Formally Verified Software-Defined Delay Tolerant Networks

Jan-Paul Ramos-Dávila
Cornell University
Ithaca, NY, USA
jvr34@cornell.edu

Alwyn E. Goodloe
NASA Langley Research Center
Hampton, VA, USA
a.goodloe@nasa.gov

Abstract

Software-Defined Delay Tolerant Networks (SDDTNs) integrate Software-Defined Networking (SDN) principles with Delay Tolerant Networks (DTNs) to address challenges in high-latency and intermittent connectivity environments. Formal verification of these systems is challenging due to the complexity of protocols like the Bundle Protocol (BP). We present NetQIR, a domain-specific intermediate representation targeting SDDTN algorithms. NetQIR provides a formal framework that captures the behavior of P4 programs focusing on the Match-Action Pipeline (MAP) algorithm. We formalize NetQIR's semantics and type system in the Coq proof assistant [2], deriving a sound, executable semantics that supports verified execution of P4 programs targeting SDDTNs.

1 Introduction

Delay Tolerant Networks (DTNs) are designed for environments with high latency or intermittent connectivity, where traditional networking protocols fail [9, 14]. Software-Defined Networking (SDN) decouples the control plane from the data plane, allowing centralized management of network resources [11, 16]. Software-Defined Delay Tolerant Networks (SDDTNs) integrate SDN principles with DTNs to manage large-scale networks, optimize resource utilization, and adapt to changing environments [3, 4, 19].

Formal verification of SDDTNs is challenging due to the complexity of protocols like the Bundle Protocol (BP) [1] and the dynamic nature of DTNs. Verifying the correctness of network programs written in languages like P4 [5] is non-trivial, given their rich type systems and operational semantics. Existing tools like p4v [15] and Vera [18] provide verification techniques but may not capture all aspects required for SDDTNs.

To address these challenges, we introduce **NetQIR**, an intermediate representation designed to facilitate type-preserving compilation from P4 to a formally verified representation in Coq [2]. NetQIR allows us to leverage Coq's powerful type system and proof capabilities to ensure the correctness of SDDTN algorithms.

2 NetQIR: An Intermediate Representation

NetQIR captures essential constructs of P4 programs targeting the Match-Action Pipeline (MAP), enabling formal reasoning about their behavior within Coq. Similar to efforts like Petr4 [10] and P4K [13], we focus on creating a formal semantics that can be encoded in a proof assistant.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CoqPL'25, January 25, 2025, Denver, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2.1 Syntax and Semantics

NetQIR includes constructs for expressions, statements, actions, tables, and programs. The syntax mirrors P4 constructs while being amenable to formalization.

```
(* NetQIR Expressions *)
Inductive expr : Type :=
  | EVal : value → expr
  | EVar : string → expr
  | EBinOp : binop → expr → expr → expr
  | EField : expr → string → expr.

(* NetQIR Statements *)
Inductive stmt : Type :=
  | SAssign : string → expr → stmt
  | SIf : expr → stmt → stmt → stmt
  | SSeq : stmt → stmt → stmt
  | SActionInvoke : string → list expr → stmt
  | STableApply : string → stmt.
```

We define operational semantics using small-step semantics. The state consists of a store mapping variables to values and a packet representing headers and metadata. Our semantics are inspired by prior formalizations of network languages [7, 8].

3 Formal Guarantees and Coq Formalization

We establish formal guarantees for NetQIR programs, providing proofs of critical properties within Coq. Our approach ensures that well-typed NetQIR programs behave correctly, similar to techniques used in [12, 21].

3.1 Packet Delivery Correctness

We prove that well-typed NetQIR programs correctly process packets according to their specification.

Theorem 3.1 (Packet Delivery Correctness). *If a NetQIR program p is well-typed under context Γ, and executing p from state (σ, pkt) to (σ', pkt') yields pkt', then pkt' is correctly processed according to p's semantics.*

Proof Sketch. We proceed by induction on the structure of p. For each construct, we rely on the typing rules and operational semantics to ensure that the transformations applied to the packet are as specified. Assignments update the state correctly due to the well-typedness of expressions. Conditional statements execute the appropriate branch based on well-typed boolean expressions. Action invocations and table applications apply well-typed actions that modify the packet state as intended. The operational semantics define how each construct modifies the packet, and the typing rules prevent runtime errors, ensuring correct packet processing. □

3.2 Type Preservation

The type system guarantees that NetQIR programs are well-typed, and the operational semantics maintain the type of expressions and packets.

Theorem 3.2 (Type Preservation). *If $\Gamma \vdash e : t$ and $e \longrightarrow e'$ under the operational semantics, then $\Gamma \vdash e' : t$.*

Proof. By induction on the evaluation steps of e. Each reduction step maintains the type, as operations are defined to be type-preserving, and the typing rules ensure that operands and results have consistent types. □

3.3 Flow Conservation

We prove that NetQIR programs conserve packet flow, meaning they do not create or destroy packets arbitrarily.

Theorem 3.3 (Flow Conservation). *A well-typed NetQIR program p preserves the number of packets, except for explicit drop actions.*

Proof. By examining the operational semantics, packets are only modified during execution. The only way to remove a packet is through a drop action, which is explicitly defined. The typing rules prevent unauthorized packet creation, ensuring flow conservation. □

4 Type-Preserving Compilation from P4 to NetQIR

We establish a type-preserving compilation from P4 to NetQIR, ensuring properties proven about NetQIR programs hold for the original P4 programs. This approach aligns with methods used in other verification frameworks [17, 20].

4.1 Compilation Function and Type Correspondence

We define a compilation function $[\![\cdot]\!]$ mapping P4 constructs to NetQIR constructs while preserving types.

Definition 4.1 (Compilation Function).

$$[\![\texttt{action}\ a(\tau_1\ x_1, \ldots)\{s\}]\!] = \texttt{action}\ a(\tau_1'\ x_1, \ldots)\{[\![s]\!]\}$$

We establish type correspondence between P4 types τ_i and NetQIR types τ_i', ensuring that the compilation preserves typing.

Theorem 4.2 (Type Preservation under Compilation). *If a P4 program p is well-typed under context Γ_{P4}, then its compilation $[\![p]\!]$ is well-typed under context Γ_{NetQIR}, with corresponding types.*

Proof. By structural induction on p, mapping P4 types to NetQIR types and aligning typing rules accordingly. This ensures that well-typedness is preserved throughout the compilation. □

5 Example: Verified MAP Program

We demonstrate our approach with an IPv4 forwarding MAP program.

5.1 P4 Program

The P4 program defines an action for IPv4 forwarding [6]:

```
action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
    standard_metadata.egress_spec = port;
    hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
    hdr.ethernet.dstAddr = dstAddr;
    hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
}
```

5.2 Compilation to NetQIR

We compile the P4 action to NetQIR:

```
Definition ipv4_forward :=
  action "ipv4_forward" [("dstAddr", TBit 48); ("port", TBit 9)] {
    SAssign "standard_metadata.egress_spec" (EVar "port");
    SAssign "hdr.ethernet.srcAddr" (EField (EVar "hdr.ethernet") "
      dstAddr");
    SAssign "hdr.ethernet.dstAddr" (EVar "dstAddr");
    SAssign "hdr.ipv4.ttl" (EBinOp Sub (EField (EVar "hdr.ipv4") "
      ttl") (EVal (VBit 8 1)));
}.
```

5.3 Verification in Coq

In Coq, we verify that this action correctly updates packet headers and metadata according to the IPv4 forwarding specification.

Theorem 5.1 (Correctness of `ipv4_forward`). *For any packet pkt with valid headers and parameters dstAddr, port, executing* `ipv4_forward` *updates pkt such that:*

$$pkt.standard_metadata.egress_spec = port$$
$$pkt.hdr.ethernet.srcAddr = original\ pkt.hdr.ethernet.dstAddr$$
$$pkt.hdr.ethernet.dstAddr = dstAddr$$
$$pkt.hdr.ipv4.ttl = original\ pkt.hdr.ipv4.ttl - 1$$

Proof Sketch. By unfolding the action definition and applying the operational semantics, we verify that each assignment updates the packet state as specified. The typing rules ensure that operations are well-typed, and the execution preserves the correctness of the packet fields. □

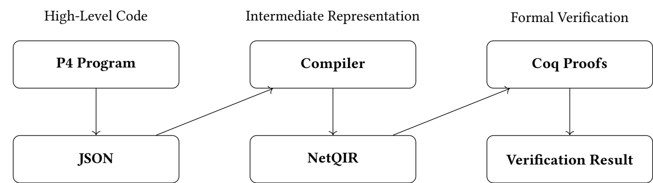

Figure 1. Verification Pipeline from P4 to Coq

Figure 1 illustrates the verification pipeline: the P4 program is serialized into JSON and passed through a type-preserving compiler to produce NetQIR, our intermediate representation. NetQIR is then used within the Coq proof assistant to formally verify the program's correctness. If the Coq proofs succeed, we obtain a verification result confirming correctness.

6 Conclusion

We have presented NetQIR, an intermediate representation facilitating type-preserving compilation from P4 to Coq. By formalizing NetQIR's semantics and type system in Coq, we provide formal guarantees about the behavior of SDDTNs. Our approach enables the verification of critical network properties and contributes to the reliability of networks in challenging environments.

References

[1] [n. d.]. Bundle Protocol Version 7. https://datatracker.ietf.org/doc/html/rfc9171. Accessed: 2023-09-01.
[2] [n. d.]. The Coq Proof Assistant. https://coq.inria.fr/. Accessed: 2023-09-01.
[3] [n. d.]. Delay Tolerant Networking Research. https://www.sandia.gov/research/research_foundations/information_science/dtn/. Accessed: 2023-09-01.

[4] [n. d.]. Interplanetary Internet Project. https://www.ipnsig.org/. Accessed: 2023-09-01.
[5] [n. d.]. P4 Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.2.3.html. Accessed: 2023-09-01.
[6] [n. d.]. P4 Tutorial. https://github.com/p4lang/tutorials. Accessed: 2023-09-01.
[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 *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 113–126.
[8] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. 2014. VeriCon: Towards verifying controller programs in software-defined networks. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 282–293.
[9] Scott Burleigh, Adrian Hooke, Leigh Torgerson, Kevin Fall, Vint Cerf, Bob Durst, Keith Scott, and Howard Weiss. 2003. Delay-tolerant networking: an approach to interplanetary internet. *IEEE Communications Magazine* 41, 6 (2003), 128–136.
[10] Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, et al. 2021. Petr4: Formal foundations for P4 data planes. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–32.
[11] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A network programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ACM, 279–291.
[12] Arjun Guha, Mark Reitblatt, and Nate Foster. 2013. Machine-verified network controllers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 483–494.
[13] Ali Kheradmand and Grigore Roşu. 2018. P4K: A formal semantics of P4 and applications. In *International Conference on Computer Aided Verification*. Springer, 678–697.
[14] Anders Lindgren, Avri Doria, Elwyn Davies, and Samo Grasic. 2016. Probabilistic routing protocol for intermittently connected networks. *RFC* 6693 (2016), 1–113.
[15] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. p4v: Practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 490–503.
[16] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. 2012. NetCore: A language for high-level network policies. In *Proceedings of the ACM SIGCOMM 2012 Conference*. ACM, 113–124.
[17] François Pottier. 2008. A capability calculus for concurrency and determinism. In *International Conference on Concurrency Theory*. Springer, 282–296.
[18] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Vera: Verification of Regular Algorithms. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. ACM, 30–36.
[19] Dominick Ta, Stephanie Booth, and Rachel Dudukovich. 2023. *High Data Rate Transport for Delay/Disruption Tolerant Networks: A Case Study*. Technical Report. NASA Glenn Research Center. https://ntrs.nasa.gov/api/citations/20220019062/downloads/HDTNSimSPACOMMApril2023_Italy.pdf
[20] Mihai Tran, Ugo Fiore, and Francesco Palmieri. 2022. SEFL: A symbolic execution framework for network functions. *Computer Networks* 207 (2022), 108805.
[21] Han Wang, Minlan Zhu, Hongyi Zeng, and Guofei Chen. 2014. NetASM: A low-level language for programming network devices. In *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 211–212.