Buffy: A Formal Language-Based Framework for Network Performance Analysis

Amir Seyhani
University of Waterloo

Junyi Zhao
Princeton University

Aarti Gupta
Princeton University

David Walker
Princeton University

Mina Tahmasbi Arashloo
University of Waterloo

Abstract

Despite recent advances in using formal methods for analyzing network performance, modeling network functionality for performance analysis remains challenging. Existing tools expect users to directly create the logical formulas corresponding to the network functionality of interest. This is often unintuitive, difficult to get right, and tightly coupled with the specific encoding and reasoning engine one chooses to use. Instead, we propose language abstractions that enable users to model network functionality and analysis tasks in an imperative solver-agnostic program, and a framework to transform them into a representation that can be analyzed by the appropriate solver. We outline our progress so far, demonstrating the potential of our approach through preliminary case studies and directions for future work.

CCS Concepts

• **Networks** → **Network performance analysis**; • **Theory of computation** → **Logic and verification**; • **Software and its engineering** → **Domain specific languages**.

Keywords

Network Performance Analysis, Formal Verification, Domain-Specific Programming Languages

ACM Reference Format:
Amir Seyhani, Junyi Zhao, Aarti Gupta, David Walker, and Mina Tahmasbi Arashloo. 2024. Buffy: A Formal Language-Based Framework for Network Performance Analysis . In *The 23rd ACM Workshop on Hot Topics in Networks (HOTNETS '24), November 18–19, 2024, Irvine, CA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3696348.3696854

HOTNETS '24, November 18–19, 2024, Irvine, CA, USA

ACM ISBN 979-8-4007-1272-2/24/11
https://doi.org/10.1145/3696348.3696854

1 Introduction

Network verification is an active area of research with considerable academic and industry impact [22, 23, 25, 26, 32]. Most existing efforts focus on analyzing a network's functional correctness, particularly its routing and forwarding. More recently, FPerf [8] and CCAC [9], for example, have shown promising results using formal methods for analyzing performance-related properties of network components.

Unfortunately, while showing tantalizing promise, these performance-analysis tools are difficult to use, even for experts in the field. One of the key problems is that the first step in any analysis is to model the network component of interest — e.g., a scheduler or traffic shaper. However, these tools offer no abstractions for doing so: The semantics of every computation and movement of packets between queues must be coded directly with low-level logical connectives (and, or, not). This leads to 100s of lines of logical variables, conjunctions and disjunctions, as if one were implementing low-level hardware for the computation. Such encodings are error-prone, hard to debug, difficult to maintain, and impossible to reuse. Without a higher-level solver-agnostic interface, creating models will remain a significant barrier to the use of formal network performance analysis tools in practice.

We believe language abstractions and compiler technology are essential in bridging the gap between general, high-level performance-oriented models for network components, and low-level logical engines capable of automated reasoning about them. Indeed, when it comes to routing and forwarding, we have already seen a number of successful language abstractions that provide high-level interfaces for human users and translate them into lower-level reasoning engines. For instance, NetKAT [7] can express data plane models and specifications and will translate them into automata for its decision procedures. Zen [11] allows users to specify route-processing functions and will translate them into logical formulae or BDDs for analysis.

In this paper, inspired by the success of existing (verification-oriented) languages, we propose Buffy, a

language-based framework for modeling and analyzing network performance. The Buffy language contains the conventional constructs of a simple imperative language (variables, assignments, conditionals, and loops) as well as a number of special built-in abstractions for managing and reasoning about packet buffers. These programs specify how packets move between (sets of) buffers in a network, any assumptions about network traffic and state, as well as performance queries of interest. Buffy will be able to compile these high-level programs into one or more backends (e.g. Z3 [14], FPerf [8], Dafny [30]) for automated reasoning.

We have designed Buffy with flexibility in mind, both for supporting various levels of "precision" in buffers and various range of verification and reasoning approaches as backends. We show the potential of Buffy in creating a high-level unified interface for network performance analysis through preliminary case studies based on FPerf [8] and CCAC [9]. We envision that Buffy will provide a flexible platform for network performance analysis, enabling experimentation with various abstractions, backend solvers, and verifiers.

2 Motivation and Overview

In this section, we use a (buggy) packet scheduler as a motivating example for our proposed approach. The scheduler is inspired by FQ-CoDel [21], Linux's default queuing discipline, and is one of the main use cases in FPerf [8].

2.1 A Fair-Queuing Packet Scheduler

Our example scheduler services queues in a (mostly) round-robin fashion, but prioritizes the first few packets of new flows so that they are not blocked by longer flows if they are short and latency-sensitive. Incoming packets are classified and assigned to one of several packet queues. The scheduler keeps two lists of pointers to queues: queues with newly started flows (*new_queues*) and other queues with outstanding packets (*old_queues*). Suppose that the incoming packet belongs to q_i. If q_i is not in either of the lists, a pointer to q_i is added to the end of *new_queues*. Otherwise, q_i will remain in its current list. On dequeue, if *new_queues* is not empty, the queue at the head of that list, q_h, will send a packet. If q_h becomes empty, it is be removed from *new_queues* and marked as inactive. If it has already sent a quantum of bytes and is no longer considered a short flow but is not empty, it is inserted into *old_queues*. Otherwise, it is placed at the end of *new_queues*. If *new_queues* is empty, the head of *old_queues* will get to send a packet next.

The bug. Our scheduler has a subtle bug – When a queue in *new_queues* becomes empty, it is immediately deactivated. So, next time it gets a packet, it is placed in *new_queues* again. Given that *new_queues* are prioritized, this can potentially cause starvation for queues in *old_queues*. The FQ-CoDel RFC [21] warns against this bug: "the queue could reappear (the next time a packet arrives for it) before the list of old queues is visited; this can go on indefinitely, even with a small number of active flows, if the flow [...] transmits at just the right rate." The RFC proposes a change to the deactivation process to avoid this problem. In this paper, we use the buggy version as our motivating example.

```
// T: number of time steps we are modeling
// ins[N]: The N input packet queues
// nq and oq: new_queues and old_queues
// q->elem[i][t]: ith element of queue q at time t
// q->enqs[i][t]: ith element to be enqueued in queue q
for (int t = 0; t < T; t++){
  expr nq_head = nq->elem[0][t];
  for (int i = 0; i < N; i++) {
    expr qi_transmits = (nq_head.valid && nq_head.val == i);
    expr qi_not_empty = ins[i]->elem[1][t].valid
                        || ins[i]->enqs[0][t].valid;
    expr to_add = oq->enqs[0][t]
    expr enq_in_oq = to_add.valid && to_add.val == i;
    expr c = implies(qi_transmits && qi_not_empty, enq_in_oq);
    add_to_solver(c);}}
```

Figure 1: Queue demotion logic in FPerf

2.2 Modeling and Analysis Challenges

Ensuring the absence of undesirable performance issues such as the mentioned bug requires formal modeling and reasoning. Despite recent advances in formal analysis tools for network performance [8, 9], this remains a challenging task.

Creating models. The enqueue and dequeue processes of our scheduler are not overly complex – they look at the head of two lists, new_queues and old_queues to decide which packet queue will transmit next. Yet, modeling such behavior directly using logical formulas is not particularly intuitive and can get complicated and error-prone.

To see why, consider how FPerf models this scheduler. It uses the C++ API provided by Z3 [14], a Satisfiability Modulo Theories (SMT) solver, to create Boolean and integer variables, and logical formulas. The (simplified) code snippet in Figure 1 shows how deciding whether to demote a queue from new_queues to old_queues is modeled in FPerf. Modeling the demotion logic in FPerf involves *directly* constructing formulas with logical operators (&&, ||, implies, *etc.*) for *each time step* and for *each possible value* of the head of new_queues. There are dozens of other similar code snippets that directly construct formulas to *enumerate* all possible distinct scenarios for all the possible states of the input packet queues and the internal lists of pointers to queues in every time step and to update the queues accordingly for the next time step. The complete FPerf implementation of scheduling logic alone is ~200 lines of code (see [1]) and there are 100s of lines of code creating additional scheduler-agnostic constraints that model the internal operations of the packet queues and lists.

Creating models at the level of individual formulas is not only tedious, but also error-prone, and difficult to navigate,

maintain, and debug. As anecdotal evidence, a few undergraduate students have tried using FPerf to add/modify use cases in research projects, and it has sometimes taken weeks to get some of the use cases to work, citing the above reasons as the main barrier for using such formal analysis tools.

Using other solvers/verifiers. To make matters worse, to use a reasoning engine other than FPerf, such as Dafny [30] for checking annotations, or a model checker such as CBMC [13] to find bugs, a significant effort is needed to create a new model from scratch. In particular, FPerf can synthesize a set of input packet traffic sequences that satisfy a given query. But, a user may be interested in simpler tasks, *e.g.,* checking if a given input workload (modeled as an input assumption in Dafny) satisfies the given query. For these simpler tasks, we can model the scheduler in an imperative programming language (*e.g.,* Boogie for Dafny, C for CBMC), which is more intuitive but, as we show in §6, this requires code transformations and annotations. We aim to provide *a single front-end language* for the user to describe their network functionality of interest and support different verification tasks using multiple back-end solvers/verifiers.

Changing the level of abstraction. There is a well-known trade-off between a model's "precision" and its analysis efficiency. For example, FPerf models a packet queue as a list, creating separate variables to represent each element of each queue at each time step. CCAC [9], on the other hand, uses a single integer variable to represent the number of bytes present in the queue, abstracting away the boundaries and details of individual elements. However, no one size fits all. We aim to support different levels of modeling abstraction, in the style of abstract data types with common methods but potentially different implementations. That is, while we provide a unified set of operations over the buffers in the language regardless of the abstraction level, we support backend implementations with different levels of precision.

Modular analysis for scalability. Existing efforts perform a *monolithic* performance analysis on the whole system. For example, in FPerf, the formulas grow larger with more components,more queues, and more time steps – this leads to scalability limitations. One motivating goal for our work is to develop modular techniques for performance analysis. To this end, we aim to support modularity in our front-end language, to enable users create models of network functionality as a set of modules composed via packet buffers.

2.3 Buffy Overview

Buffy aims to provide language abstractions for modeling and reasoning about network performance. It includes an imperative solver-agnostic language with special constructs for packet buffers. As shown in Figure 2, we envision users writing a program that describes the network functionality of interest, potential assumptions about input traffic, network state, and their queries in the Buffy language. Buffy would then transform the program into one of the several possible intermediate representations (IRs) that can be further analyzed by different back-end solvers.

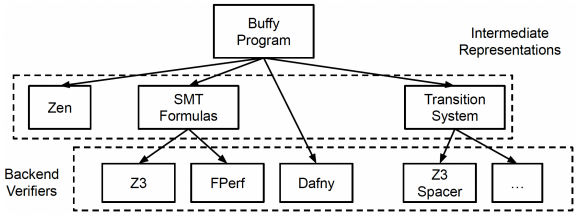

Figure 2: Buffy overview

We discuss the Buffy language in §3, our vision for Buffy's IRs and backends in §4, supporting modular analysis in §5, and our preliminary results through two case studies in §6.

3 Buffer-Centeric Abstractions

The main feature of Buffy's language, in addition to the conventional constructs of imperative programming languages, is special abstractions of packet buffers. We believe packet buffers play a central role in network performance analysis. Performance problems happen when there is contention for a shared resource, e.g., when multiple traffic streams need to share the same outgoing link. Many major performance problems arise when network queues do not drain as fast as expected and build up beyond acceptable thresholds, resulting in increased latency or packet drops. That is why there is ongoing interest in the networking community in monitoring or analyzing packet queues [9, 12, 29, 35], factoring in the impact on packet queues in designing new protocols and algorithms [4, 5, 20, 31, 34]. As a result, buffers can be thought of as a central part of network performance analysis.

That is why we have made buffers first-class citizens in the Buffy language and provide abstract operations on top of them as language constructs. This way, users can focus on modeling their functionality of interest and not worry about modeling buffers or their operations.

Buffy programs. At its core, a Buffy program describes how data move between buffers in a "time step", i.e., a single execution of the program. It takes in one or more buffers as input and one or more write-only buffers as output. Using an imperative C-like language, users can specify how to move data from the input buffers to the output buffers. The granularity of time can change depending on the network functionality that is being modeled. For a scheduler, this can be an enqueue and a dequeue operation. For a congestion control algorithm, it can be a round-trip time (RTT).

Syntax overview. Figure 3 shows a subset of Buffy's syntax, highlighting the buffer-centric constructs. Buffy supports simple standard types (integers and booleans), conventional arithmetic and boolean expressions, assignments, and control flow constructs such as conditionals and bounded loops.

```
n in Integer; o in Bool; x in Var;
f in FieldName; l in Lists; b in Buffers;

F ::= f == n //Filters
B ::= b | B |> F  //Buffers

//Expressions
E ::= x | o | n | E binop E | uop E
      | B | backlog-p(B) | backlog-b(B)
      | E[E] | l.has(E) | l.empty()

//Commands
C ::= move-p(b, b, E) | move-b(b, b, E)
      | x = l.pop_front() | l.push_back(E)
      | x = E | if (E) {C} else {C}
      | for (i in 0..n) do {C} | C; C |decls
```

Figure 3: A subset of Buffy's syntax

We opt for an imperative language to let users write programs in a more familiar format. Buffy also supports simple data structures, particularly arrays and lists.

Buffers. What sets Buffy apart is its special constructs for *packet buffers*. We have designated buffers as a special construct to represent network queues and have Buffy programs specify how data move between buffers.

We have carefully designed the expressions and commands over buffers to be (1) abstract enough, such that it would be possible to "plug-in" models for them at various precision levels, yet (2) expressive enough to allow Buffy programs to describe the desired network functionality. In particular, we allow the following operations on the buffers: (1) backlog-p(B)/backlog-b(B), which returns number of packets/bytes in the buffer B, (2) B |> F, which returns a buffer with elements of B that pass filter F, and (3) move-p(b, b, E)/move-b(b, b, e), which moves E packets/bytes from some input buffer (first argument) to some output buffer (second argument).

Example. Figure 4 shows how the scheduler from our motivating example can be implemented in Buffy. It specifies how the scheduler operates in one "time step", which, similar to FPerf, is defined as the time between two consecutive dequeue operations. It declares two "global" variables, nq and oq, for new_queues and old_queues respectively. Global variables are maintained across time steps. The scope of local variables, on the other hand, is within a single time step. The scheduler first checks if any "inactive" input buffers have received traffic and updates nq accordingly. It then checks nq and oq to decide which input buffer should transmit next. It uses the move-p function to specify how many packets should move from which input buffer to the output buffer.

Recall from §2 that the FPerf model of this scheduler has 100s of lines of code to directly generate the corresponding SMT formulas. Using Buffy, this scheduler can be described using 18 lines of imperative code and in a much more intuitive manner. Buffy would then transform the program into

```
1  fq(buffer[N] ibs, buffer ob){
2    global list nq; global list oq;
3    // update new queues
4    for (i in 0..N) do{
5       if ( backlog-p(ibs[i]) > 0 & !oq.has(i) & !nq.has(i))
6           nq.enq(i);}
7    // decide which input queue should transmit
8    local bool dequeued; local int head;
9    local dequeued = false;
10   for (i in 0..N) do {
11     if (!dequeued) {
12       head = -1;
13       if (!nq.empty()) { head = nq.pop_front();}
14       else {
15           if (!oq.empty()) { head = oq.pop_front();}}
16       if (head != -1) {
17         if ( backlog-p(ibs[head]) > 1) {
18           oq.push_back(head);}
19         if ( backlog-p(ibs[head]) > 0) {
20           move-p(ibs[head], ob, 1);
21           dequeued = true;}}}}}
```

Figure 4: Our (buggy) scheduler (§2) in Buffy

a form that can be efficiently analyzed by the user's backend solver of choice. Table 1 shows the LoC comparison of modeling some packet schedulers in Buffy and FPerf.

Buffer models with varying precision. We envision Buffy to have a library of buffer models that implement buffer operations (e.g., move, backlog, and filter) at varying levels of precision that users can try without changing their Buffy programs. We have a preliminary prototype of modeling a buffer as a list of packets and plan to add more in the future, including modeling a buffer as sets of integers each representing the total number of packets or bytes the buffer contains from different traffic classes.

Different buffer models can be useful in different circumstances. For instance, suppose in the FQ scheduler example (Figure 4), we are interested in a query that asks whether the number of packets moved from one input buffer to the output can be much larger than the others. In this specific case, packet contents do not matter since we only care about how many packets from each input buffer are moved to the output. Thus, there is no need to differentiate between packets inside a single buffer. As such, we can potentially perform the analysis by modeling each buffer as an integer representing its size (in packets or bytes). On the other hand, this may not be sufficient if packet contents and/or order are important for analyzing the modeled functionality or query.

For instance, suppose we use the number i to represent a packet from the i-th input buffer. The sequences [1, 1, 1, 2, 2, 2] are [1, 2, 1, 2, 1, 2] are two possible packet sequences entering some output buffer. They both have an equal number of packets from input buffers 1 and 2. So, a model keeping track of just packet counts cannot distinguish between them

if our query centers around packet ordering in a fine-grained manner, and we would need to model ordering also.

Composition. Thanks to their buffer-centric interface, Buffy programs lend themselves quite naturally to composition. Suppose program O_i is an output buffer in program P_1, and I_j is an input buffer in program P_2. P_1 and P_2 can be composed by "connecting" O_i and I_j. Semantically, this means at the end of the time step t, the contents of O_i will be flushed into I_j. At the beginning of $t + 1$, I_j's updated state will reflect the modifications incurred from receiving packets from O_i and potential move operations in time step t in P_2. Note that the user does not need to add extra code to perform these updates. By specifying that the two buffers are connected, Buffy will augment programs to implement the mechanics of the composition. For example, in the CCAC case study, we modularize the CCAC model into 3 Buffy programs, with communication handled by input and output buffers.

Assumptions and queries. Users can declare global variables as "monitors", which are *ghost code* that do not change the program's behavior and just observe it to track performance metrics of interest. These can be statistics about buffers (e.g., buffer size, buffer drain rate) or program-specific state (e.g., window size changes in congestion control algorithms). Users can then use `assert(E)` statements, where `E` is a boolean expression, in the program to check if these monitors have acceptable values at various points during the execution of the program and across time steps. Similarly, users can use `assume(E)` statements to specify assumptions about buffers or program state. Input traffic is a sequence of packets that are flushed into the input buffers every time step, so users can use a similar approach to specify assumptions about input traffic patterns. We will explain the usage of assumptions and queries in our case studies in §6.

4 Multiple Back-Ends

We aim to support multiple solver and verifier back-ends in Buffy, so users can choose among them depending on the verification task at hand, and also to provide a platform for comparing different approaches for the same analysis tasks.

Back-end for Z3 and FPerf. Users may want to fully specify the assumptions and the program and use Z3 to check whether the query would be satisfied. Or, users may want to use FPerf to synthesize the assumptions on the input traffic that would cause the query to be satisfied. In both cases, the IR would comprise SMT formulas, including symbolic variables and constraints. For generating SMT formulas from Buffy programs, we can leverage standard program transformations such as loop unrolling, function inlining, and Static Single Assignment (SSA) form [3]. We also plan to explore customized transformations, especially for buffer-related language constructs. The SMT problem can be written in the standard SMT-LIB format [10] supported by different SMT solvers (including Z3), or it can be constructed by using solver APIs. In future work, we plan to investigate the use of frameworks such as Zen [11] that provide high-level C++ APIs for the programmatic construction of SMT problems.

Program	FPerf (LoC)	Buffy (LoC)
Fair-Queue	197	18
Round-Robin	60	10
Strict-Priority	33	7

Table 1: FPerf vs Buffy LoC comparison

Back-end for Dafny. We would like to leverage Dafny [30], a popular modeling and verification framework. Dafny is an annotation checker, *i.e.,* it requires all loop invariants and interface specifications (requires/ensures clauses) for each method to be provided by the user. Although it is straightforward to translate a Buffy program to a Dafny program, directly trying to verify it with Dafny may not work. As we will show in our case studies (§6), coming up with loop invariants and interface specifications for network programs with multiple buffers is not easy. Instead, we use loop unrolling and function inlining in Buffy programs where we do not have easily available loop invariants and interface specifications. In addition, we use customized transformations to enforce workload assumptions on input traffic buffers.

Back-end for model checkers. To use a symbolic model checker, Buffy can transform the program into a transition system as the IR. Although we have not tried this yet in our case studies, we plan to translate a program into a system of Constrained Horn Clauses (CHC), to explore the use of the Spacer tool [27], which has been applied successfully for modular verification of programs.

5 Modular Analysis

One of Buffy's main goals is to support modular analysis, which is crucial for enhancing scalability on large systems and longer time intervals. For instance, we can leverage the existing modular verification techniques in Dafny for analyzing Buffy programs. In one of our case studies, a prior work [9] has identified many interface specifications of a component that models network paths, which helped us add interface specifications at the boundary of the Buffy program modeling that component and efficiently check the correctness of the discovered corner case without having to perform much inlining.

In future work, we plan to explore techniques to synthesize interface specifications at the boundary of Buffy programs and assumptions on input workloads. First, we will develop grammars with suitably expressive predicates on buffers that can capture interface specifications of interest for performance analysis. It has been seen that SyGuS (Syntax-Guided Synthesis) approaches with domain-specific grammars [6] are often more effective than pure solver-based

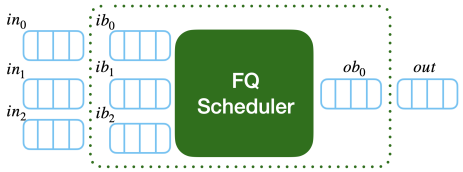

Figure 5: Schematic of the buffers for the FQ Scheduler

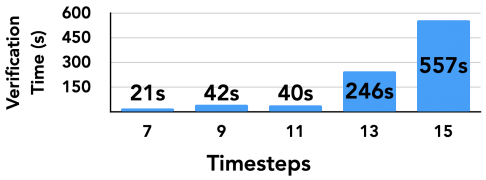

Figure 6: Dafny Verification Time

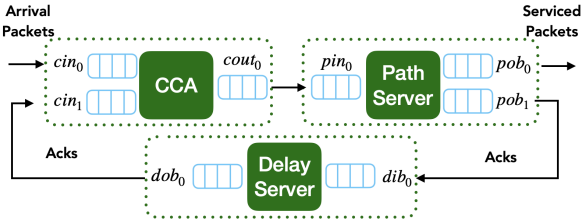

Figure 7: Buffy Model of CCAC

approaches for synthesis. We will use guess-and-check techniques, where an iterative procedure guesses some candidate guided by our grammars, and checks the candidate using a verifier. Specifically, we would like to use the Houdini algorithm [15] with Dafny to iteratively refine guesses of interface specifications. We would also like to develop CHC-based synthesis techniques, in the style of the Spacer tool [27] (§4).

6 Case Studies

FPerf [8] and CCAC [9] use Z3 as the back-end solvers. To demonstrate the benefits of Buffy, we have written Buffy programs to model one use case from each paper and used Dafny [30] as the back-end solver to check their correctness. We manually transformed Buffy to Dafny, and plan to automate the process in future work. The code samples of our Dafny implementations are available on the Github [2].

6.1 FPerf - FQ Scheduler

Figure 4 shows the Buffy implementation of FPerf's FQ scheduler, and Figure 5 shows a schematic of the buffers used in the Buffy model. The scheduler moves packets from input buffers ib_i to the output buffer ob_0. The input traffic at each time step is specified using the buffers in_j and is flushed into the input buffer ib_j of the scheduler. In this case study, we manually translated this program to a Dafny method. We had to apply several transformations, described later on, to make our program suitable for Dafny to analyze.

FPerf's query for the FQ scheduler checks for starvation – whether a buffer can take much more than its fair share of the bandwidth – and it uses the total number of dequeued packets from a buffer in a bounded (T) number of time steps as a metric. As such, we augment our program to keep track of that metric, cdeq, as ghost variables (monitors), and add assert(cdeq[T - 1] >= T/2) as the query. Moreover, FPerf synthesizes a set of conditions on the input traffic, a.k.a workload, that will satisfy the query. Dafny is an annotation checker. So, we use Dafny's havoc variables to specify symbolic input traffic (i.e., any input is possible) and use assume statements to restrict them to FPerf's synthesized traffic pattern.

Loop unrolling and method inlining. Dafny requires loop invariants and pre/post conditions for each method. In our initial Dafny implementations, the Buffy program was translated to a method that specified the scheduler's functionality for a single timestep. Within that method, there were several bounded loops. There was also a loop to execute the scheduler method for T time steps. Writing loop invariants and method pre/post conditions proved quite challenging as they would have to capture the scheduler's functionality as a set of complex predicates. Instead, we unfold the loops and inline the scheduler and other sophisticated methods to streamline the translation from Buffy to Dafny.

Symbolic inputs. Dafny supports havocs – symbolic variables with non-deterministic values that can be constrained using assume statements. However, using havoc for complex data types such as seq<int> (which we use to model packet buffers) proved to be difficult for Dafny to analyze. Instead, we had to transform our traffic variables to "structured" havoc variables – sequences of fixed shape and size with integer havoc variables inside. That is, even though Dafny is a C-like imperative language, several standard (unrolling and inlining) and customized transformations (structured havocs) were needed to make the FQ scheduler model suitable for analysis in Dafny. Buffy's goal is to relieve users of having to perform such solver-specific transformations.

Scalability. As shown in Figure 6, unfolding the loops and inlining methods resulted in the verification time increasing exponentially with increasing the total time steps (T). So, while unrolling and inlining helped streamline the translation, without modules and the right invariants and conditions at the boundaries, scalability will remain an issue, motivating our future work on modular analysis.

6.2 CCAC - AIMD Ack Burst Scenario

CCAC models how congestion control algorithms (CCAs) behave over Internet paths. It models Internet paths as a path server, which is a generalized and non-deterministic token bucket filter, followed by a fixed delay. To reproduce CCAC's model in Buffy, we decomposed it into three programs: the

CCA, the path server, and a fixed delay server. We composed these programs by connecting their buffers (see §3).

Figure 7 demonstrates this composition: The CCA has two input buffers, cin_0 for input data and cin_1 for acks coming back from delay server. The content of output buffer $cout_0$ is then flushed into input buffer pin_0 of the path server. Path server forwards the serviced packets into the output buffer pob_0 and sends the acks into pob_1. Acks are then flushed into the dib_0 of the delay server, and will go back to the CCA.

We use havoc and assume statements to create the ack burst condition discovered by CCAC and use assert statements to check that the query (occurrence of loss) is satisfied. Here, to make our program suitable for Dafny, we needed to unroll some of the loops. However, CCAC describes several conditions and invariants about its path server that we could supply for Dafny to help with the analysis. This is in contrast with the previous case study, where we had to do several forms of transformations to account for the absence of user-provided invariants and conditions. As mentioned before, in our future work, we hope to augment such user-provided annotations with automatically generated ones.

7 Discussion

We enforce some limitations at the language level to ensure efficient analysis of Buffy programs with current solvers. The flexibility of separating the modeling and backend solvers in our approach allows us to remove thse constraints if solvers become better at analyzing more complex code.

Bounded loops. We envision that Buffy programs need to loop over input buffers or data structures, and it is reasonable to assume some constant bounds for them at compile time. Moreover, a loop invariant is needed to analyze loops with variable bounds, and finding the proper loop invariant is undecidable in the general case. Thus, Buffy currently only allows loops with a constant bound. One way of program analysis in presence of loops is to have the user provide the proper loop invariant. Limiting the loops by a constant allows us to unroll the loops during the analysis, hence avoiding the need for invariants. However, with loop invariants for the loop that executes the program over many timesteps as annotations, we could scale Buffy's analysis to an arbitrarily-bounded time horizon, an improvement over tools like FPerf. We aim to explore synthesizing such invariants in the future.

Bounded arrays. Buffy requires all data structures (e.g., arrays) to have a constant upper bound on their size. This allows us to avoid using expensive SMT solver theories such as array theory, e.g., by flattening arrays, hence making the analysis simpler.

Primitive data types. Currently, Buffy only supports integers, boolean, and buffers, and array and list data structures. To keep Buffy minimal, we only added the data types and structures required by the problems and examples we have experimented with so far. However, if needed in the future, we do not foresee fundamental challenges in extending the Buffy framework to more data types and data structures.

8 Related Work

Performance-related and quantitative reasoning. FPerf [8] and CCAC [9] have demonstrated promising results in using formal methods to reason about network performance. Other work has focused on reasoning about quantities such as link loads and hop counts [24, 28, 36], as well as probabilistic reasoning about networks [17, 18]. While these works provide a range of approaches for analyzing various performance-related and quantitative aspects of networks, we focus on creating a unified, high-level, and flexible programming language for modeling network functionality and a framework to support multiple analysis backends.

Language-based frameworks for network verification. Our work is inspired by prior efforts in language-based frameworks that support network modeling and verification. Examples include NetKAT [7] for data plane verification, NV [19] for control plane verification, Batfish [16] for network analysis and simulation, Zen [11] for programmatically generating SMT formulas. Finally, there is prior work [33] that uses a language-based approach to model and analyze a particular scheduling paradigm, a tree of PIFO queues.

Verification back-ends. As described earlier, multiple backends in Buffy are related to and leverage advanced solvers/tools that have been successfully applied in software verification. These include Z3 [14], Dafny [30], Spacer [27], SyGuS-based techniques [6]. As shown in our case studies, using these methods requires some customization for buffer-based abstractions in Buffy, and we aim to further improve automation to reduce user burden where possible.

9 Conclusion

Modeling network functionality for performance analysis is a challenging task, especially if users are required to directly create low-level logical formulas. We have proposed Buffy, a high-level solver-agnostic imperative language centered around buffer abstractions for users to specify network functionality, and a framework that will automatically generate solver-friendly representations. We envision our framework will lower the barrier for utilizing formal methods for performance analysis and enabling experimentation with different abstractions and reasoning approaches.

Acknowledgments

We thank Carol Duan for early exploration of language abstractions and the anonymous reviewers for their helpful feedback. This work was supported in part by an NSERC Discovery grant and NSF grants CNS-2312539 and CCF-2107138.

References

[1] 2023. FPerf Github Repository. https://github.com/all-things-networking/fperf/tree/main. Accessed: 2024-06.

[2] 2024. Buffy Github Repository. https://github.com/all-things-networking/hotnets24-buffy. Accessed: 2024-06.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.

[4] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center tcp (dctcp). In *ACM SIGCOMM*.

[5] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM*.

[6] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *FMCAD*.

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

[8] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. 2023. Formal methods for network performance analysis. In *USENIX NSDI*.

[9] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. 2021. Toward formally verifying congestion control behavior. In *ACM SIGCOMM*.

[10] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6.* Technical Report. Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[11] Ryan Beckett and Ratul Mahajan. 2020. A General Framework for Compositional Network Modeling. In *The ACM Workshop on Hot Topics in Networks (HotNets)*.

[12] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. 2019. Fine-grained queue measurement in the data plane. In *ACM CoNEXT*.

[13] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (Lecture Notes in Computer Science, Vol. 2988)*. Springer.

[14] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *In Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[15] Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. 2001. Annotation inference for modular checkers. *Inform. Process. Lett.* (2001).

[16] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *USENIX NSDI*.

[17] Nate Foster, Dexter Kozen, Konstantinos Mamouras, Mark Reitblatt, and Alexandra Silva. 2016. Probabilistic NetKAT. In *European Symposium on Programming*.

[18] Timon Gehr, Sasa Misailovic, Petar Tsankov, Laurent Vanbever, Pascal Wiesmann, and Martin Vechev. 2018. Bayonet: Probabilistic inference for networks. In *ACM SIGPLAN PLDI*.

[19] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. 2020. NV: An Intermediate Language for Verification of Network Control Planes. In *ACM SIGPLAN PLDI*.

[20] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *ACM SIGCOMM*.

[21] Toke Høeiland-Jørgensen, Paul McKenny, Dave Taht, Jim Gettys, and Eric Dumazet. 2018. The Flow Queue CoDel packet scheduler and active queue management algorithm. RFC 8290. https://rfc-editor.org/rfc/rfc8290.txt

[22] Alex Horn, Ali Kheradmand, and Mukul Prasad. 2017. Delta-Net: Real-time network verification using atoms. In *USENIX NSDI*.

[23] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, et al. 2019. Validating datacenters at scale. In *ACM SIGCOMM*.

[24] Garvit Juniwal, Nikolaj Bjorner, Ratul Mahajan, Sanjit Seshia, and George Varghese. 2016. Quantitative network analysis. *Technical report* (2016).

[25] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header space analysis: Static checking for networks. In *USENIX NSDI*.

[26] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. 2013. Veriflow: Verifying network-wide invariants in real time. In *USENIX NSDI*.

[27] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based model checking for recursive programs. *Formal Methods in System Design* (2016).

[28] Kim G Larsen, Stefan Schmid, and Bingtian Xue. 2017. WNetKAT: A weighted SDN programming and verification language. In *International Conference on Principles of Distributed Systems (OPODIS)*.

[29] Jean-Yves Le Boudec and Patrick Thiran. 2001. *Network calculus: a theory of deterministic queuing systems for the internet.* Springer.

[30] K. M. Leino. 2017. Accessible Software Verification with Dafny. *IEEE Software* (nov 2017). https://doi.org/10.1109/MS.2017.4121212

[31] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. 2019. HPCC: High precision congestion control. In *ACM SIGCOMM*.

[32] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the data plane with Anteater. In *ACM SIGCOMM*.

[33] Anshuman Mohan, Yunhe Liu, Nate Foster, Tobias Kappé, and Dexter Kozen. 2023. Formal abstractions for packet scheduling. In *ACM OOPSAL*.

[34] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A centralized" zero-queue" datacenter network. In *ACM SIGCOMM*.

[35] The P4.org Application Working Group. 2024. In-band Network Telemetry (INT) Dataplane Specification. https://p4.org/p4-spec/docs/INT_v2_1.pdf.

[36] Ying Zhang, Wenfei Wu, Sujata Banerjee, Joon-Myung Kang, and Mario A Sanchez. 2017. SLA-verifier: Stateful and quantitative verification for service chaining. In *IEEE INFOCOM*.