Programs: Let us formalize the concept of a `node program`. The resources involved in executing a program include a single coordinator node and one or more follower nodes. The followers are constrained to be within one hop of the coordinator. We will refer to a grouping of these program resources (a coordinator and its followers) as a container. In the above example, there is a single container whose coordinator is node A; the nodes B and C are followers. Each container has an input, a running, and an output queue of flows. The input queue includes flows that the container must process. The running queue includes flows currently executed by the container, using the coordinator and follower nodes. Finally, the output queue includes the completed flows i.e., flows that are guaranteed to deliver their packets under the prescribed reliability model.

A `node program` is executed by the coordinator and followers of a container. The instruction start removes a flow from the container's input queue for processing. The instruction complete indicates that a flow completed its execution and adds it to the container's output queue.

```
fragment        := start-blk; action_blk; complete_blk
start_blk       := start(flow)
complete_blk    := complete(flow); | complete_blk
action_blk      := action | if-statement
if-statement    := if bool-expr then action else-part
else-part       := else action_blk | ϵ
bool-expr       := has(flow) | !has(flow)
action          := pull(flow) | push(flow) | wait | sleep
```

Fig. 2: EBNF rules of a program fragment

A node program is structured as an array of fragments indexed by the slot in which the fragment is executed. A fragment may include actions and if-statements that allow actions to be executed conditionally. The coordinator may execute either push or pull while the followers execute waits or sleeps. In each fragment, at most a single action can be executed at run-time to ensure that its execution time does not exceed the length of a slot.

The state of a node program currently includes only M, which keeps track of the state of each flow. When flow F_i starts executing, $M[F_i]$ is initialized to failure (F). When push(F_i) or pull(F_i) is executed successfully, $M[F_i]$ is set to success (S). Flow F_i is removed from M when complete(F_i) is executed. Conditional if-statements may query the state of the program using has(F_i), which returns true if $M[F_i]$ is successful (S).

The grammar of a fragment is included in Figure 2. A fragment has three blocks: a *start block*, an *action block*, and a *complete block*. The start block may include a single starts to begin the execution of a flow within the container. Similarly, the complete block may include one or more completes to indicate that a flow is complete. The action block may include one of push, pull, wait, or sleep in the base case. We allow for the inclusion of conditional if-then-else statements. As part of the condition, a program may use has(F_i) or its negation !has(F_i) to determine whether node A has F_i. In any slot, we constrain a node to execute at most one action per slot to guarantee its completion by the end of the slot.

Analysis of Execution DAGs: Analyzing whether the program can tolerate the prescribed single packet loss is more difficult than static, schedule-table approaches. For this simple example, we can determine this is the case by visually inspecting the execution DAG and observing that each path includes at most one failure and reaches the final state when all flows deliver their packets. The response time of each flow can also be computed as the maximum length of the paths in the execution DAG that start with the state in which the flow is released and end with the state in which the flow is completed. A general method for analyzing the reliability and latency of flows is to consider those properties along all possible execution paths. This fact should be concerning as it is known that the number of paths and reachable states can grow exponentially, leading to a state explosion problem. While we may bring techniques used in verification to mitigate the state explosion problem to bear, we found the strategy of synthesizing code with a fixed structure can simplify analysis. The synthesis procedure described next enables such simpler reliability analysis by controlling the "structure" of the generated code.

Program Synthesis: A real-time scheduler can be adapted to efficiently synthesize the fragments that will be executed in each slot. The node-synthesizer manages the container's run queue, which includes the flows currently executing inside the container. With each flow in the run queue, we associate a counter indicating the number of times it was scheduled for execution. In each slot, the node-synthesizer inspects the container's input queue to determine if any flows need to be serviced. If the queue is not empty, the flow at the head of the container's input queue is removed and added to the run queue. Its execution counter is initialized to zero. The node-synthesizer generates the fragment

that will be executed in the current slot based on flows in the run queue. We consider each flow F_i in the run queue iteratively and in order of priority. We synthesize a conditional statement that checks if F_i is not yet complete (i.e., has(F_i) is False) and, if this is a case, either a push(F_i) or pull(F_i) is generated. The execution counter of all flows in the run queue is incremented by one at the end of the slot. A flow in the run queue whose execution counter equals the maximum number of failures $R + 1$ has been scheduled in sufficient executions to tolerate R failures. Therefore, the flow is complete and can be moved from the run queue to the container's output queue. This algorithm generates the programs shown in Figure 1c, except for slot 0. In slot 0, we already know that has(F_0) is False, so the pull(F_0) is performed unconditionally (and the if-statement does not need to be generated).

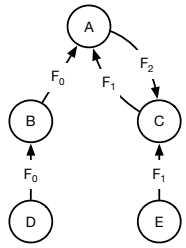
We can prove that the program tolerates the desired R failures by leveraging the structure of the synthesized code. Specifically, proving that flow F_i will reach its destination despite R packet losses requires showing that all possible program execution paths reach a state where $M[F_i] = \mathrm{S}$. The constrained structure of the program results in a constrained execution DAG similar to the one shown in Figure 1d. The proof revolves around determining the number of failures on each of the paths that reach a state where F_i was received. When programs are less regular or for different reliability models, the reliability analysis can be more complicated. In [4–6], we considered a more realistic reliability model called the Threshold Link Reliability (TLR) model. TLR models the likelihood that an action (i.e., a push or pull) of flow F_i (including both exchanges of data between sender and receiver) is successful as a Bernoulli variable $LQ_i(t)$. We assume that consecutive pushes or pulls performed over the same or different links are independent. Empirical studies suggest that this property holds when channel hopping is used [13, 15]. TLR has only one parameter – the minimum packet delivery rate m, which lower bounds the values of $LQ_i(t)$ such that $m \leq LQ_i(t)\ \forall i, t \in \mathbb{N}$. We have developed efficient synthesis techniques that ensure that when packet losses follow the TLR model, we can construct programs that guarantee the delivery of a flow's packets.

3.2 Programs on Multihop Topologies

In this section, we consider how we may construct programs for multihop topologies.

At a high level, our approach to handling multihop topologies is based on a two-level synthesis strategy. At the node level, containers synthesize their `node programs` as described in the previous section. At the network level, the problem is to arbitrate the allocation of nodes to containers to ensure nodes are in at most one container. Each node has its own container for which it is the coordinator. A container runs a node-synthesizer responsible for synthesizing programs that execute the flows in its input queue. The node-synthesizer generates code to be executed by both the coordinator and the followers (but involves no nodes outside the container). Then, the net-synthesizer runs centrally and constructs a cyclic schedule that executes or suspends containers in each slot. A node can be either a container's coordinator or a follower for a *single* container in given slot. Multiple containers may be assigned in a slot on different channels if they do not share followers or coordinators to maximize throughput. The containers that are executed and their follower nodes may change from slot to slot.

Traditional real-time schedulers can be easily adapted to operate on containers rather than flows. Specifically, we can construct the high-level program iteratively in a slot-by-slot manner. In each slot, the net-synthesizer considers the flows that are released according to their priority. Multihop flows are initially released on the link associated with their first hop. We categorize flows as *upstream* or *downstream* depending on whether they forward data to or from a common node, respectively. (This is common in wireless sensor networks that employ a base station.) If a flow is *upstream*, the flow is added to the input queue of its destination's container. Conversely, if the flow is *downstream*, the flow is added to the input queue of its source's container. Let the priority of a container be the maximum priority of the



(a) Multihop topology

Slot ⟶	0	1	2	3	4	5
$container_A$ Input	F_2	F_2	F_0, F_1, F_2	F_1, F_2	F_2	
$container_A$ Followers			B	B, C	C	C
$container_A$ Action	suspend	suspend	exec($[F_0]$, #2)	exec($[F_0, F_1]$, #0)	exec($[F_1, F_2]$, #1)	exec($[F_2]$, #2)
$container_A$ Output				F_0	F_1	F_2
$container_B$ Input	F_0					
$container_B$ Followers	D	D				
$container_B$ Action	exec($[F_0]$,#0)	exec($[F_0]$, #1)	-	-	-	-
$container_B$ Output		F_0				
$container_C$ Input	F_1					
$container_C$ Followers	E	E				
$container_C$ Action	exec($[F_1]$,#1)	exec($[F_1]$, #2)	-	-	-	-
$container_C$ Output		F_1				

(b) Operation of net-synthesizer

Slot	Node A	Node B	Node C	Node D	Node E
0	suspend sleep	exec($[F_0]$, #0) in $container_B$ start(F_0) pull(F_0)	exec($[F_1]$, #1) in $container_C$ start(F_1) pull(F_1)	exec($[F_0]$, #0) in $container_B$ wait	exec($[F_1]$, #1) in $container_C$ wait
1	suspend sleep	exec($[F_0]$, #0) in $container_B$ if !has(F_0) then pull(F_0) complete(F_0)	exec($[F_1]$, #1) in $container_C$ if !has(F_1) then pull(F_1) complete(F_1)	exec($[F_0]$, #0) in $container_B$ wait	exec($[F_1]$, #1) in $container_C$ wait
2	exec($[F_0]$, #2) in $container_A$ start(F_0) pull(F_0)	exec($[F_0]$, #2) in $container_A$ wait	suspended sleep	suspend sleep	suspend sleep
3	exec($[F_0, F_1]$, #0) in $container_A$ start(F_1) if !has(F_0) then pull(F_0) else pull(F_1) complete(F_0)	exec($[F_0, F_1]$, #0) in $container_A$ wait	exec($[F_0, F_1]$, #0) in $container_A$ wait	suspend sleep	suspend sleep
4	exec($[F_1, F_2]$, #1) in $container_A$ start(F_2) if !has(F_1) then pull(F_1) else push(F_2) complete(F_1)	suspend sleep	exec($[F_1, F_2]$, #1) in $container_A$ wait	suspend sleep	suspend sleep
5	exec($[F_2]$, #2) in $container_A$ if !has(F_2) then push(F_2) complete(F_2)	suspend sleep	exec($[F_2]$, #2) in $container_A$ wait	suspend sleep	suspend sleep

(c) Complete network program

Fig. 3: Multihop example – We build a `network program` for the topology and flows in Figure 3a. The net-synthesizer determines which containers run and their associated resources. Its operation is summarized in Figure 3b. A node-synthesizer runs in each container to generate code for the coordinator and follower nodes. The complete network program is shown in Figure 1c. In blue text, we include the operation of the net-synthesizer for the slot (which is not part of the `node program`).

flows' priorities that are either in the input queue or are executing inside the container. The net-synthesizer considers the containers in order of their priority and executes all the containers that do not share nodes in the current slot. In addition, the net-synthesizer also determines a suitable transmission channel for each container. At the end of the slot, the net-synthesizer inspects the output queue of each container. The queue will contain all the flows that have been executed in sufficient slots to ensure that they can tolerate the prescribed number of packet losses. If a flow is completed and has not reached its destination, the next link on its route is released. And, depending on whether the flow is *upstream* or *downstream*, the flow is added to either the container of the link's destination or source. The output queues of node-synthesizer are cleared at the end of each slot.

Let us consider the operation of the net-synthesizer in the multihop topology shown in Figure 3a. We constructed the example by extending the routes of F_0 and F_1 from the original example to include an additional hop. The temporal

parameters remain the same – flows have a period of six slots and a phase of zero. Figure 3b traces the state and actions of the net-synthesizer. In slot 0, flows F_0, F_1, and F_2 are released, and the first link of the multihop flows F_0 and F_1 are considered for synthesis. Since F_0 and F_1 are *upstream*, they will be added to the input queues of B's and C's containers since B and C, are the destination of flow's first hop. Similarly, since F_2 is *downstream*, it is added to the input queue of container A. To determine the actions to be performed in this slot, we consider the containers in order of priority (i.e., container$_B$, container$_C$, and container$_A$). The highest priority container is guaranteed to be executed, which requires that D is added to container$_B$ followers. Node B's node-synthesizer is invoked to generate code to run F_0. The container$_C$ has the next highest priority, and it would require node E if it were to be executed. Since E is not used by another container there is no resource conflict, and container$_B$ and container$_C$ can be scheduled in the same slot. Similarly, C's `node program` is invoked to generate code to run F_1. Finally, container$_A$ is considered. However, it cannot execute since F_2 requires node C which is already used (since it is the coordinator of container$_C$). Therefore, container$_A$ is suspended.

In slot 1, container$_B$ and container$_C$ can be executed concurrently again. At the end of the slot, the node-synthesizers running in each container place F_0 and F_1 to their respective output queues. Since F_0 and F_1 have not yet reached their destination, they are added to container$_A$, which is the source of their next-hop link. For the remaining slots of the `network program`, container$_A$ executes flows F_0, F_1, and F_2. The `node program` of container$_A$ for slots 3 – 5 is the one previously described in Figure 1c.

Our previous work – RECORP [4, 6] and WARP [5] – follow this strategy to synthesizing programs for wireless CPS. RECORP considers `node programs` that only use pushes, while WARP uses both pulls and pushes. RECORP uses an Integer Linear Program (ILP) to determine what containers run in a slot and what followers are assigned to each container. The ILP lets us identify the optimal set of containers that may run in a slot while enforcing prioritization. However, these decisions may not be optimal for longer time horizons. RECORP can synthesize programs that provide probabilistic guarantees on packet delivery within a few minutes. In contrast to RECORP, WARP employs heuristics to synthesize programs incrementally slot-by-slot on each node. As a result, WARP can synthesize and execute programs that handle hundreds of flows using today's hardware within 10 millisecond slots.

4 CONCLUSIONS AND FUTURE DIRECTIONS

Thus far, we have focused only on the challenge of handling packet failures under prescribed models of link reliability. In a nutshell, we have found that significant gains in network performance may be garnered when considering multiple execution paths when constructing programs. An inherent challenge with the proposed approach is state explosion, which can significantly hamper both the program analysis and synthesis. Indeed, our initial prototypes could only scale to handle tens of flows in moderately-sized networks. Ultimately, we have side-stepped the state explosion problem and developed practical protocols by carefully constraining the expressiveness of the language and the structure of the programs. We have been pleasantly surprised that it is sufficient to track only hundreds of states to outperform state-of-the-art scheduling approaches.

We have only scratched the surface of what is possible when programs are used to coordinate the operation of networks. Three directions for further exploration might include:

- **Richer CPS Applications:** As CPS applications grow in complexity, modeling their workloads as periodic flows becomes inadequate. It is well-understood that sensors such as microphones or cameras generate variable workloads. Similarly, our understanding of event-based feedback control is growing, and CPS applications that use it are starting

to emerge. Furthermore, handling sporadic traffic in networks poses significant challenges, particularly in centrally managed networks. Empirical studies into different application domains are needed to understand what are the communication needs of CPS applications.

- **Richer Models and Properties:** CPS application designers mainly focus on analyzing network response time and reliability due to the importance of feedback stability. This paper considers a simple reliability model where R failures can occur within the network. We should also consider more realistic models such as the TLR model. The impact of reliability assumptions on program synthesis and analysis complexity is an area to be explored. Furthermore, other specialized analyses (e.g., bounds on energy consumption) may be of interest in other CPS systems.
- **Reinforcement Learning for Synthesis:** As applications increase in complexity and designers will want to enforce/analyze more diverse properties, we will reach a point when we transition from synthesizing programs using handcrafted heuristics to reinforcement learning techniques. However, challenges abound. The number of action and the state spaces for network programs are likely to be large. Building good policies will require algorithms that reason about multi-objective optimization goals on complex data structures such as graphs or trees. Furthermore, is insufficient to synthesize efficient network programs; for synthesis to be practical it must produce programs in a timely manner.

In closing, we want to encourage the community to build on our work and extend program synthesis and analysis for wireless CPS networks.

ACKNOWLEDGMENTS

This work is funded in part by NSF under CNS-1750155.

REFERENCES

[1] [n.d.]. NSF Future Internet Design. http://www.nets-find.net/.
[2] 2012. IEEE Standard for Local and metropolitan area networks–Part 15.4: LR-WPANs. *IEEE Std 802.15.4e-2012* (2012).
[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.
[4] R. Brummet, O. Chipara, and T. Herman. 2020. Recorp: Receiver-Oriented Policies for Industrial Wireless Networks. In *IoTDI*.
[5] Ryan Brummet, Md Kowsar Hossain, Octav Chipara, Ted Herman, and Steve Goddard. 2021. WARP: On-the-fly Program Synthesis for Agile, Real-time, and Reliable Wireless Networks. In *Proceedings of the 20th International Conference on Information Processing in Sensor Networks (co-located with CPS-IoT Week 2021)*. 254–267.
[6] Ryan Brummet, Md Kowsar Hossain, Octav Chipara, Ted Herman, and David E Stewart. 2021. Recorp: Receiver-Oriented Policies for Industrial Wireless Networks. *ACM Transactions on Sensor Networks (TOSN)* 17, 4 (2021), 1–32.
[7] Richard Candell, Catherine A Remley, Jeanne T Quimby, David R Novotny, Alexandra E Curtin, Peter B Papazian, Galen H Koepke, Joseph E Diener, and Mohamed T Hany. 2017. Industrial Wireless Systems: Radio Propagation Measurements. *Technical Note (NIST TN)-1951* (2017).
[8] Baotong Chen, Jiafu Wan, Lei Shu, Peng Li, Mithun Mukherjee, and Boxing Yin. 2017. Smart factory of industry 4.0: Key technologies, application case, and challenges. *Ieee Access* 6 (2017), 6505–6519.
[9] Octav Chipara, Chenyang Lu, Thomas C Bailey, and Gruia-Catalin Roman. 2010. Reliable clinical monitoring using wireless sensor networks: experiences in a step-down hospital unit. In *SenSys*.
[10] Behnam Dezfouli, Marjan Radi, and Octav Chipara. 2017. REWIMO: A real-time and reliable low-power wireless mobile network. *TOSN* (2017).
[11] Nick Feamster, Jennifer Rexford, and Ellen Zegura. 2014. The road to SDN: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review* 44, 2 (2014), 87–98.
[12] Ken Ferens, Lily Woo, and Witold Kinsner. 2009. Performance of ZigBee networks in the presence of broadband electromagnetic noise. In *CCECE*.
[13] A. Gonga, O. Landsiedel, P. Soldati, and M. Johansson. 2012. Revisiting Multi-channel Communication to Mitigate Interference and Link Dynamics in Wireless Sensor Networks. In *ICDCS*.
[14] James Harbin, Alan Burns, Robert I Davis, Leandro Soares Indrusiak, Iain Bate, and David Griffin. 2019. The AirTight Protocol for Mixed Criticality Wireless CPS. *TCPS* (2019).
[15] Ozlem Durmaz Incel. 2011. A survey on multi-channel communication in wireless sensor networks. *Computer Networks* (2011).

[16] Ranjit Jhala and Rupak Majumdar. 2009. Software model checking. *ACM Computing Surveys (CSUR)* 41, 4 (2009), 1–54.
[17] Chenyang Lu, Abusayeed Saifullah, Bo Li, Mo Sha, Humberto Gonzalez, Dolvara Gunatilaka, Chengjie Wu, Lanshun Nie, and Yixin Chen. 2015. Real-time wireless sensor-actuator networks for industrial cyber-physical systems. *Proc. IEEE* (2015).
[18] Venkata Prashant Modekurthy, Abusayeed Saifullah, and Sanjay Madria. 2019. DistributedHART: A distributed real-time scheduling system for wirelesshart networks. In *RTAS*.
[19] Marcelo Nobre, Ivanovitch Silva, and Luiz Affonso Guedes. 2015. Routing and scheduling algorithms for WirelessHART Networks: a survey. *Sensors* (2015).
[20] Abusayeed Saifullah, You Xu, and Chenyang Lu. 2010. Real-Time Scheduling for WirelessHART Networks. In *RTSS*.
[21] Yakir Vizel, Georg Weissenbacher, and Sharad Malik. 2015. Boolean satisfiability solvers and their applications in model checking. *Proc. IEEE* 103, 11 (2015), 2021–2035.
[22] Tianyu Zhang, Tao Gong, Zelin Yun, Song Han, Qingxu Deng, and Xiaobo Sharon Hu. 2018. FD-PaS: A fully distributed packet scheduling framework for handling disturbances in real-time wireless networks. In *RTAS*.