P4AIG: Circuit-Level Verification of P4 programs

Mohammad A. Noureddine*, Amanda Hsu†, Matthew Caesar*, ‡ Fadi A. Zaraket, † William H. Sanders
*Department of Computer Science, † Department of Electrical and Computer Engineering
University of Illinois at Urbana Champaign
Urbana, IL
{nouredd2, achsu3, caesar, whs}@illinois.edu
‡ Department of Electrical and Computer Engineering
American University of Beirut
Beirut, Lebanon
fz11@aub.edu.lb

***Abstract*—In this work, we set out to develop P4AIG, a tool for the static verification of programmable data planes using sequential circuit analysis. P4AIG targets P4 programs by treating them as hardware pipelines rather than software programs. P4AIG allows for the circuit-level treatment of P4 programs, a feature not available for traditional software verification techniques. We believe that P4AIG will exploit the nature of P4 programs to achieve higher scalability in verification.**

I. Introduction

Programmable data planes allow for a significant shift in the paradigms of modern network designs. Data plane-specific programming languages, such as P4 [1], allow network designers to design, develop, modify, and test packet-forwarding protocols and pipelines in a hardware-agnostic manner. However, with great programmability comes design-time bugs, causing faults that can have drastic impacts on the performance and the security of networking systems [2].

Current approaches to testing programmable data plane implementations rely on exhaustive testing using simulation and automated packet generation software [3]. However, testing based techniques are expensive as they are searching a large space of inputs; testing must cover all possible types of packets with different protocols, both valid and invalid. Furthermore, testing can confuse bugs that occur in the data-plane program and its semantics with those that occur due to a specific compilation of the program into a hardware-specific implementation. Different vendors might generate different implementations of the same data-plane program, thus introducing potential bugs that are not derivatives of the program semantics.

Complementary to exhaustive testing are static verification techniques that leverage SAT and SMT solvers to verify annotated data-plane programs. Current verification techniques [3]–[6] treat P4 programs as traditional software programs and rely on symbolic execution as their main verification engine. However, by adopting such a software approach, these techniques become language dependent; every data-plane programming language requires different software and language-dependent verification techniques. More fundamentally, software verification techniques target Turing complete languages while data-plane programs describe restricted hardware pipelines that are bounded and simpler to verify.

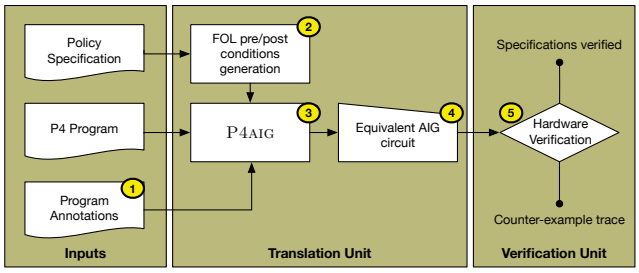


Fig. 1: High Level Overview of P4AIG

In this work, we will explore the fundamental property of data-plane programs – namely, that they describe restricted hardware pipelines – to achieve scalable verification. We propose P4AIG: a tool for the formal verification of data-plane programs using hardware verification techniques. P4AIG is inspired by the work in [7] showcasing that the translation of software programs into equivalent sequential circuits can significantly improve their verification. Specifically, we believe that the boundedness and the features of data-plane programs make them ideal candidates for such a translation: they operate on bounded inputs, theirs loops are linearly bounded by the number of packet headers, and they do not use dynamic memory allocation. In this paper, we target P4 programs, however, P4AIG can be easily applied to different languages since it performs verification at the hardware packet-forwarding level.

II. P4AIG: Circuit-level Verification of P4 Programs

Figure 1 shows the high level overview of our proposed approach. P4AIG is composed of two main units: (1) a translation unit and (2) a verification unit.

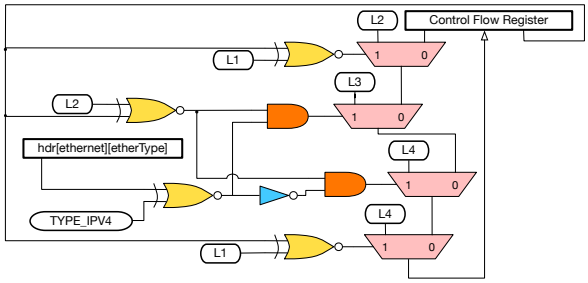
Program Annotations: First, P4AIG takes as input the original P4 program that we wish to verify. In addition, we require the application developers to annotate their code with *First Order Logic* (FOL) specifications (① [1]) in the form of in-code statements (similar to assertions in [4], [6]). These specifications identify the desired behavior of the P4 pipeline and are to be manually specified by the developers. The combination of the P4 program and the program annotations

[1] ① refers to annotations in Figure 1.

978-1-7281-3028-6/19/$31.00 ©2019 IEEE
DOI 10.1109/DSN-S.2019.00016

```
/* basic.p4: 49 - 71 */
parser MyParser(packet_in packet, out headers hdr,
        inout metadata meta,
        inout standard_metadata_t standard_metadata) {
    state start { /* @L1 */
        transition parse_ethernet;
    }
    state parse_ethernet { /* @L2 */
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            default: accept; /* def. accept: @L4 */
        }
    }
    state parse_ipv4 { /* @L3 */
        packet.extract(hdr.ipv4);
        transition accept;
    }
}
```

(a) Sample P4 parser code from `basic.p4`.



(b) Equivalent sequential circuit.

Fig. 2: *A simple parser showing the state machine that extracts and processes layers 2 and 3 information from ingress packets. We add location labels for each state (**@LX** comments in the code). On the right hand, we show the equivalent sequential circuit capturing the control flow of the parser. Ovals represent constants while boxes represent registers. Inputs into registers represent their next state that will get updated after each clock cycle. For brevity, we elide the details for the `hdr` register.*

will constitute a complete definition of the network forwarding plane that is then fed into the P4AIG translation unit (③).

Handling the Control Plane: Data plane programs, as defined by `P4`, only comprise half of a *software defined network* program; they only serve to define the forwarding behavior of the network, while the actual forwarding decision are left for the *control plane*. Practically, the control plane defines the contents of the `match-action` tables in a `P4` program. Therefore, P4AIG must be able to capture the control plane's behavior in order to accurately verify networking programs. One way to tackle this challenge is to over-approximate the behavior of the control plane using free inputs, i.e., inputs whose values are non-deterministically assigned at verification time. However, this approach suffers from two main drawbacks: (1) it introduces many false positives and (2) it explodes the search space thus increasing the verification complexity.

P4AIG overcomes this challenges by analyzing the control plane program and generating FOL invariants (②) that are then passed into P4AIG's main translation unit. These invariants will serve as constraints that P4AIG will use to prune the search space over the control plane actions. Compared to other approaches [4], P4AIG does not require control plane annotations and can easily be used to verify the same `P4` program with different control plane implementations.

Generating and Verifying Sequential Circuits: P4AIG's main translation unit (③) takes as input the annotated `P4` program as well the generated control plane invariants and produces an equivalent sequential circuit, encoded as an *And-Inverter-Graph* (AIG). Figure 2 shows a sample P4 packet parser program (`basic.p4`) [2] and the equivalent representation of its control flow as a sequential circuit.

Following the approach in [7], we will prove that the generated AIG is semantically equivalent to the original `P4` program; the generated AIG (④) encodes the bit-level semantics of the `P4` program and emulates its data plane pipeline. In addition, P4AIG translates the FOL specifications into bit-assertions that are to be used for the verification step.

P4AIG's pipeline concludes with the verification unit (⑤). It takes the generated sequential circuit along with its assigned bit-assertions and uses hardware verification tools, such as `ABC` [8], to check its validity. It reduces the size of the circuit using synthesis, rewriting, and abstraction techniques, and then verifies the specifications using circuit SAT solvers and bounded model checking. Any counter-example traces that result in a violation of the program's specifications will be passed back to the developers for debugging. We will develop P4AIG as an open-sourced tool and compare it to existing verification techniques using an array of `P4` programs.

[2] https://github.com/p4lang/tutorials/blob/master/exercises/basic/solution/basic.p4

References

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[2] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures." in *NDSS*, vol. 15, 2015, pp. 8–11.

[3] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas, "P4pktgen: Automated test case generation for p4 programs," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. New York, NY, USA: ACM, 2018, pp. 5:1–5:7.

[4] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, "P4v: Practical verification for programmable data planes," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: ACM, 2018, pp. 490–503.

[5] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging P4 programs with Vera," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: ACM, 2018, pp. 518–532.

[6] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho, and M. Barcellos, "Uncovering bugs in P4 programs with assertion-based verification," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. New York, NY, USA: ACM, 2018, pp. 4:1–4:7.

[7] M. A. Noureddine and F. A. Zaraket, "Model checking software with first order logic specifications using AIG solvers," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 741–763, Aug 2016.

[8] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 24–40.