UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GABRIEL NUNES MARTINS

Validando modelos para verificação de programas P4 por execução simbólica

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Marinho Pilla Barcellos
Co-orientador: Miguel Cardoso Neves

Porto Alegre
2018

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Profª. Jane Fraga Tutikian
Pró-Reitor de Graduação: Prof. Wladimir Pinheiro do Nascimento
Diretora do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Ciência de Computação: Prof. Raul Fernando Weber
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço, em primeiro lugar, a Carlos Alberto Martins e Patrícia Nunes, meu pai e minha mãe. Agradeço por terem me dado todas as condições necessárias para que eu me dedicasse aos meus estudos e por terem sempre me incentivado a ser uma pessoa melhor.

Agradeço ao meu padrasto, Rodrigo Franco, por ter sido sempre atencioso e prestativo, tanto no período de desenvolvimento deste trabalho, quanto nas mais diversas ocasiões ao longo dos últimos 14 anos. Agradeço também às minhas irmãs, Rafaela Martins e Luísa Franco, e aos demais membros da minha família pelo carinho e por terem me acompanhado durante a graduação.

Agradeço à minha namorada, Raquel Horvath, principalmente nestes últimos meses de graduação. Agradeço por ter estado comigo em todos os momentos, tanto de alegria quanto de tristeza, e por todo apoio e compreensão.

Finalmente, agradeço ao meu orientador, Marinho Barcellos, e, especialmente, ao meu co-orientador, Miguel Neves, por terem me guiado durante a elaboração deste trabalho, e à Universidade e demais professores, por terem me propiciado um excelente ambiente de aprendizado e um ensino de qualidade.

RESUMO

A linguagem P4 permite a programação do plano de dados de dispositivos de rede, facilitando a criação de novos protocolos e funcionalidades. No entanto, ao passo que planos de dados programáveis aumentam a flexibilidade de Redes Definidas por Software (SDN), aumentam também a chance de erros devido à possibilidade de bugs nos programas implementados. A fim de prevenir falhas provenientes da programação do plano de dados, técnicas de teste e verificação podem ser aplicadas para encontrar erros antes da implantação de softwares nos dispositivos de rede. Neste trabalho, é apresentada uma metodologia de validação de modelos para a ferramenta de verificação de programas P4 *assert-p4*.

Palavras-chave: SDN. P4. Planos de dados programáveis. Verificação e validação de software.

Validating models for verification of P4 programs through symbolic execution

ABSTRACT

The P4 programming language allows a network device's dataplane to be programmed, simplifying the introduction of new protocols and features. However, while programmable dataplanes improve flexibility for Software-Defined Networking (SDN), they also increase the chance of errors due to possible bugs in the implemented software. In order to prevent failures arising from dataplane programmability, testing and verification techniques can be applied to identify errors before a software's implementation on network devices. In this work, we present a model validation methodology for *assert-p4*, a P4 program verification tool.

Keywords: SDN. P4. Programmable dataplanes. Software verification and validation.

LISTA DE FIGURAS

Figura 2.1 Comparativo entre modelo de SDN tradicional (a) e SDN com plano de dados reprogramável (b). 13
Figura 2.2 Seções de código P4 associadas ao modelo abstrato de encaminhamento proposto por Bosshart et al. (2014). 14
Figura 2.3 Exemplo de cabeçalhos e parser de um programa P4. 15
Figura 2.4 Exemplo de bloco de controle P4, responsável por encaminhamento de pacotes IPv4. 16
Figura 2.5 Implantação de um programa P4 em um switch programável. 17
Figura 2.6 Passos para a verificação de um programa através da ferramenta *assert-p4*. 20
Figura 2.7 Gramática da linguagem de asserções de *assert-p4*. 21
Figura 2.8 Exemplo de bloco de controle de um programa P4 anotado com asserções. 22
Figura 2.9 Exemplo de tradução de cabeçalho e *parser* P4 para C realizada por *assert-p4* 23
Figura 2.10 Exemplo de tradução de ações, tabelas e blocos de controle P4 para C realizada por *assert-p4*. 24
Figura 2.11 Fragmento de programa P4 anotado com asserções e adaptação do modelo C correspondente gerado por *assert-p4*. 25
Figura 2.12 Esquema de execução simbólica de modelo C gerado por *assert-p4* (simplificado). 26

Figura 3.1 Algoritmo para validação de modelos C gerados com *assert-p4*. 28
Figura 3.2 Tempo decorrido para validação do modelo gerado por *assert-p4* para o programa Switch.p4. 37

LISTA DE TABELAS

Tabela 3.1 Versões de softwares e ferramentas utilizados para implementação do script de validação....33
Tabela 3.2 Programas P4 selecionados para o processo de validação. 34
Tabela 3.3 Tempo decorrido durante o processo de validação dos estudos de caso selecionados. 37

Tabela 4.1 Erros de compilação em modelos C produzidos pelo protótipo do *assert-p4*....38

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
BMv2	Behavioral Model version 2
DSL	Domain Specific Language
GCL	Guarded Command Language
HSA	Header Space Analysis
IP	Internet Protocol
IPv4	Internet Protocol version 4
NAT	Network Address Translation
NOD	Network Optimized Datalog
NOS	Network Operating System
P4	Programming Protocol-independent Packet Processors
PCAP	Packet Capture
POF	Protocol Oblivious Forwarding
RFC	Request for Comments
SAT	Boolean Satisfiability Problem
SEFL	Symbolic Execution Friendly Language
SDN	Software-Defined Networking
SMT	Satisfiability Modulo Theories
TTL	Time To Live

SUMÁRIO

1 INTRODUÇÃO **10**
2 FUNDAMENTAÇÃO TEÓRICA **12**
2.1 Redes Definidas por Software **12**
2.2 Linguagem P4 .... **13**
2.3 Verificação de Redes **14**
2.3.1 Tipos de propriedades 15
2.3.2 Ferramentas de verificação 17
2.4 *assert-p4* **19**
2.4.1 Especificação de asserções 20
2.4.2 Geração de modelos 21
2.4.3 Execução Simbólica 24
3 VALIDAÇÃO DE MODELOS C **27**
3.1 Metodologia **27**
3.1.1 Geração de pacotes de entrada e regras de encaminhamento 28
3.1.2 Geração e execução de modelos C 29
3.1.3 Processamento em um dispositivo de rede 30
3.1.4 Comparação de resultados 31
3.1.5 Implementação 31
3.2 Seleção de Estudos de Caso **33**
3.3 Resultados **34**
3.3.1 Falhas de modelagem 35
3.3.2 Desempenho 36
4 REPRODUTIBILIDADE **38**
5 CONSIDERAÇÕES FINAIS **41**
REFERÊNCIAS **43**

1 INTRODUÇÃO

Avanços na área de Redes Definidas por Software (SDN) tornaram possível a programação do plano de dados de dispositivos de rede. Usando linguagens como P4 (BOSSHART et al., 2014), operadores de rede podem desenvolver, customizar e implementar novos protocolos e serviços. Apesar desta flexibilidade permitir o rápido atendimento de novas demandas das aplicações, mudanças não-verificadas em funcionalidades da rede podem introduzir falhas, comportamentos inesperados e brechas de segurança (DOBRESCU; ARGYRAKI, 2014; LOPES et al., 2016).

Processos de engenharia de software aplicam técnicas de verificação e validação para determinar se um software se comporta da maneira esperada (WALLACE; FUJII, 1989). No contexto de verificação de redes, ferramentas como Anteater (MAI et al., 2011), Veriflow (KHURSHID et al., 2013) e NetPlumber (KAZEMIAN et al., 2013) foram desenvolvidas para captura de bugs em redes aplicando algumas destas técnicas. Entretanto, tais ferramentas não são capazes de verificar planos de dados programáveis, uma vez que possuem um modelo estático de encaminhamento de pacotes embutido.

Recentemente, a ferramenta *assert-p4* (FREIRE et al., 2018) foi apresentada como alternativa para verificação de redes cujo plano de dados pode ser programado através de P4. Para este fim, a ferramenta gera um modelo C a partir do programa P4 e faz sua verificação através de execução simbólica. Entretanto, até o presente momento, não há prova de que o processo de tradução de código P4 para C é correto.

Este trabalho objetiva a validação dos modelos C gerados pela ferramenta *assert-p4*, a fim de mostrar que o comportamento destes é consistente com o programa P4 sendo verificado. Também, apresenta o resultado de um esforço de documentação do protótipo apresentado por Freire et al. (2018), que, além de facilitar a utilização da ferramenta, torna possível a reprodutibilidade dos experimentos descritos no artigo.

A metodologia de validação proposta neste trabalho, inspirada na estratégia apresentada por Stoenescu et al. (2016), compara pacotes de saída emitidos pelos modelos gerados por *assert-p4* e por um switch virtual, de forma que a identificação de divergências nos pacotes aponta inconsistências no comportamento do modelo. Através da aplicação deste procedimento de validação, foi possível a identificação e correção de bugs no processo de tradução de código P4 para C, aumentando, consequentemente, a confiabilidade do processo de verificação de *assert-p4*.

O restante deste trabalho está organizado da seguinte forma: o Capítulo 2 aborda

conceitos fundamentais sobre verificação de redes, execução simbólica e a ferramenta *assert-p4*; o Capítulo 3 desenvolve a metodologia de validação dos modelos C gerados pela ferramenta de verificação, enquanto o Capítulo 4 apresenta os esforços para permitir a reprodutibilidade dos experimentos de Freire et al. (2018) e operacionalização de *assert-p4*; finalmente, o Capítulo 5 sintetiza os tópicos discutidos introduzindo assuntos a serem explorados em trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os principais conceitos relacionados a este trabalho. Primeiramente, são introduzidos fundamentos de Redes Definidas por Software e a linguagem P4. Após, são apresentadas ferramentas e técnicas para a verificação de redes. Finalmente, é apresentada a ferramenta de verificação *assert-p4*.

2.1 Redes Definidas por Software

Em uma arquitetura de rede tradicional, o plano de controle e o plano de dados são acoplados e embutidos nos dispositivos de rede. O plano de controle é responsável pela configuração do dispositivo, enquanto o plano de dados é responsável pela movimentação, de fato, dos pacotes na infraestrutura de rede. Neste modelo, o controle do funcionamento da rede é descentralizado, sendo necessária configuração individual de cada dispositivo de rede. Nesta arquitetura tradicional, o desenvolvimento de novas funcionalidades e aplicações é complexo e limitado (BENSON; AKELLA; MALTZ, 2009).

Redes Definidas por Software (do inglês, *Software-Defined Networking* - SDN) é um paradigma de rede em que o plano de controle e o plano de dados dos dispositivos são desacoplados. Desta forma, roteadores e switches se tornam simples encaminhadores de pacotes enquanto a lógica de controle é centralizada e implementada em um controlador, também conhecido como sistema operacional de rede ou NOS[1] (KREUTZ et al., 2015).

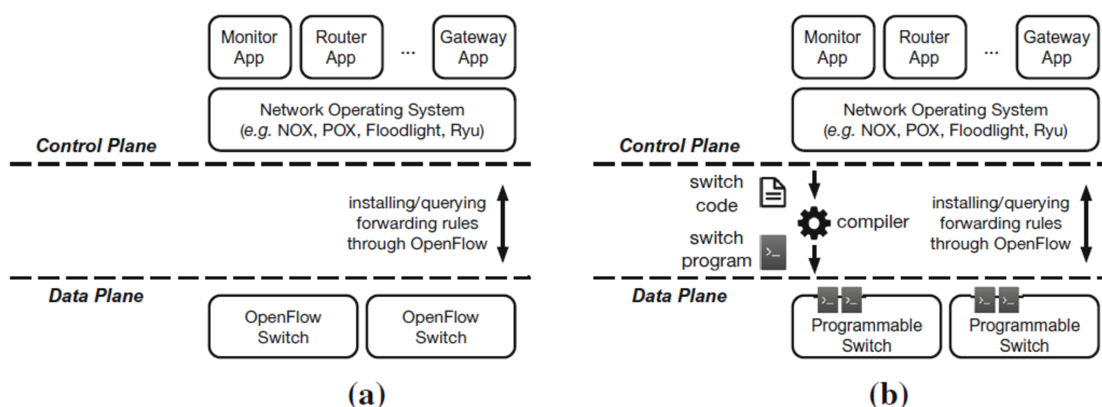
Em um sistema operacional de rede como NOX (GUDE et al., 2008), operadores de rede desenvolvem aplicações que realizam funções como roteamento IP, monitoramento de enlaces, balanceamento de carga, entre outros. Utilizando APIs bem definidas como OpenFlow (MCKEOWN et al., 2008), o controlador pode inserir, modificar e remover regras de encaminhamento de roteadores e switches, alterando o comportamento destes dispositivos de maneira dinâmica. Esta habilidade de programar em alto nível a rede através de aplicações que executam em um NOS é considerado um dos maiores benefícios de SDNs (KREUTZ et al., 2015).

Ainda que inicialmente SDNs tenham facilitado a reconfiguração de dispositivos de rede, a modificação destes dispositivos está limitada às funções e protocolos definidos por seus fabricantes (CORDEIRO; MARQUES; GASPARY, 2017). Para endereçar esta limitação, surgiu o conceito de programabilidade do plano de dados, onde, através

[1] Do inglês, *Network Operating System*.

de linguagens de domínio específico (DSL[2]) como P4 (BOSSHART et al., 2014) e POF (SONG, 2013), tornou-se possível a implementação de novos protocolos e comportamentos de encaminhamento em dispositivos de rede. A Figura 2.1 ilustra a diferença em SDNs com a presença de plano de dados programáveis.

Figura 2.1: Comparativo entre modelo de SDN tradicional (a) e SDN com plano de dados reprogramável (b).



Fonte: (CORDEIRO; MARQUES; GASPARY, 2017).

2.2 Linguagem P4

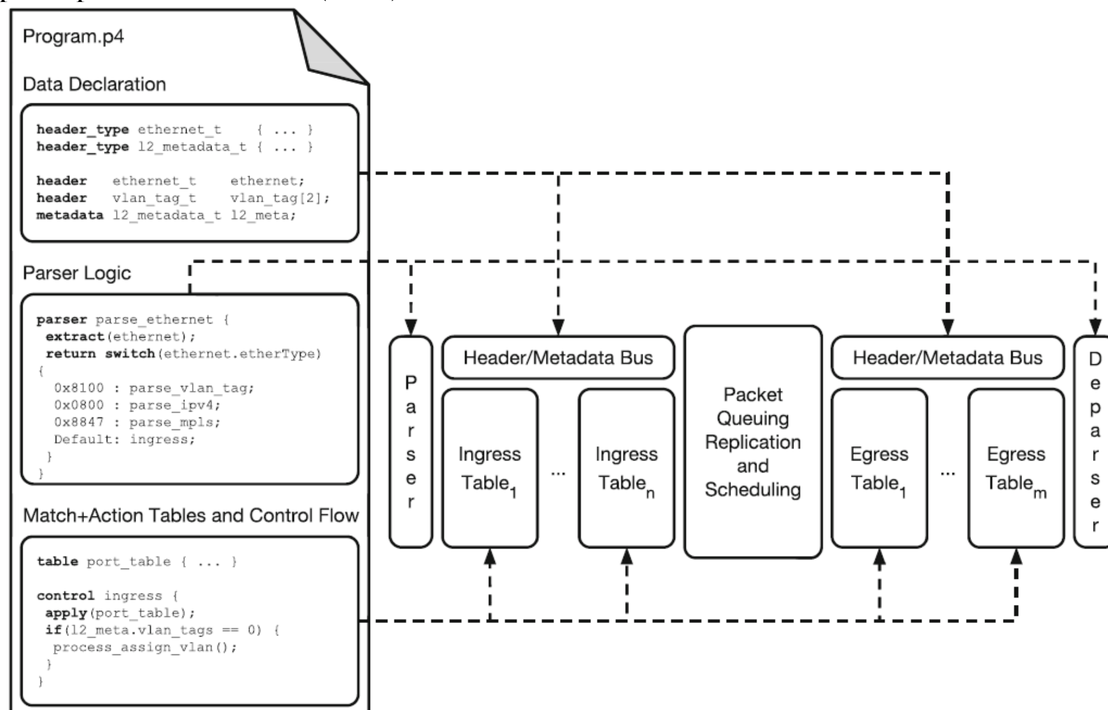
P4 é uma linguagem de programação que possibilita a configuração do plano de dados de dispositivos de rede (BOSSHART et al., 2014). Através de P4, operadores podem definir a forma como pacotes são processados em um *switch* sem conhecer detalhes do hardware do dispositivo.

Um programa P4 é logicamente organizado em seções (declaração de dados, *parsing*, tabelas de encaminhamento e blocos de controle) segundo um modelo abstrato de encaminhamento (BOSSHART et al., 2014). Este modelo, definido por Bosshart et al. (2014), generaliza a forma como pacotes são processados em diferentes dispositivos de encaminhamento. A Figura 2.2 ilustra o mapeamento de seções de código P4 a elementos do modelo abstrato de encaminhamento.

Os principais elementos de um programa P4 são os *parsers* e os blocos de controle com suas tabelas e ações. Um parser é uma máquina de estados cujo objetivo é a extração de dados dos cabeçalhos de um pacote e metadados associados ao processamento desse pacote. Os blocos de controle, por sua vez, realizam o processamento dos dados extraídos pelo *parser*, executando ações segundo regras de encaminhamento definidas para tabe-

[2]Do inglês, *Domain Specific Language*

Figura 2.2: Seções de código P4 associadas ao modelo abstrato de encaminhamento proposto por Bosshart et al. (2014).



Fonte: (CORDEIRO; MARQUES; GASPARY, 2017).

las, posicionando cabeçalhos em pacotes de saída (*deparsing*) e realizando cálculos de soma de verificação. A Figura 2.3 mostra a definição de um parser que realiza a extração de cabeçalhos no pacote recebido pelo switch P4, enquanto a Figura 2.4, por sua vez, exemplifica um bloco de controle responsável pelo encaminhamento de pacotes IPv4.

Desenvolvedores P4 elaboram programas para um switch alvo baseado em uma arquitetura específica definida pelos fabricantes do dispositivo, que também devem fornecer um compilador compatível. O processo de compilação gera uma configuração que implementa a lógica de encaminhamento descrito no programa P4 e uma API para comunicação entre o plano de controles e plano de dados. A Figura 2.5 ilustra os passos necessários para a implantação de um programa P4 em um dispositivo.

2.3 Verificação de Redes

O diagnóstico de problemas em redes é um processo necessário para evitar prejuízos causados pelo funcionamento incorreto de serviços e aplicações, o que motiva o desenvolvimento de ferramentas para auxiliar o processo de verificação de redes (MAI et al., 2011; KHURSHID et al., 2013). Para tal fim, foram desenvolvidas ferramentas utilizando diversas técnicas de verificação para analisar propriedades distintas do funcio-

Figura 2.3: Exemplo de cabeçalhos e parser de um programa P4.

```
header ethernet_t {
   bit<48>  dstAddr;
   bit<48>  srcAddr;
   bit<16>  etherType;
}

header ipv4_t {
   bit<4>   version;
   bit<4>   ihl;
   bit<8>   diffserv;
   bit<16>  totalLen;
   bit<16>  identification;
   bit<3>   flags;
   bit<13>  fragOffset;
   bit<8>   ttl;
   bit<8>   protocol;
   bit<16>  hdrChecksum;
   bit<32>  srcAddr;
   bit<32>  dstAddr;
}

struct headers {
   ethernet_t ether;
   ipv4_t     ipv4;
}
```

```
parser MyParser(packet_in packet,
   out headers hdr,
   inout metadata meta,
   inout std_metadata_t standard_metadata) {

   state start {
      transition parse_ethernet;
   }

   state parse_ethernet {
      packet.extract(hdr.ether);
      transition select(hdr.ether.etherType) {
         0x800: parse_ipv4;
         default: accept;
      }
   }

   state parse_ipv4 {
      packet.extract(hdr.ipv4);
      transition accept;
   }
}
```

Fonte: Adaptado de <https://github.com/p4lang/tutorials/blob/master/exercises/basic/solution/basic.p4>.

namento das redes.

A verificação de redes pode ser realizada através de análise do plano de controle e do plano de dados. Abordagens de verificação sob o plano de controle são geralmente limitadas à verificação de protocolos específicos ou *firewalls* (MAI et al., 2011). Por outro lado, abordagens de verificação sob o plano de dados verificam o modelo de encaminhamento de pacotes diretamente e são capazes de encontrar bugs no software de switches e roteadores.

2.3.1 Tipos de propriedades

A área de verificação do plano de dados é tradicionalmente dividida em duas categorias ortogonais: verificação de rede e verificação de dispositivos de rede. Enquanto a primeira opera sob um único modelo gerado a partir da combinação de todos os planos de dados dos dispositivos da rede e suas configurações, a segunda aborda os softwares

Figura 2.4: Exemplo de bloco de controle P4, responsável por encaminhamento de pacotes IPv4.

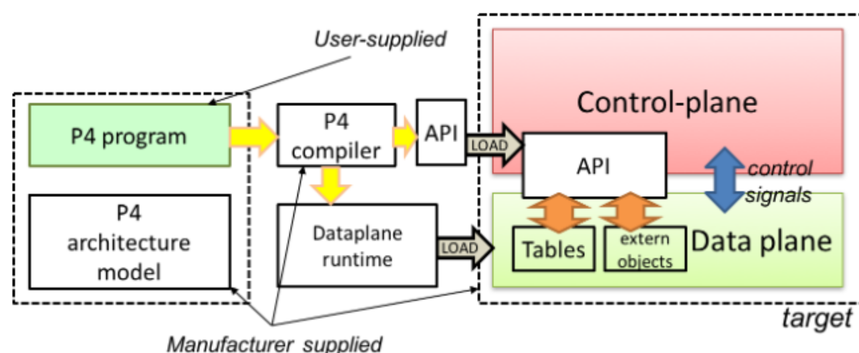
```
control MyIngress(inout headers hdr,
   inout metadata meta,
   inout std_metadata_t standard_metadata) {

   action drop() {
      mark_to_drop();
   }

   action ipv4_forward(bit<48> dstAddr,
                       bit<9> port) {
      standard_metadata.egress_spec = port;
      hdr.ether.srcAddr = hdr.ether.dstAddr;
      hdr.ether.dstAddr = dstAddr;
      hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
   }

table ipv4_lpm {
      key = {
         hdr.ipv4.dstAddr: lpm;
      }
      actions = {
         ipv4_forward;
         drop;
         NoAction;
      }
      size = 1024;
      default_action = drop();
   }

   apply {
      if (hdr.ipv4.isValid()) {
         ipv4_lpm.apply();
      }
   }
}
```

Fonte: Adaptado de <https://github.com/p4lang/tutorials/blob/master/exercises/basic/solution/basic.p4>.

implantados em cada um destes dispositivos individualmente (ZAOSTROVNYKH et al., 2017).

Durante o processo de verificação, são analisadas *propriedades de rede*, relativas ao comportamento da rede como um todo, e *propriedades de software*, que se referem ao comportamento individual de um dispositivo. Exemplos de propriedades de rede são conectividade (pacotes emitidos por um hospedeiro eventualmente alcançam seus destinatários), ausência de laços (pacotes nunca voltam a um mesmo ponto da rede com exatamente os mesmos cabeçalhos e conteúdos) e controle de acesso (pacotes emitidos por hospedeiros não-autorizados são descartados) (CASADO; FOSTER; GUHA, 2014). Propriedades de software, por outro lado, envolvem *crash-freedom* (não há sequência de pacotes que cause interrupção do funcionamento do dispositivo de rede), *bounded-execution* (nenhuma sequência de pacotes causa a execução de um número infinito de instruções) e conformidade com especificação de um protocolo (o programa implementa corretamente os comportamentos do protocolo conforme definido em sua RFC (DOBRESCU; ARGYRAKI, 2014).

Figura 2.5: Implantação de um programa P4 em um switch programável.



Fonte: <https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html>

2.3.2 Ferramentas de verificação

As ferramentas de verificação de redes existentes utilizam uma variedade de técnicas dependendo das propriedades a serem verificadas, sejam elas no plano de controle ou no plano de dados. A seguir, são descritas algumas ferramentas para análise de propriedades sob o plano de dados, com maior enfoque às que analisam propriedades de programa.

As ferramentas de verificação do plano de dados analisam propriedades a partir de um *snapshot* da rede, que contém informações sobre o estado dos dispositivos em um dado momento (como, por exemplo, as regras de encaminhamento instaladas). Anteater (MAI et al., 2011) converte o plano de dados em fórmulas lógicas e propriedades de rede em instâncias do problema da satisfatibilidade booleana (SAT)[3], que podem então ser verificadas através de um solucionador[4]. HSA (KAZEMIAN; VARGHESE; MCKEOWN, 2012) propõe a modelagem do processamento de pacotes como transformações algébricas sobre um espaço geométrico para verificação de propriedades como alcançabilidade e ausência de laços. NetPlumber (KAZEMIAN et al., 2013) estende HSA realizando análises incrementais, o que possibilita a verificação de propriedades de rede em tempo real.

A elaboração de técnicas e ferramentas de verificação capazes de comportar mudanças no modelo de encaminhamento de pacotes facilita a análise de planos de dados programáveis. Em NOD (LOPES et al., 2015), tanto a definição de propriedades quanto do modelo de rede é realizada utilizando Datalog, permitindo a definição de propriedades de alto nível e alterações no modelo de rede a partir de inserções de novas regras.

[3]O problema da satisfatibilidade booleana (SAT) consiste em verificar se existe uma valoração possível para variáveis de uma fórmula booleana de tal forma que esta seja satisfazível ("verdadeira").

[4]Solucionadores são ferramentas desenvolvidas para resolver eficientemente instâncias de problemas computacionais (como, por exemplo, SAT).

Baseado em NOD, Lopes et al. (2016) propôs uma solução que traduz programas P4 para regras em Datalog, possibilitando alterações automáticas no modelo de rede, além da identificação de bugs como má-formação de pacotes. A ferramenta p4v (LIU et al., 2018), por outro lado, gera modelos de rede através da conversão de programas P4 para linguagem de comandos com guarda[5] (GCL), a partir dos quais é possível, com auxílio de um *solucionador*, verificar propriedades específicas de programa.

Para verificação de propriedades de programa, ferramentas utilizam, predominantemente, a técnica de execução simbólica (DOBRESCU; ARGYRAKI, 2014; STOENESCU et al., 2016; ZAOSTROVNYKH et al., 2017; FREIRE et al., 2018; STOENESCU et al., 2018). Devido à popularidade desta técnica, este trabalho dá maior enfoque às suas características comparado às outras metodologias descritas anteriormente. A execução simbólica consiste na exploração de todos os caminhos de execução factíveis de um programa (BALDONI et al., 2018). A cada bifurcação no programa sendo verificado, novos caminhos de execução são criados com restrições de factibilidade[6]. Tais restrições são analisadas através de solucionadores SMT[7] e, caso não sejam atendidas, a execução do caminho é interrompida.

Dobrescu e Argyraki (2014) propuseram uma ferramenta capaz de verificar propriedades de filtragem e *crash-freedom* simbolicamente executando módulos do roteador Click (KOHLER et al., 2000). Zaostrovnykh et al. (2017), por sua vez, utilizou execução simbólica para verificar formalmente um NAT implementado em C. SymNet (STOENESCU et al., 2016) é uma plataforma que executa simbolicamente modelos SEFL[8], gerados a partir da conversão de configurações de switches e roteadores, para verificação de propriedades do plano de dados. Vera (STOENESCU et al., 2018) realiza conversão de programas P4 em instruções SEFL que são simbolicamente executados na SymNet para verificação automática de propriedades de programa. De maneira similar, a ferramenta *assert-p4* (FREIRE et al., 2018) - descrita com detalhes na Seção 2.4 - traduz código P4 anotado com asserções para código C, que é verificado através de execução simbólica na plataforma KLEE (CADAR et al., 2008).

[5] *Guarded Command Language* é uma linguagem imperativa que define comandos que somente podem ser executados caso determinada proposição (ou seja, sua guarda) seja "verdadeira".

[6] Por exemplo, um caminho pode ter a restrição $x > 0 \land y < 5$, que é traduzida como *a variável simbólica x somente pode assumir valores maiores que 0 e a variável simbólica y somente pode assumir valores menores que 5.*

[7] *Satisfiability Modulo Theories* (SMT) é um problema de decisão similar ao SAT, porém com fórmulas estendidas à lógica de primeira ordem.

[8] Do inglês, *Symbolic Execution Friendly Language* (em português, "linguagem propícia à execução simbólica"). SEFL é uma linguagem para modelagem de redes desenvolvida por Stoenescu et al. (2016).

Validação de modelos. Muitas ferramentas de verificação geram modelos para análise de propriedades tanto de rede quanto de software. Entretanto, é necessário que estes modelos reflitam com precisão a realidade, tornando imprescindível a sua validação. A seguir, são exemplificadas duas estratégias distintas empregadas para validação de modelos.

Lopes et al. (2016) formalizou a semântica operacional de P4 para demonstrar que a conversão de programas P4 para modelos Datalog é correta. A semântica operacional *big-step* da linguagem define relações no formato $S, \mathcal{E} \xrightarrow{instr} S'$, que descreve o estado antes (S) e depois (S') da execução de uma instrução ($instr$) em um determinado ambiente de execução ($\mathcal{E}$), formalizando o comportamento das expressões da linguagem. A partir de regras de semântica operacional, a conversão em regras Datalog pode então ser realizada de maneira simples e intuitiva (LOPES et al., 2016).

Para validar modelos SEFL, Stoenescu et al. (2016) propôs uma metodologia de validação que utiliza o resultado do processamento de pacotes em um dispositivo de rede real para gerar restrições aos caminhos de execução simbólica do modelo sob a plataforma SymNet. Para validar um caminho de execução, seu processamento é temporariamente interrompido e suas restrições de factibilidade são extraídas. Estas restrições (descritas em fórmulas booleanas) são resolvidas com o solucionador Z3 (MOURA; BJØRNER, 2008) e utilizadas para gerar um pacote concreto. Como próximo passo, o roteador Click (KOHLER et al., 2000) processa este pacote e emite um pacote de saída, que é convertido em novas restrições a serem adicionadas ao caminho de execução sendo validado. O processamento do caminho é retomado, finalmente, após a inserção das novas restrições de factibilidade. Este procedimento é repetido para diversos caminhos gerados através de testes de alcançabilidade sob o modelo SEFL. Caso as novas restrições de pelo menos um dos caminhos de execução não sejam atendidas, o modelo é classificado como inválido.

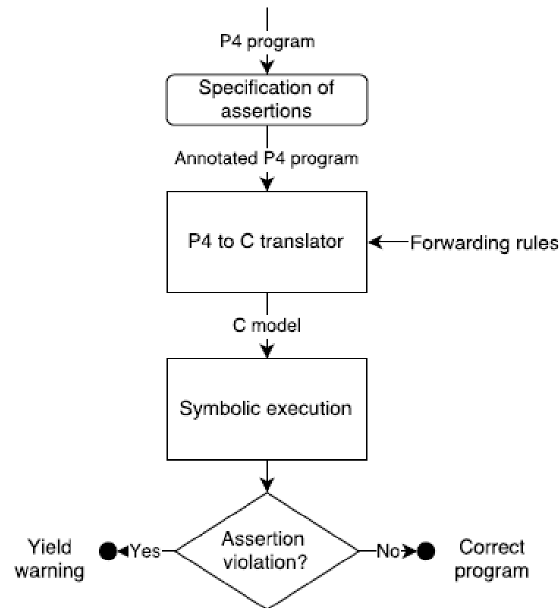
2.4 *assert-p4*

Os resultados obtidos no presente trabalho tornam mais confiável a verificação de programas P4 realizada por *assert-p4* (FREIRE et al., 2018). Para possibilitar a compreensão das contribuições descritas nos próximos capítulos, é necessário entendimento aprofundado desta ferramenta.

assert-p4 é uma ferramenta baseada em asserções e execução simbólica para verificação de programas P4 (FREIRE et al., 2018). A partir de uma linguagem de asserções,

desenvolvedores definem propriedades desejadas no programa P4, que são então verificadas através da execução simbólica de um modelo C gerado a partir do código P4. A Figura 2.6 mostra o fluxo de execução de *assert-p4* para a verificação de um programa.

Figura 2.6: Passos para a verificação de um programa através da ferramenta *assert-p4*.



Fonte: (FREIRE et al., 2018)

2.4.1 Especificação de asserções

Para expressar o funcionamento esperado em um programa P4, (FREIRE et al., 2018) desenvolveu uma linguagem de asserções para capturar características do processamento de pacotes no plano de dados de um dispositivo. Tais asserções podem ser inseridas através de anotações do tipo $assert$ no código P4.

A Figura 2.7 especifica a gramática da linguagem de asserções. As asserções podem ser classificadas em dois tipos: restritas à localidade, que testam o valor de uma variável no ponto do programa em que a asserção é especificada (por exemplo, *o campo* TTL *do cabeçalho IPv4 deve ser maior que* 0), e não-restritas à localidade, que, por sua vez, definem propriedades em mais alto nível que devem ser aplicáveis em todo o programa a partir do seu ponto de definição (por exemplo, *se a ação* $drop$ *foi executada, o pacote não deve ser encaminhado até o final da execução do programa*).

Cada asserção é composta por conjuntos de expressões booleanas e aritméticas, além dos métodos *forward*, *traverse_path*, *constant*, *if*, *extract_header* e *emit_header*. Tanto as expressões quanto os métodos atuam sobre um ou mais valores, campos de ca-

Figura 2.7: Gramática da linguagem de asserções de *assert-p4*.

```
b ::= v                m ::= forward()
    | f                    | traverse_path()
    | m                    | constant(f)
    | !b                   | if(b, b, [b])
    | b || b               | extract_header(h)
    | b && b               | emit_header(h)
    | b == b           i ::= v
    | b != b               | f
    | i >= i               | i * i
    | i <= i               | i / i
    | i < i                | i % i
    | i > i                | i + i
    | i == i               | i - i
    | i != i
```

Fonte: (FREIRE et al., 2018)

beçalho ou cabeçalhos, e são, ao final, avaliadas para "verdadeiro" ou "falso". O método $forward()$ retorna "verdadeiro" se o pacote não for descartado até o final da execução do programa. $traverse_path()$ indica se determinado bloco (por exemplo, bloco de controle) será percorrido até o final da execução do programa. Já o método $constant(f)$ retorna "verdadeiro"se o campo f não for modificado a partir da posição da asserção até o final da execução do programa. $if(b_1, b_2[, b_3])$ avalia a expressão b_2 caso b_1 seja verdadeiro ou b_3 caso contrário. $extract_header(h)$ é avaliado para "verdadeiro"se o cabeçalho h é extraído do pacote de entrada em algum momento do programa. Finalmente, $emit_header(h)$ é avaliado para "verdadeiro" se o pacote de saída transmitido contiver o cabeçalho h. Os métodos apresentados por esta linguagem de asserções possibilitam a especificação de propriedades que seriam difíceis ou impossíveis de expressar usando apenas asserções tradicionais (FREIRE et al., 2018).

A Figura 2.8 exemplifica um bloco de controle anotado com asserções. A asserção da linha 6 especifica que pacotes marcados para serem descartados não podem ser encaminhados: se um caminho executar a ação $Drop$ ($if(traverse_path(), ...)$), então o pacote deve ser descartado até o final da execução deste caminho ($!forward$). Já a asserção da linha 23 especifica que pacotes encaminhados devem conter TTL maior que 0: se o pacote for encaminhado ($if(forward(), ...)$), então o campo ttl do cabeçalho IP deve ser maior que 0 ($headers.ip.ttl > 0$).

2.4.2 Geração de modelos

Após anotação com asserções, o próximo passo para verificação com *assert-p4* é a conversão do programa P4 em um modelo C, que será posteriormente executado sim-

Figura 2.8: Exemplo de bloco de controle de um programa P4 anotado com asserções.

```
1   control TopPipe(inout Parsed_packet headers,
2      out OutControl outCtrl) {
3
4      action Drop() {
5         outCtrl.outputPort = DROP_PORT;
6         @assert("if(traverse_path(), !forward())");
7      }
8
9      action Set_dmac(bit<48> dmac) {
10        headers.ethernet.dstAddr = dmac;
11     }
12
13     table dmac {
14        key = { nextHop : exact; }
15        actions = { Drop; Set_dmac; }
16        default_action = Drop;
17     }
18
19     apply {
20        ...
21        dmac.apply();
22        ...
23        @assert("if(forward(), headers.ip.ttl > 0)");
24     }
25  }
```

Fonte: (FREIRE et al., 2018), adaptado.

bolicamente. Neste passo, estruturas do código P4 (cabeçalhos, tabelas, ações, blocos de controle e *parsers*) são traduzidos em estruturas semanticamente - e ocasionalmente sintaticamente - similares da linguagem C. A Figura 2.9 exemplifica a modelagem C para cabeçalhos e *parsers* P4, enquanto a Figura 2.10, por sua vez, mostra a tradução de ações, tabelas e blocos de controle.

A tradução de cabeçalhos P4 é intuitiva devido à similaridade estrutural entre *headers* P4 e *structs* C. Cada campo do cabeçalho P4 é mapeado para um membro da estrutura C, onde o tipo deste é selecionado apropriadamente de forma que o tamanho em bits do campo seja preservado.

O *parser* do programa P4 é traduzido em múltiplas funções C (uma para cada estado) e uma função extra, utilizada para realizar a definição simbólica de cabeçalhos e para invocar a função que modela o estado inicial do parser ($start$).

Cada tabela do código P4 é traduzida em uma função C. Caso fornecidas entradas para as tabelas de encaminhamento ao *assert-p4*, o corpo da função C consiste de uma sequência de construtores $if/else$, onde cada condicional if modela uma entrada, enquanto o bloco $else$ modela a ação padrão da tabela, caso especificada. Já nos casos onde

Figura 2.9: Exemplo de tradução de cabeçalho e *parser* P4 para C realizada por *assert-p4*.

Estrutura	Código P4	Modelagem C

Cabeçalho	**header** ethernet_t {
	   bit<48> dstAddr;
	   bit<48> srcAddr;
	   bit<16> etherType;
	}
		typedef struct {
		   uint8_t isValid : 1;
		   uint64_t dstAddr : 48;
		   uint64_t srcAddr : 48;
		   uint16_t etherType : 16;
		} ethernet_t;

Parser	**parser** TopParser(**packet_in** b,
	   out Parsed_packet hdr) {
	
	   state start {
	      transition parse_ethernet;
	   }
	
	   state parse_ethernet {
	      b.extract(hdr.eth);
	      transition select(hdr.eth.
	         etherType) {
	         0x0800: parse_ipv4;
	         default: accept;
	      }
	   }
	}
		Parsed_packet hdr;
		
		void TopParser() {
		   make_symbolic(hdr);
		   start();
		}
		
		void start() {
		   parse_ethernet();
		}
		
		void parse_ethernet() {
		   hdr.eth.isValid = 1;
		   switch(hdr.eth.etherType) {
		      case 0x0800:
		         parse_ipv4(); **break;**
		      default:
		         accept(); **break;**
		   }
		}

Fonte: (FREIRE et al., 2018), adaptado.

não são fornecidas entradas, é definida uma variável simbólica e um bloco $switch/case$ contendo todas as ações possíveis da tabela.

Similarmente a tabelas, ações P4 são modeladas como funções C e também variam de acordo com o fornecimento ou não de entradas de tabela, uma vez que os parâmetros para as ações (e, consequentemente, para as funções) são definidos nestas regras de encaminhamento. Caso entradas sejam fornecidas, os valores especificados nas regras são associados aos parâmetros da função; caso contrário, os parâmetros recebem valores simbólicos.

Como blocos de controle P4 são constituídos por diversas ações e tabelas, a modelagem em C é feita ao longo de diversas funções. Variáveis locais aos blocos de controle são definidas globalmente no código C com identificadores únicos. A função principal de um bloco de controle é gerada a partir do bloco $apply$ e é constituída de diversas chamadas

Figura 2.10: Exemplo de tradução de ações, tabelas e blocos de controle P4 para C realizada por *assert-p4*.

Estrutura	Código P4	Modelagem C

Ação	**action** forward(**bit**<9> port) {
	metadata.egress_spec = port;
	}
		void forward() {
		uint32_t port;
		make_symbolic(port);
		metadata.egress_spec = port;
		}

Tabela	**table** forward_table() {
	actions = {
	forward;
	NoAction;
	}
	key = {
	hdr.eth.dstAddr : **exact;**
	}
	size = 32;
	default_action = NoAction;
	}
		void forward_table() {
		int symbol;
		make_symbolic(symbol);
		switch(symbol) {
		case 0: forward(); **break;**
		default: NoAction(); **break;**
		}
		}

Bloco de Controle	**control** ingress(**inout** headers
	hdr, **inout** metadata meta) {
	
	apply {
	forward_table.apply()
	}
	}
		// variáveis globais
		headers hdr;
		metadatada meta;
		
		// bloco de controle
		void ingress() {
		forward_table();
		}

Fonte: (FREIRE et al., 2018), adaptado.

às funções que modelam as tabelas e ações associadas.

2.4.3 Execução Simbólica

Após a geração do modelo C, o próximo (e último) passo de verificação é a execução simbólica do modelo na plataforma de execução KLEE. À medida que a exploração de caminhos de execução do modelo é concluída, é informado ao usuário se alguma asserção previamente definida é violada.

Considere o fragmento de programa P4 e do modelo C gerado por *assert-p4* apresentado na Figura 2.11. Este programa realiza encaminhamento de pacotes IPv4 e foi anotado com uma asserção para garantir que pacotes descartados não sejam encaminhados. A semântica da asserção pode ser traduzida como: *se a ação drop for executada, então o pacote não pode ser encaminhado*. Tanto a tabela quanto suas ações são mo-

deladas em funções C com o mesmo nome. Por fins de simplicidade, o corpo da ação $forward$ e da função correspondente no modelo é omitido. A asserção anotada na ação $drop$ é modelada por duas variáveis, $send_pkt$, que recebe valor 1 caso o pacote seja encaminhado (método $forward$ da asserção), e $traverse$, que recebe valor 1 caso a função $drop$ seja executada (método $traverse$). A asserção é violada caso ambas variáveis possuam valor 1 (ou seja, a ação $drop$ foi executada e o pacote foi encaminhado) ao final da execução de um caminho, representado pela função $check_assertions$. A Figura 2.12 ilustra parte do processo de execução simbólica do modelo C deste exemplo, conforme descrito a seguir.

Figura 2.11: Fragmento de programa P4 anotado com asserções e adaptação do modelo C correspondente gerado por *assert-p4*.

```
1  action drop() {
2    mark_to_drop();
3    @assert("if(traverse, !forward)");
4  }
5
6  table ipv4_exact {
7    key = { hdr.ipv4.dstAddr: exact; }
8    actions = {
9      forward;
10     drop;
11   }
12 }
```

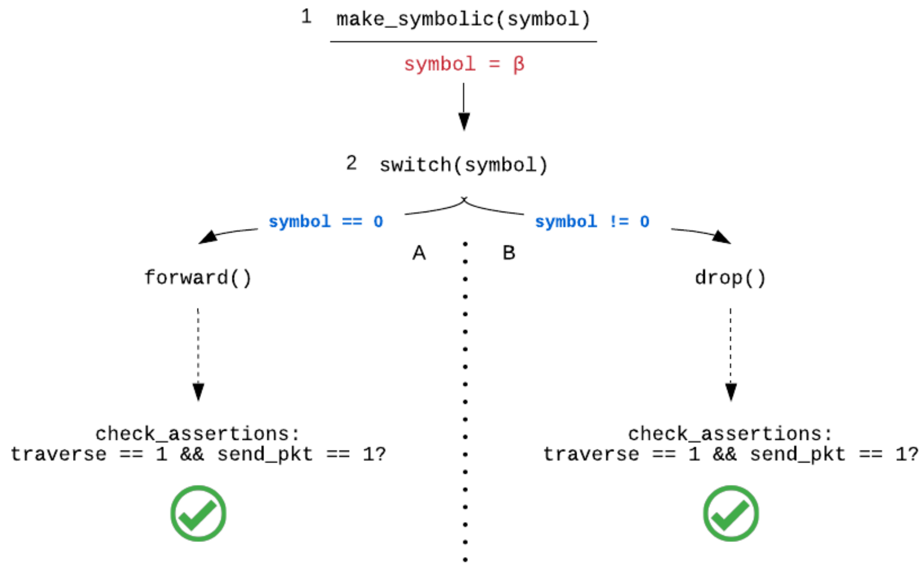
```
1  int send_pkt = 1;
2  int traverse = 0;
3
4  void drop() {
5    send_pkt = 0;
6    traverse = 1;
7  }
8
9  void forward() { ... }
10
11 void ipv4_exact() {
12   int symbol;
13   make_symbolic(symbol);
14   switch(symbol) {
15     case 0: forward(); break;
16     default: drop(); break;
17   }
18 }
19
20 ...
21
22 void check_assertions() {
23   if (traverse && send_pkt) {
24     printf("Error!");
25   }
26 }
```

Fonte: Autoria própria.

Ao percorrer a função $ipv4_forward$, a plataforma de execução simbólica atribui à variável $symbol$ um valor simbólico (1). Em seguida, a instrução $switch$ é executada. Uma vez que a variável $symbol$ possui um valor simbólico, a *engine* gera dois novos caminhos de execução: um em que a variável $symbol$ receberá valor igual a 0 e, portanto, executará a função $forward$ (A) e outro em que $symbol$ receberá valor diferente de 0 e executará a função $drop$ (B). Estes novos caminhos, então, são simultaneamente executa-

dos até o seu final. Como um último passo, a função $check_assertions$ será executada por ambos caminhos a fim de verificar se a asserção foi violada. Neste exemplo, nenhum dos dois caminhos ilustrados termina com valor 1 atribuído a ambas variáveis $traverse$ e $send_pkt$; logo, a asserção não é violada.

Figura 2.12: Esquema de execução simbólica de modelo C gerado por *assert-p4* (simplificado).



Fonte: Autoria própria.

3 VALIDAÇÃO DE MODELOS C

A verificação de propriedades sobre o *modelo* de um programa P4 ao invés do seu código original tem duas vantagens principais: permitir a utilização de ferramentas já consolidadas no processo de verificação; e tornar este processo mais simples, abstraindo detalhes de menor relevância. Entretanto, ao se verificar o modelo de um programa P4, assume-se como premissa que este modelo está correto, o que por vezes não é verdade (ZAOSTROVNYKH et al., 2017). Na prática, é comum que verificações efetuadas sobre modelos incorretos resultem em ataques e/ou falhas de grandes proporções (como, por exemplo, os ataques detalhados por Vanhoef e Piessens (2017)). Por essa razão, propomos uma metodologia para a validação dos modelos C gerados pela ferramenta *assert-p4*, a fim de garantir que estes representam fielmente os programas sendo verificados.

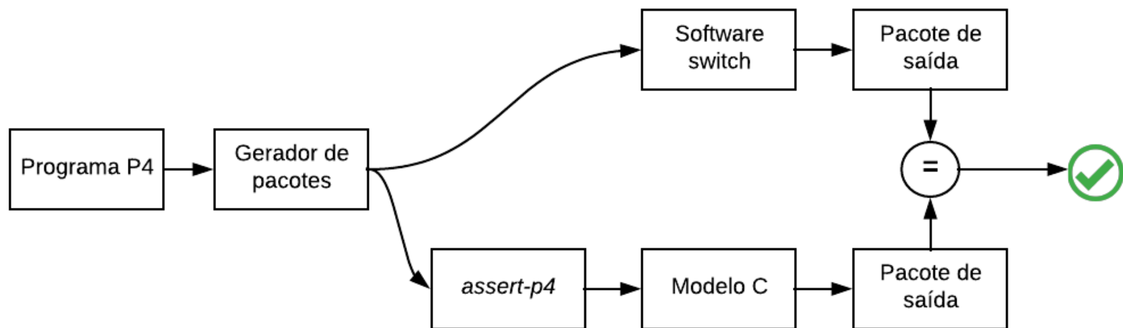
3.1 Metodologia

A metodologia descrita neste trabalho se baseia na comparação entre o modelo e o programa original através de testes de entrada e saída. A ideia-chave é testar, para entradas aleatórias (isto é, pacotes de dados), se o modelo é capaz de produzir os mesmos pacotes de saída que um dispositivo de rede configurado de maneira equivalente (ou seja, com o mesmo programa P4 e as mesmas regras de encaminhamento). A Figura 3.1 ilustra a metodologia proposta.

De acordo com a figura, um gerador de pacotes é utilizado para produzir tanto os pacotes de entrada quanto as regras de encaminhamento necessárias para gerar o modelo C e configurar o dispositivo de rede. Por simplicidade, neste trabalho consideramos o uso de software switches como dispositivos de rede. No entanto, a mesma metodologia pode ser aplicada para outros tipos de dispositivos (por exemplo, NetFPGAs, interfaces de rede inteligentes e switches de hardware). Em trabalhos futuros, pretendemos explorar diferentes dispositivos e suas particularidades (por exemplo, a necessidade de se modelar *externs* distintas).

Embora esta metodologia não garanta o mesmo nível de confiança que uma prova formal (como a de Lopes et al. (2016)), o fato de tratar tanto *assert-p4* quanto os dispositivos de rede utilizados nos testes como caixas pretas reduz o esforço necessário para suportar mudanças na especificação da linguagem P4, uma vez que não é preciso construir uma nova prova a cada atualização da linguagem. De forma semelhante, a metodologia

Figura 3.1: Algoritmo para validação de modelos C gerados com *assert-p4*.



Fonte: Autoria própria.

deste trabalho também é mais simples que a proposta por Stoenescu et al. (2016), uma vez que não requer a manipulação de *solvers* para gerar e analisar fórmulas lógicas que representem as informações tanto do modelo quanto dos respectivos casos de teste. A seguir, descrevemos em maiores detalhes cada etapa da metodologia proposta, bem como os desafios enfrentados na sua concretização.

3.1.1 Geração de pacotes de entrada e regras de encaminhamento

Cada caso de teste utilizado na validação de um modelo C é composto por um pacote de entrada e um conjunto de regras de encaminhamento. Neste caso, o número de combinações dessas duas variáveis pode ser infinitamente grande, tornando inviável explorar todos os casos de teste possíveis. Dessa forma, é necessário explorar o espaço de possibilidades de maneira minimamente satisfatória através da geração de casos de teste heterogêneos (isto é, que exploram diferentes caminhos de execução dentro de um programa ou modelo).

Desafio #1: Heterogeneidade dos casos de teste. O principal problema na utilização de um esquema de validação baseado em testes de entrada e saída é garantir que haja diversidade nos caminhos percorridos dentro do programa ou modelo. Por exemplo, se todos os pacotes de entrada compartilharem uma mesma sequência de cabeçalhos, os mesmos estados do *parser* serão executados em todos os testes, potencialmente ocultando falhas decorrentes da modelagem de outros estados. Quanto maior for a heterogeneidade dos casos de teste, maior será o nível de cobertura do procedimento de validação, o que aumenta o nível de confiabilidade do modelo. Para garantir a heterogeneidade dos casos de teste, adotamos técnicas como a exploração alternada de prefixos durante a implementação da metodologia de validação proposta. Por exemplo, se um teste percorre sequen-

cialmente os estados A e B do *parser*, o próximo teste não deve, caso possível, percorrer estes estados nesta mesma ordem.

Ao terminar a exploração de um caminho, são coletadas as diversas restrições para sua execução (por exemplo, os estados A e B do *parser* devem ser percorridos e as ações X e Y devem ser executadas). A geração de um teste, neste caso, consiste em materializar um conjunto de cabeçalhos (ou seja, um pacote) e regras de encaminhamento de forma que um programa P4 percorra o caminho ao processar estas entradas. Entretanto, há casos em que não é possível a geração de um pacote devido à sequência de estados percorridos no *parser*. Portanto, ao gerar casos de teste através de exploração de caminhos, devem ser desconsiderados estes "casos vazios".

3.1.2 Geração e execução de modelos C

A ferramenta *assert-p4* é capaz de gerar modelos com características distintas dependendo do cenário de verificação sendo abordado. Por exemplo, é possível considerar conjuntos específicos de regras de encaminhamento ou ainda assumir que estas contêm valores simbólicos quando se deseja uma análise mais abrangente envolvendo também o plano de controle. No entanto, em ambos os casos as variáveis de entrada dos programas (isto é, os campos de cabeçalhos que receberão as informações dos pacotes) são modelados como variáveis simbólicas. Neste caso, é necessário modificar os modelos gerados pela ferramenta a fim de tornar possível a leitura das informações de entrada referentes a cada caso de teste.

Desafio #2: Adaptação dos modelos para entradas concretas. Uma vez que *assert-p4* gera modelos apropriados para a execução simbólica (por exemplo, os cabeçalhos são inicializados com valores simbólicos), o algoritmo de validação deve realizar alterações estruturais de forma que o modelo seja capaz de processar entradas concretas referentes aos casos de teste gerados no passo anterior. Particularmente, cada modelo deve ser ajustado conforme as regras de encaminhamento descritas no caso de teste e deve ser capaz de processar os cabeçalhos contidos no pacote de entrada. Porém, quaisquer ajustes devem ser realizados de maneira menos intrusiva possível, uma vez que, dependendo das modificações realizadas, o processo de validação não estará mais validando o modelo que é utilizado de fato na verificação.

Como *assert-p4* é capaz de modelar regras de encaminhamento ao gerar um modelo C, este primeiro ajuste é trivial - basta gerar um novo modelo para cada caso. No

entanto, a modelagem do pacote de entrada é mais desafiadora: uma vez que um programa P4 pode definir cabeçalhos customizados e a ordem destes em um pacote, não é possível atribuir o conteúdo dos cabeçalhos de um pacote de teste diretamente às variáveis do modelo. Como solução, propomos ajustar o modelo de forma que este seja capaz de extrair cabeçalhos - ou seja, modificá-lo para que, à medida em que estados modelados do *parser* sejam percorridos, sejam progressivamente atribuídos valores aos cabeçalhos correspondentes - da mesma forma como ocorre em um dispositivo P4 real. Outra alteração necessária é a definição da porta de entrada do pacote no modelo. Neste caso, propomos que esta informação seja configurada antes do estado inicial do *parser*, para que esta informação esteja disponível antes do início do processamento de cabeçalhos (como em um dispositivo de rede).

O modelo C gerado por *assert-p4* não possui estruturas para exportar o resultado do processamento de pacotes, mas sim para indicar se asserções são violadas durante o processo. Portanto, um último ajuste é necessário para que este simule também o *deparser* P4, transformando os valores definidos nos cabeçalhos ao final da execução em um pacote, que deve ser então exportado juntamente com a porta de saída correspondente.

Conforme apresentado acima, propomos a alteração do modelo somente em dois locais: nos estados do *parser* e no *deparser*. Conforme o modelo abstrato de encaminhamento sob o qual a linguagem P4 foi construída, estes blocos restringem suas operações a, basicamente, extração e emissão de cabeçalhos, respectivamente. Sendo assim, ao ajustarmos somente estes locais do processamento de pacotes, diminuímos a chance de realizar alterações disruptivas no modelo.

3.1.3 Processamento em um dispositivo de rede

Para o procedimento de validação, um dispositivo de rede deve ser configurado com o programa P4 que foi utilizado tanto para geração de pacotes de entrada e regras de encaminhamento, quanto para a geração do modelo C. Após esta configuração, devem ser enviados ao dispositivo pacotes de entrada e capturados os pacotes de saída emitidos.

O envio de pacotes para processamento no dispositivo é trivial. Para cada caso de teste, as regras de encaminhamento devem ser configuradas e, em seguida, o pacote de entrada enviado através do enlace conectado à porta de entrada correspondente ao caso de teste (em conformidade com a porta informada ao modelo C).

Por outro lado, o processo de captura de pacotes de saída é mais trabalhoso, uma

vez que não se sabe, a priori, por qual interface do switch o pacote será emitido. Uma solução "força-bruta" é o monitoramento de todos os enlaces conectados ao switch, pois desta forma será possível capturar um pacote independentemente da porta utilizada para sua emissão. Entretanto, posto que o objetivo da captura de pacotes de saída é verificar se estes são produzidos da mesma forma no modelo e no dispositivo, basta monitorar a porta de saída indicada ao final da execução do modelo C para o mesmo caso de teste - se o pacote de saída for emitido em uma porta de saída diferente, a não-captura do pacote já é informação suficiente para verificar uma divergência de comportamento entre o modelo e o dispositivo de rede.

3.1.4 Comparação de resultados

Para determinar a validade de um modelo, os pacotes de saída emitidos pelo modelo e pelo dispositivo de rede devem possuir exatamente o mesmo conteúdo. Ademais, a porta através da qual estes pacotes foram emitidos também deve ser a mesma.

A validade do modelo é refutada caso haja divergência de resultados em pelo menos um dos casos de testes. Entretanto, ainda que o modelo e o dispositivo de rede produzam os mesmos pacotes de saída para todos os pacotes de entrada, não é *garantida* a validade do modelo, mas sim que este *se comporta de maneira consistente com o dispositivo*. A garantia de validade do modelo depende uma prova formal, como proposto por Lopes et al. (2016), mas deixamos a elaboração de tal prova para trabalho futuro, em um momento em que a especificação da linguagem P4 esteja mais madura e estável.

3.1.5 Implementação

Para avaliar a metodologia apresentada neste trabalho, utilizamos a ferramenta p4pktgen (NÖTZLI et al., 2018) para geração automática de casos de teste. Através de execução simbólica, p4pktgen gera pacotes de entrada e regras de encaminhamento, que podem ser utilizados para percorrer caminhos de execução dentro de um programa P4.

Por mais que p4pktgen aplique diversas técnicas para otimizar o processo de execução simbólica (como *backtracking* de prefixos e solução incremental de SMTs), a geração de casos de teste para todos os caminhos possíveis dentro de um programa é extremamente custosa e, na prática, impossível para programas complexos (NÖTZLI et al., 2018). Nestes casos, configuramos p4pktgen para que realizasse uma exploração alternada dos caminhos, gerando apenas um pacote de teste para cada caminho dentro do *parser* (ou

seja, não geramos casos de teste variando regras de encaminhamento para um mesmo pacote). Em alguns casos, p4pktgen não é capaz de formar um pacote de entrada que resultaria na execução de determinados caminhos, denominados por (NÖTZLI et al., 2018) como "prefixos impossíveis". Portanto, após o término da geração de casos de teste com a ferramenta, filtramos todos os "casos vazios" (que não possuem um pacote definido).

A alteração do modelo C gerado pelo *assert-p4* é realizada de forma automática para cada caso de teste através de análise e modificação das linhas de código que compõem o modelo. Através desta análise, são localizadas as instruções que modelam os cabeçalhos, o *deparser* e os estados do *parser*, de forma que possam então ser inseridas as funções auxiliares que realizam emissão e processamento de cabeçalhos dos pacotes de saída e entrada, respectivamente.

Após a conformação aos casos de teste, é realizado o processo de compilação seguido de execução dos modelos C. O resultado de cada execução é obtido através de impressões em *stdout*[1] que apresentam o conteúdo do pacote de saída e a porta através da qual este foi emitido. O conteúdo do pacote é representado por uma sequência binária de caracteres (0 ou 1) ou $null$, enquanto a porta de saída é representada por um número natural ou -1, onde $null$ e -1 são utilizados para indicar o descarte do pacote de entrada.

Para concretizar a estratégia de validação, utilizamos o software switch BMv2 (P4 LANGUAGE CONSORTIUM, 2014) conectado a enlaces virtuais para realizar o processamento de pacotes. P4 Language Consortium (2014) define este software switch como a implementação-referência de um switch P4, o que justifica a sua escolha como dispositivo de rede neste trabalho.

A configuração das regras de encaminhamento do BMv2 antes da transmissão de cada pacote é realizada através do script *simple_switch_CLI*, entregue juntamente com o software switch. Os pacotes são enviados ao switch individualmente através dos enlace virtuais conectados às portas de entrada do dispositivo. Esta transmissão é realizada utilizando a biblioteca Scapy (BIONDI, 2005) e a representação PCAP dos pacotes de entrada emitida por p4pktgen.

Para capturar o resultado do processamento de cada pacote de entrada, são utilizadas threads que monitoram os enlaces conectados ao switch. O período de monitoramento dura 1 segundo ou o tempo necessário até a captura de um pacote de saída. Para tornar este processo mais eficiente, o algoritmo utiliza a porta de saída informada pelo modelo C para monitorar apenas um enlace por vez, da seguinte forma: em casos onde um pacote

[1]Canal de saída entre um programa e o sistema operacional.

de saída é emitido pelo modelo C, é realizado monitoramento no enlace conectado à porta correspondente; já em casos em que o modelo C aponta o descarte do pacote de entrada, não é necessária a captura de pacotes do switch, mas sim a análise do *log* de execução do BMv2 para verificar se um pacote foi descartado no período. A estratégia de captura descrita aqui é mais rápida e eficiente que a proposta por Stoenescu et al. (2016), uma vez que a metodologia para validação de modelos SEFL especifica um período fixo de 1 segundo de espera por pacotes, além de requerer o monitoramento de todos enlaces do dispositivo de rede.

A implementação do algoritmo foi realizada em Python e utiliza diversas ferramentas, que devem estar disponíveis no momento de execução do script de validação. A versão das ferramentas utilizadas está descrita na Tabela 3.1. Uma vez que p4pktgen não evidencia um número de versão, é apresentado abaixo o *hash* da versão do código na plataforma GitHub.

Tabela 3.1: Versões de softwares e ferramentas utilizados para implementação do script de validação.

Software/Ferramenta	Versão
Python	2.7.12
gcc	5.4.0
p4c-bm2-ss	0.0.5
p4pktgen	9d34be4
Scapy	2.2.0

Fonte: Autoria própria.

3.2 Seleção de Estudos de Caso

Para avaliar a efetividade da metodologia de validação proposta na identificação de inconsistências entre programas P4 e seus respectivos modelos, coletamos um extenso conjunto de aplicações disponibilizadas na Internet. Essas aplicações estão listadas na Tabela 3.2 e provêm de tutoriais, trabalhos acadêmicos e produtos de código-aberto. A tabela apresenta ainda características relevantes de cada aplicação, como o uso de registradores para armazenar estado e de pilhas de cabeçalhos para facilitar a manipulação do conteúdo dos pacotes. Por fim, um link aponta para o endereço de origem de cada código-fonte.

Devido a limitações na ferramenta utilizada para a geração de pacotes (isto é, casos de teste), não foi possível efetuar o processo de validação dos modelos em todos os programas listados na Tabela 3.2. Por exemplo, p4pktgen não suporta o uso de registradores

nem de pilhas de cabeçalhos. Nesse caso, utilizamos três dos programas P4 selecionados para avaliação: Stag, Timestamp Switching e uma versão modificada de Switch.p4[2]. Ainda assim, consideramos relevante listar todas as aplicações P4 encontradas, uma vez que essa informação não está disponível de forma sucinta na literatura e também pode ser útil para outros autores.

Tabela 3.2: Programas P4 selecionados para o processo de validação.

Programa	**Registradores**	**Pilha de Cabeçalhos**	**Código-fonte**
DAIET (SAPIO et al., 2017)	X	X	<https://goo.gl/CzmzY5>
Dapper (GHASEMI; BENSON; REXFORD, 2017)	X		<https://goo.gl/fWkdLK>
MRI		X	<https://goo.gl/3wGpFW>
NetCache (JIN et al., 2017)	X		<https://goo.gl/djesV7>
NetChain (JIN et al., 2018)	X	X	<https://goo.gl/vv8eFq>
NetPaxos (DANG et al., 2015)	X		<https://goo.gl/FvAQnn>
nocc (JEPSEN et al., 2018b)	X		<https://goo.gl/e8CcNm>
P4-CoDel (KUNDEL et al., 2018)	X		<https://goo.gl/vE2DH6>
Linear Road (JEPSEN et al., 2018a)	X		<https://goo.gl/j8axc1>
PRECISION (BEN-BASAT et al., 2018)	X		<https://goo.gl/FKrmci>
SketchLearn (HUANG; LEE; BAO, 2018)	X		<https://goo.gl/1w211g>
SpeedLight (YASEEN; SONCHACK; LIU, 2018)	X		<https://goo.gl/ajYZRS>
Stag (LOPES et al., 2016)			<https://goo.gl/ABocVN>
Switch.p4	X	X	<https://goo.gl/EcKZ5S>
SwitchPointer (TAMMANA; AGARWAL; LEE, 2018)	X	X	<https://goo.gl/eY3mLU>
Timestamp Switching			<https://goo.gl/a6ubpC>

Fonte: Autoria própria.

3.3 Resultados

Durante a aplicação da metodologia proposta para validação dos modelos dos programas P4 selecionados na seção anterior, encontramos diversas falhas de modelagem por parte da ferramenta de verificação *assert-p4*. Embora tais falhas não tenham impossibilitado a identificação de bugs em programas P4 com esta ferramenta (conforme apresentado por Freire et al. (2018)), a presença de inconsistências no modelo pode levar a resultados de verificação incorretos em outras circunstâncias. Nesta seção, discutimos algumas das falhas de modelagem identificadas, bem como o desempenho da metodologia proposta em termos do tempo de execução para diferentes programas.

[2] As modificações realizadas no Switch.p4 foram: remoção de pilhas de cabeçalhos; remoção de chamadas externas (por exemplo, contadores); substituição de campos de cabeçalho de 128 bits para campos de 64 bits; substituição do tipo de combinação de chaves de tabela para *exact*.

3.3.1 Falhas de modelagem

Entre as principais falhas de modelagem encontradas estão divergências sintáticas e semânticas entre o modelo e o programa original. Essas divergências são observadas através de inconsistências entre os pacotes de saída do modelo e do dispositivo de rede (por exemplo, portas de saída ou valores de cabeçalhos distintos), e ressaltam a importância e a efetividade da metodologia de validação proposta. Abaixo, são discutidas algumas destas falhas[3].

Modelagem da função *verify*. A função $verify(x, y)$ em P4 possui a seguinte semântica: se x for "falso", então um erro do tipo y deve ser emitido[4]. Verificamos que esta primitiva estava sendo modelada com semântica incorreta, informando erro de execução no modelo C caso x fosse "verdadeiro", comportamento oposto à especificação da linguagem P4.

Precedência de operadores booleanos. A especificação da linguagem P4 determina que operadores booleanos devem ser processados da esquerda para a direita (P4 LANGUAGE CONSORTIUM, 2018), enquanto a linguagem C apresenta uma ordem definida[5] de precedência de operadores (C++ COMMUNITY, 2017). Tal divergência no processamento de expressões não era considerada durante o processo de tradução realizado pelo *assert-p4*, causando a avaliação incorreta de expressões booleanas. Este problema pôde ser corrigido através da inserção de parênteses para garantir a ordem de precedência da esquerda para a direita.

Modelagem da ação padrão de uma tabela. Quando não são fornecidas regras de encaminhamento para uma tabela, sua ação padrão deve ser executada independente do conteúdo do pacote de entrada sendo processado. Identificamos que *assert-p4* não realizava a modelagem de ações padrão, salvo situações em que tal ação fosse redefinida por uma regra de encaminhamento.

Processamento de arquivos de regras de encaminhamento vazios. Um arquivo de regras de encaminhamento vazio (isto é, que não contém nenhuma regra) pode ser fornecido ao *assert-p4* para a verificação de cenários onde nenhuma tabela de encaminha-

---

[3] As falhas de modelagem descritas neste capítulo foram corrigidas à medida em que foram identificadas através de alterações no código-fonte da ferramenta *assert-p4*, já contidas em sua última versão disponível.

[4] Segundo a especificação da linguagem P4, fica a cargo do fabricante do dispositivo determinar o tipo do erro e o comportamento específico do programa ao notificá-lo durante a execução.

[5] A precedência de operadores booleanos na linguagem C segue a seguinte ordem decrescente: ! e $\sim$ (negação e negação bit-a-bit); $\ll$ e $\gg$ (*shift* bit-a-bit); $<, \leq, >$ e $\geq$ (comparadores de grandeza); $==$ e $!=$ (comparadores de igualdade); & (conjunção bit-a-bit); $\wedge$ (disjunção exclusiva bit-a-bit); | (disjunção); && (conjunção); || (disjunção).

mento possui entradas, devendo portanto serem executadas suas respectivas ações padrão. Observamos que, ao receber arquivos de regras de encaminhamento vazios, *assert-p4* criava incorretamente entradas simbólicas para todas as tabelas ao invés de modelar suas ações padrão.

Mapeamento de ações de regras de encaminhamento. Como visto na Seção 2.4.2, tanto ações como tabelas são modeladas como funções no modelo C. Ao realizar a modelagem de uma regra de encaminhamento, por vezes não eram inseridas corretamente no corpo da tabela a função que modela a ação definida pela regra. Por exemplo, suponha uma tabela com possíveis ações $copy_to_cpu$ e $copy_to_cpu_with_reason$. Ao modelar uma regra que utilizasse a ação $copy_to_cpu$, *assert-p4* inseria incorretamente, em alguns casos, a chamada de função correspondente a ação $copy_to_cpu_with_reason$.

Mapeamento de chaves de regras de encaminhamento. A ordem dos valores de chave utilizados para selecionar qual ação deve ser aplicada durante o processamento de uma tabela segue a ordem de definição das chaves no código P4. Por exemplo, se uma tabela P4 define as chaves $key = \{a : exact, b : exact;\}$ e uma regra de encaminhamento para esta tabela é definida como $table_add\ table\ action\ 0\ 1\ \implies$, a combinação chave-valor deve ser $a == 0 \wedge b == 1$. Observamos que *assert-p4* não preservava esta ordem de chaves ao modelar a estrutura condicional que seleciona qual ação executar, levando a combinações chave-valor incorretas.

3.3.2 Desempenho

Para avaliar o desempenho da metodologia proposta, executamos sua implementação em uma máquina virtual com 3 núcleos de processamento AMD Ryzen 5 1600 @ 3200 MHz e 4GB de memória RAM, executando sobre o sistema operacional Ubuntu 16.04.5 LTS. Como métrica, consideramos o tempo necessário para concluir o processo de validação de cada estudo de caso selecionado (Stag, Timestamp Switching e Switch.p4).

A Tabela 3.3 apresenta os resultados obtidos para diferentes quantidades de casos de teste. Note que o tempo total de validação, em segundos, compreende tanto o tempo necessário para a ferramenta p4pktgen gerar os casos de teste quanto o tempo gasto processando os pacotes de entrada. Uma vez que p4pktgen não é capaz de explorar todos os caminhos possíveis de Switch.p4 devido ao seu tamanho (NÖTZLI et al., 2018), o processo de geração de pacotes para este estudo de caso foi interrompida após 3 horas. Nos demais casos, todos os caminhos foram testados.

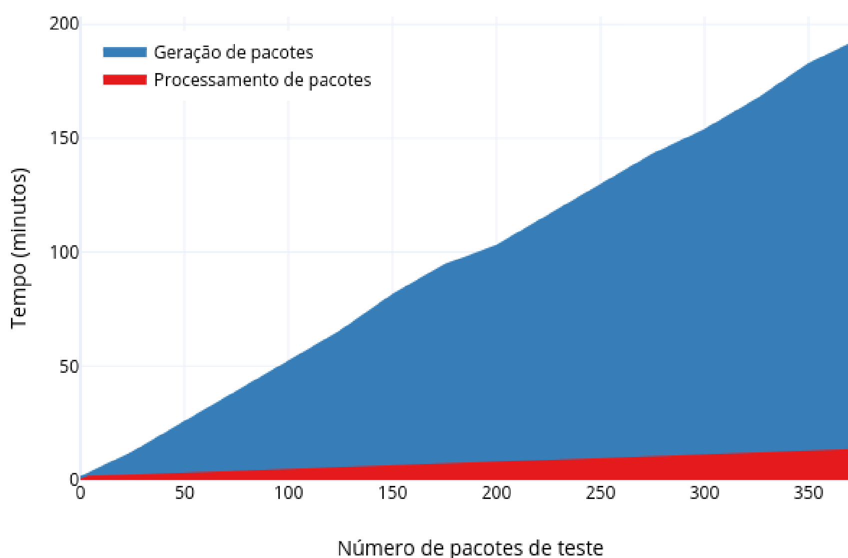
Tabela 3.3: Tempo decorrido durante o processo de validação dos estudos de caso selecionados.

Estudo de Caso	Pacotes	Tempo de Validação (s)	
		Total	p4pktgen
Timestamp Switching	9	10,4s	0,8s
Stag	16	11,6s	1,1s
Switch.p4	100	3106,8s	2849,6s
	200	6149,6s	5705,4s
	375	11588,2s	10800,0s

Fonte: Autoria própria.

A Figura 3.2 ilustra o tempo necessário para a validação dos modelos gerados para o programa Switch.p4. Em linhas gerais, pode-se perceber que o tempo necessário para a validação dos modelos cresce linearmente em função do número de casos de teste executados (isto é, do número de pacotes gerados e processados). Além disso, também observamos que à medida que o número de casos de teste aumenta, o tempo gasto gerando os pacotes de entrada e as regras de encaminhamento passa a predominar em relação ao tempo de processamento desses pacotes. Como trabalhos futuros, pretendemos investigar formas de reduzir este tempo, permitindo a execução de validações mais robustas.

Figura 3.2: Tempo decorrido para validação do modelo gerado por *assert-p4* para o programa Switch.p4.



Fonte: Autoria própria.

4 REPRODUTIBILIDADE

O protótipo da ferramenta *assert-p4* apresentado por Freire et al. (2018) permite demonstrar o funcionamento de um mecanismo de verificação de programas P4 baseado em asserções e execução simbólica. Entretanto, para que seja possível a reutilização da ferramenta bem como a reprodutibilidade dos experimentos por terceiros, é necessário não apenas que a mesma seja funcional, mas que esteja propriamente documentada (POULIN, 1994).

Saucez e Luigi (2017) definem um trabalho como *reproduzível* se for possível obter os mesmos resultados reportados em múltiplas tentativas, usando sistemas diferentes dos utilizados nos experimentos originais. Para possibilitar a reprodutibilidade, é necessária disponibilização e documentação dos artefatos produzidos, bem como informações de dependências para construção de um ambiente no qual seja possível a execução de tais artefatos.

Para permitir a reutilização da ferramenta *assert-p4*, conduzimos atividades tanto de operacionalização quanto de depuração e documentação. No que diz respeito à implementação, corrigimos diversos erros no protótipo e desenvolvemos scripts para automatizar o processo de verificação através da criação de um ambiente de execução. No tocante à documentação, elaboramos guias para a utilização da ferramenta e para reprodução dos experimentos apresentados por Freire et al. (2018).

Implementação. Inicialmente, foram necessários ajustes no protótipo do *assert-p4* que, por vezes, não era capaz de gerar modelos C sintaticamente corretos. Os erros descritos a seguir impediam a continuação do processo de verificação, diferentemente dos bugs de modelagem reportados no Capítulo 3, que causavam inconsistências entre o comportamento do modelo e o comportamento de dispositivos reais configurados com o programa P4.

Tabela 4.1: Erros de compilação em modelos C produzidos pelo protótipo do *assert-p4*.

#	Erro de compilação	Regras Fornecidas	Programas
1	initializer element is not a compile-time constant	Não	Stag Timestamp Switching
2	called object type ’int’ is not a function or function pointer	Não	Stag NetPaxos
3	use of undeclared identifier ’hdr’	Não	NetPaxos
4	expected ’;’ after top level declarator	Não	Timestamp Switching
5	conflicting types for function	Sim	Stag
6	invalid suffix on floating constant	Sim	Timestamp Switching
7	expected identifier	Sim	Timestamp Switching

Fonte: Autoria própria.

Na Tabela 4.1, estão enumerados os erros de compilação encontrados nos modelos C gerados pelo protótipo de *assert-p4*. Na coluna "Regras Fornecidas", estão identificados com "Sim" os erros de compilação que ocorrem somente nos cenários em que regras de encaminhamento são fornecidas ao *assert-p4*. Na coluna "Programas", estão identificados em quais programas, especificamente, cada erro pôde ser observado.

O erro #1 ocorre quando asserções são declaradas no código P4 fora de um bloco de controle, enquanto #2 é proveniente de uma tradução incorreta da função P4 que testa se um cabeçalho é válido. O erro #3, por sua vez, acontece em situações que a variável de cabeçalhos na implementação do *parser* P4 é definida com nome diferente de *hdr*. Os erros #4 e #5 ocorrem devido a um bug na declaração das funções C correspondente a *actions* do programa P4. Já #7 e #8 ocorrem devido a um processamento incompleto das regras de encaminhamento fornecidas.

Durante o processo de desenvolvimento deste trabalho, foram documentados os erros #1 e #3 como limitações do esquema de tradução e corrigidos os demais erros através de modificações no script de tradução. A partir destas alterações, tornou-se possível a execução do protótipo do *assert-p4* para verificação de programas P4 sem a necessidade de intervenções manuais para corrigir erros no modelo C.

A fim de facilitar a execução da ferramenta, foram desenvolvidos dois mecanismos para construção de um ambiente de execução do *assert-p4*: um script para o sistema operacional Ubuntu 16.04.5 LTS (Xenial Xerus) que automaticamente instala todas as dependências da ferramenta; e um arquivo de configuração para a ferramenta de gerenciamento de sistemas virtuais Vagrant (HASHICORP, 2018), com o qual é possível a instanciação de uma máquina virtual para utilização do verificador de programas P4. Além disso, foi desenvolvido um script para encapsular todas as linhas de comando necessárias para a verificação de um programa, resumindo os passos para execução da ferramenta.

Documentação. Uma vez que *assert-p4* utiliza diversos softwares de terceiros, também é necessária a documentação das dependências da ferramenta. Para garantir a disponibilidade das versões sob as quais o protótipo inicial foi projetado, foram realizados *forks* dos repositórios na plataforma GitHub.

Além de disponibilizar um ambiente completo com instruções para a execução do *assert-p4*, foi elaborada, ainda, uma documentação específica para a reprodução dos resultados obtidos por Freire et al. (2018). Para cada software P4 avaliado através do protótipo do *assert-p4*, foi confeccionada uma página explicando o objetivo do software, o significado de cada asserção inserida no código P4, como executar a verificação e quais

os resultados esperados.

Os esforços de reprodutibilidade desenvolvidos neste trabalho foram reconhecidos através do selo *Artifacts Evaluated - Reusable*, concedido pelo Comitê de Avaliação de Artefatos do ACM CoNEXT 2018 para apenas 3 de 14 publicações avaliadas. Esta premiação reforça a importância da criação de documentação e a manutenção de artefatos, além de estimular a busca por reprodutibilidade em trabalhos futuros (BAJPAI et al., 2017).

5 CONSIDERAÇÕES FINAIS

Neste trabalho, foi apresentada uma metodologia para validação dos modelos C gerados por *assert-p4* como parte do processo de verificação de programas P4. Através desta, foi possível a identificação e correção de diversos erros na modelagem realizada pela ferramenta, tornando seus resultados mais confiáveis.

O algoritmo de validação aqui apresentado é significativamente mais simples, mas igualmente eficiente para identificação de erros de modelagem, se comparado à metodologia que inspirou sua criação, proposta por Stoenescu et al. (2016). Enquanto a metodologia para validação de modelos SEFL requer exploração e execução de caminhos em uma plataforma de execução simbólica e, separadamente, a utilização de *solvers* para criação e avaliação de casos de teste, a estratégia proposta para validação dos modelos de *assert-p4* consiste apenas na comparação do resultado do processamento de pacotes no modelo C e em um dispositivo de rede P4-compatível.

Entretanto, há limitações neste formato de validação. Uma vez que a metodologia consiste, essencialmente, na aplicação de baterias de testes nos modelos C e em um switch virtual, a qualidade e variedade dos casos de teste utilizados têm impacto direto na confiabilidade dos resultados obtidos. Para lidar com este entrave, foi definida a utilização de p4pktgen, capaz de gerar pacotes de entrada através da exploração dos possíveis caminhos de programas P4. Ainda que esta ferramenta também possua suas limitações, o planejamento descrito pelos seus autores para melhoria de desempenho e cobertura para geração de casos de testes faz de p4pktgen a melhor alternativa, até o presente momento, para preencher esta lacuna na metodologia de validação proposta.

Juntamente com a validação dos modelos C, foram realizados esforços de operacionalização de *assert-p4*, documentando suas formas de utilização e providenciando métodos para a criação de um ambiente completo para a verificação de programas P4 com a ferramenta. Tais esforços permitiram que *assert-p4* fosse reconhecido como "reutilizável" pelo Comitê de Avaliação de Artefatos da CoNEXT'18, reforçando a necessidade da documentação de protótipos e ferramentas, que por vezes é negligenciada pela comunidade científica, dificultando a reprodução de experimentos e a continuidade de pesquisas.

Como trabalho futuro, planeja-se a exploração de diferentes estratégias de execução simbólica para melhorar o desempenho do *assert-p4*, tornando factível a verificação completa de programas P4 complexos como Switch.p4. Também, projeta-se o aperfeiçoamento do esquema de tradução de código para geração dos modelos C, de forma que estes

sejam capazes de refletir, progressivamente, mais elementos da linguagem P4. Espera-se, tendo em vista o que foi exposto, que *assert-p4* possa ser difundido como uma ferramenta confiável para atividades de verificação e validação de softwares P4.

REFERÊNCIAS

BAJPAI, V. et al. Challenges with reproducibility. In: ACM. **Proceedings of the Reproducibility Workshop**. [S.l.], 2017. p. 1–4.

BALDONI, R. et al. A survey of symbolic execution techniques. **ACM Computing Surveys (CSUR)**, ACM, v. 51, n. 3, p. 50, 2018.

BEN-BASAT, R. et al. Efficient measurement on programmable switches using probabilistic recirculation. In: IEEE. **2018 IEEE 26th International Conference on Network Protocols (ICNP)**. [S.l.], 2018. p. 313–323.

BENSON, T.; AKELLA, A.; MALTZ, D. A. Unraveling the complexity of network management. In: **NSDI**. [S.l.: s.n.], 2009. p. 335–348.

BIONDI, P. Scapy: explore the net with new eyes. **Technical report, EADS Corporate Research Center**, 2005.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **ACM SIGCOMM Computer Communication Review**, ACM, v. 44, n. 3, p. 87–95, 2014.

C++ COMMUNITY. **C Operator Precedence**. 2017. Disponível em: <https://en.cppreference.com/w/c/language/operator_precedence>. Acesso em: 17 nov. 2018.

CADAR, C. et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: **OSDI**. [S.l.: s.n.], 2008. v. 8, p. 209–224.

CASADO, M.; FOSTER, N.; GUHA, A. Abstractions for software-defined networks. **Communications of the ACM**, ACM, v. 57, n. 10, p. 86–95, 2014.

CORDEIRO, W. L. da C.; MARQUES, J. A.; GASPARY, L. P. Data plane programmability beyond openflow: Opportunities and challenges for network and service operations and management. **Journal of Network and Systems Management**, Springer, v. 25, n. 4, p. 784–818, 2017.

DANG, H. T. et al. Netpaxos: Consensus at network speed. In: ACM. **Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research**. [S.l.], 2015. p. 5.

DOBRESCU, M.; ARGYRAKI, K. Software dataplane verification. In: **11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)**. [S.l.: s.n.], 2014. p. 101–114.

FREIRE, L. et al. Uncovering bugs in p4 programs with assertion-based verification. In: ACM. **Proceedings of the Symposium on SDN Research**. [S.l.], 2018. p. 4.

GHASEMI, M.; BENSON, T.; REXFORD, J. Dapper: Data plane performance diagnosis of tcp. In: ACM. **Proceedings of the Symposium on SDN Research**. [S.l.], 2017. p. 61–74.

GUDE, N. et al. Nox: towards an operating system for networks. **ACM SIGCOMM Computer Communication Review**, ACM, v. 38, n. 3, p. 105–110, 2008.

HASHICORP. **Vagrant by HashiCorp**. 2018. Disponível em: <https://www.vagrantup.com/>. Acesso em: 13 out. 2018.

HUANG, Q.; LEE, P. P.; BAO, Y. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In: ACM. **Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication**. [S.l.], 2018. p. 576–590.

JEPSEN, T. et al. Life in the fast lane: A line-rate linear road. In: ACM. **Proceedings of the Symposium on SDN Research**. [S.l.], 2018. p. 10.

JEPSEN, T. et al. Infinite resources for optimistic concurrency control. In: ACM. **Proceedings of the 2018 Morning Workshop on In-Network Computing**. [S.l.], 2018. p. 26–32.

JIN, X. et al. Netcache: Balancing key-value stores with fast in-network caching. In: ACM. **Proceedings of the 26th Symposium on Operating Systems Principles**. [S.l.], 2017. p. 121–136.

JIN, X. et al. Netchain: Scale-free sub-rtt coordination. In: USENIX ASSOCIATION. **15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)**. [S.l.], 2018.

KAZEMIAN, P. et al. Real time network policy checking using header space analysis. In: **NSDI**. [S.l.: s.n.], 2013. p. 99–111.

KAZEMIAN, P.; VARGHESE, G.; MCKEOWN, N. Header space analysis: Static checking for networks. In: **NSDI**. [S.l.: s.n.], 2012. v. 12, p. 113–126.

KHURSHID, A. et al. Veriflow: Verifying network-wide invariants in real time. In: **Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)**. [S.l.: s.n.], 2013. p. 15–27.

KOHLER, E. et al. The click modular router. **ACM Transactions on Computer Systems (TOCS)**, ACM, v. 18, n. 3, p. 263–297, 2000.

KREUTZ, D. et al. Software-defined networking: A comprehensive survey. **Proceedings of the IEEE**, IEEE, v. 103, n. 1, p. 14–76, 2015.

KUNDEL, R. et al. P4-codel: Active queue management in programmable data planes. In: **Proceedings of 2018 IEEE Conference on Network Function Virtualization and Software Defined Networks**. [S.l.: s.n.], 2018.

LIU, J. et al. P4v: Practical verification for programmable data planes. In: ACM. **Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication**. [S.l.], 2018. p. 490–503.

LOPES, N. et al. **Automatically verifying reachability and well-formedness in P4 Networks**. [S.l.], 2016.

LOPES, N. P. et al. Checking beliefs in dynamic networks. In: **NSDI**. [S.l.: s.n.], 2015. p. 499–512.

MAI, H. et al. Debugging the data plane with anteater. In: ACM. **ACM SIGCOMM Computer Communication Review**. [S.l.], 2011. v. 41, n. 4, p. 290–301.

MCKEOWN, N. et al. Openflow: enabling innovation in campus networks. **ACM SIGCOMM Computer Communication Review**, ACM, v. 38, n. 2, p. 69–74, 2008.

MOURA, L. D.; BJØRNER, N. Z3: An efficient smt solver. In: SPRINGER. **International conference on Tools and Algorithms for the Construction and Analysis of Systems**. [S.l.], 2008. p. 337–340.

NÖTZLI, A. et al. P4pktgen: Automated test case generation for p4 programs. In: ACM. **Proceedings of the Symposium on SDN Research**. [S.l.], 2018. p. 5.

P4 LANGUAGE CONSORTIUM. **Behavioral Model Repository**. 2014. Disponível em: <https://github.com/p4lang/behavioral-model>. Acesso em: 10 nov. 2018.

P4 LANGUAGE CONSORTIUM. **$P4_{16}$ Language Specification**. 2018. Disponível em: <https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html>. Acesso em: 17 nov. 2018.

POULIN, J. S. Measuring software reusability. In: IEEE. **Software Reuse: Advances in Software Reusability, 1994. Proceedings., Third International Conference on**. [S.l.], 1994. p. 126–138.

SAPIO, A. et al. In-network computation is a dumb idea whose time has come. In: ACM. **Proceedings of the 16th ACM Workshop on Hot Topics in Networks**. [S.l.], 2017. p. 150–156.

SAUCEZ, D.; LUIGI, I. Thoughts and recommendations from the acm sigcomm 2017 reproducibility workshop. 2017.

SONG, H. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In: ACM. **Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking**. [S.l.], 2013. p. 127–132.

STOENESCU, R. et al. Debugging p4 programs with vera. In: ACM. **Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication**. [S.l.], 2018. p. 518–532.

STOENESCU, R. et al. Symnet: scalable symbolic execution for modern networks. In: ACM. **Proceedings of the 2016 ACM SIGCOMM Conference**. [S.l.], 2016. p. 314–327.

TAMMANA, P.; AGARWAL, R.; LEE, M. Distributed network monitoring and debugging with switchpointer. In: USENIX ASSOCIATION. **15th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’18)**. [S.l.], 2018.

VANHOEF, M.; PIESSENS, F. Key reinstallation attacks: Forcing nonce reuse in wpa2. In: ACM. **Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security**. [S.l.], 2017. p. 1313–1328.

WALLACE, D. R.; FUJII, R. U. Software verification and validation: An overview. **IEEE Software**, IEEE, v. 6, n. 3, p. 10–17, 1989.

YASEEN, N.; SONCHACK, J.; LIU, V. Synchronized network snapshots. In: ACM. **Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication**. [S.l.], 2018. p. 402–416.

ZAOSTROVNYKH, A. et al. A formally verified nat. In: ACM. **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. [S.l.], 2017. p. 141–154.