VALIDATION OF AVAILABILITY AND POLICY BASED MANAGEMENT FOR PROGRAMMABLE NETWORKS

Ferney A. Maldonado López

Universitat de Girona

DOCTORAL THESIS

Validation of Availability and Policy Based Management for Programmable Networks

Ferney A. Maldonado Lopez

2017

Universitat de Girona

DOCTORAL THESIS

Validation of Availability and Policy Based Management for Programmable Networks

Ferney A. Maldonado Lopez

2017

[illegible] [illegible]

DOCTORAL THESIS

Validation of Availability and Policy Based Management for Programmable Networks

Ferney A. Maldonado Lopez

2017

Doctoral Programme in Technology, University of Girona
School of Engineering, Universidad de los Andes

Supervisors:
Ph.D. Eusebi Calle and Ph.D. Yezid Donoso

This thesis is presented in fulfillment of the requirement for the conferral of the degree of Doctor of Philosophy by the University of Girona and Doctor of Engineering by the Universidad de los Andes

Girona, May 29th, 2017

We approve the submission of this thesis entitled "Validation of Availability and Policy Based Management for Programmable Networks" to the School of Doctoral studies of the University of Girona.

Ferney Alonso Maldonado Lopez	[illegible]
Thesis Candidate	

Dr. Eusebi Calle Ortega	
Adviser	

Dr. Yezid Donoso Meisel	[illegible]
Co–adviser Universidad de los Andes	

Jury Tribunal

Dr.	
President	

Dr.	
Secretary	

Dr.	
Vocal	

Dr.	
Vocal	

Dr.	
Vocal	

Date of presentation	
Qualification	

Acknowledgments

First of all, the credit of this thesis is to my family who have accompanied and encouraged me to the completion of this thesis.

I would like to thank my advisors, Dr. Yezid Donoso and Dr. Eusebi Calle for their guidance throughout this research project.

I want to thank *the team*; Jaime Chavarriaga, Andrés Marentes, Carlos Lozano, Andrés Moreno, Germán Montoya, Carlos Velásquez, Diego Castiblanco and many other with whom we discussed about the project for long hours. Your comments and criticisms were highly valuable.

I am grateful to Dr. Harold Castro, and the Systems and Computer Engineering Department at Universidad de los Andes for their support.

Especially grateful to Dr. Jorge Cuellar, Dr. Radha Poovendran, Andrew Clark, Chou–Chang Yang, Phillip Lee and all the people who shared their time and experiences with me in the University of Washington. My best memories on that visit.

This work was partially supported by the Department of Science, Technology and Innovation – Colciencias with grant 528 of 2011; the project Redes interdependientes y con restricciones geográficas: Indicadores de robustez (GIROS) TEC2015-66412-R; and the Alianza CAOBA–Centro de Excelencia y Apropiación en Big Data y Data Analytics.

List of Publications

- Maldonado-Lopez, F.; Calle, E. and Donoso, Y. Checking Multi-domain Policies in SDN, Int. J. of Computers, Communication and Control, 2016, 11, 393-405, 2016, 11, 393-405 [MLCD16].
- Maldonado-Lopez, F. A. and Donoso, Y. Reliable Critical Infrastructure: Multiple Failures for Multicast using Multi-Objective Approach, Int. J. of Computers, Communication and Control, 2013, 8, 79-86 [MLD13].
- D. F. Rueda, E. Calle, F. A. Maldonado-Lopez and Y. Donoso, Reducing the impact of targeted attacks in interdependent telecommunication networks, 2016 23rd International Conference on Telecommunications (ICT), Thessaloniki, Greece, 2016, pp. 1-5. [RCM16].
- Maldonado-Lopez, F. A.; Calle, E. and Donoso, Y. Detection and prevention of firewall-rule conflicts on software-defined networking Reliable Networks Design and Modeling (RNDM), 2015 7th International Workshop on, 2015, 259-265 [MLCD15].
- Maldonado-Lopez, F.; Chavarriaga, J. and Donoso, Detecting Network Policy Conflicts Using Alloy Abstract State Machines, Alloy, B, TLA, VDM, and Z, Springer Berlin Heidelberg, 2014, 8477, 314-317 [MLCD14]
- Maldonado-Lopez, F. A. and Donoso, Y. Multicast Session Protection Planner - Tool to plan and deploy protection infrastructure: a SPEA approach CCIA, 2012, 191-200 [MLD12]
- Maldonado-Lopez, F. A.; Corchuelo, J. and Donoso, Y. Unavailability and cost minimization in a parallel-series system using multi-objective evolutionary algorithms, International Conference on Applied, Numerical and Computational Mathematics, and Proceedings of the 2011 international conference on Computers, digital communications and computing, World Scientific and Engineering Academy and Society (WSEAS), 2011, 33-38 [MLCD11]

List of Acronyms

AAA Authentication, Authorization and Accounting

ACL Access Control List

API Application Program Interface

BCP Boolean Constraint Propagation

BDD Binary Decision Diagram

CTL Computation Tree Logic

CNF Conjunctive Normal Form

DL Description Logic

DPLL David, Putman, Logemann and Loveman algorithm

FDD Firewall Decision Diagram

FIB Forwarding Information Base

HSA Header Space Analysis

KAT Kleene algebra with test

LTL Linear temporary logic

NAT Network Address Translation

NETAD Network Traffic Anomaly Detector

NOS Network Operating System

OF OpenFlow protocol

PBM Policy–Based Management

PBNM Policy Based Network Management

PDP Policy Decision Points

PEP Policy Enforcement Points

QBF Quantified Boolean formula

QFF Quantifier Free Form

QSAT Quantified SAT

RBAC Role-based Access Control

RCP	Routing Control Platform
SANE	Secure Architecture Network Enterprise
SMT	Satisfiability Module Theories
SAT	Boolean Satisfiability problem
SDN	Software–Defined Networking
TLA	Temporal Logic of Actions
ToS	Type of Service field
TRW-CB	Credit based Threshold Random Walk
WOL	Web Ontology Language

Glossary of Terms

Access control System that determines whether a request of resource is granted or denied.

By–Passing Mechanism that describes how a packet tricks the policy and by a set of genuine rules.

Checking Inspecting or testing a feature.

Correctness Assertion that a function is valid regarding to its specification. The function's output corresponds to the expected behavior.

Datapath Network device. e.g. Router, switch, proxy are examples of datapaths.

Declarative Programming Programming paradigm that describes the computational logic instead of describing the control of flow.

DPLL David–Putman–Logemann–Loveland algorithm for a search that decides satisfiability.

Elasticity Measure of robustness. It is the area under the curve of throughput versus the percentage of nodes removed.

Flowtable or *forwarding–table* is a table stored in a datapath that lists the output port for a specific packet header.

Formal Specification mathematical based technique to describe systems, behavior, domains, and outputs. It defines design properties that later will be verified.

Order Denotes the number of nodes of a graph.

KodKod SAT–based constraint solver for first order logic with relations, transitive closure, bit–vector arithmetic, and partial models.

Kripke Structure Transition system used for model checking to represent the behavior of a system.

Model–checking Given a model of a system, exhaustively and automatically check whether the model meets a given specification.

Quantifier Free Form Boolean combination of simple arithmetic constraints on integers.

Rule Mapping of a set of actions to a set of conditions.

SAT solver Solves satisfiability problem.

Skolem function Function $f(x_1, x_2, \ldots, x_n)$ that replaces the existential quantifier in a logic statement at the Skolem reduction.

Skolem reduction Reduction from formal logic statement to a Skolem normal form (SNF) that removes existential quantifiers ($\forall$ and $\exists$).

Stateful analysis Network analysis that evaluates flows, packets, flow–tables, and multiple variables from the whole network.

Stateless analysis Network analysis of flows or packets without consider other flows or packets or network variables.

Validation Testing if an application meets functional and non–functional requirements.

Verification Verify that an instance is correct. It consists of determining if a program accomplish the specification.

List of Figures

List of Tables

List of Listings

Contents

Abstract

In recent years the network architecture is transform from a monolithic to a network with multi–component and flexible systems, where the software plays the important role of facilitating such flexibility. Software-Defined Networking (SDN) is a network technology that separates control functions, network intelligence by software, and the data plane, packet forwarding in hardware. This change is important being that the architecture of the current network technology maintains these two functionalities in the same device: the router.

The separation of the data and control layers allows flexibility in the management and use of network resources because the software is specialized in controlling the traffic and simple and economic hardware is in charge of forwarding. Like other x–Defined by Software, developments that may arise from this technology are only limited by the imagination of the network application developer. Developers can build applications that control the detail of network and packet processing, from the autonomous configuration to complex operations which involve the context. It opens a myriad of opportunities to develop new functionalities that today are partially made with specialized equipment.

However, one of the most frequent problems in networks is the availability. The countermeasure is to develop mechanisms and strategies to deal with failure incidents and to recover functionality, prevent failures, and mitigate the impact of failures in network operation. SDN availability is an interesting issue that the academic and industrial communities are beginning to address, since the technology is still on development and the efforts are focused on build applications for administration to face the virtualization and dynamic configuration.

One of the challenges is undoubtedly the configuration and the programming of the network. The human factor represents between 50% and 80% of network failures due to errors and bugs in the programming of applications and the implementation of algorithms and protocols.

This doctoral thesis proposes 1) to use formal specification and verification of network functionalities in the context of SDN to reduce the impact of network failures. 2) It also presents the guide of network administration through policy implementation for security and auditing, and 3) shows the impact of failures on a representation of SDN architecture as an interdependent network model.

Resumen

En los últimos años se ha transformado la arquitectura de red desde un sistema monolítico hacia uno de múltiples componentes flexibles, donde el software desempeña el importante rol de facilitar dicha flexibilidad. Las Redes Definidas por Software (SDN) permite la separación de las funciones de control, inteligencia de la red por software de control, y de la capa de datos, el reenvío de paquetes en el hardware. Este cambio es importante dado que la arquitectura actual mantiene estas dos funcionalidades en el mismo dispositivo: el enrutador.

La separación de las capas de control y de datos permite mayor flexibilidad de administración y utilización de los recursos de red. Los desarrollos surgen de esta tecnología están únicamente limitados por la imaginación del desarrollador de aplicaciones. Este último puede diseñar aplicaciones que controlen el detalle de las operaciones de la red, desde la configuración automática de equipos, hasta el desarrollo de complejas operaciones lógicas. Existe entonces un sinnúmero de oportunidades para desarrollar funcionalidades que hoy en día se realizan con equipo especializado.

Sin embargo, uno de los problemas más frecuentes en redes es la disponibilidad. Su objetivo es desarrollar mecanismos y estrategias para actuar ante incidentes de fallo y recuperar la funcionalidad, evitar los fallos, y mitigar el impacto en la operación de la red. En SDN este es un problema interesante que la comunidad académica está trabajando, pero dado que la tecnología es reciente, los esfuerzos se han enfocando en desarrollar aplicaciones para la administración de la infraestructura ante la virtualización y la configuración dinámica.

Uno de los retos es sin duda la configuración y programación de la red. Ya que el factor humano representa entre el 50% y el 80% de los fallos de red son producidos por errores de configuración y bugs en la programación de aplicaciones y la implementación de algoritmos.

Esta tesis de doctorado propone 1) la utilización mecanismos formales de especificación y verificación de aplicaciones de red en el contexto de las SDN con el objetivo de reducir el impacto de las fallas de red. 2) también orienta la administración de las redes por medio de implementación de políticas para la seguridad y la auditoría en redes, y 3) describe el comportamiento de las fallas en una representación de la arquitectura SDN como una red interdependiente y el presenta el impacto fallas en la red.

Resum

En els darrers anys s'ha transformat la arquitectura de xarxa des d'un sistema monolític fins cap a un de múltiples components flexibles, on el software juga el paper important de facilitar aquesta flexibilitat. Les Xarxes Definides per Software (SDN) permeten la separació de les funcions de control, intelligència de xarxa, y de capa de dades, reenviant paquets en el hardware. Aquest canvi es important perquè l'arquitectura de la tecnologia actual manté aquestes dues funcionalitats en el mateix dispositiu: l'enrutador.

La separació de les capes de control y dades permet major flexibilitat d'admistració y utilització dels recursos de la xarxa. Els avenćos tecnològics d'aquesta tecnologia estan únicament limitats per la imaginació del desenvolupador d'aplicacions per xarxa que pot disseny aplicacions que controlin el detall de les operacions de la xarxa, des de la configuració automàtica d'equipaments, fins al desenvolupament d'operacions lògiques complexes. S'obre d'aquesta manera un ventall per desenvolupar funcionalitats que avui en dia es realitzen amb un equipament especialitzat.

No obstant, uns dels problemes mes freqüents en les xarxes es la disponibilitat. L'objectiu és desenvolupar mecanismes i estratègies per actuar davant incidents de fallada i recuperar la funcionalitat, evitar que les fallades apareguin, i mitigar l'impacte de les fallades en l'operació de la xarxa. En SDN aquest és un problema interesant que la comunitat acadèmica està a traballant, però com que la tecnologia ès recent, els esfor̃cos s'han enfocat en desenvolupar aplicacions per l'administració de la virtualització i la configuració automàtica.

Un dels reptes, sens dubte, és el de la configuració i programació de la xarxa. El factor humà representa entre el 50% i el 80% de les fallades de la xarxa entre errors de configuració i bugs en la programació de les aplicacions i implementació d'algorismes i protocols.

Aquesta tesis de doctorat proposa 1) la utilització de mecanismes de especificació i verificació d'aplicacions de la xarxa en el context SDN amb l'objectiu de reduir l'impacte de les fallades de la xarxa. 2) també es proposa orientar l'administració de les xarxes per mitjà de polítiques per la seguretat i la auditoria de xarxes, i 3) se descriu el comportament de les fallades en una representació de l'arquitectura SDN com una xarxa independent i s'estudia l'impacte de les fallades en aquest model de xarxa.

Chapter 1

Introduction

If I had an hour to save the World, I would spend fifty-five minutes defining the problem, and only five minutes finding the solutions.

Albert Einstein

This chapter describes the motivation for this thesis, defines the research problem, and lists the objectives and contributions achieved along the research project. At the end, this chapter describes the structure and content of this document.

1.1 Motivation

Since the mid–1970s, network functionalities, as routing and forwarding, and thus router architecture, have largely remain the same. The *router* is a fundamental device for networking, with specialized software that implements specific algorithms for routing, management, security and quality of service; but also, it has specialized hardware to perform the forwarding function. This device combines control and forwarding tasks (control and data planes), and embedded them into the same middlebox. However, the router architecture remains as a monolithic structure with its own operating system and protocol implementations. Unlike router architecture, Internet changes due to the development of new protocols, applications, and services. Recently, new requirements such as cloud computing, massive information services, and growing data–center infrastructure, demand changes on network architecture and therefore modifications in the router.

1.1.1 SDN: a flexible networking paradigm

Software–Defined Networking (SDN) is a networking approach that separates the control plane from the data plane. Figure 1.1 shows layered separation of control, the network operating system, communications protocols, and the data plane. The *control plane* is a software layer responsible for addressing, routing and managing protocols. Meanwhile, the *data plane* oversees data forwarding, that is sending packets through a specific port or interface. The

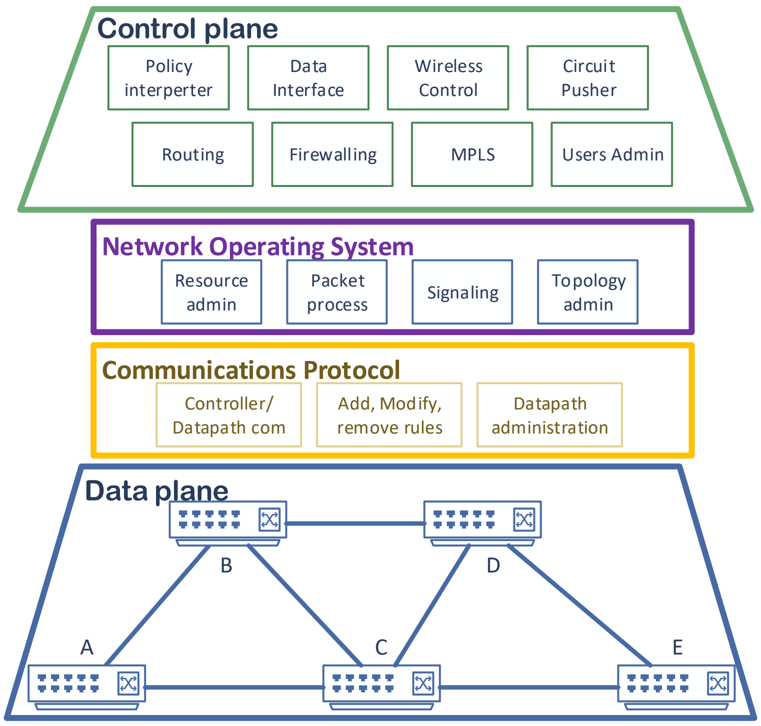


Figure 1.1: SDN layered architecture.

SDN network is governed by a centralized *controller* on a server. On top of the controller, the layer of management executes software routines, customized protocols and network applications. The controller sends and installs traffic rules on network devices by a communications protocol which accesses the data plane. OpenFlow [McK+08b] is a communications protocol that installs entries into the forwarding table of a network device (datapath). Forwarding tables store $\langle match, action \rangle$ pairs which determine the action performed over the packet if a packet header matches an entry on the table. In contrast to traditional routing–tables that store addresses and routes to particular destinations, route metrics, information related to topology, forwarding tables store the set of actions to be apply on a specific packet. SDN actions are wide and diverse, they include sending the package to the controller to be analyzed and processed by a network application, forward the packet to a specific port, modify the packet, or even discard it. SDN facilitates processing data packets in a granular way in contrast to the present network architecture. SDN allows the transition from a dated monolithic network with specialized hardware and software, to a layered and flexible architecture with open interfaces over commodity hardware. The important change is that protocols and functions are now network applications which now run on the controller.

SDN empowers programmers to develop applications to support network functionality and other complex requirements such as network assistance for computer virtualization, slicing of infrastructure for multi–tenant, and reconfiguration of network devices for dynamic demand. SDN offers a programming framework for the underlying infrastructure and allows the network management by software abstractions. Networking goals such as traffic engineering, protection, survivability and security, are designed and implemented as software applications that run on the centralized controller. Furthermore, SDN allows the virtualization of network functionalities (NFV) due to they are also applications running on the controller [Mar+14]. Firewalls, load balancers, or intrusion detection systems are functionalities, software, that process traffic packets. Network appliances are substituted by software instances that build virtual networks. We expect that SDN facilitates new developments and solutions to network problems that today are complex to solve.

1.1.2 Network failures and availability

Failures in computer systems, applications, and network components will produce expensive outages. The service disruption is classified into one or more of the following categories: nature, hardware, human error, software, system overload, or vandalism [OM08]. Nevertheless, human error is considered the largest single source of failure and outages [BP01], and causes 62% of network downtime [Ker04], and according to Juniper, human error is culpable for 50 to 80% of network outages [Jun]. For example, a common failure occurs due to firewall misconfiguration, it can block important traffic if its configuration is inaccurate. Moreover, finding configuration errors is difficult and depends on the amount and complexity of the rules [Woo10].

Techniques for mitigating these failures vary from the testing of correctness, which evaluates the behavior before the production deployment [Abr96], to tools which check low–level configuration files accomplish behavior constraints [MWA02]. In order to detect conflicts, inconsistencies or bugs, network administrators can test the configuration of each device, rely on monitors to observe the traffic to confirm correctness; or use *formal methods* to specify a model of the intended behavior of a network, and run an automatic configuration of each middlebox for de SDN case.

Formal methods in networking are mathematical techniques for specification, development and verification of network functionality. This mathematical analysis contributes to increasing network reliability. In recent years, formal verification of software has gained importance due to new advances in search algorithms, machine learning, and hardware power. The verification by formal methods helps to reduce the gap between requirements and configuration, through the validation of congruence of the model of requirements and the model of configurations.

The core of formal methods is the Boolean satisfiability problem (SAT), that consists of, given a formula, finding a satisfying assignment or proving that none exists. For that, SAT solvers are effective tools for solving constraint satisfaction problems [MZ09] even if it is an NP–complete problem [PW10]. Over the course of this thesis, we use lightweight formal methods to find topology and policy inconsistencies and validate configurations. We focus on network invariant checking, firewall rule validation and configuration–implementation accomplishment for SDN networks.

SDN approach faces different challenges and opportunities, especially those related with programming and configuration errors. SDN is vulnerable to failures, error and configuration faults, as any other software or application. In 2003 Oppenheimer et.al. [OGP03] studied over a hundred of reports of failures in the Internet services and found that operator error is the leading cause of failure, also it is the largest contributor to time to repair, and the configuration error is the largest category of operator errors.

This research studies network availability for SDN, the failure propagation between control and data planes, then it presents a framework to check applications to enhance network availability under this new SDN paradigm. This thesis also focuses on network misconfiguration by human errors, due to it is a common phenomenon and the principal cause of network outage. Finally, this thesis proposes to identify inconsistencies by lightweight formal models. We evaluate network scripts and configurations and validate that network behavior model against a specification.

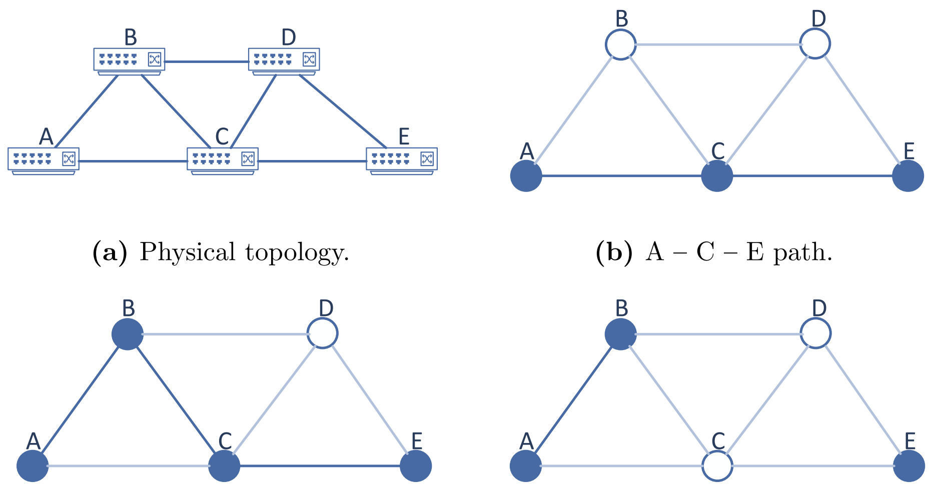
1.2 Problem Overview

Communications protocols, like OpenFlow, facilitate the interaction between components from different vendors and offer a standard API. Thus multi–vendor network–control applications can be developed. Furthermore, the SDN community develops programming languages, network operating systems, and tools for simplifying administration tasks. The complete network behavior can be specified by a set of high–level policies instead of low–level configurations on devices. SDN also provides flexibility and scalability, because the policy specification can be mapped into regular instructions for each equipment. The main advantage of SDN is allowing top–down management, from policies to packet forwarding.

However, network applications are written by humans and therefore they are not exempt from logical or syntax errors. These errors can generate availability issues, network failures and misbehavior such as forwarding address errors, blocking or interruption of connections, oscillation, black–holes, false failure advertisement, disconnection, or forwarding inconsistencies which trigger packet loss which results in availability issues. Even, the ordering is highly important. Rules in the datapath should be ordered and that arrangement influences the decision taken by the datapath.

Consider the network example depicted in figure 1.2 and suppose that the network administrator configures a path between nodes A and E. The administrator may define a high–level policy thereby: *IT members in the subnetwork A can access the video database in subnetwork E.*

The policy by itself does not guarantee the correct network behavior nor the underlying topology can implement the policy. We can model the topology and the path as constraints for the implementation. Figure 1.2a shows a topology example that implements the policy. In the SDN environment, the network administrator can write policies on the control plane. Then, the controller automatically translates policies into rules and installs forwarding rules according to the network application. The controller also decides the distribution of rules into datapaths. For example, figures 1.2b and 1.2c are two paths between nodes A and E that accomplish the policy. The controller implements one of the possible configurations according to physical topology and other requirements —such as mandatory datapaths, included/excluded nodes



(a) Physical topology.

(b) A – C – E path.

(c) A – B – C – E path.

(d) No path satisfies the policy.

Figure 1.2: Multiple configuration of paths between A and E datapaths.

for the path, available network resources, permissions, etc.—. The controller implements the path that satisfies the policy and the restrictions. For example, the figure 1.2c is the resulting path if we add a new restriction that avoids the node D. Finally, sometimes no policies can be implemented. Suppose a new policy that avoids nodes C *and* D. Figure 1.2d shows that no path can satisfy this policy. Int this way, the reader can note that bad–defined policies may generate erratic behavior in the network.

We show that important amount of infrastructure downtime is caused by human misconfiguration because it is a error prone task. Also, policy definition is a high–level abstraction of the expected behavior and its definition not always can be implemented by the infrastructure. On the other hand, because of the number of network devices, users, flows and rules, the validation and correction of rules could generate other policy violations. Even worse, human rule construction without verification could introduce security vulnerabilities.

For the reasons given, an automatic mechanism becomes necessary to validate the conformity between the high–level policy and the low–level implementation. In order to achieve this goal, we expose three issues to the discussion: 1. Description of high–level policies and logical topology specification in SDN to build a model that can be comparable. 2. Description of device implementation from datapaths. 3. A mechanism to validate if the specification, given by the policy, agrees the datapath implementation and report a list of inconsistencies.

1.3 Proposed Solution

The proposed solution in this thesis comprises three modules: a model of topologies, failures and policies; a model of implementations from datapaths; and a mechanism of validation. Figure 1.3 depicts the proposed framework which has two interpreters, one for policy and topology, and another for device configurations. After these two inputs are interpreted, we build two predicate–relational models and then they are evaluated to identify coherence or conflicts.

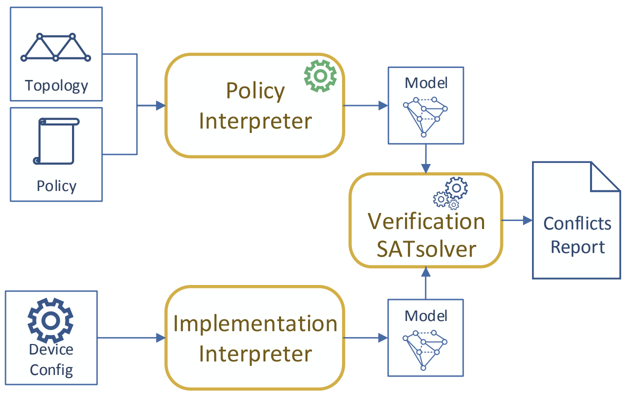


Figure 1.3: Proposal of general framework to validate policies, topologies and device configurations.

Diverse verification tools based on tests have been proposed to validate policies and restrictions for regular networks. Off–line validation programs specifies unitary and integrated tests, runs experiments, and identifies if the implementation fulfills the requirements. On–line validation programs are based on performance experiments which consist in configuring datapaths, verifying data transmission, and measuring and comparing with the expected result. These mechanisms can check if the configuration fulfills the policy but they cannot identify if instructions in the datapath contradict the policy, or if there are conflicts within the set of instructions. A policy implementation is said to be *closed* if the set of network policies are implemented by the set of configurations on datapaths, and procedures outside the policy cannot be executed by the datapath configuration. For example, consider a network policy that *Allows IT department members to access the database server*. The configuration on datapaths must allow connections from IT members to the database server, but also, the configuration must not allow connections form members outside IT department to the database server.

We propose to validate network misbehaviors on three domains: 1. wrong or neglected specification of infrastructure or applications, e. g. the topology described by the documentation does not coincide with the infrastructure; 2. errors in policy design, at the controller (procedure errors at control plane), e. g. a policy can apply restrictions for null or impossible traffic, visitor user can access and modify private records; 3. unsorted instructions on datapaths (errors at data plane implementation), e. g. if the administrator allows connections to 192.168.10.* should be instructed as 1) allow 192.168.10.0/24 and 2) deny *.*.*.*, not the other way.

We propose to use lightweight formal methods for checking network policies against a set of restrictions of a SDN environment. Specifically, we rewrite network policy validation as a SAT problem, and then use predicate logic transformation to find inconsistencies, conflicts and examples of configurations. Besides, we use a predicate solver to represent formulas and

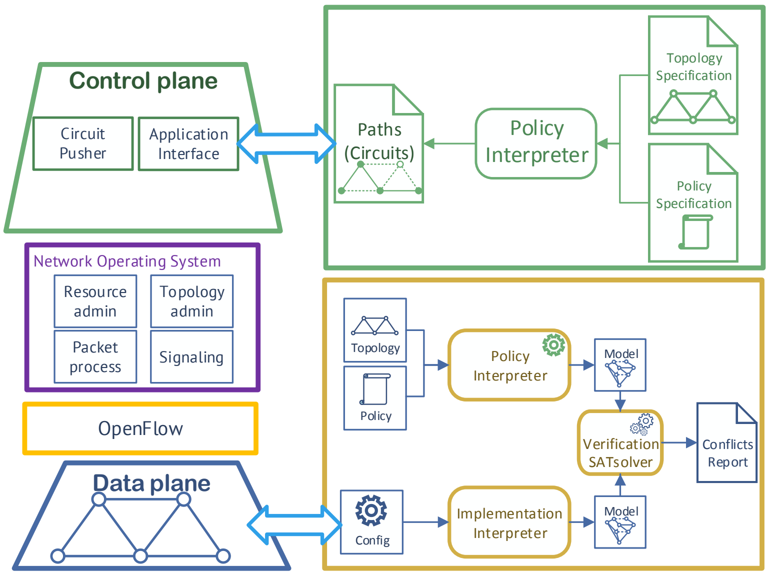


Figure 1.4: Framework developed on this thesis.

restrictions in Conjunctive Normal Form (CNF). To assist this work, we employ Alloy as the modeling language based on first–order and relational logic [Jac03]. Figure 1.4 presents the framework developed on this thesis. In short, we model network topology, policies and configurations on Alloy, use the solver to find conflicts, and map findings to specific anomalies.

In this research project, we process high–level network policies that later are interpreted by SDN instructions and OpenFlow commands. Then, we validate forwarding configurations on datapaths against a policy set. Thus, we explore the verification of closed–policy implementations. We consider that this work contributes to facilitate network availability on SDN.

1.4 Research Objectives

The objective of this thesis is to study network availability by the validation of network policies in Software–Defined Networking. For this purpose, we develop a strategy for mapping network constrains to logical–relational predicates and operate them.

The specific objectives for this thesis are:

- To represent the network infrastructure as a formal model that allows being evaluated (verifiable) under the specification of functional requirements in predicate–relational logic.
- To model SDN as an interdependent graph and analyze the impact of failures on this architecture that attempt the availability.
- To represent three network functionalities (network invariants, firewall, auditing) as formal representations that are tested using a model finder tool.

- To design a lightweight formal model for network policies that can be adopted under SDN paradigm.
- To implement a validation process using Alloy modeling language to test our approach.
- To analyze the validity of solutions through a case–verification.

1.5 Contributions

This thesis presents a series of works oriented to Policy–Based Management (PBM) which is a framework that aims to simplify and automatize network administration. SDN is suitable for PBM, given that the controller has a complete view of the network; therefore, it is able to apply network–wide policies.

In this manner, the contributions of this thesis are:

- A diagnosis of research in programmable–network management, and the identification of error reduction using formal validation of policies for SDN.
- A model to check network invariants and SDN topology features.
- A representation of SDN network as an interdependent graph and the analysis of its robustness after a sequential targeted failures.
- A set of modeling expression with which a network administrator could write high–level policies which are translated into forwarding rules regardless of devices, vendors, or operating systems.
- A model to check firewall rules and conflicts, and another to check if a policy is begin enforced by a foreign network domain.

1.6 Methodology

The methodology followed in this research thesis is in two ways, from the specification at control plane, and from the data plane to the validation. Figure 1.5 describes these two processes. In the first process, we model topologies, failures and policies in a predicate–relational language. Then, we develop an interpreter that transforms the specification into logical circuits. The data plane of SDN receives the logical circuits and installs them into the SDN datapaths In the second process, we grab the configuration from datapaths and use another interpreter to build the configuration model. Then, we use a solver of satisfiability (SAT) to compare these two models and report the congruency.

In the practice, we use this methodology for each contribution of this thesis. Specifically, we follow this list of tasks:

1. Architecture definition of Software-defined Networking. The documentation process, collecting the specification on SDN, focused on OpenFlow.
2. Identification of network properties to be verified under topology failures.

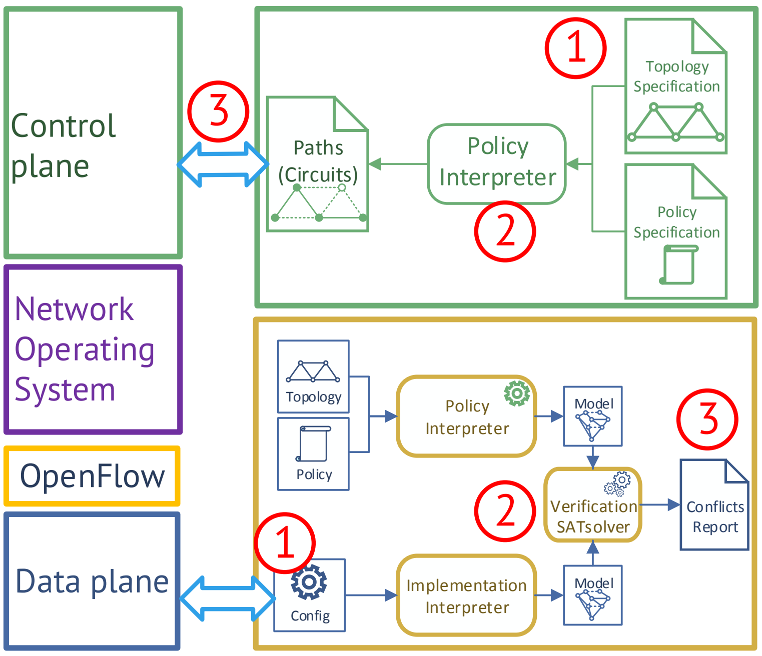


Figure 1.5: Methodology followed in this thesis.

3. Construction of a lightweight formal model, in Alloy language, that describes the network properties and a set of tests (violations).

4. Execution of experiments to find inconsistencies in the model (instances and counterexamples) using the modeling tool. Report the cases where the verification was positive, there were no inconsistencies between the model and the case, and negative, the case has induced false tests.

5. Simulation scenarios are implemented to collect data and evaluate the model, cases and instances.

6. Write reports and develop a plan to release them in proceedings and articles.

1.7 Outline of the Thesis

This document comprises six chapters. Chapter 2 has three parts. The first one offers a compact review on SDN and highlights the fundamental issues regarding this thesis. The second part presents essential background on logical systems used by verification. Finally, the third part includes the state of the art on verification techniques for networking and how they have influenced this thesis regarding to topology, firewalls and policy enforcement.

Once we present the fundamental concepts and related work, chapter 3 shows the initial model to verify network topology, paths, and traffic flows. Also, this chapter presents the basis of network modeling used along the document, the semantic function used to find conflicts and the case of network invariants from a path–definition language.

Chapter 4 offers a wider view of network availability applied to SDN environments. In this chapter, we model a SDN network as an interdependent network and analyze the robustness of its topology under the sequential targeted attack. This chapter finalizes with tests of three link patterns as result of a simulation.

Since it is necessary to apply this verification technique to a networking case, chapter 5 is dedicated to a firewall case on SDN networks. We tackle inconsistencies and conflicting configurations in an application of firewall for SDN that can generate availability issues. This chapter shows work on security, policies of filtering, firewalls and access control verified using a model finder based on Alloy. Moreover, it presents a strategy to reduce the number of variables, and hence the scope complexity in seeking solutions and counter examples.

While previous chapters show how to validate if the configuration corresponds to the policy, chapter 6 demonstrates if a policy is achieved from the forwarding information allocated on datapaths. This chapter describes a mechanism to monitor and audit network policies in a foreign domain. It is a challenging approach that verifies configuration and policies in the opposite direction, from the data plane to the verification.

Chapter 7 exhibits comprehensive framework for network policy management developed throughout this thesis. We summarize the results of our work and conclude this report, present findings and limitations, and describe some objectives that may addressed in future research on validation for networking.

Finally, appendix A presents the proposed mapping of policies that translated paths and topologies to predicate–relational logic. Appendix B describes in detail the schema to find conflicts and inconsistencies from the semantic function. Appendix C shows the grammar used in Alloy to model the complete system presented on this document, and appendix D presents the grammar structure to parse the topologies.

Chapter 2

Software–Defined Networking and Policy Management

If I have seen further it is by standing on the shoulders of Giants.

Isaac Newton

This chapter presents a review of Policy Based Management (PBM) for networks and its applicability to Software–Defined Networking (SDN). Since all network information is centralized in a unique logical location, the controller, there is an opportunity to create more efficient ways to manage and administrate the network by policies. PBM is the bridge between the logical management and SDN. The goal for PBM is to provide trustworthy configuration, and promotes the validation and design of robust architectures.

2.1 Software–Defined Networking

Modern computer networks are large and complex, manage a variety of applications, and execute diverse protocols. The current network architecture is vertical integrated, see figure 2.1 to appreciate that the same device performs control and forwarding tasks. Each vendor's network device includes hardware, the operating system, its applications and protocol implementations. Vendors control all technical details of their products and sometimes have proprietary and non–standard interfaces which may reduce the flexibility and compatibility of the device. Besides, current network management is supported on distributed algorithms and protocols which are not as efficient as they would be if a centralized controller had a complete view.

Moreover, due to the accelerated growth of new services and Internet traffic, providers, private networks, users and applications demand more capabilities from the infrastructure. For example, virtualization, data centers and cloud computing require dynamic and fast network technology to fulfill their requirements of elasticity [VN11]. Network *operation* will be software applications that control packets, monitor the network and administrate the

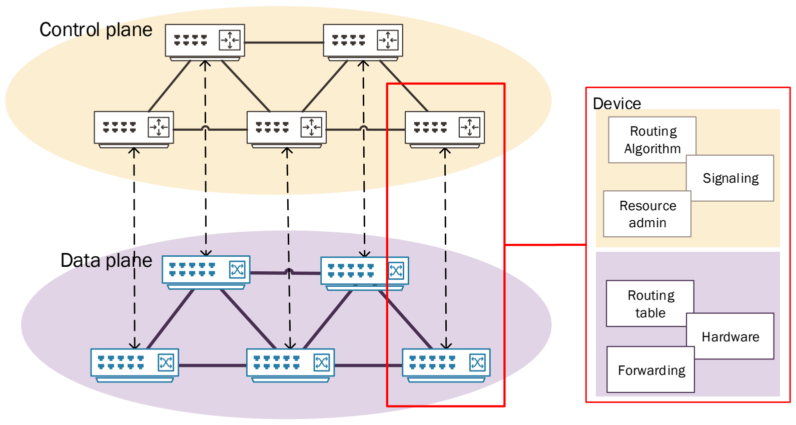


Figure 2.1: Control and forwarding tasks in current network architecture.

infrastructure.

During the last decade, this requirement is a concern for the computer networking community, and the development of programmable infrastructures is needed to support dynamic network requirements. That programmable approach must facilitate interoperability of devices from multiple vendors, facilitate the creation and management of dynamic topologies, and offer acceptable performance levels. The programmable network also has to guarantee security, reliability and failure resistance. That is why programmable networks like SDN have become more attractive for the community.

The process towards *Programmable Networks* is tightly related with computer virtualization because the concept behind is similar: software is used to emulate hardware interfaces. The *Click Modular Router* was designed as an architecture to build and configure routers by software [Koh+00]. Later, the idea of separating forwarding and routing was proposed as a Routing Control Platform (RCP) [Fea+04; Cae+05] which calculated and selected paths over a network without the complexity of a complete distributed system. Here, the concept of a software platform able to control network functionality, in addition to routing, began to take shape. Then, the project 4D [Gre+05] proposed the simplification of the architecture and managing the network in four dimensions: decision, dissemination, discovery and data. *Decision* involved all decision of the control, including reachability, load balancing, security and configuration of interfaces. *Dissemination* dimension is provided by an isolated communication channel between decision entities and datapaths. *Discovery* dimension detects new physical devices and establishes their governance. Finally, the *Data* dimension processes each packet according to the network state as the output of decision dimension.

A *centralized server* that makes interdiction decisions was proposed as Secure Architecture Network Enterprise (SANE)[Cas+06]. Here, the Domain Controller (DC) acts as a protection layer at link level, and decides how to dispatch the traffic instead of distributed devices like firewalls or routers. Posteriorly, Ethane [Cas+07; Cas+09] extended that SANE for corporate networks which had two components: the controller which decides how a packet

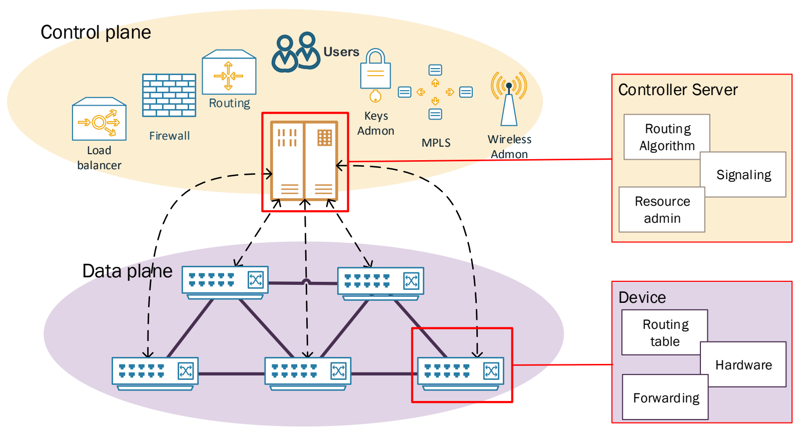


Figure 2.2: Control and forwarding tasks in SDN architecture.

should be forwarded, and the datapaths —regular switches— which had a forwarding table called Forwarding Information Base (FIB) to forward packets, and a secure channel to the controller. This work was one of the earliest to propose clear separation of control and forward tasks.

2.1.1 SDN architecture

In the current architecture, network devices are *monolithic* and operates the control and the data planes. Figure 2.1 shows the common network architecture where the device performs both tasks, and dotted lines represent the communication channels between control decisions and hardware instructions. The control plane manages addresses, routing and logic resources between network and link layers. The forwarding plane manipulates the packet and administrates device's ports and interfaces. Each device deals only with partial information and constitutes a portion of the distributed system to perform fundamental tasks such as addressing or routing.

SDN separates the control plane from the data plane, also called forwarding plane. SDN comprises three layers, upper layer corresponds to protocol logic and applications for network administration, the second layer is the Network Operating System (NOS) as support of the control plane, and lower layer is the data plane and forwarding. Figure 2.2 shows the SDN architecture and its layer division by purpose. The logical channel between controller and forwarding, dotted lines in the figure, operates the OpenFlow protocol [McK+08a]. The *controller* is responsible for running network functionality. In addition, it can execute other high–level applications to control the traffic. Figure 2.3 shows architectural differences between the current network and the SDN paradigm. In SDN, each layer is executed on different hardware, they are physically decoupled. The controller runs on a server, and the forwarding is performed on simple datapaths (switches).

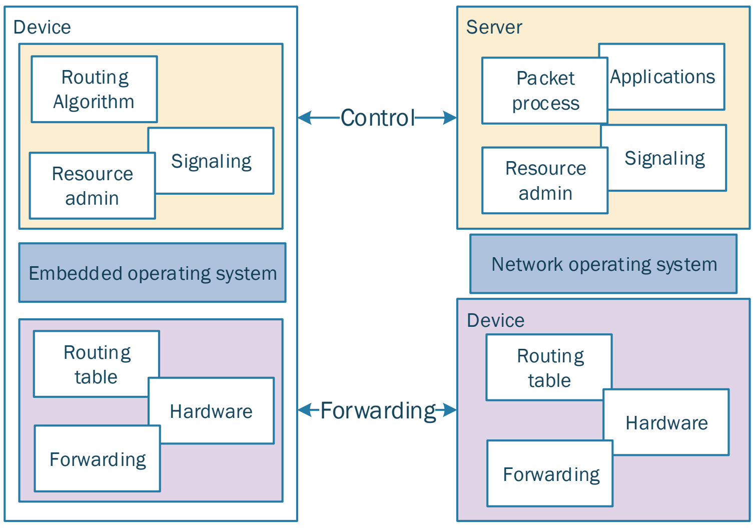


Figure 2.3: Network architecture comparison: Current network vs. SDN.

OpenFlow

OpenFlow protocol (OF) is an open implementation of a protocol that communicates the controller and the datapath. It allows to program the datapath and modify the forwarding table, commonly called flowtable. [McK+08a]. A secure channel is established to connect the controller and the datapath through the OF protocol, so the controller can install/delete/modify rules into datapath's flowtable. Datapaths store the flowtable with forwarding actions for each match entry with current-protocols and regular Ethernet fields. OF defines three atomic actions: SEND, DROP, and SEND_TO_CONTROLLER to perform over a packet when it reaches the datapath. Figure 2.4 shows the header fields used by OF to manage traffic. The forwarding table stores is a list of packet–headers matchers and the correspondent action to be executed to a packet.

In Port	VLAN ID	Ethernet			IP				TCP	
		Src	Dst	Type	Src	Dst	Proto	ToS	Src	Dst

Figure 2.4: Packet fields for header matching in OpenFlow version 1.0.

The controller can modify entries from the flowtable, independently of the user and operation. Then, OF allows the creation of new flow rules that does not interrupt other users network utilization. Using the OF syntax shown in figure 2.5, the network administrator can program the datapath behavior, and decide how the packets are processed by the device.

SDN can also offers *isolation* which allows independent network domain for each user, and facilitates to create *virtual networks*. Other extensions for OF have been developed and depend on the protocol version, [Ope11; Ope12; Ope13].

Initially, OF was created for research purposes, using Ethernet switches and standard interfaces. Then, multiple vendors offered APIs and protocols based on OF to manipulate the flowtable on their devices and called it *OF enabled*. Using this architecture, network

Rules	$r ::= \langle pat, pri, t, [a_1, \ldots, a_n] \rangle$
Patterns	$pat ::= \{h_1 : n_1, \ldots, h_k : n_k\}$
Priority	$pri ::= n$
Timeout	$t ::= n \mid$ None
Actions	$a ::=$ output (op) $\mid$ modify (h, n)
Header	$h ::=$ in_port $\mid$ vlan $\mid$ dl_src $\mid$ dl_dst $\mid$ dl_type $\mid$
	nw_src $\mid$ nw_dst $\mid$ nw_proto $\mid$ tp_src $\mid$ tp_dst
Ports	$op ::= n \mid$ flood $\mid$ controller

Figure 2.5: OpenFlow syntax.

administrators are able to share resources in a dynamic way because it allows creating virtual networks, independent network domains for multiple groups of users, and have a fine–grain control of the traffic.

Forwarding devices

A datapath is an OF enabled device, switch, router or any forwarding device, responsible for storing the *flowtable*, also called forwarding table, with a set of actions for each entry. This table stores a list of match-action pairs ⟨`match,action`⟩. If a received packet matches any header–rule of the table, the device executes the associated action. On the other hand, if the packet header does not match any entry, the device sends a request to the controller. A request contains the header, destination and received ports, and in some cases, it includes the singular packet. After receiving the request, the controller can create or modify a rule, and update the forwarding table on the device. An OF device checks matching using current protocols and Ethernet fields. Figure 2.4 shows possible header-fields used by OF. Controller and network applications can process packets using these fields with OpenFlow. The registered actions that a device can perform are: `send` to a specific device-port, `drop` the packet, or `send to controller` request.

The network controller

The controller acts as an intermediate layer (2.5 layer) that manages data plane devices and holds the network state. Normally, the controller is a centralized entity, has a database with topological information, and it supports the intelligence behind the network. Under this perspective, network layer devices do not have to calculate and handle routing tables, as the traditional L3 devices did. Logical sets of rules, such as routing protocols, are applications that run on the controller and set forwarding instructions on each device under the controller's domain. All routing information, such as addressing, spanning trees or path calculation, comes from the controller. Moreover, the controller is the only authorized entity to add, modify and delete flow rules on datapaths. For instance, the controller can also execute the path computation element protocol (PCEP)[a] that computes paths over a topology. The controller, and its NOS, has two interfaces: north and south. The northbound is an Application

[a]IETF RFC 4655 defines the path computation element (PCE), path computation client (PCC), and the protocol.

Program Interface (API) with network applications that provides controlling services. And the southbound communicates the controller with datapaths.

In summary, OpenFlow facilitates the separation of control and forwarding functions. Network equipment is reduced to devices that receive and forward packets according to its flowtable. Regularly, network devices report unknown or new traffic to the controller, and it updates its network information. Diverse implementations were developed to optimize flowtables on datapaths. For example, DevoFlow [Cur+11] uses the ASIC hardware to separate long flows from small ones, and central control has only to focus on significant flows and reduces the amount of flowtable entries. In this way, it minimizes requests to optimize the interaction with the control plane. Later, Nguyen et al. [NSBT14] proposed an optimization problem of rule selection and placement on switches and solved it with Integer Linear Programming which includes in and out points, and consider that routing policy can be ignored if those points are respected. Therefore, its allocation procedure maximizes the rule-allocation that satisfies the out points.

2.1.2 Software for Software–Defined Networking

One of the main advantages of SDN is that they are supported in programming languages and frameworks. Declarative programming proved to be a suitable candidate to program network functionality.

Network Operating Systems

Diverse controller platforms as NOS have been developed. This concept was originated from the 4D project that aims at a new, more manageable, system that facilitates configuration and control [Fea+04; Gre+05; Mal+04]. NOX [Gud+08] was a pioneer approach of a network operating system; the first OpenFlow controller. NOX was designed as a centralized system that maintains the network state, on a database, and would allow to program using high-level abstractions. Multiple works followed NOX and new approaches were implemented for specific features [Kim+12]. Beacon [Eri13] and Floodlight [Big12] are network controllers implemented with Java, which support event–based and threaded operations. However, having a centralized controller causes a scalability issue. Onix [Kop+10] and HyperFlow [TG10] distributed the controller and defined methods to maintain consistency. On the other hand, Maestro [CCN11] exploited parallelism on a single machine and reduced the overhead, thus reduced the bottleneck, compared with NOX [Gud+08]. Maestro is also modular and allows to programmer to build personalized views of the network state. Recently, other platforms have gained popularity, for example IRIS [LPSY14] which offers a more scalable and available controller.

As testing tools, software for network functionality had a significant rise with easy prototype platforms such as Mininet [LHM10], which offers run, debug, and test software for networks with limited resources.

Programming languages and compilers for SDN

Programming languages for networking showed an important evolution in recent years. After the advent of SDN, many programming languages for networks were created. All new programming languages have the high-level abstraction in common instead of flow or forwarding level. In this section, multiple programming languages are presented, and their features are highlighted.

Declarative programming is a paradigm that allows building program elements and structures in terms of computation logic, rather than describing how the algorithm is implemented by the flow control. Under this programming paradigm, a program is a deduction of a formal logic over a logical space. Some examples of declarative programming are SQL, regex, logic and functional programming and configuration managers. Due to SDN programming is focused on describe the behavior of packets in the network, declarative programming fits this need, and helps to model the graph representation by the formalism. One of the most representative languages is Frenetic [Fos+11], which is a declarative programming based on NOX. The network administrator composes network policies in the Frenetic language, then the compiler translates policies into stream queries and transformations. With Frenetic it is possible to install low–level rules at datapaths and its performance is comparable with NOX over OpenFlow implementations. Rule composition was one of the most important advancements on network programming.

A packet is perceived as an input, whereas a network functionality, routing for example, is a composition of functions of a datapath after another. NetCore offers foundations for supporting parallel composition, uses predicates for filtering, and actions modify the packets [MFHW12]. NetCore is a high-level declarative language that describes the desired behavior of the network but does not detail the implementation of that behavior. With NetCore it is possible to express packet forwarding policies for SDN. Pyretic [Mon+13] was a language that developed parallel and sequential composition of network modules. Pyretic abstracted high-level modules and operates in parallel where multiple policies can act over the same packet. Merlin is another declarative language based on logical predicates and regular expressions with which a network administrator can write network policies [Sou+13]. Merlin compiler uses a constraint solver and heuristics to allocate network resources, find paths and assign bandwidth. Regular expressions include union, concatenation, and Kleene star. NetKAT [And+14] has axiomatic semantics and compiler based on Kleene algebra for reasoning about networks, and Boolean algebra about predicates. Network is viewed as an automaton that *moves* packets from a node to another within its topology. NetKAT defined a finite automaton, and used regular expressions to represent network infrastructure and Boolean reasoning of predicates with Kleene algebra with test (KAT). Other tools to simplify programming for SDN is a language independent system called Maple [Voe+13]. With Maple, an administrator writes general network forwarding as functions f in a general-purpose programming language; those are called *algorithmic policies*. In theory, the function f is applicable to every packet, but in praxis Maple identified reusable forwarding instructions, recorded the invocation of the function f, and generalized outcomes and dependencies to other packets. Flowlog [NFSK14] is a tireless language for controllers and represents all three layers into a single abstraction. Its syntax is a mixture of SQL and rule–based languages that describes forwarding tables. Flowlog runs verification based on Alloy to check program correctness and topology properties.

Debugging the network

Debugging network applications is also a critical procedure for SDN. OFRewind [WLSF11] is a network debugger that stores and replies network events to reproduce errors and failures, and helps to identify root causes. NetSight [Han+12] is a network debugger, analog to `gdb` for programming. *OFf* [DSB14] is a debugger for SDN, as a regular programming tool, moreover it included packet tracing, replay, alerts and other visualizations of network behavior. More information about languages, debuggers and tools for network programming can be found on the survey of languages for SDN [Fos+13]. Handigol *et al.* proposes a debugger for SDN called `ndb` that traces sequences of events, backtracks errant packets, and implements breakpoints [Han+14]. The packet backtrack computes the forwarding sequence where a packet goes through once it reaches the breakpoint line. In this way, the network programmer is able to identify all forwarding details for each node, including flow-table states, flow matching, and ports. Moreover, it can check the correctness of forwarding.

SDN architecture has three layers: control plane, data plane and controller state. Flowlog is a language that abstracts all three layers into a unique abstraction. It is based on SQL and rule-based languages. Flowlog programs are compiled to a lightweight formal modeling and verification tool called Alloy.

2.1.3 Network applications

Network functionalities are implemented as protocols, or packet processing in middleboxes. The first step on the road towards SDN was implementing switching and forwarding, which was achieved by OpenFlow [McK+08b]. Later, multiple functions were brought as software applications, taking advantage of the programmable network. Also, network configurations that respond and adapt to application needs, facilitate network processes, or create additional functionalities are now possible under the SDN paradigm. In this section, some network applications are shown and discussed.

Application–aware networking

With SDN, the network programmer automatizes configurations to satisfy custom requirements. From traffic engineering to network security, SDN applications have steadily grown. Das et al. [Das+11] proposed to configure OpenFlow for multiple services over a regular packet network. In this way, OF can have differential traffic operations and use multiple metrics and applications. The traffic is processed in accordance with network resources, and application requirements. For example, for BigData applications like Hadoop, SDN is used to support scheduling jobs, integrate the topology and reconfigure routing [WNS12]. For datacenter networks, SDN is a cornerstone element given that the multitenant environment requires isolation by applications, users, even routing mechanisms over the fat–tree topology.

Network topologies for data centers and its fat–tree architecture was designed to have *optimal routes* from any-to-any node. SDN can change the topology configuration in contrast with the one–size–fits–all architecture used on datacenters [WSY11]. For this reason, SDN has impacted in the virtualization of network functionalities. FlowVisor [She+09] is the virtualization at datapath level that allows to create logical networks using the same forwarding infrastructure. FlowVisor acts between switches and multiple OF controllers which allows to isolate research traffic along production traffic. According to Kop, network services for multitenant in datacenters take advantage of SDN [Kop+14]. They work over non–virtualized networks by encapsulation of packets and the delivery is not based on regular addresses. Then, security policies are added to tenant's traffic and the tenant information is hidden. Hence, this traffic is isolated, even unable to be tracked for the datacenter hypervisor.

Network administration

Load balancing is the strategy for traffic distribution across multiple resources. However, this practice is expensive and becomes a single point of failure, so it would be convenient to divide traffic over shared resources as servers using OF. A naive approach installs *microflows* on each datapath which will act as a load balance on each node; however, this strategy will overload the controller. Wang et al. [WBR11] used OpenFlow *wildcards* and proposed algorithms to compute them and automatize load-balancing policies. Hedera [AF+10] is a dynamic-flow scheduler that acts as a centralized load balancer and aggregates network resources on datacenters, especially on those with multi-rooted tress topology.

Route analytics and traffic management are also improved with SDN implementation. Agarwal et al. [AKL13] showed reduction of packet–loss and delay in a network, even when SDN is incrementally instituted. As complement of the routing control platform [Cae+05], the route–flow control platform (RFCP) is a hybrid controller–centralized routing engine that rules multi–vendor networks under the same dashboard [Rot+12].

Administration of mobile networks can also benefit from SDN. Odin [SSZMF12] is a framework to control WLANs, it offers security services of Authentication, Authorization and Accounting (AAA); managing of mobility, interference and channel configuration; and network administration tools such as QoS constraints and load balancing. Odin *agents* run on APs, and the *master* runs on the centralized sever which controls all WLAN related operations.

Security and Fault management

Network security is the network characteristic better served by having a centralized administration. Security interests on SDN go from the centralized controller to secure dataflows. The Secure Architecture Network Enterprise (SANE) [Cas+06] was one of the earlier approaches to secure data flows by a centralized entity called *domain controller* (DC). SANE is an architecture that acts as a protection layer, at link level, where all interdiction decisions are made by a centralized server instead of distributed devices like firewalls or routers. SANE is an access control architecture that involves decisions of ACLs, packet filters, NATs and middleboxes. SANE was created to deal with network threats like worms or malware, and its principal advantage is to allow high-level policies and then translate into common configuration scripts. Domain Controller (DC) is the server that takes all decisions; it has a complete view of the network and grants access based on roles. SANE offers the authentication service that secures communications and maintains the guarantee of authenticity. Moreover, network services are published to be accessed through the DC instead of using a DNS. A general overview of security for SDN can be studied at [FDFE14; AX15].

OF allows the implementation of more complex security applications like quarantine procedures and malicious mitigation. FRESCO [Shi+13] is an application development framework to design and compose security detection and mitigation modules. It has a monitor and 16 modular libraries within an API. Each modular library includes the interfaces: input, output, event, parameter and action. This API allows legacy application to trigger any of its modules. For instance, PDI–based applications can execute detection scripts and trigger responses which generate new flow rules. Network administrator can adapt the current security solutions (applications) using FRESCO. Regarding to insider attack, FRESCO implements a signature to identify who established the new rule to trace the adversary. Also, the signature is used to apply authority and permission.

Anomaly traffic detection was also benefited from network functionalities implemented by software. Mehdi et al. [MKK11] implemented the Credit based Threshold Random Walk (TRW-CB), rate–limiting control, the maximum entropy detector, and Network Traffic Anomaly Detector (NETAD), common algorithms for anomaly detection, and claimed imperceptible delay in communications on low speed connections such as home or small offices. OpenFlow uses the Link-Layer Discovery Protocol (LLDP) to discover link or node failures and trigger restoration procedures. All failures are reported to the controller which should handle the failure. However, this load can be large and adds more complexity for the fault management and the recovery processed at the controller. Kempf et al. [Kem+12] proposed to locate a monitoring function able to dispatch failure alert messages and handle some failures without overloading the controller. Nevertheless, adding more functionality to datapaths contradicts the function separation of control and forwarding planes. While it is true that this kind of procedure increases the reliability, and reduces the downtime, also increases the complexity at the simplistic datapath.

2.1.4 Challenges for Software-Defined Networking

Although SDN has been widely supported by the community, there are many open challenges which require developing new tools and exploring other fields to capitalize on the flexibility of the paradigm [Wic+15]. Dynamic service, QoS and application performance are a must on these programmable networks. The control plane must be able to determine the destination of each packet, no matter the granularity level of the policy or its specific flow. Now, the network is flexible enough and new tools to measure performance and parameters of quality are needed.

On the other hand, SDN is exposed not only to physical but to logical failures in the design, development or implementation. Previous network paradigm had a totally distributed system, partial information and convergence issues; now, every control decision remains in the centralized controller. Therefore, the controller can calculate and monitor new flows and real–time switch performance. The problem is to have a reliable controller, or whether it should be in only *one* place, or logically centralized but physically distributed remains undecidable [Dix+13]. The controller placement problem tries to solve where to locate the controller to achieve a trustworthy service level and the maximum performance [HSM12].

We present a comparative table, table 2.1, that denotes the advantages and disadvantages of SDN compared to traditional architecture based on four network administration domains: scalability, performance, interoperability, and security.

Table 2.1: Advantages and disadvantages of SDN regarding traditional networking architecture.

	Advantages	**Disadvantages**
Scalability	• Centralized point of configuration. • Commodity hardware can be automatically programmed.	• Large networks with multiple controllers inherit synchronization problems. • Back–end databases for large amount of flows may generate consistency issues.
Performance	• Global network view, algorithms are efficient and avoid redundant overhead. • SDN facilitates dynamic configuration for large infrastructures as Datacenter networks.	• Information from all datapaths is sent to unique controller can generate bottlenecks. • Software bugs and errors are propagated to all network devices. • The first packet of a flow forces the installation of new rules from the controller.
Interoperability	• Old behavior con be programmed, then full compatibility with legacy hardware. • SDN benefits for one domain networks.	• Traditional networks have implemented multiple QoS resources and solutions which must be reprogrammed. • Multi–domain requires authorization and authentication.
Security	• Statistics collection of traffic rules allow identify attacks at early stages. • Flow–base forwarding facilitates the access control.	• High exposure to new or unknown vulnerabilities. • Centralized control is hard to replicate and diversify in case of compromise it.

Moreover, SDN inherits all the problematics of software. Legacy tools and network security mechanisms must be updated to SDN applications. Now, SDN is a huge software project with its implications of versioning, debugging, large deployment, testing and verification. Aside from traditional security considerations, SDN paradigm also introduces new kinds of network vulnerabilities such as application isolation and eavesdropping [WH13; Feh13].

The *softwarization of telecommunications* offers more flexibility and power because of the programming nature. Multiple studies demonstrate that human errors are the most frequent cause of incidents, even more critical than malicious acts [LSK09]. The goal of our framework is to minimize automatic/human errors. This framework comprises four stages: error avoidance, correction, interception, and correction. This thesis is focused on the use of formal methods for the first stage, specifically it uses model checking to *avoid* and *identify* failures in network functionality. We address the concept of anomaly, policy violation, and inconsistency from the network environment to *system's conflict*. Then, we check that network properties are congruent in the system model. Finally, we verify network functionality on the SDN scenario.

2.2 Verification and Model Checking for Networking

Program analysis is a set of mechanisms to prove program properties, like robustness, with respect to a *formal specification* for a set of functions. It is used to describe complex systems and reason about them. Recently they have been adopted to check software functionality, build reliable systems, and model and test critical infrastructure. Nevertheless, formal verification is gaining more attention by researchers and it is being applied to diverse areas of computation, of course networking is not an exception. There are multiple verification techniques based on formal methods for networking [QH15]. They consist in using computer techniques based on mathematical logic to specify, analyze, and validate network functionality. It is a natural step due to the clean state developed by SDN, and the opportunity to develop, debug, test, and run network functionality in a *soft* environment.

Dynamic analysis is performed during execution time of a program and *validates* if the execution fulfills the requirements. Common methods in this classification defines *test cases* such as unity, integration, and system test, and then the tested program must run those tests. Monitoring / profiling is a kind of analysis that measures program complexity and use of resources. This analysis is focused on program optimization. For example, slicing is the reduction of statements of a program and validates its behavior after a given set of variables.

On the other hand, static analysis is executed about the code —or the model of it— instead of the execution. This analysis *verifies* if a program satisfies the requirements. Figure 2.6 shows the taxonomy of program analysis approaches [NNH99]. In networking, this analysis means either checking the fixed configuration of a network or only its state. Symbolic simulation, SAT, and model checking are types of static verification. Incremental verification is based on static verification but it checks incrementally from previous solutions or results. The model specification is focused on the system behavior. Therefore, the model is defined by mathematical objects as sets, sequences, relations, functions and mappings. The model may also include invariant properties, and pre/post conditions. Model checking [CE82] is a formal verification technique that explores solutions over a *model* which represents

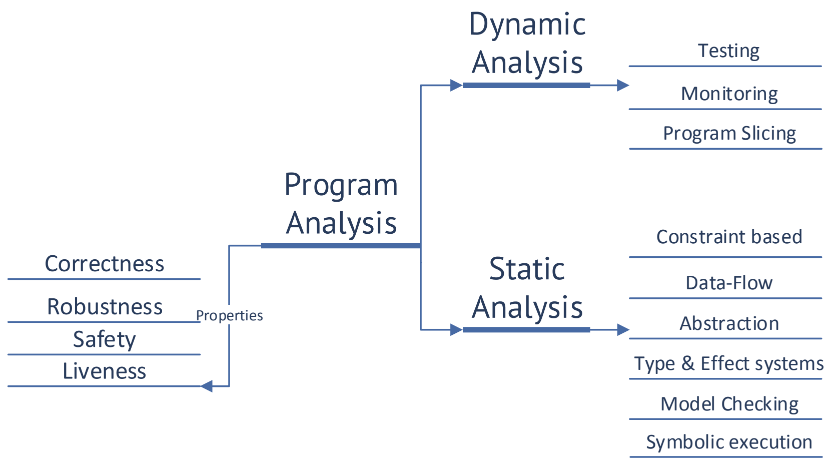


Figure 2.6: Taxonomy of program analysis

the state of a system. $\mathcal{M}$ is said to model formulae ϕ, noted as $\mathcal{M} \models \phi$, if ϕ is true in the representation of the model $\mathcal{M}$ [Cla08]. In a nutshell, in this thesis we build the formulae that represents the system network behavior and check if it is true under its specification. This software verification approach sets values to program arguments and explores the system's transitions. Model Checker is an instrument to verify the correctness of a program that relies on the counterexample principle. To prove if a property P is held, it is enough to discover a counterexample that states that P is not valid or possible.

2.2.1 Logical Systems

In this section, we present multiple logical frameworks used to verify and validate network properties in this thesis. The propositional and first–order logic can represent behaviors in the network, as graph. With HOL logic we can validate the specification of computational programs in terms of pre and post conditions Temporal logic is mostly used for communication protocols because it allows to represent reactive actions and timer events. And relational logic to represent and calculate data structures.

Propositional logic is formed with propositional variables[b], and predicate logic that includes propositional symbols, predicates, functions, quantifiers, equalities, and variables into its formulae. This logic was devised by Hilbert and Skolem, it includes two common quantifiers: (∃) exists and (∀) for all. First–order logic operates objects, relations, functions, and quantifiers which are defined over individual elements. Multiple analyzer tools are based

[b]A variable that can be `TRUE` or `FALSE`.

on first–order logic, for example, Vampire's prover or Alloy[c] analyzer [d]. See § 2.2.3 for these tools.

High-Order Logic (HOL) is more expressive and allows to express any mathematical theory. However, the validity of high-order logic is not even semi-decidable. *Hoare Logic* is another class of logic systems that is used to model computer programs. It defines the partial correctness specification in terms of precondition-command-postcondition $\{P\}C\{Q\}$. If the command C is executed in a state P, the state Q is satisfied. Hoare logic is specially situated for checking sequences of commands.

Meanwhile, modal logic models a variety of *modes of truth.* Created by Leibniz and revived by Kripke, it uses expressions such as *possible* and *necessary* worlds. The statement is true in at least one world, under specific conditions, in the possible expression; on the other hand, the expression *necessary* models a statement that always is true no matter the conditions. As we will see later, the possible worlds and Kripke models are powerful groundwork for model checking.

If it is necessary modeling reactive systems, temporal logic is the most suitable. For example, this logic system is fitted for network protocols. It is also convenient for modeling concurrent finite–state systems. Temporal logic combines transitory operators like G(globally) for always true, F(eventually) a preposition will be true in the future, X for next and U for until. There are two kinds of temporal logic: Linear temporary logic (LTL) describes the time as a line and a sequence of events; and Computation Tree Logic (CTL) which models an event and the branches of many possible futures. CTL has been used for checking network systems. For example, ConfigChecker [ASA11], and network update [Rei+12] used CTL for verification on networking.

Finally, relational logic is a formal semantic, created by Codd, and used to model data structures as relational databases. It is relevant for this work because it combines first–order operators with relational calculus.

For networking, diverse schemes have been proposed to formalize networks. For instance, Karsten et al. [KKPB07] presented a functional abstraction of protocols, layers and middleboxes which were presented as a set of axioms for forwarding mechanisms. This formulation allowed to define fundamental concepts for networking such as *naming* and *address*, then composing communication protocols was possible by a set of well–defined primitives. They implemented the Universal Forwarding Engine able to interpret the algebraic model, translate meta–compilation to Click router implementation, and validate its implementation. This validation could be formal following the Hoare–logic prove for correctness.

In this thesis, we use the basis of prepositional and relational systems to build a model, a specification, and a set of rules that describes the behavior of the network, and the network state. We use Alloy model–finder to create, operate, and check relational structures. Moreover,

[c]Declarative specification language to express constrains and behavior of a software, `http://alloy.mit.edu/alloy/`.

[d]vprover `http://www.vprover.org/`.

the analyzer uses relational logic plus the *transitive closure* which adds expressiveness as it describes the constraint propagation. Alloy is a *model finder* that uses first–order logic and a domain, and finds an interpretation where the formula is satisfiable [Jac02]. It uses diverse satisfiability solvers able to determine a vast number of constraints and variables. Now, we describe the verification based on the satisfiability property as the basis for our proposal.

2.2.2 Verification by Satisfiability: the SAT

Boolean Satisfiability problem (SAT) determines if the true or false variable assignment will make the entire formula true. SAT problem is represented by $S = \langle X, D, C\rangle$, where X is an n–tuple of variables $X = (x_1, x_2, \dots, x_n)$, D is an n-tuple of domains $D = (D_1, D_2, \dots, D_n)$ such that $x_i \in D_i$, and C is a t-tuple of constraints $C = (C_1, C_2, \dots, C_n)$. Constraints are translated to CNF by:

i) simplification reduction, $P \rightarrow Q$ is rewritten as $\neg P \vee Q$, and $P \leftrightarrow Q$ is rewritten as $(P \vee \neg Q) \wedge (\neg P \vee Q)$;

ii) replace $\neg\forall xP$ with $\exists x\neg P$, $\neg\exists xP$ with $\forall x\neg P$, $\neg(P \wedge Q)$ with $(\neg P \vee \neg Q)$, $\neg(P \vee Q)$ with $(\neg P \wedge \neg Q)$, and $\neg\neg P$ with P;

iii) quantifiers are moved replacing $\forall xP \wedge Q$ with $\forall x(P \wedge Q)$, $\exists xP \wedge Q$ with $\exists x(P \wedge Q)$, $Q \wedge \forall xP$ with $\forall x(Q \wedge P)$, $Q \wedge \exists xP$ with $\exists x(Q \wedge P)$, $\forall xP \vee Q$ with $\forall x(P \vee Q)$, $\exists xP \vee Q$ with $\exists x(P \vee Q)$, $Q \vee \forall xP$ with $\forall x(Q \vee P)$, $Q \vee \exists xP$ with $\exists x(Q \vee P)$.

iv) Initialize the set $\gamma = \{\}$, the Skolemize the formula $\exists x\forall y\forall zA$ with the Skolem constant $\forall y\forall zA[x(\gamma)/x]$, and $\forall y\exists zP(y, z)$ is replaced by $\forall yP(y, f(y))$ with Skolem function.

v) Disjunctions are distributed replacing $P \vee (Q \wedge R)$ with $(P \vee Q) \wedge (P \vee R)$, and $(Q \wedge R) \vee P$ with $(Q \vee P) \wedge (R \vee P)$.

The first operation simplifies the formula, then the resulting is a formula which contains only $\forall, \exists, \wedge, \vee$ and $\neg$ connectors. The second step applies Morgan's laws to reduce negations to atoms, the *literal normal form.* The third operation move quantifiers, the variables are renamed to be applied to the scope of each formula, the *prenex normal form.* The fourth step is to *Skolemize* the assertion, by removing *existential* quantifiers and adding Skolem constants and functions, the *Skolem normal form.* Finally, disjunctions are distributed and the formula is written as conjunctions and disjunctions, this is the *conjunctive normal form.*

For example the formula $\forall Y(\forall x(f(Y, X) \vee g(X)) \rightarrow g(Y))$ is rewrite as:

i) $\forall Y(\neg\forall X(f(Y, X) \vee g(X)) \vee g(Y))$

ii) $\forall Y(\exists X(\neg f(Y, X) \wedge \neg g(X)) \vee g(Y))$

iii) $\forall Y(\exists X((\neg f(Y, X) \wedge \neg g(X)) \vee g(Y)))$

iv) $\exists X((\neg f(Y, X) \wedge \neg g(X)) \vee g(Y)\gamma = \{Y\}(\neg f(Y, x(Y)) \wedge \neg g(x(Y))) \vee g(Y)$

v) $(\neg f(Y, x(Y)) \vee g(Y)) \wedge (\neg g(x(Y)) \vee g(Y))$

A formula is said *satisfiable* if there is a variable assignation whose evaluation is true, and unsatisfiable otherwise. In most of the cases, checking SAT is a NP-Complete problem, depending of the scope [PW10]. SAT problem in propositional logic was proved to be NP-complete. Thus, it is intractable but solved in practice. Quantified SAT (QSAT) is the canonical base of the SAT problem whose variables are quantified, it means they have universal or existence quantifiers. Quantifiers make the QSAT more expressive than SAT but it is a PSPACE–complete problem [KT06]. Additionally, Satisfiability Module Theories (SMT) are generalizations of SAT problems and included a combination of theories in first-order logic.

The verification problem can be expressed as a model $\mathcal{M} \models \phi$, where ϕ is a specification. The verification consists in determining whether ϕ is true on the world described by $\mathcal{M}$. Diverse problems in computation can be expressed as propositional satisfiability problems. For example, detecting conflicts, or inconsistencies, at network-device configuration can be re-formulated as a SAT problem where rules, flows and constraints are written as propositional logic formulae. This thesis uses *model* verification in order to validate network configurations and the specification achievement.

2.2.3 Solvers for satisfiability and verification

Traditional methods solve the SAT problem as a constraint decision problem. From the original David, Putman, Logemann and Loveman algorithm (DPLL)[e] to the most recent advances, have shown efficient strategies to solve SAT problems such as incremental solution or variable resolution [GPFW96]. Numerous tools, called SAT solvers, can calculate solutions of millions of variables in seconds. Verification was benefited by SAT solutions, as was shown for VeriSol, a framework for verification based on SAT [Gan].

Rapidly, SAT solvers have increased efficiency using diverse data structures and heuristics. For example, Chaff [Mos+01] improved performance due to the implementation of Boolean Constraint Propagation (BCP). MiniSAT [SE05] reduces redundant literals which produce conflicts, hence its lower memory consumption compared with the DPLL algorithm. Z3 is a Microsoft SMT solver that combined multiple solvers for SATisfiability. Yices [DM06] is another SMT that involves linear arithmetic and recently supports SMT-Lib notation. Unfortunately, QSAT and SMT solvers do not scale properly. Most of them need specific-data structures to improve the performance.

A Binary Decision Diagram (BDD) [Bry86] is a data structure that represents operations over Boolean functions. In short, it is a rooted, directed, and acyclic graph, where each branch means a decision over the formulation, see figure 2.7. Each leaf is the decision over a variable —or set of variables on the assignation— and the path through the tree denotes the variable assignation process. DPLL and other branching schemes use BDDs or reduced forms of the structure to perform the search. Biere et al. [BCCZ99] showed how the DPLL procedure can replace BDDs, and finding counterexamples was faster and reduce the space of BDD

[e]Davis–Putman–Logemann–Loveland algorithm is a backtracking search algorithm for deciding satisfiability published in 1962.

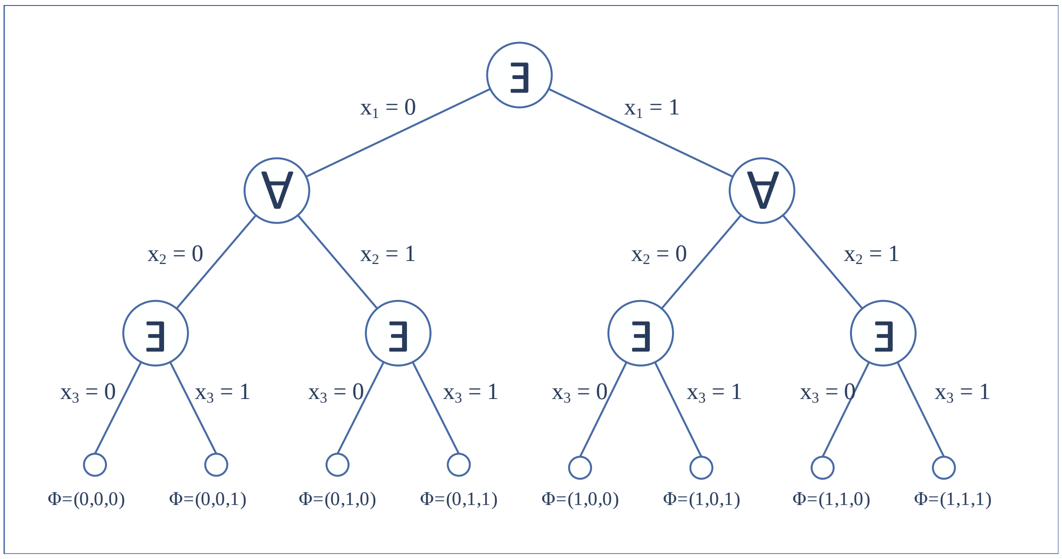


Figure 2.7: Each branch represents a decision over a variable. Leaves show the evaluation over the formula.

execution. Moreover, it presented how to apply SAT procedures to symbolic model checking. They also introduced the *bounded model checking*, a procedure that uses LTL to reduce the satisfiability problem. This bounded model, is an important solution that we will use later by using the Alloy tool given that it allows us to analyze larger systems.

Symbolic execution [BEL75] is a technique to validate a system by introducing *symbols* as inputs instead of data. Then, the program is executed, and the analysis determines which part of the software is executed by the inputs, or the conditions over them. The most significant issue is the *path explosion*, it means the program has to validate all the feasible paths for a run. Symbolic execution can be used for testing sets of inputs, with related characteristics instead of testing singular inputs. This technique was used by Cloud9, a platform based on clusters, that performs parallel symbolic execution for real–software testing[BUZC11].

Recently SDN started to use verification mechanisms for its software. For example, McGeer [McG12] modeled OF networks as high–level Boolean functions and demonstrated that the verification problem is NP-complete. Moreover, he showed that verification of OpenFlow rules is polynomial if the ruleset is restricted to prefixed rules. Older verification tools were used for networking. For instance, SPIN [Hol97] is a verifier for distributed systems, that was designed for detecting errors in distributed protocols and telephone exchanges. NuSMV [CCGR99] is an extension of the symbolic model checker SMV that integrates BDD search and LTL logic. Alloy is also another symbolic model checker that builds Boolean formulas, and has an interface with solvers.

Now, we present the construction of a model for networking, and how the fundamentals on this section are applicable to network specification and implementation.

Table 2.2: Classification of errors in system programs.

Error type	**Description**
Group A	The requirement or the algorithm that solves it is misunderstood. Machine configuration and architecture is chosen but it is known. Dynamic behavior is wrong, out of synchrony, incorrect resource allocation, functionality is not activated in the sequence.
Group B	The implementation of the algorithm is incorrect. Initialization fails, or the buffer is not cleared after its use. Register does not contain the expected value. Fields change their values but the size is not validated. Invalid counting and calculations.
Group C	The expected input is bad–formatted. Fuzzy instructions at operation. Spelling error in messages.

2.3 Verification meets Networking

As any other system program, there are errors in networking software with different causes. Table 2.2 shows a summary with the most common errors in system programs following the Endres classification [End75]. There are three classes of errors. Group A errors are those related to the problem understanding or the chosen algorithm is inadequate to solve the problem. Group B errors are those related with incorrect implementation, the translation to a program language is inaccurate. Group C errors are those related to the user interaction, they are the result of unattended or malicious use.

Since today's networks are more complex, their configuration is extremely large and more error prone than ever. Then, validating the functionality of software for networking is an imperative necessity. On the one hand, network administrators have tools based on tests. Generally, they use monitors and tools based on ICMP, or traffic to test network functionality. However, they also need to verify the coherence between policies and their configuration as a complete system state. As shown in § 2.2, SAT and SMT solvers become important tools for this kind of verification. Moreover, verifying network functionality for checking SDN software could offer greater guarantees of its functionality. Although verification and model–checking are large and mature areas, this thesis is focused on showing how SAT–based verification can be exploited for SDN. Verifying a network configuration is comparable to verifying a state machine because a datapath can be represented as a state machine that processes packets. Moreover, the reachability problem in a finite state machine is a PSPACE-complete problem.

This thesis aims to validate the specification to identify group A errors, and compare the model against the implementation result, group B errors. After this exercise, we expect to identify the match between specification and implementation, and observe lacks in the problem understanding process or faults/neglects in the implementation. Network properties studied on this thesis are described on table 2.3. Reachability basically denotes connectivity as in graph theory. Way–point and avoidance properties force to have or exclude nodes in paths. Loop–free and black holes are properties related to forwarding consistency, they advocate for

Table 2.3: Proposed network properties to be validated for this thesis.

Property	**Description**
Reachability	denotes that there exists a physical path that connects the two nodes. This property regards only topology —physical connectivity.
Loop–free	Routing ability to avoid paths with loops.
Black holes	Routing ability that prevents packets from getting lost. It is forwarding consistency between datapaths whose routing policy discard packets.
Way point	Mandatory node, describes a node which must be part of a path.
Avoidance	Exclusion of a node, describes a node that must not be part of a path.
Fruitful rule	All forwarding rules in a datapath are productive. Every forwarding rule belongs to a flow. It avoids bogus rules.
Settled rule	Every packet after reaching a datapath must match a forwarding rule. It avoids unsettled rules and delay while a new forwarding rule is set.
Non–Bypassing	A set of rules accomplishes a policy even after a packet modification. Policies can have overlapping rules which could be used to bypass the datapath.

successful delivery of a packet. Bogus are spurious rules in contrast with the optimal use of flowentries in a datapath; they denote those packet headers with no entries in the flowtable. Finally, the non–bypassing property means that a network policy is always achievable. Other properties that were not having into account: for example, *NoForgottenPackets* is a property that clears all packets at each datapath, or *DirectPaths* that maintains a path for all packets for a dataflow [Can+12].

2.3.1 Verification of Network Topology and its Invariants

The fundamental verification task is the topology validation. The main objective is to guarantee the fundamental functionality of a network, commonly referred as *network invariants* over the graph. Assuring these properties, the network could avoid severe failures such as service oscillation, black–holes, false failure advertisement, disconnection, or reduction of availability, and security vulnerabilities. Meanwhile, forwarding inconsistency can trigger packet losing instead.

Network verification can also be done by dynamic analysis based on *testing*, *monitoring*, and execution *slicing*. A common technique for dynamic analysis is to send test packages, establishing network monitors, or forwarding synthetic traffic for test with tools such as the automatic test packet generator (ATPG) [ZKVM14]. We divide the schemes to check network invariants for the data plane and control plane. Data plane verification identifies correctness at packet level, and determines forwarding coherence in the datapaths. Control plane verification assures the correctness of high–level protocols and accomplish the specification. Commonly, both verifications are executed to guarantee that specification is correct and the datapath implementation complies it.

Verification on networking has been done for some time. An important work on this field was done by Feamster and Balakrishnan who found group A error at configuration faults in BGP using a routing configuration checker called *rcc* [FB05]. And recently a work done by Yin et al. [YCZ10] found errors in Internet routers. Specially, they focused their attention on opensource implementation of routing, and documented the network behavior triggered by these errors.

This thesis is focused on misconfigurations and errors produced by human factors; although, other origins are considered as failure (malfunction) such as links failures, or shutdowns, see chapter 4. This work addresses the verification of specification of procedures and functionality. Additionally, we use *static analysis* to describe sets of packets and device configurations —e.g. routing— as filters over a packet set. Diverse works used static analysis for checking reachability. For example, Xie et al. [Xie+05] modeled packet filters and routing protocols by static analysis to verify reachability on IP networks. They also applied packet transformations to model Network Address Translation (NAT) and Type of Service field (ToS) modifications. Subsequent works used the abstraction from the static analysis in addition of Boolean satisfiability for error detection. Saturn [XA07], for example, used the advantages of SAT solver to model control and data flow. Metarouting [GS05] was a project, from a previous developed Routing Algebra framework (RAML), that defined routing protocols by a high–level declarative language. Routers compile the routing metalanguage and run the specified protocol. The protocol mechanism is separated from the routing policy —how routes are compared—. Naraing [Nar05] verified the network configuration using model finder Alloy and modeled a fault–tolerant VPN. It was one of the first to use model finders to solve configuration problems for realistic scale and complexity.

Anteater [Mai+11] is the iconic tool that uses static analysis of invariants in the data plane. It represented forwarding information as Boolean functions, and built multiple instances of Boolean satisfiability problems from symbolic packets, network constraints and invariants (§ 2.2.2). Then checks the network state, and reports counterexamples by using a SAT solver, in § 2.2.3. However, Anteater assumes that the packet does not change while it is passing through the network. Later, a technique called Header Space Analysis (HSA) [KVM12] was created to statically verify network specification, and its implementation was called *Hassel*. HSA could identify forwarding loops and traffic isolation. HSA took the complete packet header and expressed it as bit-string of length L, used it as geometric space, and defined operations –algebra– over the space. Network devices were modeled as functions that transform L-dimensional spaces. Otherwise, network and topology functions were expressed as header transformations.

Verification for SDN In recent years, research in formal verification for networking has grown, especially with model checking. However, the main concern is the explosion of states, packets in transit, and network functions and protocols. Since OpenFlow does not support hardware isolation, the flowtable is shared, and thus it could generate conflicts among flows, hence network misbehavior. Natarajan et al. [NHW12] proposed a conflict detection algorithm based on Description Logic (DL) and Web Ontology Language (WOL). It used a multidimensional prefix–tree structure (called *trie*) to create a hybrid hash–trie arrangement for the flowtable to identify conflicting flow entries.

Testing the correctness of network forwarding is a major issue in SDN networks. Data plane is complex with multiple middleboxes, and network policies are high–level descriptions of intended behavior. FlowTest [FS14] is a framework focused on testing stateful data plane devices such as firewalls, proxies or IDS, and dynamic network policies. It explores the space state of the data plane to verify the network behavior about the policy definition. FlowTest uses Datalog, propositional and first order logics to model the network state and generate test plans. Also, it uses machine states and CBMC[f] to find counterexamples.

Network Abstractions Sethi et al. [SNM13] created an abstraction of SDN controller and modeled learning switch and firewall applications. They implemented these abstractions that simulate the controller behavior using Murphi language[g]. Guha et al. [GRF13] built a verified controller that reasons the network behavior with Coq[h] and interprets the NetCore [MFHW12] language and compiler. Inputs are recognized as instructions in NetCore, which after being transformed into OpenFlow flowtables are analyzed in Coq by Featherweight, the proposed formal–model for OpenFlow. Another abstraction, but this time in the forwarding plane, was done by Mirzaei et al. [Mir+16] that modeled internal states of a network and OF switches using Alloy. They analyze flowtable entries, receiving messages, pipe–lines, forwarding processes, and transitions between states. Later, Frenetic [Fos+11] or Machine–verify [GRF13] only verify that coding compilation succeeds, but they do not check program correctness.

ConfigAssure [Nar13] was a project that aimed to reduce errors by manual configuration, and the ability of an adversary to gain information from the configuration file. ConfigAssure is a specification language and a synthesis engine that specifies requirements for multiple layers and allow composition, and reduces manual errors. The verification engine checks if the specification is correct, and a moving–target agent modifies the configuration file to increase the obfuscation. It also had a database with first order constrains in Quantifier Free Form (QFF) arithmetic, and used powerful solvers such as Kodkod[i] as SAT solver and Prolog. Finally, Kinetic [Kim+15] is a SDN controller and language based on Pyretic, that allows expressing dynamic policies with verification. This language allowed to program the dynamic behavior of a network and verify the reactive configuration to the changes. Kinetic verifies the language by a model checking based on CTL (see § 2.2.1).

[f]CBMC is a bounded model checker for C and C++ to verify arrays, pointers and exceptions to find bugs in programs.

[g]Murphi is a model checker based on explicit state enumeration. It is used on microprocessor industry for verifying protocols. http://formalverification.cs.utah.edu/Murphi/.

[h]Coq is a formal language to describe mathematical definitions, algorithms, and theorems to facilitate theorem proofs. https://coq.inria.fr/.

[i]Kodkod is a SAT-based constraint solver for first–order logic. http://alloy.mit.edu/kodkod/

Checking Invariants for Software-Defined Networking

After the SDN movement, all previous methods for verifying network invariants were updated to the new requirements. OpenFlow was the trigger that changed the verification of network functionality to more formal validation mechanisms. Early verification tested OpenFlow switches and the compatibility of devices from different vendors. Later, SOFT [Kuz+12] used symbolic execution for each switch in isolation and wanted to identify inputs that produce different outputs which generated inconsistencies.

Incremental computation is a technique that adds new rules to previous results and searches for potential violations only over the common rules. Veriflow [KZCG12; Khu+13] used incremental computation to verify rules before they were inserted or modified. Veriflow is located between the controller and datapaths, and verifies invariants while a forwarding rule is processed. Another way to verify invariants was proposed in NetPlumber [Kaz+13], a tool that used HSA [KVM12] to identify invariant violations such as paths with loops and reachability. NetPlumber included Flowexp, a language based on regular expressions and FML [Hin+09b], that can describe network policies. High–level network policies will be discussed on Section 2.4. Afterwards, Lopes et al. [LBGV13] verified the reachability property, detection of cycles, and forwarding loops on network configurations. They mixed and added incremental analysis to Veriflow and NetPlumber and claimed faster results after the verification tool was optimized for the network domain. Moreover, they create Datalog[j] network model and showed that is possible to verify networking domain with a general tool such as Prolog. NICE [Can+12] also applied model–check techniques to explore the complete space and used symbolic execution to examine all possible paths, packets and network events. However, its path search creates an extensive number of variables which add complexity on large scale applications. Recently, new verification of network invariants such as *Atomic Predicates* (AP) claimed to be more efficient because they reduce the set of predicates [YL16]. Any set of rules can be reduced to a unique and minimal set of Atomic Predicates. Then, those predicates are represented into a BDD (see Section 2.2.3), and process the diagram with well–known BDD algorithms.

We continue using this principle, using light–weight formal methods over a redesigned structure which allows us to verify the network behavior, not only from the configuration but also from the high–level management policy.

2.3.2 Verification of Network Security Properties

The security property is one of the most notable functions to be verifiable with formal methods. In this area, the Firewall receives special attention because it is the keystone in network security. Failures in firewall configurations are common and produce large losses in corporate networks. Some studies suggest that those failures by configuration are the result of bad–practices and follow a pattern [Woo04]. Their study is so important because they can generate security flaws, blocking appropriated, allowing undesired (malicious) traffic or simply contradict a security policy.

[j] Declarative logic programming language similar to Prolog. Commonly, it was used as query language for databases, data integration, networking and security.

A varied representation of formal evaluation of firewall policies has been developed. A model for firewall policies and their implementation of algorithms is presented by the *Firewall Policy Advisor* [ASH04b], which used a BDDs for discovering policy anomalies. Later, this work was extended to multiple firewalls [ASH04a], and those models were extensively used. The classic Fireman [Yua+06] used static analysis to verify firewall policies. It verified inconsistencies by a symbolic model checking for all packets along all possible valid paths into a set of configurations; however, it did not verify against a high–level policy, only checked if there were inconsistencies in the set. Another seminal model for analyzing firewall configuration is the Firewall Decision Diagram (FDD) [GL07] that represents a security policy in a directed and non–cycled graph, where edges represent constraints and final nodes represent firewall decisions. Later, formal verification algorithms [Liu08] also used FDDs to check misconfiguration, and validate firewall policies, but this time against the formal security policy.

Other works have used model–checking techniques with SAT–solvers and compared them with BDD approaches. Jeffrey et al. [JS09], for example, used data structures combined with BDDs and compared them with SAT solvers; and found efficiency performing over BDDs. Model–checking for firewall reasoning is a NP–complete problem instead of PSPACE–complete problem because there is not cyclicity in the firewall configuration. So, translating it into a SAT problem is appropriate to analyze this kind of problems. The same argumentation was used by Zhang et al. [ZMMN12] who implemented SAT to create equivalences and discover redundancies between two firewalls. Moreover, they used a Quantified Boolean formula (QBF) solver to optimize the Access Control List (ACL) set of rules. FLOVER [Son+13], also used a model–checking based on Yices[k], to find if flow-policies violate the security policy. It focuses on identifying *by–passing* violations, see figure 2.6, to avoid a policy with a set of genuine rules.

If verifying the complete network behavior is the concern, ConfigChecker [ASMEAE09] aimed to verify both properties: reachability and secure configuration for the entire network. The network is represented by a state machine and its behavior, the end–to–end functionality is represented by a BDD. Each state denotes location and the packet header, and transitions are processes in the network. Another version of ConfigChecker [ASA11] used CTL to model network properties in a time line and BDDs to perform firewall verification with symbolic reachability analysis. FlowGuard [HHAZ14] is a framework to detect and resolve secure policy violations that checks flow and identifies policy violations after a network update, and proposed five strategies to solve them. FlowGuard creates and checks firewall authorization *spaces* which represent a series of packets that are disjoint, allowed or denied. In this sense, those spaces are equivalent and have the same functionality to header spaces described in HSA [KVM12].

[k]Yices is a SMT solver for satisfiability. `http://yices.csl.sri.com/`.

2.3.3 Issues and Limitations of Verification

Software verification over applications or protocols is a hard issue. In the case of SDN policies, the problem is not an easy one because the network state, albeit bounded, is too large. Monitoring every variable of the network state is inconvenient and unreal, and the changes in the network add complexity to the verification process. As was shown in § 2.1.1, the length of the network state depends on three aspects: the topology size, the set of flow–entries per datapath, and the granularity of each rule. Even when the topology is fixed, the scope in dynamic analysis is related on the packet–header length instead of the number of packets in the network. Although the header of each packet has a limited size, the explosion of variables is huge, and the number, type, and content of a control message adds more states to the problem formulation. This issue can be lessened with improved structures, solvers and algorithms, as those developed by Pătraşcu and Williams [PW10] which identified reductions that can lead into faster SAT algorithms. Additionally, the verification of larger topologies for dynamic checking has been shown to not properly scale to practical data–center networks. However, the model can be extended to larger networks, in some cases generalized, by network patterns and abstractions.

Reviewed tools and mechanisms are focused on verifying only a plane, the control or the forwarding plane, but they do not verify the specification, the control, neither confirm the implementation in the forwarding plane. We declare the complete process is verifiable at specification, control level, and datapath–level implementation. This thesis aims to reduce configuration errors by the human factor which causes more than 62% of network downtime [Ker04]. We think that programmable infrastructure, governed by *well–defined* policies, dramatically reduces configuration errors and contribute to a reliable network systems. To do this, next Section introduces policies for networking. There, we analyze several descriptions of specification, security and SDN policies and languages. Then, we discuss three steps to validate policies for SDN: topology, rule–sets, and finally, network policies.

2.4 Network Policies and Device Configurations to SDN

This Section describes the Policy–Base Network Management (PBM), languages for specification, strategies for validation, and its application to SDN. A policy specifies an abstraction, from the high–level like the natural language, to bitwise instructions at low–level. A policy maps a previous system–state, an action that produces a state–transition, and the resulting system–state. We focus on PBM paradigm because it allows to separate high–level directives from functionality at low–level implementation. A network policy is a set of *conditions*, *constraints*, and *settings* about how specific types of traffic must be managed in a network, which *users* are authorized to access to *network resources*, and the *circumstances* under which they can access.

The IETF [Ste+99] begun the standardization by a draft, best known as POLICY in the RFC3198 [Wes+01], which describes network policies and services, and establishes the policy terminology for PBM. The PBM system is composed of: a repository, a set of Policy Decision Points (PDP), and a set of Policy Enforcement Points (PEP). The repository stores the policies; it can be a Directory Enabled Network (DEN)[1] which is a specification that defines network entities and binds network services with clients. PEP is a network node or datapath, and PDP is a node that takes decisions, for example the centralized server on the SDN environment. Boutaba and Aib [BA07] presented a compact review of PBM frameworks, languages and tools.

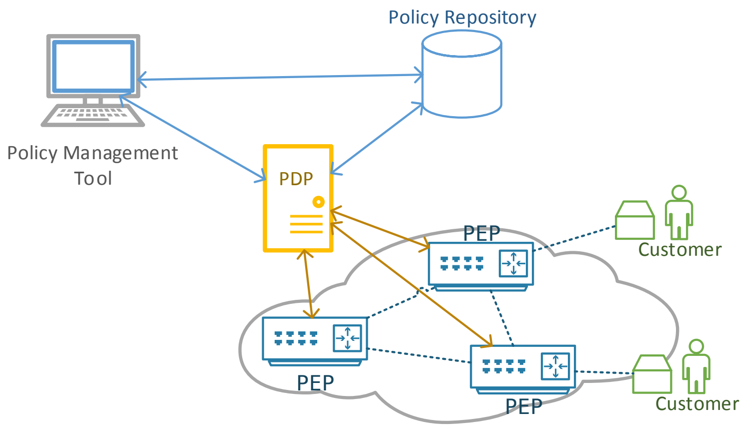


Figure 2.8: Policy-Based Management Architecture [Wes+01]

We use the set of definitions for action, condition, event, rule, and policy taken from [Str03]; and the terminology for Policy Based Network Management (PBNM) from the RFC3198, see table 2.4. Policies are defined by an administrator who specifies how network devices must handle traffic. An earlier policy system for networking was Inter-Organization Networks (ION) developed by Deborah Estrin in 1985. ION proposed to have discretionary and non–discretionary access–control based on the packet header, and implemented it in a gateway, responsible for making decisions of control, but a gateway cannot properly scale and hold all possible policies. The centralize access–control was implemented on a server that controls the end–hosts. Another framework was the Common Open Policy Service (COPS), by the RFC 2748 [Dur+00]; it is a protocol to exchange signaling defined between the policy server (PDP) and client (PEP) for IP networks. The policy decision point (PDP) is also a centralized controller that enforces policy statements and prioritizes traffic based on three classes: delay–tolerant, bulk data, or real–time traffic.

2.4.1 Policy Specification Language

Specification is the unmistakable way to represent a policy by an expressible language. The simplest manner to express a policy is the conditional–action, the AC structure. The structure is: ***if*** *condition* ***then*** *actions*. A language that uses this structure was proposed by Clark [Cla89]. It creates an Administrative Region (`AR`) which is a set of network devices and links, and defines the *term* that comprises source (`Hs`) and destination(`Hd`), user class (`UCI`), and global conditions(`Cg`). Source and destination specify host address and AR identification, and entry / exit ARs. The syntax and the example of the policy are shown in table 2.5.

[1]DEN is a specification in object–oriented model of network elements and services into a repository, independent of repository and access protocol, and a map of this biding information in LDAP or X.500 protocol.

Table 2.4: Policy definitions [Ste+99]

Policy	Set of rules to administer, manage, and control access to network state.
Rule	Map that relates a set of conditions to a set of actions. Those actions are enable once conditions are satisfied. The rule also contains information how actions are applied, the order that actions should be executed, and exceptions during execution.
Condition	Evaluation of a set of events and determines if a policy must be executed. A condition is the necessary state that defines if the policy-rule actions are executed; if this condition evaluates to TRUE, the rule should be enforced. Policy-clause is a set of policy-conditions.
Conflict	A policy conflict occurs when the conditions of two rules are satisfied, over the same object, but these actions contradict each other.
PDP	Policy Decision Point is a logical entity that makes decisions, for itself and other network elements.
Target	Target of a policy is the set of entities which are affected by a policy. e. g. The target of a policy that configures the network are the services running on the network.

Table 2.5: Example of policy language proposed by Clark [Cla89]. `Hs` is the source host address, `ARs` is the source AR, `ARent` is the entry AR, `UCI` is the User Class Id, and `Cg` are global conditions.

Syntax	`((Hs,ARs,ARent),(Hd,ARd,ARexit),UCI,Cg)`
Example	((a.b.c.d,1,-), (w.x.y.z,3,-), University, Unauthenticated UCI)

However, this policy representation did not describe low–level detail nor specify path structure. Low-level policy languages involve path specification and the implementation of traffic forwarding. At low–level, the policy abstracts the implementation on the device, it basically models packet forwarding and transformations.

Another policy structure is the tuple *event-condition-action* (ECA), or *event* ***generates*** *action* ***if*** *condition*, a declarative way common on database systems. This structure allows to consider the environment and interaction into the policy. An example of this structure is the Policy Description Language (PDL) [LBN99] which has a policy server able to provide centralized control over a soft switch. It was essential to detail our work on network policy validation for programmable networks. Table 2.6 shows the syntax of PDL, and an example. Here the event *~hour* means a sequence that ends with the event *hour*, *window5* event is triggered after the second occurrence of *hour* and 5 hours later of the first event *hour*.

Ponder [DDLS01] is a declarative, object–oriented and strong typed language to specify security policies which correspond to device implementations of access control for distributed systems. Ponder included *Obligation policies* which follow the ECA paradigm, and describe the set of mandatory actions when specific events occur and the set of conditions are satisfied. Also, Ponder used the concept of role, similar to RBAC, to facilitate the implementation in the industry. For Ponder, a *policy is a rule that defines a choice in the behavior of a system*,

Table 2.6: Syntax of PDL policy [LBN99]. Events are considered sequences as list, conjunction &, or disjunction |. Terms can be constants, functions or expressions like $e[k].m$ where k is the repetition, and m is an attribute of event e.

Syntax	¡policy¿ := ¡event¿ **triggers** pde **if** ¡condition¿ **causes** ¡actions¿ ¡events¿ := $[e_1, \ldots, e_n] \mid [e_1 \& \ldots \& e_n] \mid [e_1 \mid \ldots \mid e_n]$ ¡condition¿ := $p_1, \ldots, p_n$ where p_i is a predicate ¡predicate¿ := $t_1 [= \mid \neq \mid < \mid \leq \mid > \mid \geq] t_2$ ¡term¿ := ¡constant¿ — $f(t_1, \ldots, t_n)$ — $e[k].m$ ¡action¿ := $a(t_1, \ldots, t_n)$
Example	*hour, ~hour* **triggers** *window5(Start = hour[1].Time)* **if** *hour[3].Time - hour[1].Time = 5* **causes** *reroute(trunk1)*

in multiple domains. A domain is a group of objects to which policies apply and can be used to partition the objects in a large system according to logical or geographical boundaries, object type, responsibility and authority. Table A.1 shows the syntax and an example of Ponder language.

XACML [GM03] is a language for specifying network–security policies for access–control. It is an XML–based language that works by queries where the policy server responds to a request and allows or denies a set of successive actions; in some cases, it returns indeterminate or not applicable responses. A XACML policy contains: subjects, targets, resources and actions. Target is the condition to satisfy by the subject, and resources and actions, it is the policy output. Figure A.1 shows the basic syntax and a small example.This language is implemented as the IETF Policy Framework [Ste+99] with PDP, PEPs and policy repositories.

Another language for networking specification that come from databases, is called Network Datalog (NDlog) [Loo+06]. NDlog was a language for specification of networks that created foundations for recursive querying to specify routing protocols, QoS constraints, and path constraints. NDlog allows to write a specification as a set of rules and a query, and validates if rules, expressed as conjunction of predicates, can generate an output that satisfies the query. The query result is a set of paths that satisfies the constraint set. It is called a pathfinder language because it operates based on queries over a graph. Ethane was another approach that provides network-wide access-control by definition of policies and high-level principals [Cas+07]. With Ethane, administrator defines fine–grained policies which are declared over high–level names, include the path, and the network binds the packet and the source. Ethane created a language called *Pol-ETH* which followed the AC paradigm.

Our primary work is influenced by the Path-Based Policy Language (PPL) [SLX01] which is a language focused on paths and flows. It allows network administrators to specify *a*) the network topology, nodes and links; *b*) flow paths, *c*) network traffic, and *d*) policy rules. Also, there are PPL compilers [SLX01][Guv03] that *a*) verify the policies to detect some conflicts, and *b*) transform them into configuration scripts for the involved network elements.

However, these PPL compilers precalculate all possible paths, symbolically at least, before finding policy conflicts and verifying the validity of the configuration. This *path explosion* is an exhaustive one for the NP–complete search, and combines all possible node set and policy paths. Chapter 3 illustrates the adaptation of PPL to SDN-based topology validation. Moreover, it describes how to use the PPL language to specify network topologies, traffic and policy rules, and presents the notion of policy conflict.

2.4.2 Policy Analysis and Verification

The elements to be checked in a policy are: *a*) syntax, *b*) consistency between rules that compose a policy, *c*) the topology that implements the policy, and *d*) whether the policy is congruent with the implementation. Modal conflicts are related to syntax conflicts. Detecting this kind of conflicts is *easier* because they depend on the *grammatic form*. In general, testing policy output and behavior could be a more complex process. The reasons of unwanted behavior are difficult to identify, debug, or even diagnose. As seen in § 2.2, there are two methods to test the policy behavior. First, the network manager can test the implementation measuring the performance over the real network, sending testing traffic and identifying whether the result satisfies the policy goals. Second, the policy reasoning validates the expected behavior using logical tools, such as simulation, symbolic analysis or another logical instrument. The main objective is to determine a useful policy set (rules at low–level) that achieves the intention of a network policy in any of its dimensions like quality, security, or access control.

Detecting policy conflicts

Testing a new configuration before it is put into action is a common practice. Using the production infrastructure could be unsuitable, but having parallel infrastructure is expensive. *Shadow configuration* [AWY08] allowed to have a pair of configuration files, which are executed in the current infrastructure but isolated of production. Shadow configuration files of a set of routers can be tested without launching them into operation. The shadow configuration only carries testing packets while the production configuration delivers real traffic. However, this is a practical test that involves the complete deployment, and network administrators can only observe the measures and performance over testing traffic.

Chomicki et al. [CLN03] proposed a formal framework which used logic programming for conflict resolution on ECA rules. Rules, conflict detection and solutions are defined by axioms, and applied logic programs to identify constraint violations. This framework finds the system states that cannot execute specific actions by *monitors* which filter policies and cancel some actions to meet the constraints. Some conflicts occur because resources are insufficient and the assignation of individual capability cannot satisfy the complete set of policies. Corybantic [Mog+13] is a framework to design SDN controllers to avoid resource competition on networks with multiple controllers. It allows composition of controller modules and aims to optimize network–wide objectives. Corybantic decides which policy has priority over other to maximize a *global* benefit. In the background this is a multi–objective optimization problem that is transformed into a single–objective by normalizing the function output.

Statesman [Sun+14] is a service that administrates multiple network applications while maintaining network safety and invariants. It has three network states: observed, proposal and target. The observed is the actual network state, then applications present their proposal state, and finally Statesman decides the secure and safe target state. Statesman uses a dependency model to denote domain–specific dependencies between states. Finally, it evaluates the network invariants and accepts the network change. A controller can be seen as a complex system composed of various modules and each of them decides if it applies a specific policy. In synthesis, the controller solves an optimization problem to allocate resources which achieve the network–wide objectives. Athens [AuY+14] is a mechanism to solve the allocation problem by a democratic —voting— procedure. Each module has a policy evaluator based on the change of the network state before and after a decision is applied. Then, Athens compares the set of multiple decisions and their combinations, and selects the optimal one after each module evaluates their allocation options (precision), and the normalization of those metrics with other modules (parity).

Model checkers also have been used for verifying SDN policies. Kuai [MDTW14] is a distributed enumerative model–checker for SDN that takes as input the implementation of the controller, in a command language similar to Murphi [Dil96]; a network topology, a finite set of switches and links; and the safety property. Then, it uses a finite–state model checker, as seen in § 2.2.2, and applies a partial order reduction technique to reduce the state space. Then, Kuai performs automatic abstraction to identify unbounded packets by its *counter abstraction* which records the instances of a packet flow. Kuai distributes the solver over a cluster to execute the solver and has an interface with the POX controller. NetPlumber [KCZ13] verifies network policies on real time based on Header Space Analysis (HSA) [KVM12]. It creates the *plumbing graph* which consist of all possible paths for flows. Nodes in this graph are rules in the network, and edges denote dependency between rules. A path shows the connection —or dependency— of diverse rules. The graph can establish *filters* between two nodes, and deduct the packet header that intersects both rules.

2.4.3 Network Security and Firewall policies

Security policies are focused on definition of rules for access control and bind users and conditions to resource access. A security policy can be a formal representation of its entities and functions. At low level, security mechanisms implement the controls described by the policy within a context specified in the model. ACL is an early expression of security policies, which can be described as matrices of resources and attributes. For example, the Bell–LaPadula model defines policies as a state machine and transitions between states are the policy outline. On the other hand, Role-based Access Control (RBAC) model specifies resource access based on the roles within the company instead of independent users.

Listing 2.1: Syntax of firewall policy advisor [ASH03].

```
<order> <prot> <src_ip> <src_prt> <dst_ip> <dst_prt> <action>
```

A security policy is a set of rules, written with the syntax shown in listing 2.2, which incorporate order, protocol, source ip and port, destination ip and port, and action. The action is *accept* or *deny*, and order is a priority mechanism, where the first rules have precedence over the following. The example shown in listing 2.1 is a set of rules that are congruent to a security policy.

Listing 2.2: Example of firewall rules [ASH03].

```
1  tcp 140.192.37.20  any *.*.*.*        80  deny
2  tcp 140.192.37.*   any *.*.*.*        80  accept
3  tcp *.*.*.*        any 161.120.33.40  80  accept
4  tcp 140.192.37.*   any 161.120.33.40  80  deny
5  tcp 140.192.37.30  any *.*.*.*        21  deny
6  tcp 140.192.37.*   any *.*.*.*        21  accept
7  tcp 140.192.37.*   any 161.120.33.40  21  accept
8  tcp *.*.*.*        any *.*.*.*        any deny
9  udp 140.192.37.*   any 161.120.33.40  53  accept
10 udp *.*.*.*        any 161.120.33.40  53  accept
11 udp 140.192.37.*   any 161.120.35.*   any accept
12 udp *.*.*.*        any *.*.*.*        any deny
```

According to Al-Shaer and Hamed [ASH03], there are five kinds of conflicts or anomalies: shadowing, correlation, generalization, redundancy, and irrelevance. *Shadowing* is the rule that is never reached because another rule with higher relevance is applied first but their actions are different. *Correlation* occurs when two rules have conflicting packet filters. *Generalization* arises when a high-priority rule filter is a subset of another, it is a kind of shadowing but with the same action. *Redundancy* are two rules whose result is practically the same over the same rule filter. And *irrelevance* occurs if the elimination of a rule does not change the whole policy effect.

Firewall Policy Advisor [ASH03] is a tool to analyze firewall rule relationships and discover policy anomalies by a state–diagram algorithm. Also, it has a policy editor to create, insert, modify, and delete free–of–conflict rules. Besides finding conflicts on one firewall configuration, other algorithms were developed to analyze intra-firewalls [ASH04a]. These conflicts were defined as a distributed system where previous conflicts were redefined as an ordered sequence of multiple rule sets over a flow path. Inter–firewall anomalies are shadowing, spuriousness, redundancy, and correlation. Shadowing and redundancy maintain the same concepts for intra–firewall analysis. *Spuriousness* occurs if a firewall allows traffic that later another firewall rejects. *Correlation* is the mismatch of rules in predecessor and successor set of rules —upstream and downstream firewalls.

2.4.4 Policies for Software-Defined Networking

SDN is built under policy abstraction which describes the high–level behavior is disengaged from the data plane. For example, the policies *VoIP calls cannot be terminated in mobile devices*, and *access to database server is only possible from IT department desktop computers* describe the high–level behavior, not the detailed implementation. Data plane has a set of rules in the datapath with header and port abstractions. For example: *if (packet_header matches* `tcp 140.192.37.20 161.120.33.40 80`*) then send to port 4* identifies a set of packets and forwards them through an interface. Then, a low–policy is a set of atomic operations or primitives over packet–header attributes and performed by switches. A policy is equivalent to a set of pairs $\langle key, primitive\rangle$ where the key can match with the packet header and primitive is the action: `SEND`, `DROP`, `SEND_TO_CONTROLLER`, or `OVERWRITE`. Hence, this set of primitives conforms a policy implemented as a map into the forwarding table. Moreover, controller can specify QoS and the datapath can control the flow rate, the queue management and preferences.

One of the first policy languages for SDN was the *Flow–based Security Language* (FSL) [Hin+09b] for wireless networks and designed for NOX [Gud+08]. With FSL it is possible to specify policies of access controls, isolation, and communication paths. FSL focused on end–to–end reachability and path selection, without specific thought to network monitoring. Its syntax is shown on table A.3. A flow is specified as a tuple of: source and destination users (Us,Ut), hosts (Hs,Ht), access points (As,At), protocol (Prot), and the communication direction is a request or response (Req). Since FSL was designed for SDN, the policy in FSL is mapped to a *rule* in NOX; it means a matching of packet–headers. The evolution of FSL is the *Flow-based Management Language* (FML) [Hin+09a]; it is a high–level declarative language able to express common configurations and network–wide polices into a single framework. FML was designed to normalize policy configurations of diverse devices such as ACLs, VLANs, NATs, or routing. Software languages such as FML and Merlin [Sou+13] allow network administrators to define these policies in a declarative way. Listings in table A.3 show the syntax and an example of a policy that *denies all flows from unknown users*. OpenSAFE [BRA10] is an administration framework that monitors traffic to collect statistics, detect anomalies, or obtain forensic evidence. The monitor uses *span ports* to duplicate traffic at a point of observation, and forward that traffic to a collector, generally an IDS or a *sink*. OpenSAFE expresses routing policies for monitoring using the ALARMS language which describes a path as inputs, selections, filters and sinks. ALARM syntax and policy example are shown in table A.4.

Another fundamental problem on SDN is the deployment of rule–flows on datapaths. The solution identifies the part of the policy to be implement on a flow–table of a given datapath and install the segment of rules that accomplish the general goal. This automatic transformation maps policies to rules over multiple switches. After the network application produces a configuration, the configuration is deployed into forwarding-tables on switches and other datapaths. The newer policy on a few datapaths can generate inconsistencies with previous implementations. Those inconsistencies during updating deployment can generate failures such as hiccups, loss of connection, or forwarding loops.

Reitblatt et al. [Rei+12; RFRW11] proposed a set of update operations to guarantee that a single policy —the older or the newer— applies over a packet. They created the abstraction for packets and flows, and designed mechanisms to maintain the consistency. A formal model of OpenFlow network validates the update consistency and checks the network correctness. Later, Kang et al. [KRRW12] proposed an axiom set for policy transformations. A policy and a topology are the inputs, then the policy is described as simple forwarding rules and are implemented in a *chain of switches*. The transformations must hold along the switch path, consider the consistency of flow-rules, avoid redundancies, and guarantee the implementation of the policy.

Other authors suggested a third layer over the controller. For example, Yageneh [YTG13] proposed a third plane called *behavioral abstraction*. SIMPLE [Qaz+13] is a policy–enforcement layer for middleboxes with higher–layer functionality like firewalls, IDSs or proxies. SIMPLE expressed traffic as dataflows and allowed the integration with legacy middleboxes. The traffic is the result of function composition, and the management is presented an Integer Linear Problem (ILP) formulation with datapath constraints, sequences of datapaths, and load balancing. An example and the syntax for the SIMPLE language is shown on table A.5.

Procera [VKF12] is a functional reactive language that processes high–level policies based on reactive programming which interacts with the environment. With this language embedded as domain–specific in Haskell it is possible to declare and compose network policies in reactive and temporal manners that includes time, usage, authentication status, and traffic flow. Although performance and scalability of Procera is not clear, it is a valuable achievement for developing a language able to express powerful policies. Procera allowed to respond to continuous changes of the network, and this programming reacts to low–level events [KF13]. *Flowexp* is a language that symbolizes the set of conditions on the path and header flows with regular expression. Listing A.1 shows the grammar for Flowexp. Invariant checking is accomplished by *probe nodes* which have filters and tests for those filters. Probe nodes display an alarm if none of the probes satisfy the test expression.

Resonance and Pedigree [NRFC09; Fea+10] were two approaches that suggested the separation of network control and forwarding by a high–level policy definition for the control of access and traffic flow. Lithium [Kim+12] is an event-driven network controller based on NOX [Gud+08] that defines policies in terms of events in four domains: time, users, history and traffic. Then, the network dynamic generates those *action conditions* and trigger predefined reconfigurations. Nevertheless, this approach enriched the expressiveness of a policy but added more variables for verification. One of the most representative languages based on NOX is Frenetic [Fos+11]. With it, the network administrator composes policies and the compiler translates policies into stream queries and transformations.

2.5 Summary

This chapter presents fundamental concepts and related works to this research thesis. Concepts introduced in this chapter formulate the Software–Defined Networking context, its architecture, network–operating systems and frameworks to develop network applications. We introduce other fundamental concepts related to verification strategies, logical systems, and the keystone of this thesis: the *satisfiability problem*, which allows us to model networks and rules as predicates that later will be solved to validate network–wide policies. Then, we present related works on the issue of verification of network functionality, and finally, we describe the Policy–Based Management architecture and show related work on policy specification, languages, and validation to SDN.

Now we present a summary taxonomy which clarifies the problem associated with validation of policies on SDN networks, other approaches and its more relevant characteristics. We divide our research project into three steps: network invariant analysis, rules/policies for security, and general policy validation.

For the network invariant analysis, we propose a taxonomy that shows multiple solutions and their techniques to verify network invariants, shown in table 2.7. VeryFlow [Khu+13] is a layer between the SDN controller and datapaths that checks network invariant violations in real time that a rule is updated (insertion, modification and deletion). FlowChecker [ASAH10] proposed a verification mechanism to find misconfigurations on OpenFlow flow–tables. It works as an independent application and uses Binary Decision Diagram (BDD) (see BDD in Section 2.2.3) and a computation tree logic language (see CTL in Section 2.2.1) to write queries or properties that the administrator requires verifying. Anteater [Mai+11] (2.3.1) was the start point for this thesis, because it used static analysis of invariants in the data plane, it represented forwarding information as Boolean functions, and it built multiple instances of Boolean satisfiability problems from symbolic packets, network constraints and invariants. NICE [Can+12] also applied model–check techniques to explore the complete space and used symbolic execution to examine all possible paths, packets and network events. However, its path search creates an extensive number of variables which add complexity on large scale applications. Other solutions, as the proposed by Kang et al. [KRRW12], present an axiom set for policy transformations. The verification consists of checking the consistency of flow rules, avoid redundancies, and guarantee the policy implementation in datapaths. Kuai [MDTW14] is a distributed enumerative model–checker for SDN that uses the controller implementation, the topology specification, and a security property in Murphi language. Then, the model of switches, firewalls and routers are verified on a cluster, and the result of bugs or inconsistencies is reported. SymNet is a tool that uses Symbolic Execution to check network properties by tracking possible values in a header field as the packet goes through the network. SymNet also modeled middleboxes as transformation constraints over header fields. Recently, Atomic Predicates [YL16] used predicates, as we did on this thesis, to check network properties and showed time and space efficiency.

Table 2.7 summarizes the related work on network–invariants. First, we identify the plane where other solutions are applied. Data plane is verified by [Mai+11], and [Kan+13]. The control plane is analyzed by [ASAH10], [Kaz+13], [Khu+13], [SPNR13] and [MDTW14]. And both are studied by [Can+12] and our solution [MLCD14]. The next criterion is the formal logic used on the solution. We classify the solution using model–checkers which are applied by [ASAH10], [Can+12], [Kan+13], and [MDTW14] as opposed to [SPNR13] who created a live execution with symbols to verify the model. The flexibility and universality of predicates is a natural descriptor of network invariants; however, the predicate representation is used only for documentation in [Kan+13] but is the main part of the descriptor language in [ASAH10],

Table 2.7: Comparison of verification techniques for network invariants.

		Logic								**Network Properties**								
Reference	**Control / Data**	**Model–Checking**	**Symbolic Execution**	**Predicates**	**Algebra**	**BDD**	**SAT**	**Graph**	**Temporal logic**	**Reachability**	**No–Loops**	**Black Holes**	**Waypoint**	**No Delay / Direct**	**Avoidance**	**Slicing**	**Consistency**	**Tools**
FlowChecker [ASAH10]	C	•		•					•	•							•	BuDDy
Anteater [Mai+11]	D	•					•			•	•	•					•	Boolector, PicoSAT, PrecoSAT
NICE [Can+12]	C/D	•	•								•	•	•	•				STP constraint solver
Formal modeling & verification[Kan+13]	D	•			•					•	•		•					ACSR, VERSA, UPPAAL-TA
NetPlumber HSA [Kaz+13]	C							•		•	•	•	•			•		Python, Hassel
Veriflow [Khu+13]	C							•		•	•						•	
SymNet [SPNR13]	C		•					•		•	•						•	Haskell, Scala, Antlr
Network conflict detection [MLCD14]	C/D			•	•		•			•	•	•	•		•			Alloy
Kuai [MDTW14]	C	•							•	•	•							Murphi, PReach
Atomic Predicates [YL16]				•		•				•	•	•	•			•		

[YL16], and our work [MLCD14]. Regarding checking invariants, reachability and loop–free are essential properties which are verified for almost all the related works; blackholes and waypoints are invariants focused on routing consistency. Avoidance invariant, as complement of waypoint, is only considered in our work [MLCD14] and compared with Anteater which used a similar structure, we checked more invariants and operate in both planes. In summary, with the Network Conflict Detector [MLCD14] we can check multiples invariants in both planes using predicates and relational algebra in the control and data planes. This work and its results are presented in chapter 3.

Table 2.8: Comparison of verification techniques for security properties.

	Logic				Firewall					Policy				
Reference	**Model Checking**	**Symbolic**	**xDD**	**SAT/SMT**	**Shadowing**	**Redundancy**	**Correlation**	**Generalization**	**Spuriusness**	**Non–Bypass**	**Consistency**	**Compactness**	**Equivalence**	**Tools**
Firewall Policy Advisor [ASH04b]			•		•	•	•	•	•					
Fireman [Yua+06]		•	•		•	•	•	•	•		•			BuDDy
Firewall Decision Diagram [GL07]			•								•	•		
Formal Verification [Liu08]		•												
FW configuration model [JS09]	•			•							•			Minisat, BuDDy
ConfigChecker [ASA11]	•	•	•		•									
Verification and Synthesis [ZMMN12]				•		•						•	•	Minisat, DBepQBF
FLOVER [Son+13]	•		•	•						•	•			Yices
Flowguard [HHAZ14]			•		•	•	•	•						
FireWell [MLCD15]				•	•	•	•	•	•	•				Alloy

The second contribution of this thesis is related to identification of conflicting rules in firewall, and security policy violations. The first work included in our analysis is the *Firewall Policy Advisor* [ASH04b], which used a BDDs for discovering policy anomalies. Fireman [Yua+06] used static analysis to verify firewall policies by a symbolic model checking for all packets along all possible paths valid into a set of configurations. Later, the FDD [GL07] represented a security policy in a directed and non–cycled graph, where edges represent constraints and final nodes denote firewall decisions. Liu [Liu08] wrote formal verification algorithms which also used FDDs to check misconfiguration, and validate firewall policies, but this time against the formal security policy. The policy included rule consistency in firewalls and the compactness of the set of rules. Jeffrey and Samak [JS09] modeled firewalls and verified policy consistency, including *cyclicity*, when a firewall enters in a loop or a rule is never *reached*. They compared the efficiency of BDDs structures in contrast to SAT solvers, and found that SAT outperforms BDDs. ConfigChecker [ASA11] used BDDs to verify firewall configuration by symbolic reachability analysis. Zhang et al. [ZMMN12] implemented SAT to create equivalences and discover redundant rules between two firewalls, and they used a QBF solver to optimize the ACL set of rules. FLOVER [Son+13] used a model–checker based on Yices to find rules which violate the security policy, especially *by–passing* violations to avoid a policy with a set of genuine rules. FlowGuard [HHAZ14] detects and solves secure policy violations by checking flows after a network update, and proposed five strategies to solve them. FlowGuard creates and checks firewall authorization *spaces* which represent series of packets that are disjoint, allowed or denied. In this sense, those spaces are equivalent and have the same functionality to header spaces described in HSA [KVM12]. Our work, called FireWell [MLCD15], creates a model from a set of firewall rules and translates it into a SAT model. Then the model is processed by a SAT–solver to find inconsistencies. Finally, the model is tested for a set of policy constraints that includes the by–passing violation. Some of this related work did not verify against a high–level policy, only checked if there are inconsistencies in the set of rules. We use the model of policies and check it against the model or firewall rules to find inconsistencies.

In the next chapter, we present the model and verification of topology invariants based on paths. Later, in chapter 5 we show policy verification for firewall rules, and in chapter 6 we present a more general verification that models three elements: i) a network policy, ii) a network topology, and , and iii) the set of configurations which are essential to identify congruence and policy enforcement.

Chapter 3

Topology and Path Verification

Testing can only prove the presence of bugs, not their absence.

Edsger W. Dijkstra

This chapter presents the process of verification that validates network invariants and policies according to their path definition. Formal definitions of topology, path, flow, and policy are used to model network policies which later will be analyzed through a model finder. This mechanism allows us to check the SDN implementation of policies on nodes and paths, instead of developing external implementations to the specification language [Mai+11]. Moreover, we present a compiler for Path–Based Policy Language (PPL) [MLCD14] based on Alloy that finds policy–topology conflicts.

Policy–based Network Management (PBNM) aims to reduce complexity and errors in network configuration. There are two approaches for using PBNM to validate network configurations. One detects conflicts, inconsistencies and bugs in *each* device configuration. It is called the *post–configuration* checking. Other focus on avoiding errors by automatic generation of configurations. The automatic configuration is created according to the specification of the *expected* behavior of a network. The result of the pre–configuration checking is, for example, a set of configuration scripts [Ste+99].

Topology checking is one of the essential analysis in networking. It helps us to check if at topology level the SDN application is consistent without resorting to higher layers. Essential network capabilities, called *network invariants*, respond to fundamental operations over the graph. For example, vertex reachability resolves packet loss and forwarding inconsistency; verification over paths identifies and avoids loops and black–holes; and verification of path–policy checks if a path goes through a specific node or avoids it.

3.1 Abstractions for Topologies, Flows and Policies

This section shows the formalism used as basis for checking network invariants. For the purpose of this thesis, four invariants are considered: *1*) node isolation, *2*) reachability, *3)* loop–free, and *4)* node avoidance and waypoints paths. Assuring these properties, could avoid severe networking errors such as disconnection, service oscillation, black–holes, false failure advertisement, or availability and security vulnerabilities.

3.1.1 Network topology

We use regular graph theory definitions to describe the policy language for computer networks [Lew09; Die10].

Definition 3.1 (Network)**.** *A network or graph is a duple $G = (N, L)$ such that: N is the set of nodes (vertices) in the network, and $L \subseteq N \times N$ is the set of links (edges) between nodes in the network.*

The node set of a graph is denoted as $N(G)$, and the link set as $L(G)$. The set of all links in L with the node n is denoted by $L(n) = \{(x, y)|(n, m) \cup (m, n) \in L\}$, links that contain the node n. Two nodes m, n of G are *adjacent* if (m, n) or (n, m) are links of G. The set neighbor(n) is the set of nodes connected to n such that $\{m|(n, m) \cup (m, n) \in L\}$. We sometimes write $n \rightarrow m$ instead of (n, m). The number of vertices of a graph $|G|$ is its *order*, and the number of links is denoted as $||G||$. The *degree* $d_G(n)$ of a node n is the number $|L(n)|$ of links at n, this is equal to the number of neighbors of n.

A well-formed network G satisfies the following properties:

i) connected graph: $\forall n \in N|\,\text{links}(n) \neq \emptyset$, i.e. there are not nodes without links to other nodes;

ii) self–loop: $\nexists n \in N|(n, n) \in L$, i.e. there are not links from a node to itself.

These constraints are later verified as part of network invariants in § 3.4.1.

Definition 3.2 (Path)**.** *A path is a graph $P = (N, L)$. Where the set $N = \{n_0, n_1, \ldots, n_k\}$ is a sequence of nodes; L is the set of links $L = \{(n_0, n_1), (n_1, n_2), \ldots, (n_{k-1}, n_k)\}$ that connects the sequence from n_0 to n_k .*

A network path can be described as a list of nodes that maintains a sequence $P = n_0 n_1 \ldots n_k$,, where n_0 is the source node and n_k is the target node. The number of links of a path is called its *length*, and a path of length k is denoted as P^k.

3.1.2 Transitive Closure and Reachability

Given a path P, its link set $L = \{(n_0, n_1), (n_1, n_2), \cdots, (n_{k-1}, n_k)\}$, and link a binary relation on the set of nodes N. If $a, b \in N$ and $(a, b) \in L$ then $(a, b) \in$ link. The *transitive closure* link$^+$ is the transitive relation on the set N such that link $\subset$ link$^+$, and link$^+$ is minimal. In other words, if $a, b, c \in N|(a, b), (b, c) \in$ link then $(a, c) \in$ link$^+$. Thus, the binary relation link$^+$ contains all possible pair of end–points of subpaths from $P = (N, L)$.

Given n a node of the graph, links$(n) = \{m|(n, m) \in \text{link}^+\}$ denotes the set of end–points (paths and subpaths) from n. Therefore, the following properties hold:

i) node n_j is *reachable* from n_i if and only if $n_j \in \text{links}(n_i)$, i.e. there is a path from n_i to n_j;

ii) a node n_i is not in the path if $\text{links}(n_i)$, i.e. the path does not have loops;

iii) all nodes in the path are also nodes in the topology, $\forall n_i : n_i \in \text{links}(n_i)$, and $n_i \in N(G)$.

A path $P = n_0, \ldots, n_{k-1}$ forms a *cycle* if the link (n_k, n_0) is added. Therefore $C = P + n_k n_0$ is a cycle.

Definition 3.3 (PDP). *A partially defined path p' is an incomplete definition of path. It could even be a subset of links, intermediate, sources or destination nodes.*

Definition 3.4 (P^+). *Given p' a PDP, the function $P^+(p')$ extracts the set of paths that match a partially defined path p'.*

Given $\mathcal{P}$ the set of all paths in G, a PDP $p' = (n_i, n_j)$ the superset $P^+(p')$ is the set of paths that have the link (n_i, n_j), i.e. fulfills the partial definition. $P^+(p') = \{p \in \mathcal{P} : (n_i, n_j) \in L_P \wedge n_i, n_j \in N_P\}$. This notion completely generalizes network and path definitions in the base case proposed by Stone [SLX01].

3.1.3 Categories and Sequences

The administrator needs to define patterns to classify network traffic. For this work we use *categories* and *sequences* to group traffic characteristics. *Categories* act as traffic labels. For example: traffic can be categorized as *interactive*, *streaming* or *bulk data*. Basic set operations apply to categories and their elements. We define *sequences* to model ordered sets and their elements with a precedence function and asymmetric relation. With these ordered structures, the network administrator can operate policies and describe actions where a traffic value is *lower* than a constraint.

We also define *categories* as set of elements. In our model, elements from the same category are disjoint, for all $C, C' | C \cap C' = \emptyset$. However, we can declare *super categories* or sets of categories which can be aggregated to achieve more complex structures. For example, the network administrator of a campus requires classifying users and profiles; namely: faculty, staff, and students. The label for this category *usergroup*. See figure 3.1 as an example of traffic classification and the model for a network policy.

A *sequence* arranges elements under the ordering function, most of the time the precedence relation ($\prec$). Given a sequence T, as an ordered set, such that two elements $t, t' \in T$ hold an ordering relation $t \prec t'$. The *zero* element is part of the ordered sequence such that $\forall t \in T, 0 \prec t$. We also use the functions higher and lower; $\forall t, t' \in T$, $\mathsf{higher}(t, t') = true \iff t' \prec t$. We model sequences of elements to provide ordering and preference abstraction. Thereby, we can compare and define customized ordering functions. For instance, priority on traffic can be defined using *low*, *medium* and *high*, and solve the preference function $\mathsf{higher}(a.priority, b.priority)$.

3.1.4 Traffic flow

The keystone of Policy–Based Network Management is the *traffic flow* concept. A traffic flow models an end-to-end connection in a data network. We define characteristics of data which are carried through the network topology and define metrics to apply to the flow.

Definition 3.5 (Traffic Flow)**.** *A flow f is a tuple (P^+, k, t) such that:*

i) $P^+(PPD)$ is the set of paths that match a partially–defined path PPD,

ii) $k \subset K$ is a traffic category,

iii) $t \subset T$ is a sequence.

As seen in the previous definition, a flow uses categories and sequences to represent the traffic classification by the administrator. The network administrator writes sets to classify traffic: T for sequences and K for categories. As a result, sequences, categories, paths, and super set of paths, are descriptors of the traffic flow.

So far, a network administrator can detail the topology: nodes, links, and paths. Then she specifies the traffic using sequences and categories; thereby, she may model network characteristics and constrains such as delay, jitter or bandwidth. For example, a network administrator specifies a traffic flow for video transmission, which uses the UDP protocol. Figure 3.1 shows the model of this flow specification. Note that a flow specification may include specific IP or ranges of addresses to define origin and destination hosts or subnetworks. Moreover, profiles, permissions, users, groups or subnetworks and IP–address ranges may be modeled using categories. Table 3.1 presents an example of flow structure. This model comprises topology, traffic flow, and categories and sequences for the management. See § 5.3.3 for IP addresses and other partial order sets, to realize how network addresses are represented as categories.

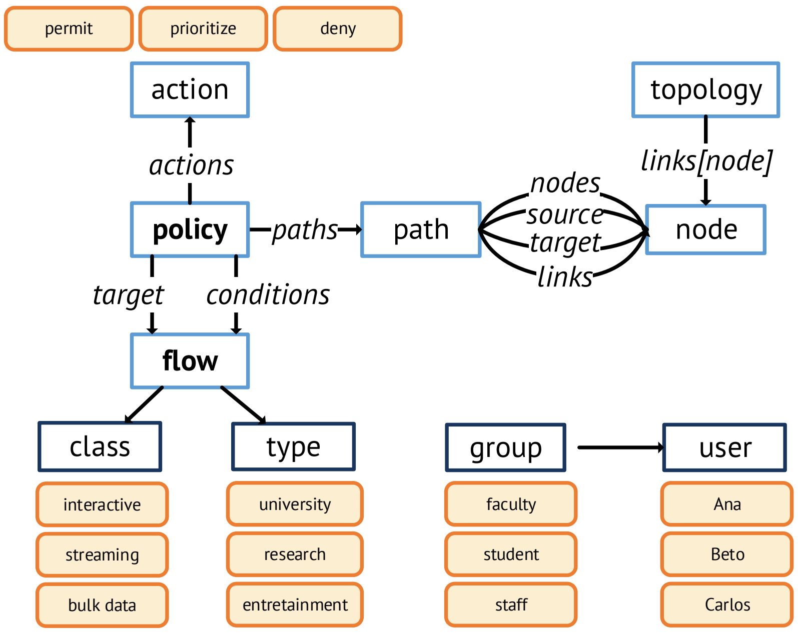


Figure 3.1: Model of network language for policy and flow.

Table 3.1: Example of topology abstraction and traffic flow. Management is an extra category for the flow example. Language is flexible enough to add more categories and sequences as necessary.

Topology	Nodes	Alpha, Bravo, Charlie, Delta, Echo
	Links	Alpha → Bravo, Alpha → Delta, Bravo → Charlie, Charlie → Delta, Bravo → Echo, Delta → Echo, ···
	Paths	⟨ *,Charlie,* ⟩ Origin: Alpha Destination: Echo
Flow	Protocol Origin Destination Class Type Delay	UDP 192.168.5.* 192.168.7.* Video Research 50ms
Management	Profile User Priority	faculty Professor X Low

3.1.5 Network policy

Policy is the relation map between conditions, paths and traffic flows, and the resulting action. The topology abstraction represents the low–level network in terms of nodes, links, and paths. At high–level, the network administrator establishes categories and sequences for traffic flows. Furthermore, she classifies the flows by profile, user, service or permission. Finally, the traffic flow characteristics are included into the model. Figure 3.1 shows the complete model and the relations within the elements to compose the policy.

Definition 3.6 (Policy)**.** *A network policy π is a tuple $\pi = (P^+, f, C, \alpha)$ such that:*

i) P^+ is a super set of paths from a PDP,

ii) f is the target flow composed of customized categories and sequences,

iii) C is a set of conditions over the flow f, and

iv) a is an action, commonly from $A = \{permit, deny\}$, that resolves whether allow the flow f over the set p.

Action a is decisive and produces a network configuration that allows or denies the traffic flow. Assume the topology comprises five nodes and six links as shown in figure 3.2. For example, the network administrator wants to apply the policy:

User Ana with profile IT member, who is in the subnetwork Alpha (192.168.5.), is allowed to access the video database in subnetwork Echo (192.168.7.*), and it must go through the router Charlie, exclusively from Monday to Friday.*

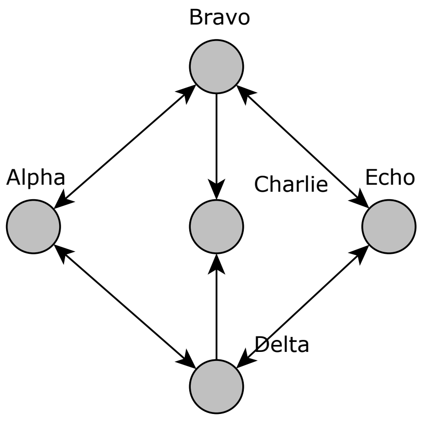


Figure 3.2: Topology example to validate network invariants.

Now the administrator details the policy. She defines the path, constraints, categories, sequences and actions. To do that she follows the steps:

1. path `A_E` := `<Alpha,*,Charlie,*,Echo>`, and the super set P^{+}(`A_E`) is calculated;
2. the traffic flow: `traffic_class` = `video`, `protocol` = `UDP`;
3. the conditions `Mon ≤ day ≤ Fri, user = Ana`, and `Ana ∈ IT member`;
4. finally the policy decision `permit`

Implementation function

Now that the topology, the flow and the policy is specified, an implementation function is needed. This function maps the set of possible paths, denoted by ω, which can implement the policy π. Another interpretation for the implementation function ω is: it maps a set of constraints into a configuration script. The definition of this function is described in more detail in § 3.1.6. For now, the implementation function generates a configuration from the policy, and the policy is represented as the conjunction of constraints 3.1.

$$\begin{aligned}\omega \;=\; & \mathit{impl}(p : \mathit{path} \in p^{+}(\texttt{a\_e}) \land \\ & \texttt{traffic\_class} == \texttt{video} \land \\ & \texttt{protocol} = \texttt{UDP} \land \\ & \texttt{day} \geq \texttt{Mon} \land \texttt{day} \leq \texttt{Fri} \land \\ & \texttt{Ana} : \mathit{user} \in \texttt{IT member})\end{aligned} \tag{3.1}$$

Note that IP addresses, user groups, traffic class and protocols should be modeled as *categories* and use set operators. On the other hand, the `day` type should be modeled as a *sequence* to compare them using $\leq$ and $\geq$ operators. As a consequence, this representation of a policy allows us to logically solve this *predicate* and prove a formal solution using a model finder [MLCD14].

3.1.6 Policy conflicts and semantics

To define the set of paths that implements a policy and then identify policy conflicts and violations we follow the guidelines of Harel and Rumpe [HR04] to specify a modeling language L describing the *syntactic domain* $\mathcal{L}_L$, the *semantic domain* $\mathcal{S}_L$, and the *semantic function* $\mathcal{M}_L : \mathcal{L}_L \to \mathcal{S}_L$, also traditionally written $[\![\cdot]\!]_L$.

Definition 3.7 (Policy Semantic)**.** *The semantic of a policy* $[\![\pi, G]\!]$ *is the set of paths that implements the flow over a path on* G *and achieves the policy.*

$[\![\Pi]\!] = \Omega$ is the semantic of all network policies and produces the set of all paths implemented on the network. The complete network configuration is denoted by $\boldsymbol{\Omega}$, and following:

i) The semantic function $[\![\pi, G]\!]$ is the set of tuples (*path*, *traffic*, *conditions*) in the network G that satisfies the policy π.

ii) A policy π is valid in a network G, if $[\![\pi, G]\!]$ is not empty

The semantic of a policy is equivalent to the implementation function described above. The implementation describes the set of actions a node (datapath) should execute to comply with the policy.

Definition 3.8 (Policy Conflict)**.** *A policy conflict occurs when a set of policies are not implemented by any path.*

- $[\![\Pi]\!] = \emptyset$ i.e. when $\exists \pi \in \Pi |\text{implements}(f)| = 0$.
- Given two valid policies π_1 and π_2, they are not conflicting in the network G if $[\![\pi_1 \cup \pi_2, G]\!]$ is not empty. That is, two policies are not conflicting if there is a set of tuples (*path*, *traffic*, *priority*) in the network G that satisfies both policies.
- In contrast, two valid policies π_1 and π_2 are conflicting in G if $|[\![\pi_1 \cup \pi_2, G]\!]| = 0$

Definition 3.9 (Minimal diagnosis)**.** *Given a set of policies* $\pi \subseteq \Pi$ *such that* $[\![\Pi \setminus \pi]\!] \neq \emptyset$*, the minimal set* π *is the minimal diagnosis.*

The next step is to create a *verifier* able to calculate the semantic function $[\![\pi, G]\!]$ and verify if that set is empty and the minimal diagnosis of that set. Next section describes a policy language used in our research project. Then, we use a model–finder to calculate the semantic function that satisfies the policy set. If the model–finder does not find any instance for the model (the set is empty), the policy set is considered invalid or conflicting.

3.2 The Path–Based Policy Language

Since a more formal characterization of topologies, flows, policies and semantics, this section describes a language focused on network paths and data flows called PPL[SLX01]. Path–Based Policy Language (PPL) is a language for Policy-based network management. With PPL, a network administrator can specify the set of rules about how the network must deal with specific categories and sequences of traffic. With it, network administrators are able to specify *a*) the network topology, nodes and links; *b*) flow paths, *c*) network traffic, and *d*) policy rules. Listing 3.1 shows the syntax of policy in PPL language. `Class` is a traffic classification that uses conditional operators. Instead, `Type` is a traffic classification that uses inequality relations to compare elements over an ordered set.

Listing 3.1: Syntax of policy in PPL language [SLX01].

```
define node [node_name]* ;
define link link_name < node_name,node_name >;
define path path_name { [node_name, | *,] };
define class class_name { [class_list | class_element] };
define sequence seq_name { [class_list | class_element] };
<condition> := { [ <var> GTEQ|LSEQ|NTEQ|EQUAL <constant>]};
<target> := {[ <var> ==|!= <constant> ]};
<action> := DENY|[PRIORITY | PERMIT | HOPCOUNT := <constant>]
<policy> := <policyID> <User_ID> {<path_list>}
{<target_list} {<condition_list} {<action_list>};
```

The existing PPL [SLX01; Guv03] compiler performs some validations on the policy rules and detects conflicts such as contradicting rules for the same network segment. However, that compiler only detects conflicts using a subset of the language and cannot detect conflicts related to contradicting rules with security controls on users and groups.

3.2.1 Network topology

PPL provides a set of `define` statements that can be used to define network nodes and links. For instance, listing 3.2 is the PPL specification of the example network with five nodes shown in Figure 3.2.

Listing 3.2: Definition of topology example in PLL language.

```
define node Alpha, Bravo, Charlie, Delta, Echo;
define link Alpha_Bravo <Alpha, Bravo>;
define link Alpha_Delta <Alpha, Delta>;
define link Bravo_Charlie <Bravo, Charlie>;
define link Charlie_Delta <Charlie, Delta>;
define link Bravo_Echo <Bravo, Echo>;
define link Echo_Delta <Echo, Delta>;
```

3.2.2 Path specification

In addition to nodes and links, PPL supports the definition of paths in order to define rules over the traffic that goes through them. Network administrators can set a path specifying all the nodes and links in the path, or by using *wildcards*. For instance, a path between *Alpha* and *Echo* may be defined specifying all the nodes using the expression `<Alpha,Bravo,Echo>`. In addition, a set of paths can be defined using *placeholders* that can be replaced by any set of nodes. Following the example, the complete set of paths between *Alpha* and *Echo* can be defined as `<Alpha,*, Echo>`, i.e. denoting all the paths that start in one node and ends in the other.

3.2.3 Policy representation

PPL supports the definition of constraints over the traffic going through nodes, links or paths in a network. A PPL rule comprises: a unique identifier (`policyID`), the ID of the policy creator (`userID`), the set of paths affected by the policy (`paths`), the type of traffic that the rule affects (`target`), the environment that determines when the policy must be applied (`conditions`), and the description of intended behavior (`action_items`). For instance, a network administrator, called Smith, defines the policy *the video traffic cannot be transmitted through Charlie*, using the the PPL statement shown in listing 3.10.

Listing 3.3: Policy example denies video traffic.

```
policy1 smith {<*,Charlie,*>} {traffic_class=video} {*} {deny}
```

Policy conflicts

A policy conflict occurs when *two or more* policies, in some of its rules, cannot be satisfied at the same time. For instance, there is a conflict if another administrator, let say Neo, defines a policy *Video traffic can be transmitted from Alpha through Charlie to Echo* using the statement in listing 3.4.

Listing 3.4: Policy example permits video.

```
policy2 neo {<Alpha,*,Charlie,*,Echo>} {traffic_class=video}
        {*} {permit}
```

Considering the above mentioned policies, there is a *contradiction*, hence a policy conflict. Basically, the first policy denies the video traffic that goes through *Charlie*, and the second policy permits that traffic in the same node.

3.3 Modeling Policies and Reasoning about Conflicts

To overcome the limitations of the existing PPL compiler, we propose a compiler that uses Alloy to reason about the network policies and detect conflicts. This section introduces Alloy and describes how it is used to specify the network elements and policies, and how we detect conflicts. Additionally, we show how to translate PPL statements into an Alloy model, and use the Alloy Analyzer to determine conflicts. First, the syntax of Alloy is depicted on listing 3.5. A complete definition of grammar for Alloy is presented in appendix A. Then, a network (graph) is modeled in Alloy, following the definition created on § 3.1.1. Additionally, a relational model is used to describe traffic classification and paths § 3.1.3, and shows policy creating and conflict identification by its semantics § 3.1.6.

3.3.1 Alloy in a nutshell

This section introduces basic concepts about Alloy and the satisfiability problem. Also, it shows how the network is modeled, and how the network model and policies are written in Alloy language.

Alloy is a model–checker with a specific declarative language based on first–order and relational logic. It is used to describe a system, called a *model*, in terms of signatures and relationships; and constraints over the signatures with *facts* and evaluations over the model with *predicates*. Alloy is also used to find examples, called *instances*, or counterexamples of the model using a bounded exhaustive search. Internally, Alloy translates the model into a set of CNF expressions and evaluates it using a SAT solver.

A *solution* is an instance or counterexample found by the solver. It denotes an *instance* where the model is satisfiable, or points those restrictions which cannot accomplish the system definition as a counterexample.

Since Alloy uses relational language, each declaration is a set noted by `sig` *`name`* {*`relations`*} {*`conditions`*}. Moreover, Alloy allows data multiplicity such as `all` and `some` for universal and existential quantifiers, in addition to `no` for none, `lone` to define at most one, `one` for exactly one, and `set` for any number. Relations on Alloy are declared by using the colon (`:`) operator. The signature `A { f : B }` denotes that there is a relation $f : A \to B$. A brief language syntax is shown in listing 3.5 and a complete reference and examples are in appendix A.

Listing 3.5: Summary of Alloy's syntax.

```
sigDecl ::= [abstract] [mult] sig name,+ { decl,* }
decl ::= [disj] name,+ : [disj] expr
mult ::= lone | some | one
block ::= { expr* }
factDecl ::= fact [name] block
predDecl ::= pred [qualName] name [paraDecls] block
funDecl ::= fun [qualName] name [paraDecls] : expr
paraDecls ::= ( decl,* ) | [ decl,* ]
```

3.3.2 Network Topology Model

Network nodes and links can be modeled in Alloy using signatures and relations. Node set is declared using the reserved word `abstract` because concrete elements are defined later. We instantiate the set of nodes and links from the topology as shown in listing 3.6. Consider the network example depicted on figure 3.2 and the example shown in PPL definition [SLX01]. Note that the network topology is also declared as an abstract element with two relations: nodes and links. `nodes` is a relation that contains *some* elements of type *node*, and `links` is another relation (node → node) map. This kind of relation is useful because it helps to define network invariants in a straight way.

Listing 3.6: Definition of topology, nodes and links in Alloy.

```
abstract sig node{}

abstract sig topology{
  nodes : some node,
  links : node -> node
}

// declaration of instances for nodes and links
one sig Alpha, Bravo, Charlie, Delta, Echo
extends node{}
one sig topologyOne extends topology {}
{
  nodes = Alpha + Bravo + Charlie + Delta + Echo
  links = Alpha->Bravo + Alpha->Delta +...
}
```

3.3.3 Path Model

Network paths are also modeled in Alloy by signatures and predicates. In short, a path is a tuple of a set of nodes and links. Two of these nodes are the source and the target. In addition, we specify path restrictions: each path is acyclic, loops are not allowed, the target node must be reachable from the source, and all nodes in the path from the source must be included in the set of nodes. The network model used on this thesis exploits that property to represent the *reachability* on a graph; see § 3.1.2. Nevertheless, network policy languages can describe parts of a path, even sets of paths.

This model follows the definition that utilizes the transitive closure shown in § 3.1.1. Note that *links* represents a binary relation from node to node. Alloy denotes the transitive closure with the hat symbol (ˆ) over the relation. Line 7 in listing 3.7 shows the abstraction that forces the no–loops invariant, line 8 shows reachability for the source node, and line 9 guarantees that all nodes in the path are defined in the topology. A *predicate* is a reusable constraint [Jac06], and evaluates the conditions over the instances and the parameter. The following listing 3.7 shows the specification of the example. Note that paths and the predicate that evaluates the validity of the path `<Alpha,*,Charlie,*,Echo>`.

Listing 3.7: Signature and instance definitions of a path in Alloy.

```
abstract sig path {
    nodes : some node,
    links : node -> node,
    source : one nodes,
    target : one nodes
} {
    no n : nodes | n in n.^(links)
    target in source.^(links)
    source.^(links) in nodes
}

pred isValid [p : path, t : topology] {
    p.nodes in t.nodes
    p.links in t.links
}
```

We create an example path using a more complex expression like *all routes that contain the link between ALPHA and BRAVO, and node CHARLIE as target.* It means a path represented by {`*,A,B,*,C`} that is defined in listing 3.8.

Listing 3.8: Path example modeled in Alloy

```
// Path <Alpha, *,Charlie, *, Echo>
pred isAlphaCharlieEcho ( p : path ) {
    isValid[ p, topologyOne ]
    p.source = Alpha
    p.target = Echo

    Charlie in p.nodes
    Charlie in p.source.^(p.links)
}

sig _AB_C_path extends path { } {
    isValid[ this, t1 ]
    source = Alpha
    target = Charlie
    Alpha + Bravo + Charlie in nodes
    Alpha -> Bravo in links
}
```

3.3.4 Flow Model

As policies are not only declared in terms of infrastructure —nodes or paths—, we model a network connection as a data *flow.* A flow describes data and topology characteristics for a communication session. For our design, the network flow consists of a path or set of paths, traffic sequences and categories, and other constraints, like times (days or hours) in which the flow is allowed, or grouping other characteristics such as users for whom the policy applies. Note in listing 3.9 that a path is a feature of a flow, and other features, time or user alike, should be defined as multiple domains on different sets —signatures.

Listing 3.9: Example of flow specification.

```
sig network_flow {
    paths : some path,
    traffic_seq : one traffic_seq,
    traffic_category : one traffic_category
}
```

3.3.5 Policy Model

According to our abstraction § 3.1.5, a policy is the aggregation of a flow, conditions, and an action, for a simple case: deny or permit. Policy rules are modeled as *facts*. A fact, in Alloy language, is a constraint that must be hold on the entire system. Hence, facts describe the mandatory set of rules to consider the model as valid. For our example, consider a network policy that *denies traffic through Charlie–node and carries video*, as seen in listing 3.10. This policy has three attributes: 1) the path that includes the node *Charlie* is defined as shown in listing 3.10, 2) there is not a flow where *Charlie* is in the path, and 3) the traffic type defined for this flow is video.

Listing 3.10: Policy instance with path, flow type, and action.

```
1 // smith <*,C,*> {video} {deny}
2 fact policy1 {
3     no f: network_flow{
4         all p: f.paths {
5             is_C_path [ p ]
6         }
7         f.flow_type = traffic_type_video
8     }
9 }
```

The policy states that there is not a flow such that there is a path with the node *Charlie* on it and has video as flow type. Note the lines 3 and 4 of example listing 3.10; they state that there is not a network flow with paths that go through the node Charlie using a predicate.

3.3.6 Policy conflict Model

The purpose of this approach is to identify network policy conflicts; the `fact` description available in Alloy is used to model network policies. In addition to the fact definition, Alloy uses multiple satisfiability solvers to identify the set of policies that invalidates the model.

For this work, we use *kodkod*[a] as constraint solver and its *Unsat–Core* module which finds an instance for the formula if the model is satisfiable —a possible implementation— or a counterexample of unsatisfiability, in case a conflict exists, with the minimal unsatisfiable core extractor [TCJ08].

Recall the policy that *permits video traffic over a path which starts in Alpha, ends in Echo, and goes through the node Charlie*, see Neo policy in listing 3.4. The model for this policy is depicted in listing 3.11 with the path $\langle A - * - C - * - E \rangle$, and contradicts with the policy in listing 3.10. Compared with the previous policy shown in listing 3.10, a conflict arises because one of the policies cannot be applied without contradicting the other. Here, both policies apply over video traffic; the first policy denies traffic from any source to any destination that goes through Charlie; the second one specifically allows video traffic over Charlie. Hence, these two policies cannot be applied at the same time.

Listing 3.11: Policy example: a contradicting policy.

```
// neo <A,*,C,*,E> {video} {permit}
fact policy2 {
    some f: network_flow {
        some p : f.paths {
            isAlphaCharlieEcho [ p ]
        }
        f.flow_type = traffic_type_video
    }
}
```

3.4 Path, Policy and Conflict Detection with Alloy

In summary, the model written in Alloy comprises: *1*) network topology abstraction with nodes and links; *2*) detailed paths with complete or partial description of nodes, links, source or destination; *3*) high–level communication constraints represented as flows which include paths, sequences, and categories for traffic; and *4*) policies described as facts. A policy uses previous definitions to limit the communication description. Model components are summarized in table 3.2.

We verify topologies, paths, and flows, at low–level infrastructure because in the SDN environment there are forwarding tables in datapaths that we check against the high–level policy. At this point, we need to model paths and flows to identify inconsistencies at datapath configuration analysis.

3.4.1 Checking topology invariants

This section shows topology invariants checked by our implementation in Alloy. We present here a constructionist schema, from topological basis to more complex structures of

[a]Kodkod is a constraint solver for first order logic. It allows set relations, transitive closure, bit–vector arithmetic, and partial models. `http://alloy.mit.edu/kodkod/`

Table 3.2: Model components and Alloy instructions.

Model component	**Alloy definition**
Node	Node signature, abstract and specific.
Network Topology	Abstract and specific signature that includes nodes, links and invariants. Links are node relations, and no–loop and reachability are invariants.
Path	Abstract and specific signatures. It includes nodes and links; is the physical support of communication. Invariants include source and destination reachability, and nodes and links that are part of the valid topology.
Flow	Abstract and specific signatures. It includes sequences, categories, and other communication attributes.
Policy	A fact is used to describe a policy. It describes if the policy denies (no f: flow such as ...) or allows (exists f: flow such as) a flow. Also it can include path and flow attributes on its constraints.

policies. Then correctness and performance tests are evaluated with artificial topologies and other tests.

Isolation

Recall that node degree $d_G(n)$ is the number $|L(n)|$ of links at n, equivalent to the neighbors of n. A node of degree 0 is called *isolated*. This invariant validates if there exits an isolated node in the topology; see figure 3.3b. In Alloy, this verification ensures that $\forall n \in N : |L(n)| > 0$. The verification of non–isolation is shown in listing 3.12.

Listing 3.12: Alloy representation of isolated test.

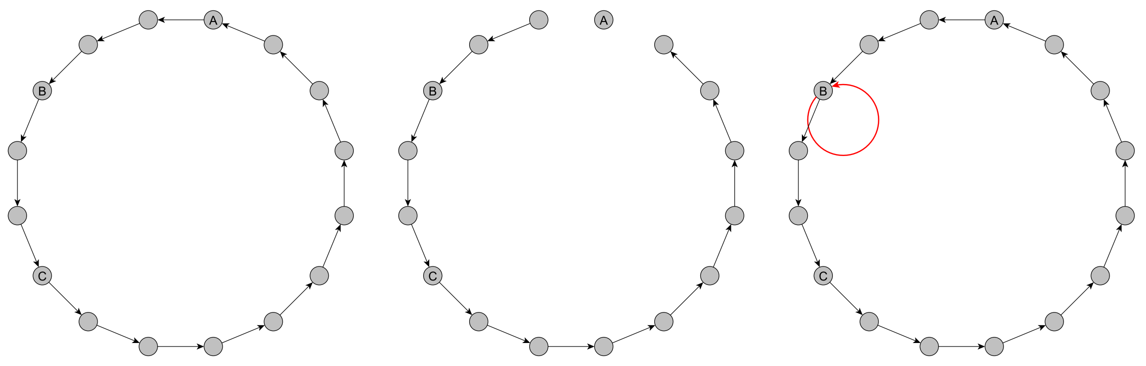
```
fact NonIsolation{
  all t:Topology, n: Node |
  n in t.nodes implies some n.(t.links) + t.links.n
}
```

Self–loop

Self–loop is an edge that connects a node to itself, see figure 3.3c. The objective is to guarantee that there are not links from a node to itself and holds the condition $\nexists n \in N : (n, n) \in L$. The verification of self–loop is shown in listing 3.13.

Listing 3.13: Alloy representation of self–loop test.

```
fact SelfLoop{
  all t:Topology, n: Node |  no n in n.(t.links)
}
```



(a) Topology base. **(b)** Isolation of a node. **(c)** Self–loop over a node.

Figure 3.3: Topology invariant examples over nodes.

Reachability

The reachability property consists in the existence of a path that *links* a node n to any other node m within the graph G. Note that every node is reachable from any node in the topology base, figure 3.3a. In short, $\forall m \in N(G)$ exits a path $P \subseteq G|P = (n, \ldots, m)$. The verification of reachability uses the expression of *reachability relation* over a graph G and the transitive closure R^+ over L, defined in § 3.1.2. Hence, m is reachable from n if and only if $m \in \text{links}(n)$. Alloy implementation of this property is shown in listing 3.14. Note that transitive closure is denoted by the hat (ˆ) operator.

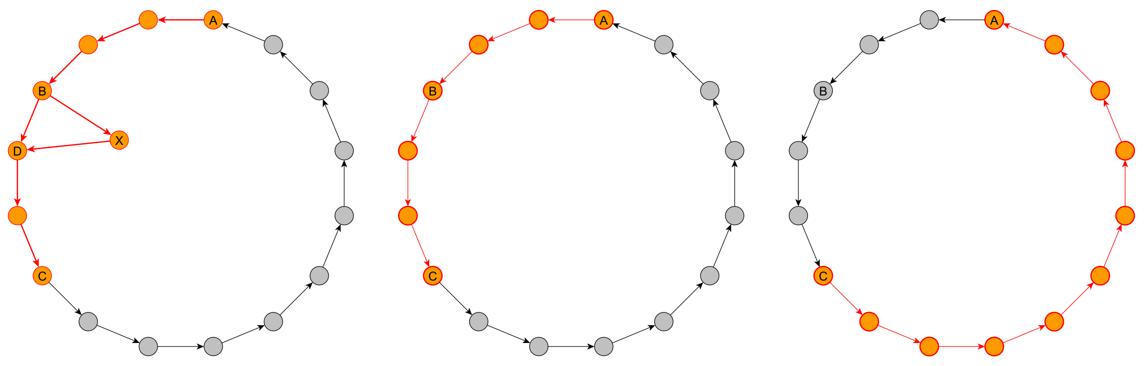
Listing 3.14: Alloy representation of reachabilility test.

```
fact Reachability{
  all t:Topology, n,m: Node |
  n in t.nodes and m in t.nodes and m in n.^(t.links)
}
```

Now we define invariants over paths instead of tests over nodes. Recall that paths are defined by facts in the Alloy representation. Those *facts* may include start and target nodes, intermediary nodes, and partial paths as those described in definition 3.3.

Cycle–Path

Recall that a path forms a *cycle* if given a path $P = n_0, \ldots, n_{k-1}$ a new link $n_k \rightarrow n_0$ is added. $C = P + n_k n_0$ denotes the sequence of nodes of a cycle. We identify path cycles by using the transitive closure. For this invariant, we want to identify if a node n is included into the set of nodes reachable from n. A path with no cycles is said to be *acyclic*. In other words, we say that a path P is acyclic if a node $n \in P$ is not reachable from itself $\nexists n \in P | n \in \text{links}(n)$.



(a) Cycle over a path. **(b)** Waypoint over a path. **(c)** Avoidance over a path.

Figure 3.4: Examples of invariants for paths.

Listing 3.15: Alloy representation test of a path with cycles.

```
fact NoCycle{
  all t:Topology, p: Path | let n in p.nodes
  no n in n.^(p.links)
}

// Path from node A to C.
one sig p1 extends Path{} {
  source = a
  target = c
  nodes = a + c + b + d + x
  links = b->d + b->x + x->d
}
```

The example in figure 3.4a shows a path from A to C, and listing 3.15 shows a path definition with the links $(B, D), (B, X)$ and (X, D). This path conflicts with the invariant because there is no way to have a path that both: a) contains those links, and b) does not have cycles. Also note that the set is defined by the addition operator $(+)$ between elements.

Waypoint

A waypoint obliges a path to go through a specific node. Being w a node in the graph, the path P complies with the waypoint if $w \in N(P)$, belongs to the nodes in the path. Waypoint can also force to include sets of nodes or links. Listing 3.16 shows the implementation of a policy that coerces node B in the path. In the same way, figure 3.4b shows the path which accomplishes the policy restriction.

Listing 3.16: Alloy representation test of a path with a mandatory node.

```
// Policy with a waypoint over B
fact WaypointB{
  b in p1.nodes
}

one sig p1 extends Path{} {
  source = a
  target = c
}
```

Avoidance

As opposed to Waypoint, avoidance rejects topological elements from a path. Here the model finder is important because it can find a path description that satisfies the restrictions. Being n a node in the graph, the path P complies the avoidance if $\nexists n \in N(P)$. The model finder will detect a path without the node in mention. Listing 3.17 shows the implementation for the path from C to A without going through B.

Listing 3.17: Alloy representation test of a path with node avoidance.

```
// Policy avoids ing over B in the path from C to A
fact Avoidance{
  no b in p1.nodes
}

one sig p1 extends Path{} {
  source = c
  target = a
}
```

3.4.2 Checking invariants and policies with Alloy

Testing Topologies

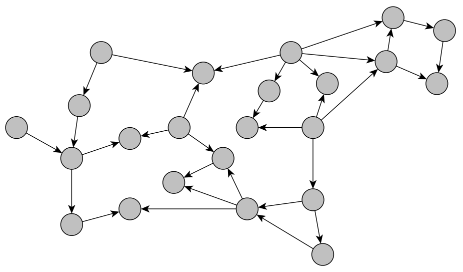
We test our implementation over well–known topologies to verify effectiveness and the time to find instances. The test is composed of several network topologies. Some tests are done with synthetic topologies, such as rings, other with well–known topologies such as AboveNet, a topology from the Topology–Zoo[b]; AN1755, a topology from the Rockefuel project[c]; and *ta2*, from the SNDlib[d]. Figure 3.5 shows the networks used for testing.

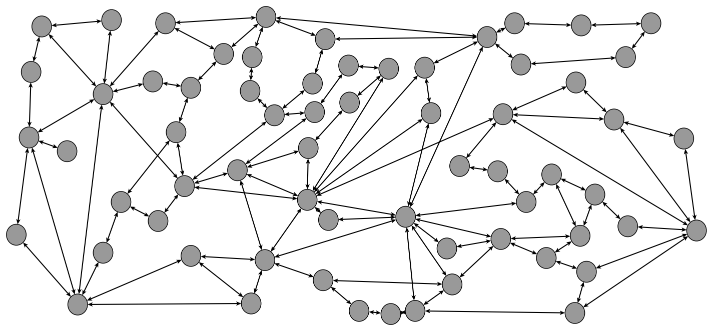
[b]http//:www.topology-zoo.org/dataset.html
[c]http://research.cs.washington.edu/networking/rocketfuel/
[d]http://sndlib.zib.de/

Invariant measures

Model–checker creates a CNF from the model and invokes the SAT solver to find instances, see § 2.2.2. Generally, we can obtain some information from the solver, such as the number of variables of the model and execution time. The *number of variables* gives information about the model complexity. It denotes the number of variables after model transformation, skolemizations, and implication reductions. Also, we obtain the time spent for the CNF and the time to find an instance or counterexamples that contradict the model specification.

Figure 3.6 shows the time spent by the solver to create the CNF and finding an instance that corroborates or contradicts the model. The non–isolation verification, figures 3.6a and 3.6b, is built from the AN 1755 topology (figure 3.5c) and its counterexample after a node isolation. The number of nodes as the independent axis is a variable from the same topology.


(a) AvobeNet with 23 nodes and 31 links.


(b) *ta2* topology with 65 nodes and 128 links.

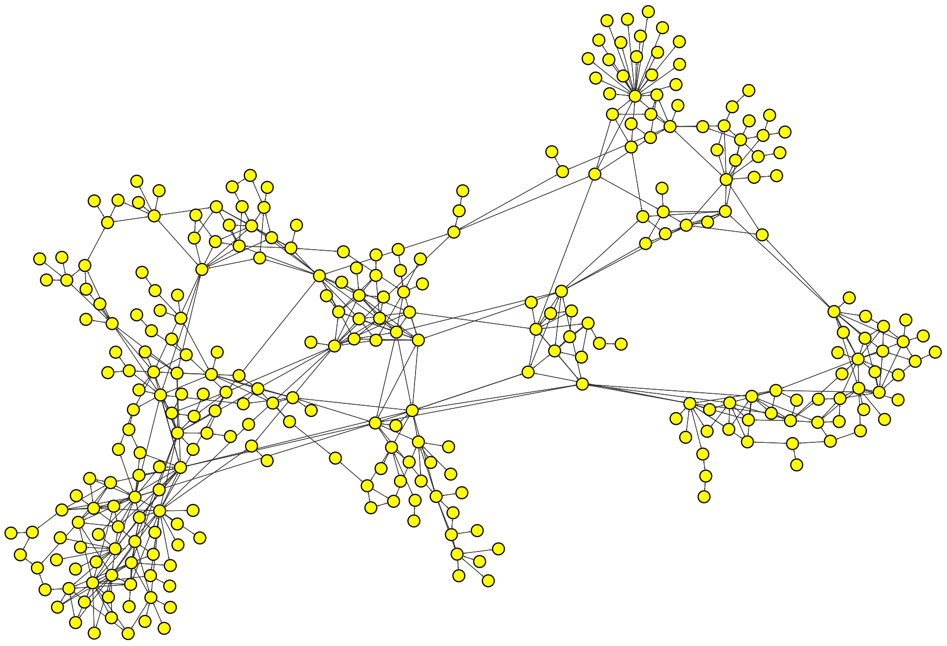
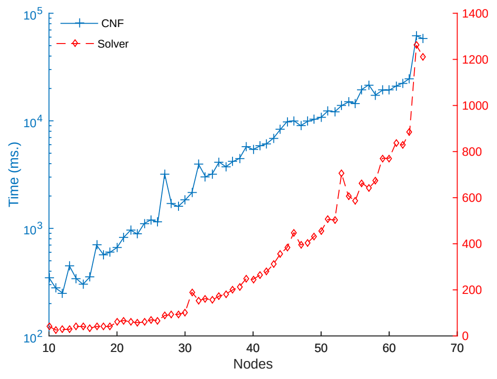
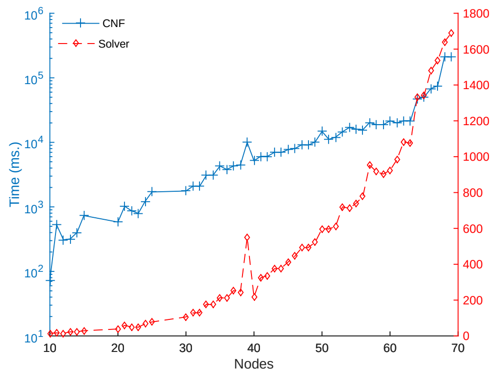

(c) Topology from AN 1755 with 300 nodes and 1097 links.

Figure 3.5: Topology examples used to test network invariants.

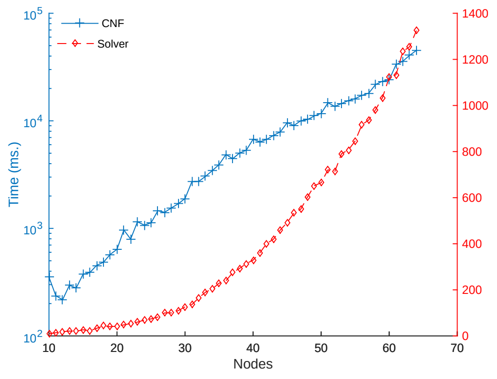
Reachability tests are reported in figures 3.6c and 3.6d. They are created from a Ring topology to guarantee that we will always have a true validation, and after removing a node the solver finds the counterexample. Blue line indicates the time to create a model for the topology and the predicate to check invariants. Note that the time to create a mode is consistent in all cases. It takes about a second to create a model which includes between 20 to 30 nodes, and 10 seconds to build a model with 50 nodes. Red line indicates the times spent by the SATsolver to check the invariant. Note that the time of finding a counterexample varies compared to the time of computing a valid case because of the way in which the assert is written. For example, the time of finding a false invariant is lower that finding a true invariant. This is because checking a false case is enough to declare a false invariant. On the other hand, the solver must check every case to declare an invariant as true.



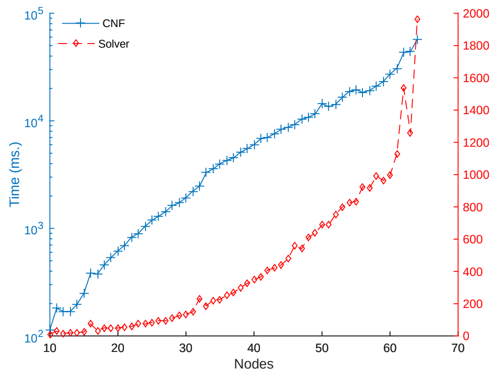
(a) Non–isolation invariant false.



(b) Non–isolation invariant true.



(c) Reachability invariant false.

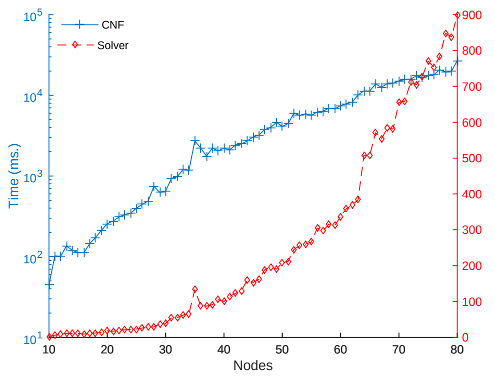


(d) Reachability invariant true.

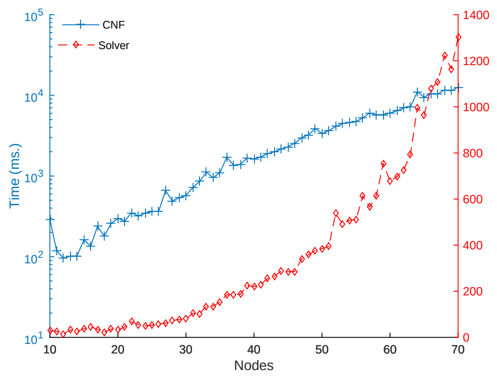
Figure 3.6: Time for CNF creation and model solving for node invariants.

For path invariants, we present a similar graphic, see figure 3.7. Here the spent time to create the model and the predicates for invariant checking is consistent in all cases and ranges between 0.1s for 10 nodes and 10s for topologies with 70 nodes. However, the difference between a model that checks the existence of a property compared to a model that guarantee its non–existence is remarkable. For example, red lines in figures 3.7a and 3.7b show the time of finding an instance and its invariant as true and false. Note that finding an instance with true invariant takes more time than a false invariant. It means, checking a path that does not have a cycle is more expensive than verifying if a path that has one, because the solution has to evaluate all paths defined in the model.

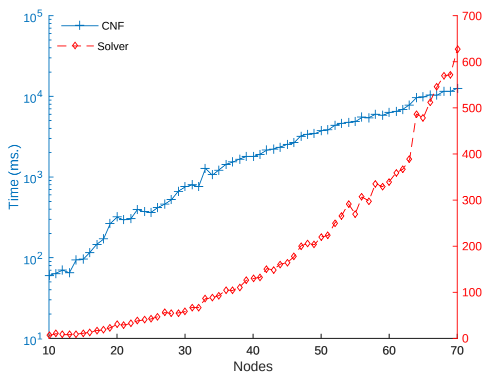
Meanwhile, finding a counterexample is faster because the solution only needs a case where the model is unsatisfiable to report the model is inconsistent with the policy. Note that the validation time for the valid scenarios is greater that the validation time for false scenarios, figures 3.7b, 3.7d and 3.7f, and are consistent for all experiments. However, the complexity of creating a model remains the same, see the right axis, no matter if the model is evaluated valid or invalid.



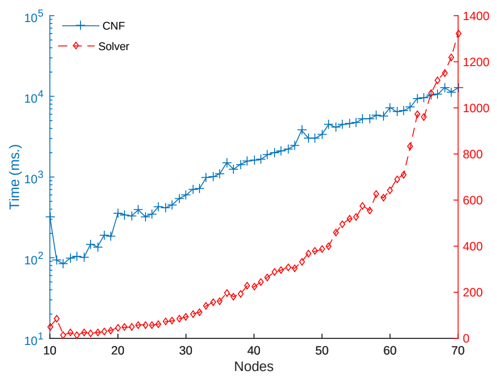
(a) NoCycle invariant false.



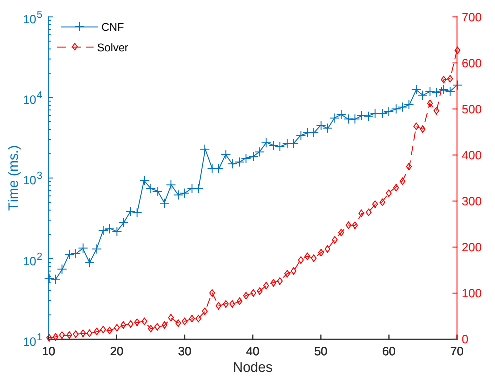
(b) NoCycle invariant true.



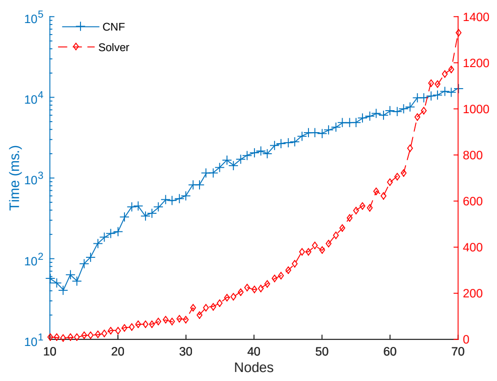
(c) Avoidance invariant false.



(d) Avoidance invariant true.



(e) Waypoint invariant false.



(f) Waypoint invariant true.

Figure 3.7: Time for CNF creation and model solving for path invariants.

3.5 Summary and Conclusions

This chapter presents the implementation of a PPL verifier supported on Alloy that allows the verification of paths and policies, and detects conflicts in policy rules. This verifier exploits relational logic to explore the paths in the network, and reasons about constraints defined on them in a simpler and more complete way than previous compilers. In detail, this chapter shows how nodes, links, and network topology are described in relational logic. Then, the description of policies using the *fact* definition as holds the model properties. Finally, this chapter shows how the unsatisfiability solution is used to identify facts contradictions, and hence conflicting and violation of policies.

Logic and relational language is used to describe topologies, paths, flows and policies in a declarative way. Also, it helps to translate this model into multiple solvers of satisfiability, especially Kodkod is used because it can calculate the minimal set of unsatisfiable arrange by using the Unsat-Core solver. The solver can determine which signatures and facts produce the unsatisfiability conflict, then our tool maps them and returns the specification inconsistencies. The presented approach is able to detect a more diverse and wider number of policy conflicts compared with previous PPL compilers [Guv03; Sto00].

The experimental PPL verifier is implemented and its details can be found in [MLCD14]. It takes a PPL specification, produces an Alloy model and uses the solvers to detect conflicts. The compiler is evaluated using several example networks, including some used in seminal work about PPL [Guv03; Sto00]. For instance, that work included an example set of policies on a network of 10 nodes and another set on a network of 80 nodes. These sets are used to compare the performance and completeness of the previous compilers [Guv03] and are part of our evaluation.

Advantages of using a logical–relational language

This chapter shows the creation of a PPL verifier based on Alloy to transform these rules into operational parameters for each entity in the corresponding network device, and overcome the contradiction and security conflicts.

The existing compilers use algorithms that first expand all the path definitions in sets of paths, then detect the segments where the paths overlap, and finally determine if there are conflicting conditions in the common segments [Guv03]. In contrast, this approach translates PPL into Alloy models. This translation offers some advantages over the previous work:

- *Extended support for wildcards in paths.* PPL is a language that uses wildcards ($*$) to denote unknown sets. In other approaches, those wildcards helped to describe complex paths, multiplicity of nodes, or unbound restrictions. It allowed paths specifications such as {`A,*,E`}, but not complex path declarations like {`*,B,C,*,D,*`} [Guv03]. The approach used on this chapter is able to interpret more complex paths. For this reason, this approach can reason about a bigger set of policies.

- *Extended support for user and group policies.* Today's compiler supports conditions over network user profiles but it does not consider the relations among groups and users.

Existing compilers find a conflict if some rules permit and other deny the traffic from the same user, but not if a rule permits the traffic from a user and another rule denies the traffic of the user's group [Guv03]. Our approach overcomes this issue including relationships between groups and users. Other approaches for role–based access control [PSS11] were the motivation to include it into our model.

- *Extended support for action conflicts.* Not only deny and allow are actions for network policies. Network managers could create alarms which will be triggered after a condition. Since actions set is defined into our model, other software actions can be included. Moreover, previous compiler cannot find policy conflicts on more than two policies. Finding the minimal unsatisfiability core, it is possible to identify if there are conflicts that involve more than two policies.

The existing PPL compilers have limited support to detect when two rules try to set different values to attributes such as the *priority* or the bandwidth assigned to a specific type of traffic [Guv03]. This issue is overcome by adding additional constraints that check if two rules do not include conflicting assignations on attributes.

Besides denying or permitting a type of traffic through a path, other policy languages such as Flow-Based Management Language (FML) [Hin+09b] support policies to force that specific types of traffic should go through or avoid a network node. This kind of policies are included in our model and their translation to Alloy abstraction is done through aggregation of constraints.

This chapter establishes the foundations of verification of topologies, paths and policies based on paths. The next chapter elevates the abstraction level to security policies based on firewall rules. Next chapter presents the model and verifier of firewall policies in a SDN environment.

Chapter 4

Network Robustness: Targeted Attacks on Interdependent Networks

Things break and complex things break in complex ways.

Steve Bellovin

This thesis is focused on providing a novel mechanism to enhance network availability. Chapter one presents lightweight formal methods to find network inconsistencies between the specification and the topology, and how the inconsistency may produce erratic functionality. This chapter analyses network availability from a complementary point of view: the topology robustness. We model the control and data planes as an interdependent network, and demonstrate how failures in the data or control plane are propagated in the interdependent network. It also presents an interrelating work oriented to increasing the robustness of interdependent networks and its application to SDN.

This representation includes the set of links which connect control plane with data plane, for the case of SDN networks [SH15]. We describe network attacks to model a pattern of failures. In this chapter, we focus on two types of multiple attacks: sequential and recalculated, and show the propagation scheme and its impact. We model the set of interconnection links, that connect the interdependent networks, following three patterns: high–high, high–low, and random; according to the betweenness centrality in the control plane (network A) and the data plane (network B). Later, we expose those three models to sequential targeted attacks and the measure the robustness of the network with the ATTR metric. Results show the interconnection pattern that is more and least affected to this type of attacks, and shows the critical spots that should be addressed in case of having targeted failures.

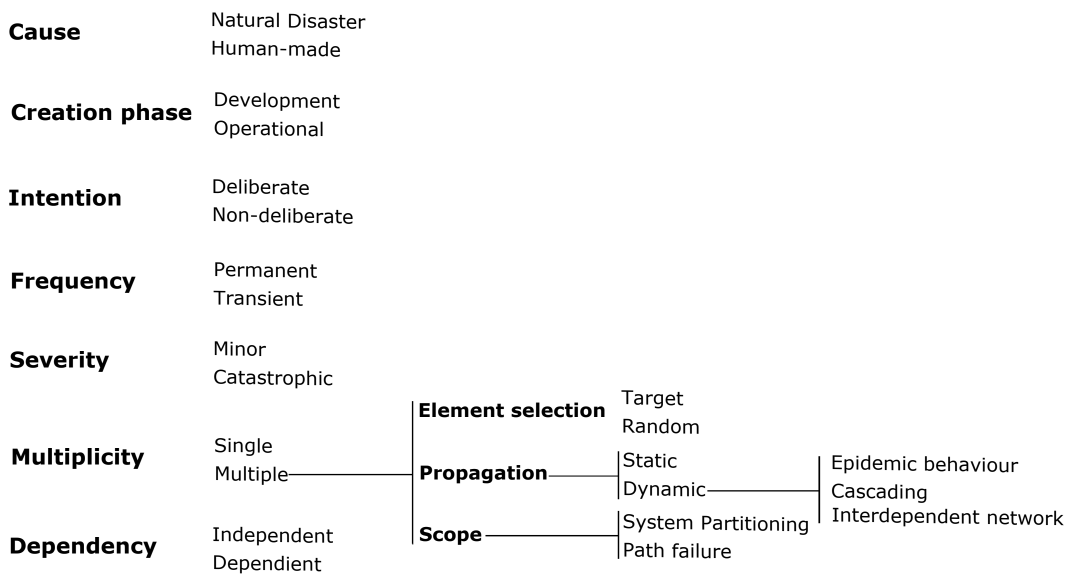


Figure 4.1: Taxonomy of failures adapted from [Seg11].

4.1 Network failures

A *failure* is an event that deviates the network performance from the normal operation, it is the transition from a correct to an incorrect state. The period of incorrect (failed) state is called *service outage.* The transition between incorrect to correct state is a *service restoration.* A *fault* can be internal or external, and cause errors if there is not a mechanism for detecting and isolating the fault, a *fault tolerance* mechanism. An *error* is a system state that can conduce to a failure; it is the manifestation of a failure. Then, after a failure occurs, network can start a *recovery* process and later the *restoration. Robustness* is the ability to continue normal operation even under attacks or failures [EK13].

Failures in a network can be single or multiple. *Single failures* are mainly related to network components e.g. cable cuts, equipment breakout, which can disrupt their normal operation. There are several methods and techniques for dealing with single failures so that service continuity is not compromised or quickly restored. On the other hand, networks can suffer due to *multiple failures*, disruption of multiple network elements simultaneously, which occur due to a wide variety of causes. Failures could be physical or logical. Physical failures are related to edge or vertex failures, principally hardware, in contrast, software failures affect the node logic (behavior) e.g. routing algorithms or controllers in SDN environment. Software failures are more complex and there are many variables that can lead to errors and malfunction.

Failures are classified based on the characteristic that affects the normal operation. Figure 4.1 presents a taxonomy of failures, according to intention, non–deliberate failures are accidents, such as misconfiguration or operational mistakes; and deliberate failures are attacks. According to source, failures can be natural disasters, human–caused or produced by a system, module or piece of software which can trigger security or availability issues. For example, Wool et. al studied firewall configuration errors [Woo04], and [LSK09] shows how human failures could affect the information privacy. Moreover, The Yankee Group found that 62% of network downtime is caused by human error and 80% of IT budget is spent in maintenance tasks [Ker04]. Also, there are failures due to unusual but legitimate operation. Overload traffic is a non–malicious request for a service higher than the normal operation can supply. This exceptional traffic is commonly caused by a *flash crowd* when the service experiments huge volume of demand. Other kind of failures are those present for interdependent networks; they experience a cascading effect from the interaction with other networks. Those failures occur because of intense network interconnectivity. The taxonomy of failures presented in figure 4.1 will be used to classify network vulnerabilities and their correspondent failure.

4.1.1 Multiple Failure Models

Network infrastructure is susceptible to multiple failures, and over a communication path they cause significant service disruption. Consequently, multiple failures can be modeled as patterns and can be grouped according to their characteristics. In Data Centers for example, 41% of link failures between two and four devices, but 10% involve more than four devices [LHKA12]. Reliability and security constraints denote challenges for management, and business productivity depends on network infrastructure working properly. However, networks suffer disruptions due to planned maintenance, failures, or misconfigurations. Multiple failure propagation schemes, most of the times, are complex and difficult to address, continue being an active research field [Cal+10]. Better understanding of failure propagation is the way to discover how many phenomena occur in the network.

Failure propagation in cascade is also a well–studied topic, and it is widely used in communication and transport networks. Those models have physical descriptions, and for complex networks the interaction between individuals is also modeled [CLM04]. Additionally, the model includes the degradation process after the failures, and describes how an action can affect other nodes in the network.

Like immune and biological systems, epidemic failure behavior is a recent approach to modeling multiple failures in networks. Recently, epidemic models were developed to explain how those failures follow susceptible-infected (SI) and susceptible-infected-disabled (SID) models [Cal+10].

However, these studies were focused on individual networks only. They do not consider the interaction with other networks and how a functional system may alter the other. Modern networks depend in depth from each other, and those vulnerabilities were exposed and modeled, as *information cascade* and *cascading failure* [SQZ14].

One of the model for multiple failure is the *sequential attack*. This model of failure consists of a sequence of node disconnection. Node disconnection is calculated once *a priori*, or recomputed after each disconnection. Sterbenz et al. used betweenness centrality (bc) for a planned network attack. They calculate the bc value for all nodes, order nodes by its value and remove nodes in that order [Ste+11a]. They showed that removing a few nodes reduced the packet–delivery ratio to less than 60%, thus the damage is greater than random attacks, link–centrality or node degree. Meanwhile Sydney et al. recalculate the betweenness centrality after each node is removed [SSYS10]. Schneider [Sch+11] proposed an onion–like structure to increase the robustness and mitigate malicious attacks based on recalculation of node degree after a removal. Disconnecting nodes based on Miuz measure generated more damage than other centrality measures [BRSBJ15]. A variation of sequential attack consists in recalculating the centrality measure of the network. Then, the attack eliminates the node with the highest centrality measure. The remaining nodes are sorted again as a new attack sequence, and it restarts the attack again.

4.1.2 Robustness

Robustness measures the impact of a vertex or edge removal in the integrity of the network[EK13]. In other words, it is the network's ability to continue performing well after failures or attacks. Robustness can be measured as the fraction of nodes removed before a complete collapse [HKYH02].

Centrality is a measure of importance for a vertex or edge in a network. Centrality measures can be adapted to measure robustness because if a node (or edge) is important for the network, removing it will affect the network integrity and functioning. One of the robustness metrics is the degree centrality (d_c), which ranks the nodes according to their degree. A node with higher degree is more influential and removing this node may considerably affect the network robustness. Closeness centrality (c_c) ranks the average distance from a node to any other node in the network. Meanwhile betweenness centrality (b_v) measures the importance of a node in the set of shortest paths of the network. It counts the number of shortest paths that go through a specific node. There is also the edge betweenness (b_e) with edges as parameter, and the average of both values.

In physics, network robustness is related to percolation and the study of the largest connected component P_∞. If $P_\infty \sim 0$ means the network is ruined, and if $P_\infty \sim 1$ means the network is completely functional. $P_\infty(p)$ denotes the giant connected component after the fraction $1 - p$ of nodes is eliminated from the network. The value p_c denotes the point after which the graph disconnects, it is the *phase transition*. The *Schneider* robustness metric considers the size of the biggest component after a node removal, and proposed a model for designing networks to improve robustness according with this measure [Sch+11]. *Miuz index* shows the impact of disconnecting a node and comparing the sizes of remaining connected components [BRSBJ15].

The *average two–terminal reliability* (ATTR or A2TR) represents the probability of connectivity between two randomly chosen nodes. It calculates the number of nodes per network component over the number of nodes of the complete graph. If the network is connected, then the $ATTR = 1$ because it divides the number of node pairs for each component over the total number of pairs in the network.

Average node degree ($\langle k \rangle$) denotes how well connected a network is. It is said that a network with high $\langle k \rangle$ has greater strength. The *Assortativity coefficient* (r) is the correlation between the degree of connected nodes. It measures the tendency of a node to connect other nodes with similar degree.

Starting from a connected network, *vertex connectivity* (κ_v) is the minimal number of nodes to be removed in order to disconnect a network. Likewise *edge connectivity* (κ_e) is the minimal number of links to disconnect the network. Other metrics are based on distances such as the *average shortest path length* ($\langle l \rangle$), and the network diameter ($\bar{d}$) that is the length of the largest shortest path. *Efficiency* (E) is a metric of dispersion that includes the distance between two nodes d_{ij}. The number of *disjoint paths* between two nodes measures the amount of spare paths for one–to–one traffic. The *clustering coefficient* (C) measures the number of triangles over the portion of vertices. In other words, it measures if two nodes shared the same neighbor. High cluster coefficient means high robustness because there are more alternative paths between two nodes.

Some other metrics are based on operations over their matrix representation or spectrum. For example, operating the adjacency matrix, it is possible to obtain the largest eigenvalue (λ_n) to estimate its robustness. This measure is related with the vertex centrality. Graphs with small diameter and high λ_n have multiple paths between two vertices. The *Symmetry Ratio* denotes the rate of eigenvalues of the adjacency matrix over the network diameter; the lower the symmetry the more robust the network is. Another measurement that arises from matrix operations is the *Laplacian* matrix $\mathsf{L} = \Delta - A$, where Δ is the degree matrix and A is the adjacency matrix. The *algebraic connectivity* (λ_2) is the second smallest Laplacian eigenvalue and quantifies how difficult it is for a network to break into parts after node elimination. Also, *natural connectivity* ($\bar{\lambda}$) uses the average of adjacency eigenvalues to estimate the redundancy of alternative paths between two nodes. Finally, the number of *spanning trees* (ξ) is also another measurement of robustness which can be written as a function of Laplacian eigenvalues.

Other metric of robustness are composed of multiple network measurements and consider topology features. The *R–value* is a model that includes a vector of multiple metrics and a service vector of components (weights) which emphasizes a metric over the other. R–value is typically normalized between 0 (no robustness at all) and 1 (perfect robustness). *Elasticity* is a robustness measure as the area under the curve of throughput vs percentage of remaining nodes in the network under attack [SSSK08]. *Viral conductance* (VC) measures the network robustness as the ability of a network to spread a virus under a specific epidemic model [YKS11; Mie12].

However, these metrics are focused on measuring the robustness in single networks. They do not consider how the network is affected when it interacts with another network.

4.1.3 Survivability and Resiliency for Communication Networks

Survivability denotes the capability of a system to fulfill its mission in a timely manner, in the presence of threads such as attacks or natural disasters. Ideally, the network aims to maintain an operational state. To keep the functional state, a component such as hardware or software monitors network performance. The monitor tests network health and produces alarms when the performance is under an acceptable level. After a failure that degrades the service, the system triggers recovery procedures. The recovery enables the system to support critical or gradated services as the situation allows them.

The process of failure recovery comes from self-healing systems, which are autonomous systems able to adapt themselves from failures and errors. The recovery process has two phases: fault management, and reversion. Fault management comprises detection, localization, and notification of a failure; and the reversion phase is in charge of recovery and restoration. For a success process, both *protections* and *restoration* actions are needed. Protection actions take place before a failure occurs, and restoration is the process launched once the failure has been detected to recover a previous functional state. Resilience is the ability of a communications network to provide and maintain an acceptable level of service in presence of faults and challenges to normal operation [Ste+11b].

Failure detection is a *daemon* process that monitors the network for anomalies, these can be degradation of a service, or even interruption. After that, the system should be able to identify the failure and its localization. Subsequently, it notifies the information necessary to begin the recovery procedure. After knowing the details of the problem, the *recovery* process is initialized. It provides an alternative service associated to a failure scenario. Finally, the process of *normalization* or reversion restores the functional state.

Fault tolerance is the ability of a system to tolerate faults such that service will not fail. For that reason, it is considered a subset of survivability processes. Fault tolerance relies on *redundancy* to compensate for random uncorrelated failure of components. Although it is not sufficient to provide coverage in the face of correlated failures.

Disruption Tolerance is the ability to tolerate disruptions in connectivity, mobility, delay and tolerance of energy. Due to changes in the communication environment, it is difficult to maintain stable connections between users. The study of dynamic network behavior started with MANETs with dynamic routing mechanisms for nomadic members. Another interesting field is Delay-tolerance networks like satellite communications. A recently, energy-constrained networks, e.g. wireless sensor networks. Nowadays the techniques for disruption tolerance in networks have reaches other field, including VANET vehicular networks.

Traffic Tolerance is the ability of handling unpredictable demands of traffic without blocking or degrading significantly the service, even when there is congestion. The main topic in resilience, due to its variability and demand of resources, is traffic. Also, traffic tolerance includes isolating the effects from high traffic demand to other links and nodes. The traffic can be generated from legitimate users or might be consequence of malicious attack such as Distributed Deny of Service attack (DDoS). DDoS detection is a needed characteristic of the system, since the network must differentiate DDoS from legal traffic. However, sophisticated DDoS are indistinguishable from normal traffic.

Dependability quantifies the reliance on a service. It is composed by two aspects: availability and reliability. The basic measures of dependability are: The Mean Time To Failure (MTTF), the expected value from the failure density function, and the Mean Time To

Repair (MTTR), the expected value of the repair density function. The mean time between failure is calculated as: $MTBF = MTTF + MTTR$. *Availability* is the probability that a system will operate when it is required, and is calculated as $A = MTTF/MTBF$. *Reliability* is the probability that a system remains operating for a period of time, and is calculated as $R(t) = P[\text{noFailure}[0,t]] = 1 - Q(t)$ where $Q(t)$ is the failure cumulative distribution function. Availability and reliability depend on application requirements. For transactional systems it is important to have a low MTTR. Whereas for session and connection oriented systems, where MTTF is important to have high MTTF values.

Protection Mechanisms

When we talk about protection, we refer to redundant resources used once a failure is detected. There are spare resources used depending on the failed section: vertex, edge, path or tree. Redundant resources are precalculated backup paths to switch to once a failure occurs [Wu95]. Protection for single failures is also a well–studied problem; multiple failures were also studied but their complexity arose; because of this, new approaches use more intelligent techniques to deal with multiple failures in graphs.

Disjoint path is a common approach from graph theory that consists in finding alternative paths which do not share nodes or links. The problem is to find a viable set of disjoint paths for a graph with low complexity for large networks [Tor92]. *Path protection* is a disjoint path used to protect a working path, a path is being used by the service. There are four schemes using dedicated paths to protect working paths: 1+1, 1:1, 1:N, M:N [HH07]. First, in the $1+1$ scenario there is a dedicated backup path, redundant, to protect a working path. In other words, there are two paths, which can be disjoint, to carry data. $1+1$ protection mechanism sends data traffic in both, working and backup, paths. Once a failure is detected on a path, the system continues using the available path. Similarly to the previous scenario, the $1:1$ protection mechanism has a dedicated backup path, but just one is used as working path. The backup path starts its operation and maintains the service when a failure is detected. The $1:N$ scenario has a dedicate backup path to protect N working paths. If any of the working paths fail, the data is switched to backup path. Then, the remaining $N-1$ paths are left without protection. Finally, the $M:N$ scheme is similar to previous the previous one, there are M dedicated backups, where $1 \leq M \leq N$, to protect up to N working paths. *Shared backup path protection* is a mechanism that protects several working paths reserving a shared backup link. This mechanism is similar to the $1:N$ protection strategy. However, there are several approaches in literature that create protection mechanisms for multiple and simultaneous failures [JOK03]. *p-cycles* is another mechanism that uses pre–configured protection cycles, a path that forms a cycle [Sch03]. First, nodes or links to be protected are selected, then a set of cycles is calculated to meet service restrictions, and the alternative path configuration is set. Once a failure is detected, the pre–configured path is applied and the configuration changes. This mechanism is principally applied to optical networking. Multiple algorithms that use p-cycle have been developed and today diverse mechanisms are based on it [KAJ09].

Failures in multicast networks is a regular problem, and the solution is finding redundant spanning trees to the multicast transmission [LCCM10]. A resource allocation mechanism that protects the multicast transmission against link failures is called *shared backup path protection* (SBPP). This mechanism is a set of multiple disjoint paths which can deal with single or multiple failures [JOK03; Pat+02] For example, autonomic computing implements multicast communication and has a set of shared resources, such as routers, in case of failures [PSB11].

Nevertheless, natural disasters and terrorist attacks increase the possibility that, when a networks failure occurs, multiple links that belong to the same shared risk link group (SRLG) fail simultaneously . Shared Risk Link Groups (SRLGs) are sets of links that share common resources. Multiple schemes have been developed to plan and deploy networks to avoid that situation and recover from those failures [RSM03].

Resiliency in SDN

Recall that once a datapath receives a packet, confirms if its header matches an entry of the FBI (Forwarding Information Base) and sends the packet to the indicated port; or send it to the controller if the header does not match any entry. If a failure occurs, the SDN datapath is recovered and installs fresh entries on its FBI. The time in which a FBI entry is renewed depends on the timer expiration in the datapath or the controller logic. Failure recovery in SDN is specific logic at the controller that install fresh forwarding rules in the datapath after a failure. Those applications are in charge of responding to failures, and other disruption events should have restoration and recovery logic. The important challenge is to identify if a failure occurs and take the proper process to refresh the path and avoid the failure. Controllers implement mechanisms to recover from a failure, for example, NOX implements L2–Learning which has a map of MAC addresses and ports [Gud+08]. L2–Learning implements the aging timer. When the datapath receives a packet, learns the source address and timestamps the FBI entry. If there are not more packets from that source, the datapath deletes the entry once its aging timer expires. Each controller has a mechanism to discover the topology, establish paths and install entries along the datapaths. Failure detection depends on the process to discover the topology.

When a link between the controller and a datapath fails, the following procedure is performed: *1*) The datapath detects the failure. *2*) datapath notifies the controller, *3*) controller program computes the repair actions, *4*) sends (pushes) an update datapaths, and *5*) the datapath updates forwarding tables. As opposed to traditional network, where notifications flooded the net, SDN transmits failure notifications straightforward to the controller. Moreover, controllers are more powerful to calculate backup configuration and can perform more calculations in contrast to small CPUs in the switches. However, this assumption is not true when the failure is between the device and its controller. In this case, the controller network needs to be re-established before the data network. Controller uses an IGP (Internal Gateway Protocol) that converges first before switches communicate. Therefore, the controller needs an out–of–band channel, hence, failure recovery process needs to pre–calculate backups paths.

The SPARC[a] project stated that network should recover from a failure within 50ms. Fast recovery is possible if a new entry is installed once a failure is detected. A solution is to detect the change in a link and identify all paths that use that link. Then recalculate a path for the affected nodes, delete the entries that use the affected link and reinstall fresh routes [Sha+11].

AFRO (Automatic Failure Recovery for OpenFlow) is a system that automatizes failure recovery for OF [Ku13]. After a failure is detected, it monitors and records the controller state before a failure occurs. Then a new instance of the controller is created without the network element that notices the failure. Moreover, it replies the inputs before the failure, installs forwarding rules to avoid the failed nodes, and its new state is consistent with the pre–failure. After a network topology change, the network forwarding state is restarted. Later the network state is recovered as the controller installs forwarding rules according to its control logic, initial configuration, and external events.

The main objective is to separate the applications that run on the controller from the failure recovery application; instead a runtime system recovers the network from failures. Thus, reduces the inclusion of new bugs.

However, the centralized controller is the major exposure in performance and reliability. The controller can be replicated or divide the network on multiple domains to segment it and reduce the risk. Controller replication also allows load balancing to handle multiple datapaths and end–to–end flows compromising the performance. Nonetheless, the replication adds more complexity because both have to maintain a strict consistency after topology updates and failures. Following this, datapath must be able to report the link status to the controller set to maintain the actual *state* of the network.

4.2 Targeted Attacks on Interdependent Networks

Real world networks are interdependent; i.e. they interact with each other. For example, the power grid network and Internet are coupled together. Two networks are said to be interdependent if the behavior or reliability of one depends on the state of the other. This system is composed by multiple networks, for example two networks A and B. When nodes in one network fail, they produce failures in nodes from the other network [PBH10]. Also Buldyrev showed that two networks A and B are coupled, and failures in nodes from A may generate a process of cascade failures in nodes from network B [Bul+10]. In contrast to single networks, interdependent networks have unique properties such as the first–order percolation transition caused by cascading failures.

4.2.1 Interdependent Network Model

Interdependent models are used to represent multilayer networks, such as overlay networks. Random graphs are a common tool to depict a backbone network because they properly

[a] *Split Architecture for Carrier–grade networks* is a FP7 project that aims to implement a new split in the architecture of Internet. `http://www.fp7-sparc.eu/`

describe its topological properties. We use the interdependent network model proposed by Buldyrev [Bul+10] for describing our targeted attack model.

An interdependent network comprises two networks (or layers) A and B. Each is an undirected network with its nodes and links $A = (N_A, L_A)$ and $B = (N_B, L_B)$. Nodes in A and B following a pattern of connection with a degree distribution $P_A(k)$ and $P_B(k)$. For example, they are randomly connected. We assume the number of nodes is the same $|N_A| = |N_B|$. The interdependent network is the result of connecting those *layers* with a set of interdependency links or *inter–links* L_I which connects both networks (layers). Then the complete graph is the set of nodes and links of both graphs plus the interdependency links $G = (N_A \cup N_B, L_A \cup L_B \cup L_I)$.

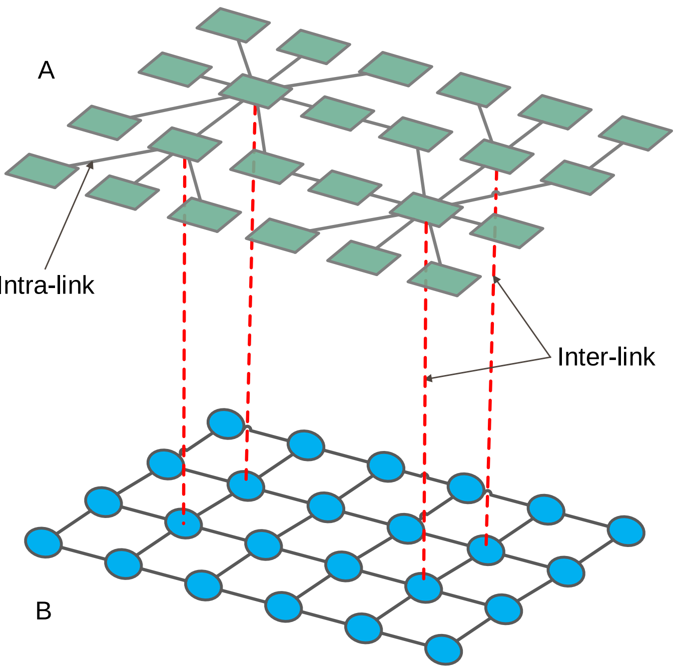


Figure 4.2: Illustration of interdependent network model. Squares represent nodes in network A, circles represent nodes in the second network B. Continuous lines represent intra–links (connectivity links), and doted lines denote inter–links (dependency links).

We may assume the interdependency links L_I are one–to–one and bidirectional $L_I = \{l_i : (n_{Ai}, n_{Bi})\}$ thus the set has $|L_I| = |N_A| = |N_B|$ interdependent links.

Let A_A and A_B the adjacency matrices of A and B respectively. Then, the adjacency matrix for the complete graph G is

$$\begin{bmatrix} A_A & 0 \\ 0 & A_B \end{bmatrix} \tag{4.1}$$

Where $2|N_A|$ is the dimension. After interactions are introduced, the interconnection matrix A_I represents the inter–links (n_{Ai}, n_{Bi}) between the networks A and B. The interdependency matrix is included into the original adjacency matrix and the result is 4.2.

$$\begin{bmatrix} A_A & A_I \\ A_I^{\intercal} & A_B \end{bmatrix} \tag{4.2}$$

Since inter–links are bidirectional, the connection from A to B is denoted by A_I, while connections from B to A are denoted by A_I^{T}.

4.2.2 Targeted attacks

Before a targeted attack is accomplished, graph's elements (nodes or links) are ranked in accordance with a centrality measure (e.g. degree, betweenness, closeness, eigenvector) and labeled. $c_{S_1} \geq c_{S_2} \geq \cdots \geq c_{S_N}$, where c_{S_i} is the centrality value of node S_i. The same ranking process is performed with the graph G_2, after the rank we have the list $c_{T_1} \geq c_{T_2} \geq \cdots \geq c_{T_N}$, where c_{T_i} is the centrality value of node T_i. Then graph elements are removed in accord with the ordered list, from high to low centrality value with the purpose of maximizing the impact of the attack in the network. Figure 4.3 shows a targeted attack in interdependent networks. Each node in A depends on one node in B. Interdependent links are denoted as dashed lines. One node in A is attacked by its centrality measure. Then the attack is propagated to its correspondent node in B. Finally, the node in B suffers the attack and propagates it.

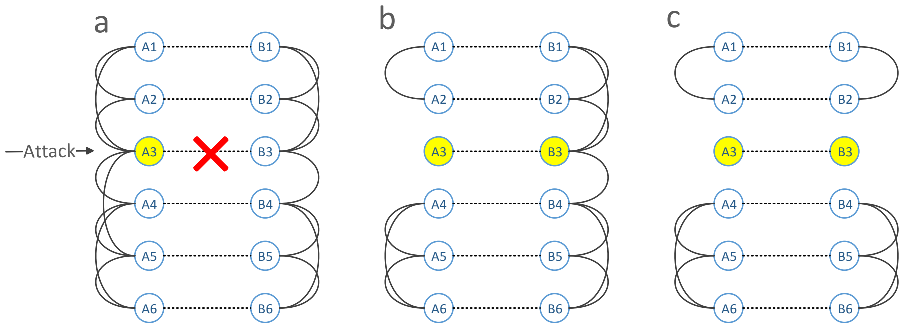


Figure 4.3: Targeted attack in interdependent networks.

For example, in backbone telecommunication networks, the most vulnerable routers can be identified by the number of shortest paths that pass through a given router, or by the number of physical links from one router to others. Moreover, other real world measures, such as the number of users potentially affected and socio–political and economic considerations can also be used to rank the nodes to be removed in telecommunication networks [MMCM12]. So, in order to identify the most relevant nodes, centrality metrics are considered to select the nodes to be eliminated in the targeted attack.

It is possible to determine the node more sensitive where an attack may produce greatest damage [SSYS10]. The network structure produces an effect of targeting nodes, after its removal, and measures other metrics with potential importance beyond node degree or betweenness [IKSW13]. However, when one network interacts with another, the critical parts of a network may change due to failure spreading between them. Thus, identifying the robustness change when two independent networks interact worth it, and it is shown in the next section. Moreover, the model for interdependent networks is adopted from the ER model, and is called ER–ER model.

Interdependent Link Pattern and Matrix

In this section, we analyze the interdependent links, their representation in a matrix and their pattern that connects the network A with B. Recently [CD15] showed that in order to analyze the impact of a targeted attacks in the robustness of A when an attack occurs in B, and vice versa, three link patterns are considered for the interdependency matrix D:

High centrality The interdependency matrix is based on *high centrality*, denoted by D_h, if the link dependency links is described by $S_i \leftrightarrow T_i$. It means, the node with the highest centrality measure in the graph A is linked to the node with the highest centrality measure in the graph B. Due to the number of nodes is the same in both graphs, there is a one–to–one correspondence between nodes of A and B.

Low centrality The interdependency matrix is based on *low centrality*, denoted by D_l, if the link dependency is described by $S_i \leftrightarrow T_{N-i}$. It means, the node with the highest centrality measure in the graph A is linked to the node with the lowest centrality measure in the graph B, and vice versa. Due to the number of nodes is the same in both graphs, there is also a one–to–one correspondence between nodes of A and B.

Random The interdependency matrix is based on *random* interdependency, denoted by D_r, if the link dependency $S_i \leftrightarrow T_j$ as a randomly one–to–one correspondence between nodes in A and B. It means, nodes are connected randomly without considering their centrality measure. Thus, a random pattern model is generated for the interdependency matrix.

Cascading failures in Interconnected networks

Cascading failures are the result of removing a fraction $1-p$ of nodes in a network. Under the supposition of having an interconnection link for each node in A with only a node in B. After removing a fraction $1-p$ of nodes in network A, the edges associated to them are also eliminated, and affects the dependent nodes in the network B. In the meanwhile, nodes affected in B are eliminated and propagates the failure to nodes and links in the network B. Figure 4.4 shows an example. Node $A1$ fails, by an attack or breakdown, and its attached links in the network, with $A2$ and $A3$, are disabled. Moreover, the failure spreads the damage to the node $B1$ in the interdependent network B. Node $B1$ fails afterwards and its links with $B2$ and $B3$ are also affected.

4.2.3 Targeted Attacks and Robustness in Interdependent Networks

This section presents an analysis of the impact of targeted attacks in interdependent networks. We model a targeted attack as a series of attacks over the most important nodes according to a centrality metric. This work is different from the previous analysis of cascading failures in interdependent networks caused by random failures [Bul+10]. Huang et al. studied the effect of targeted attacks based on node degree in interdependent networks and they

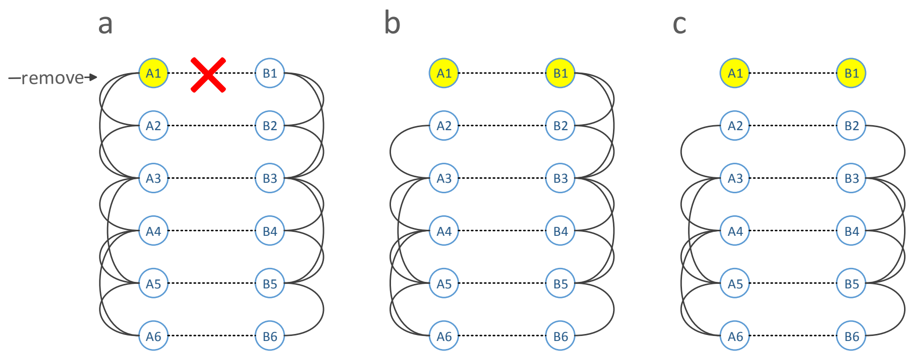


Figure 4.4: Cascading failures in interdependent networks. a) A1 node is attacked or removed. b) failure is propagated to its corresponding node B1. c) network B suffers a cascading failure.

compare the robustness for interdependent and single networks [Hua+11]. Those *important* nodes are identified using the centrality measure, betweenness in this case, and are removed following the sequence [Ken+14].

Our interdependency model describes an attack over one node in A and its respective damage only on the node that is directly connected in B, without spreading to others located in B. We use ER random networks because they have shown high vulnerability to series of targeted attacks based on nodal betweenness centrality b_c [SSYS10] and [IKSW13].

The purpose of this work is twofold, first to analyze the impact of interconnecting two ER telecommunication networks, which are highly vulnerable to targeted attacks based on nodal betweenness centrality (b_c); and also to minimize the impact of this kind of attack in the robustness of this ER-ER interdependent network by selecting the best *interdependency matrix*.

4.2.4 Sequential targeted attacks in Interdependent Networks

A sequential targeted attack can be generated using centrality measures as those shown in § 4.1.2. There are two modes of sequential targeted attacks, the first one consists in generating *a priori* ordered list based on a centrality measure and systematically eliminating nodes following the list. The second one consists in recalculating the centrality measures after a node (or link) is eliminated and then restarting the attack. Also, attacks can be performed following the ordered list or by assignation of probabilities for being removed. Huang et al. developed a mathematical framework for robustness of interdependent networks. This framework assigns probability of removing a node based on its degree. They also analyzed multiples cases where the *protection* (they have less probability to be attacked) varies from low to high degree nodes [Hua+11]. Others like Du et al. calculated the robustness for interdependent networks by connectivity and dependency links under targeted attacks. They used three measures to simulate sequential targeted attacks: internal degree only, external degree only, and both internal and external degree [DDTL16].

Consider a sequential attack based on higher degree in figure 4.5. There are interdependent two interdependent networks A and B with their respective nodes and links. Interdependent links are the doted lines, and there is an interdependent connection per node. a) the node with the highest centrality measure is selected for being attacked, in this case the node $A3$. b) all links connected to the attacked node are removed, and the attack is propagated to its corresponding node $B3$. c) node in B suffers the attack and propagates the damage. Links related to $B3$ are also eliminated. d) next node is selected to be attacked according to the highest measure, in this case $A4$. e) attack is propagated to its corresponding node $B4$. f) $B4$ suffers the attack and propagates the damage.

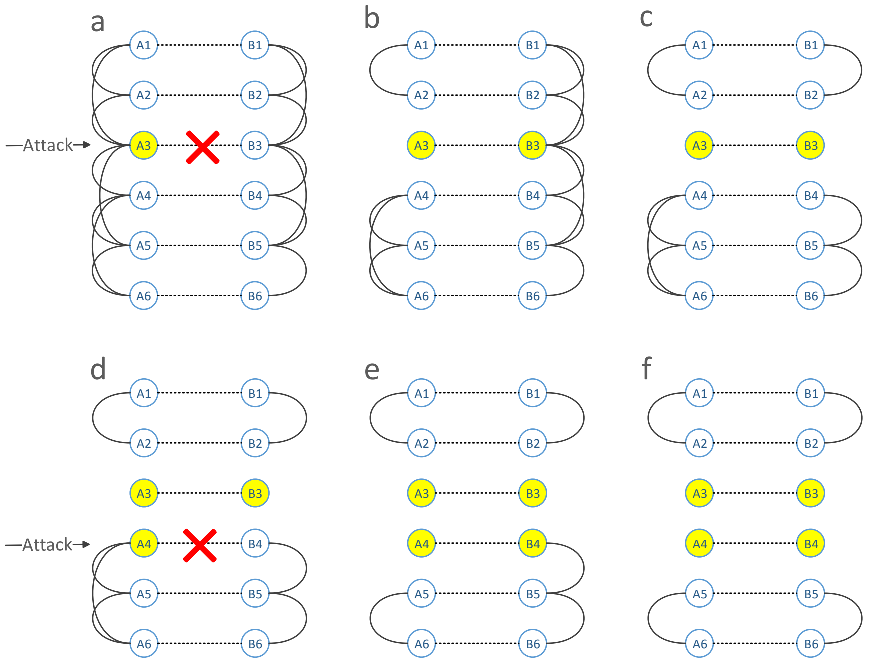


Figure 4.5: Sequential target attack in interdependent networks.

Consider nodes for each network $N_A = \{n_{A1}, \ldots, n_{Ak}\}$, $N_B = \{n_{B1}, \ldots, n_{Bk}\}$, their corresponding set of links L_A and L_B, and the interdependency link set $L_I = \{(n_{Ai}, n_{Bi}) | 1 \leq i \leq k\}$. The complete graph G^0 at initial point is in equation (4.3).

$$G^0 = (N_A \cup N_B, L_A \cup L_B \cup L_I) \tag{4.3}$$

Then, nodes from A are sorted and form the sequence of vulnerable nodes $V := (v_1, \ldots, v_k) | v_i \in N_A,\ \delta(v_1) \geq \delta(v_2) \geq \ldots \geq \delta(v_k)$ where the ordering function is the centrality value $\delta(v_i)$, in this case the highest degree. The attack is done by *removing the node with highest vulnerability* (centrality value) v_i along with its links. Fail propagation is the induced damage on the interdependent nodes of v_i in network B. The set inter$(v_i) = \{x : x \in B, (v_i, x) \in L_I\}$ denotes the nodes in the network B which have interdependent links with the node v_i. Recall that link set of a node n in the network B is $L_B(n)$ according to § 3.1.1. Now the resulting graph G^1 after the first attack is described in equation 4.4. Nodes of the resulting graph are in expression 4.4b, and links in 4.4c.

$$G^1 = (N^1, L^1) \tag{4.4a}$$

$$N^1 = N_A \setminus v_i \cup N_B \setminus \text{inter}(v_i) \tag{4.4b}$$

$$L^1 = L_A \setminus L_A(v_i) \cup L_B \setminus L_B(\text{inter}(v_i)) \cup L_I \setminus (v_i, \text{inter}(v_i)) \tag{4.4c}$$

This process of *recalculating* centrality measures and removing the highest ranked element is repeated until the desired fraction of elements has been removed.

In summary, sequential targeted attacks are used to model some failure scenarios in telecommunication networks. For example, the most vulnerable routers of a backbone network can be identified to protect the network's function. If a router fails, its functions can be distributed to any one router in the network. Then, the failure of one router will affect the importance of the remaining ones. So, the sequential targeted attack is appropriate to model network vulnerability.

4.3 Targeted Attacks on ER networks: Case Study and Results

We model the interdependent networks as two Erdös–Rényi random graphs (ER), because it is the most common model for the physical topologies from the AS perspective, and allows us to compare with a standard model. ER random model describes the existence of an edge between any pair of nodes that is uniformly distributed at random. For a graph $G = (V, E)$ with n vertices and m edges, the maximum number of possible edges is $n(n-1)/2$. Then, the probability of having an edge if it is randomly distributed is $p = 2m/(n(n-1))$ [Lew09].

4.3.1 Models of Random Network

In the ER model a graph is $G(n, p)$ with n nodes and probability p of connection between any two nodes. This is a discrete case where the probability of choosing d vertices from n is $\binom{n}{d}$, and the probability that those vertices have edges is p^d; then $\binom{n}{d}p^d$ is the probability that a vertex has edges to d other nodes. However, that vertex must not have edges to the rest $n-d$ nodes, it occurs with probability $(1-p)^{n-d}$. Therefore, the probability of a vertex having d edges, or degree d is:

$$\Pr(d) = \binom{n}{d} p^d (1-p)^{n-d} \tag{4.5}$$

And the probability of having a graph with m edges is

$$\Pr(m) = \binom{\binom{n}{2}}{m} p^m (1-p)^{\binom{n}{2}-m} \tag{4.6}$$

The mean value for m edges is

$$\langle m \rangle = \sum_{m=0}^{\binom{n}{2}} m\Pr(m) = \binom{n}{2} p \tag{4.7}$$

The mean degree of a graph $G(n, p)$ is

$$\langle k \rangle = \sum_{m=0}^{\binom{n}{2}} \frac{2m}{n} \Pr(m) = \frac{2}{n}\binom{n}{2} p = (n-1)p \tag{4.8}$$

The degree distribution is derived from Binomial distribution

$$\Pr(k) = \binom{n-1}{k} p^k (1-p)^{n-1-k}. \tag{4.9}$$

but if p is small, the binomial distribution is approximated to the Poisson distribution with parameter $\langle k \rangle$

$$\begin{aligned} \Pr(K = k) &\approx \frac{((n-1)p)^k}{k!} e^{-(n-1)p} \\ &\approx \frac{\langle k \rangle^k}{k!} e^{-\langle k \rangle}. \end{aligned}$$

Small–world Networks with small average distance between nodes are called small–world. The diameter of the network decreases as the logarithm of the network size. For example, ER networks hold this small–world characteristic. The Watt–Strogatz model, uses a ring topology where each node is connected to its closest $n/2$ neighbors, then edges are relocated with probability p_r to other vertices chosen at random. If $p_r = 1$ then the resulting topology is ER. The original topology is obtained if $p_r = 0$.

Scale–free Network has a topology which follows a power–law distribution of its node degree. This network model is used to describe large networks.

Exponential Networks Those are networks with a degree distribution that follows an exponential expression where β is the parameter that defines the distribution as follows:

$$\Pr(k) = \frac{1}{\beta} e^{\frac{k}{\beta}}. \tag{4.10}$$

4.3.2 Backbone Telecommunication Networks: Case Study

In this case study, we model two interdependent backbone telecommunication networks as two Erdös–Rényi (ER) graphs, and three interconnection patterns. We use the ER-ER topology which has two single network topologies generated from an ER random graph model with the same number of nodes ($|N| = 500$), and different probability of connection pc. The variation of connection probability pc implies a variation in the number of edges in the resulting graph, but holds the Poisson nodal degree distribution.

Single network analysis

We generate ER graphs with different pc values and measure graph properties such as the number of nodes ($|N|$), number of links ($|L|$), average nodal degree ($\langle k \rangle$), maximum degree (k_{max}), average shortest path length ($\langle l \rangle$), diameter (D) and assortativity coefficient (r). Topological properties of the used networks for this case are in table 4.1. We generate a set of graphs with pc as parameter, given that the probability of connection to have a Grand Component (GC) is at least $\ln n/n$; for $N = 500$ is 0.0125 and 0.5 is the reference value. We observe that networks exhibit assortative values close to zero $[-0.0069, 0.0028]$, the average nodal varies from 6.24 to 19.96, which denotes a dependency of bc (0.0125 to 0.04) , and small values for the average shortest path (3.59 to 2.38). The network diameter is also small, ranging from $D = 7$ for the smallest connection probability to $D = 44$ with the highest $pc = 0.04$ used in the test.

Table 4.1: Topological properties for simulated networks.

cp	L	N	$\langle k \rangle$	$k_{\max}$	$\langle l \rangle$	D	r
0.0125	1560	500	6.2409	14.6666	3.5923	6.9	-0.0018
0.0152	1912	500	7.648	16.9666	3.2735	5.9	-0.0029
0.0184	2293	500	9.1741	19.6	3.0362	5.2	-0.0026
0.0224	2802	500	11.2105	22.5666	2.8170	4.9	0.0028
0.0272	3397	500	13.5904	25.4333	2.6577	4.0	-0.0001
0.0330	4115	500	16.4616	29.7333	2.5257	4.0	-0.0085
0.0401	4992	500	19.9682	34.6	2.3870	3.9666	-0.0069
0.5	62340	500	249.3609	283.6	1.4972	2.0	-0.0043

For this case, we measure the robustness with the Average Two–Terminal Reliability (ATTR) metric, as shown in equation (4.11), where c is the number of network components, k_i is the number of nodes in the component i, and N is the number of nodes in the network.

$$\text{ATTR} = \frac{\sum_{i=1}^{c} k_i(k_i - 1)}{N(N-1)}. \tag{4.11}$$

Networks are created with a fixed number of nodes, $n = 500$, and the probability of two node connection cp varies from 0.0125 to 0.5. The percentage of failures is the fraction of nodes which are removed from the original network. ER networks are very vulnerable to sequential targeted attacks based on centrality, for example the betweenness centrality (bc). We generate multiple ER networks with the parameters showed in table 4.1, and face them to sequential (figure 4.6) and recalculated targeted attacks (figure 4.7). Figure 4.6 shows the robustness of a network (ATTR) of ER networks with different probability of connectivity under sequential targeted attack based on the *nodal betweenness.*

On figure 4.6, we can observe that robustness of ER networks during a sequence attack based on nodal betweenness depends on the connectivity probability (cp). Networks with

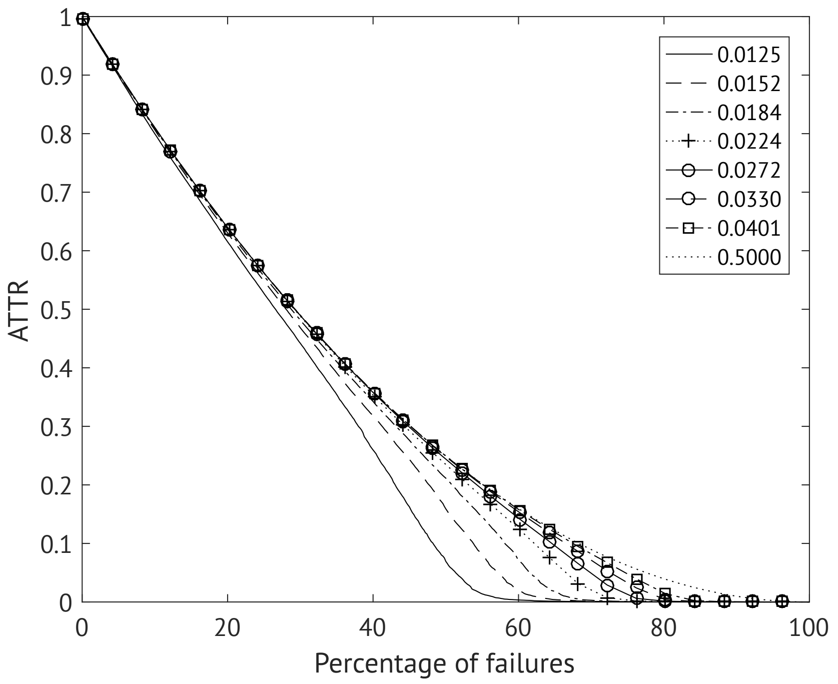


Figure 4.6: Robustness of ER–networks under sequential targeted attack.

higher *cp* are more reliable under sequence targeted attacks than those with lower *cp*. For this case, a network with *cp* = 0.0125 and 40% of failures has an ATTR measure near to 0.25, in contrast with ER–network with *cp* = 0.04 and the same 40% of failures has an ATTR of almost 0.4. It means that the connection probability affects the performance under sequential targeted attacks.

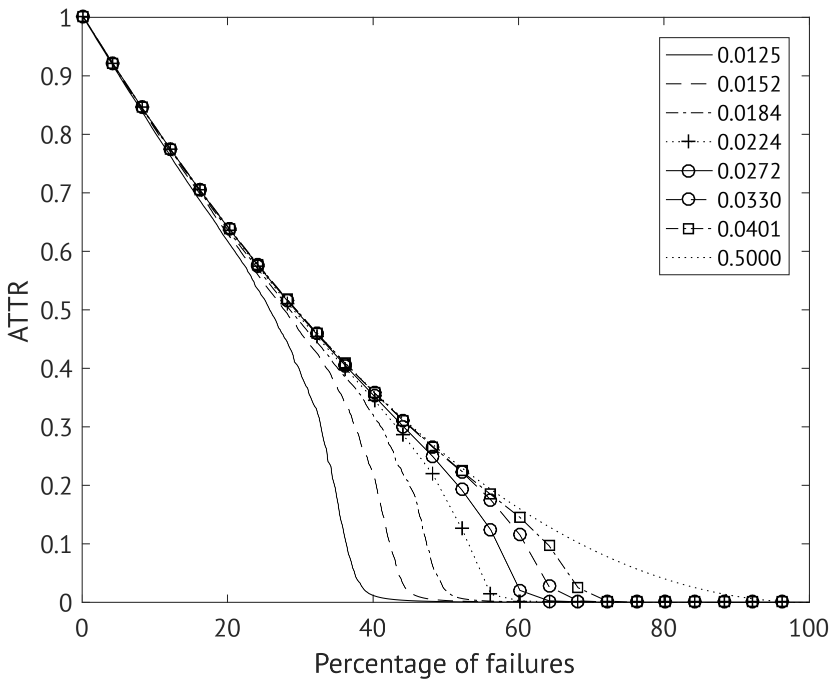


Figure 4.7: ATTR of ER networks against recalculated targeted attack.

Next simulation is also conclusive. We use networks with the parameters of the previous one, we face them to recalculated target attacks, and measure the robustness of the network. Remember the recalculated attack, once a node is removed because of its score, recalculates the centrality measurement, and sorts the vulnerable target again to identify the next node to be attacked. Network robustness under recalculated targeted attacks is shown in figure 4.7. In this figure, the reduction in terms of ATTR for fraction of failures from $30\% - 40\%$ is more accentuated than sequential targeted attack. A network with $cp = 0.0125$ and 40% of failures has a ATTR measure near to 0, instead an ER–network with $cp = 0.04$ and the same 40% of failures has an ATTR near to 0.35. Therefore, the connection probability deeply affects the performance under recalculated targeted attacks.

Targeted attacks in reverse order As interesting result, figure 4.10 shows the robustness for networks after sequential and recalculated attacks. However, the order of attack is the reverse to those shown in figures 4.6 and 4.7. In this case, the first attack is to the node with lower centrality measure, it means from the least important vertex to the highest.

Those results show that network robustness in reverse attack is independent of the connection probability for the generated random graphs, and it decreases equally for every network as shown in figure 4.8a. Alternatively, in the case of recalculated attacks, the behavior exhibits differentiation depending on the connection probability as figure 4.8b shows.

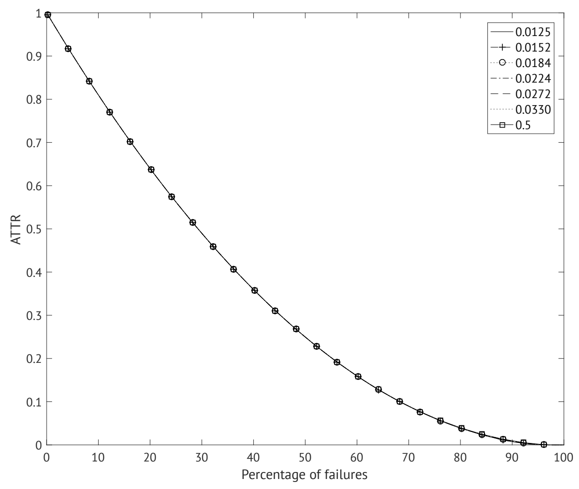
Interconnection link patterns

We analyze the impact of sequential and recalculated targeted attacks over two interconnected networks. We follow the same strategy used above, and compare ATTR measurements after sequential and recalculated attacks over networks with multiple connection probability cp. We present three interconnection patterns: high–high, high–low, and random connectivity.

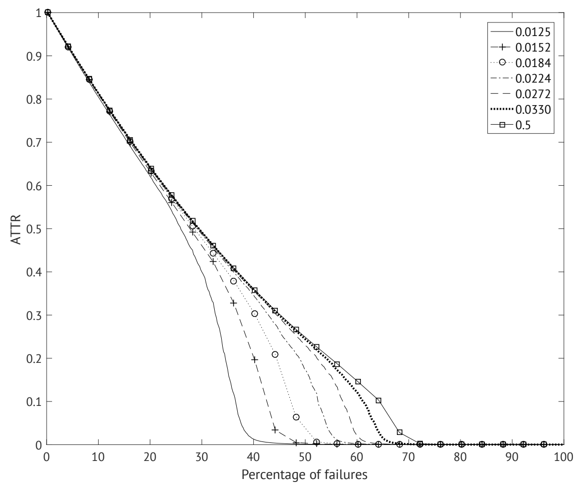
For our case study, both networks are created with the same number of nodes $|N| = 500$, and the connection probability varies from 0.0152 to 0.5. Then, we create three patterns for interconnection and evaluate the robustness (ATTR) under sequential and recalculated attack. For illustration, our examples are depicted for the network A, randomly created with connected probability $pA = 0.0125$, and the network B with connection probability pB which varies from 0.0152 to 0.5.

High–High Network nodes are *scored* and sorted after a centrality measurement. For this case, nodal betweenness. Then, we *wire* high centrality nodes from one network to high centrality nodes in the second network. The robustness of the interdependent networks, linked using the high–high pattern, after sequential and recalculated attacks is shown in figure 4.9.

We can observe how the robustness in terms of ATTR decreases under recalculated attack for both networks. As it was expected, the robustness of A decreases faster if the network



(a) Robustness of ER–networks under sequential targeted attack in reverse order.

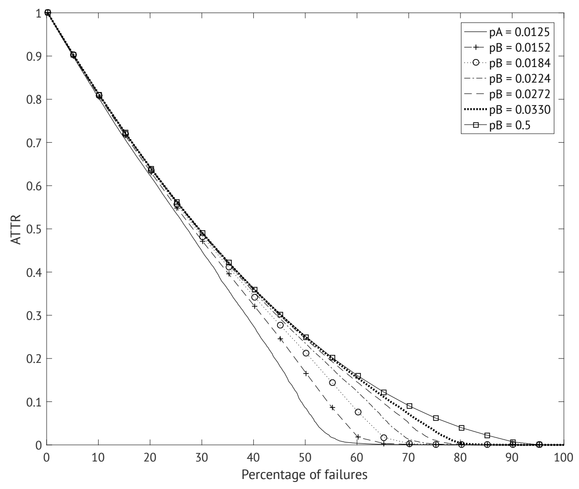


(b) Robustness of ER–networks under recalculated targeted attack in reverse order.

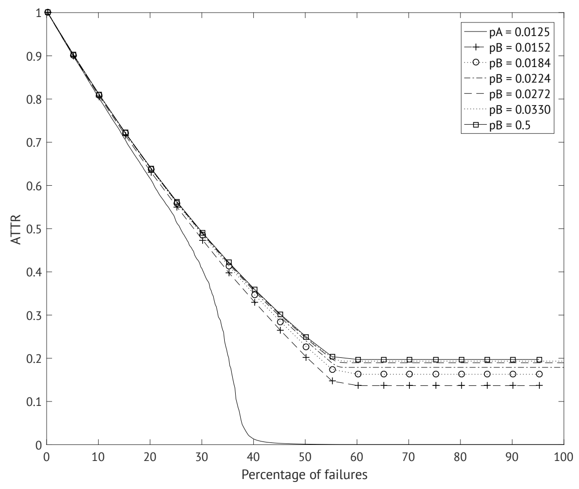
Figure 4.8: Average two–terminal reliability for sequence and recalculated attacks in reverse order.

faces a recalculated attack rather than if it faces a sequential attack. Moreover, the robustness of the interdependent network B also decreases in a similar way to an attack over a single network. However, the robustness behavior in the network B decreases much faster if the interdependent network faces a recalculated attack and its link pattern is high–high. For example, the robustness after a sequential attack is close to 0.45 for the 30% of node failure and close to 0.30 for 40% of node failure as shown in figure 4.9a. Instead, the robustness after a recalculated attack is close to 0.40 for the 30% of node failure and almost zero for 40% as shown in figure 4.9b. After 55% of failures the network is completely damaged, there are practically no vertices in the network A and the attack is not spread to B, then the ATTR remains the same.

High–Low In this pattern the nodes in graph A with high centrality measurement are connected to nodes with low centrality measurement in the graph B. Also, nodes are *scored*



(a) ATTR of ER networks against sequential targeted attack.



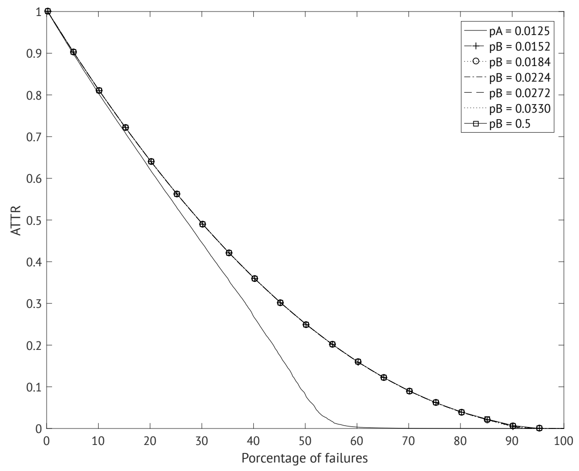
(b) ATTR of ER networks against recalculated targeted attack.

Figure 4.9: Average two–terminal reliability for sequence and recalculated attacks with high–low link pattern..

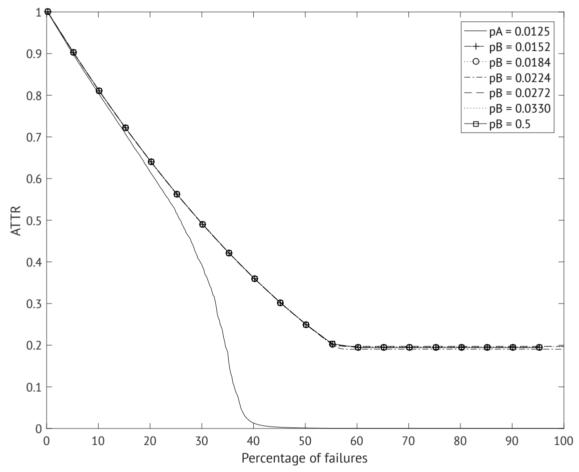
and sorted after a centrality measurement to interconnect both networks.

Like the results shown in figure 4.8a, there are not variations in robustness in the network B related to the connection probability figure 4.10a. The robustness of the interdependent network behaves in the same way as the single network under sequential targeted attack in reverse order. In the case of recalculated targeted attack, see figure 4.10b, the robustness in the network B follows the shape described in figure 4.9b but there are no variations after using multiple connection probabilities for the network B.

Random interconnection Finally, the third pattern is a random interconnection between two networks. This pattern holds the restriction of having at most one link per node on each network to interconnect to the other.

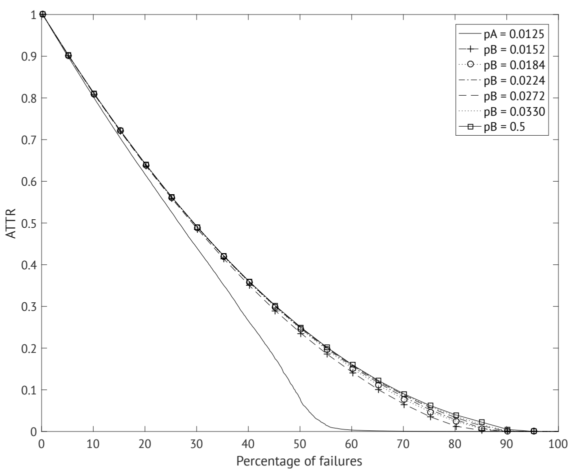


(a) ATTR of ER networks against sequential targeted attack.

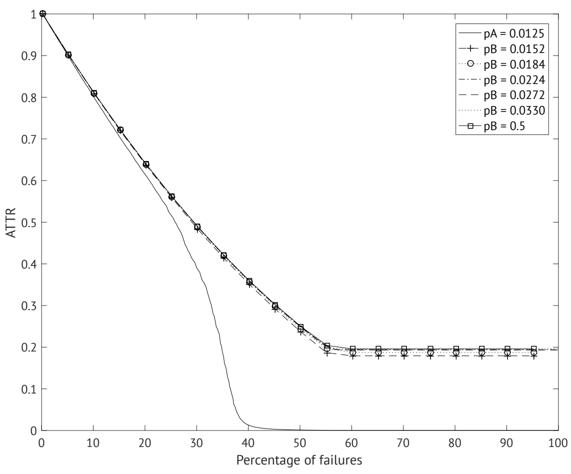


(b) ATTR of ER networks against recalculated targeted attack.

Figure 4.10: Average two–terminal reliability for sequence and recalculated attacks with high–low link pattern.



(a) ATTR of ER networks against sequential targeted attack.



(b) ATTR of ER networks against recalculated targeted attack.

Figure 4.11: Average two–terminal reliability for sequence and recalculated attacks with random link pattern.

4.4 Summary and Conclusions

In this chapter, we proposed to model the control and data planes of SDN as an interdependent network. This representation includes the interaction *links* between two planes. Also, we introduced the model of targeted attacks on interdependent networks to simulate how this type of attack affects the network. Then we evaluate the robustness using the ATTR measure over two–ER networks following the parameters shown in table 4.1. Then, those networks are *wired* to test three patters: high–high, high–low, and random between those two *planes*. The first network is attacked under sequential and recalculated targeted attack, nodes with higher betweenness centrality measurement are attacked first. The analysis is done over the robustness of the second network that is affected by the first network attack.

The pattern high–high shows the major dependency with the connection probability. In this case, the robustness of the dependent network is affected directly by its centrality measurement. On the other hand, if the pattern is high–low, the robustness in the dependent network does not depend on the connection probability. This is the optimistic case, where there is no dependency and this behavior occurs due to the pattern propagates the failures to those nodes with lower risk in the network $\boldsymbol{B}$, and coincides with the behavior of a single network. If the network is attacked with a recalculated process, the robustness for the network A dramatically decreases. Instead for the dependent network $\boldsymbol{B}$, its robustness measurement decreases with the same rate and it does not present too much variation compared with the sequential attack. For random link pattern the robustness behavior is similar to high–low pattern. It presents a small variation related to the connection probability $p\boldsymbol{B}$.

Recent literature proposed a design of interconnection links to increase the robustness and reduce the cascade effect of a random targeted attack [CD15], the next step on this direction is to propose optimization mechanisms that reduce the impact, according to a robustness measure. To achieve this goal, we require a mechanism that migrates network controllers, changes the link pattern according to the robustness evaluation, and responds to deliberate attacks. Moreover, we expect to reproduce and analyze multiple cases with different random network models such as the Watt–Strogatz, Scale–free or exponential models.

Chapter 5

Verification of Security Policies

Each problem that I solved became a rule which served afterwards to solve other problems.

René Descartes

This chapter presents our framework to verify firewall rules, based on predicate logic and satisfiability, called *FireWell*. FireWell is able to model firewall policies as formal predicates to validate, detect and prevent conflicts for SDN environments. FireWell uses a satisfiability solver based on Alloy[Jac03] which finds examples and counterexamples of implementations within a relational model. In addition, we present the FireWell implementation and test it using the Floodlight controller and a firewall application.

An advantage of SDN is that it allows us to use well–defined software expressions and predicates to regulate network behavior. Current SDN controllers, such as *Floodlight*, offer a framework to develop, test and run applications that control the network operation, including the firewall function. However, they are not able to validate firewall policies, detect conflicts or avoids contradictory configurations on network devices. Some compilers only detect conflicts in a subset of policies: hence, it cannot detect conflicts among different sunsets of policies, for example, it would not detect a conflict between a security policy and a permission policy.

In the heart of FireWell there is a verification engine that interprets firewall rules written by the administrator for a SDN controller and helps to validate rules and identify conflicts. FireWell interprets network topology as logical abstractions in Alloy. Then, SDN firewall policies are translated into logical *predicates* in Alloy that establish a relational model. Later, Alloy validates these rules using its satisfiability solver (SAT), and returns logical conflicts. Later, FireWell uses these results to report the set of rules that conflict. In addition, we model sets to overcome user-group relations, use strings of elements to detail priorities and verify the list of predicates using Alloy. Finally, we implement this compiler and show how we face these limitations.

The model of firewall rules, conflicts, and an example scenario are on § 5.2 The FireWell description, design features, architecture and implementation details are in § 5.3. Section

§ 5.3.3 explains how FireWell translates firewall rules into Alloy, and illustrates the detection and interpretation of policy conflicts. Finally, the discussion of advantages and limitations of our approach and conclusions are in §§ 5.4 and 5.5.

5.1 From Middlebox to Network Application

The management of security in network infrastructure is a complex duty. Network security involves applications, services and access control which mainly relies on firewall configuration and operation. Recall that 50% to 80% of infrastructure failure is caused by misconfiguration [AWY08; KD12]. Specifically, configuring network security policies such as firewall rules is an error prone task. Automatic tools for management aim to reduce the time required to configure network devices and the problems caused by configuration errors.

Configuration of firewalls is a well-known problem [Woo10]. Traditional firewall middlebox suffers from several drawbacks such as 1) contradictory rules that define different actions for the same packet (and must be arranged or prioritized to apply one over the rest); 2) inconsistent rules affecting the same traffic but defined on different firewalls (and must be redefined according to a global *policy*); 3) pointless and impractical rules could be included increasing complexity but do not add functionality, on the contrary, they increase the processing time and the complexity of detecting configuration errors; and 4) policies in several middle–boxes lack of consistency, it means, all firewalls must process the same packet with the same *policy*.

5.1.1 Firewall functionality

Firewall is a middlebox that controls and monitors the network against unauthorized traffic. It is widely used not only to restrict external and private traffic, but also to grant access to applications and control user behavior. A firewall filters packets according to a *security policy* which is represented as a list of *rules* to decide which actions apply to a packet. A rule has the structure $\langle predicate \rangle \rightarrow \langle action \rangle$ that means *if predicate is valid, then the action takes place.* A predicate is a logical expression evaluated over a set of packet fields such as protocols, source and destination addresses and ports. The action is a forwarding decision, commonly *accept* or *deny.* A packet matches a rule if its header satisfies the predicate. Rules are sorted, therefore if a packet header matches a rule predicate, the corresponding action is executed, and the following rules are discarded. To find conflicts on firewall configuration, it is necessary to analyze the previous $(i - 1)$ rules and evaluate them against the i^{th} rule.

Here in SDN environment, the firewall is an application running on the controller server. Now firewall operation is also distributed across the switches. The interesting issue, in this scenario, is *how to validate if firewall rules conflict.* To solve this problem, we first introduce how to identify conflicts in the SDN firewall application.

5.1.2 Firewall and Security rules in SDN

Recall that SDN separates control and data planes § 2.1. This functional separation gives major flexibility, permits network administrators to create software routines, and allows dynamic configurations. Now, applications operate and control the network. As with many other network functionalities, firewall is adopted as an application for SDN controller.

SDN introduces a different scenario for firewall functionality. The firewall application can modify the forwarding table of a device. This flexibility increases the risk of firewall–rule conflicts unlike the pre–SDN scenarios. Detecting and preventing conflicts in firewall rules for SDN is different than firewall-middlebox because, as any other SDN application, the output of the control plane is a set of forwarding rules for datapaths. Firewall functionality and other rules are distributed in multiple datapaths. The verification of those forwarding rules must guarantee the inter–operation of multiple devices and network applications, not only the firewall rules. The main goal is ensuring that all involved network devices satisfy the high–level policies.

5.1.3 FireWell Proposal

We propose to verify firewall rules and forwarding rules from datapaths as product of this SDN application. The verification engine is based on Alloy and finds examples (or counterexamples) of networks that exhibit a topology and support a set of constraints of firewall rules. Alloy[a] is a first order relational language to model and explore solutions for a formulation based on predicates. In our proposal, the network topology, traffic flows and policies are written as predicates and then are evaluated. Alloy has demonstrated being suitable to solve that kind of satisfaction problems.

Network security is a field that uses logic verification in extensible way [APS14]. For instance, access control mechanisms, such as RBAC, were verified to find inconsistencies and contradictions using Alloy [PSS11]. We propose to use Alloy, as a formal language, to define and analyze models based on relational first–order logic. It has been used to analyze inconsistencies in rules and policies in multiple domains [PSS11]. Using Alloy, we can translate specifications in PPL into a relational model and determine if the set of network policies can be implemented in a concrete network topology without conflicts.

The *Path-Based Policy Language* (PPL) is a well-known language for Policy-based network management. Using PPL, network administrators can specify the set of rules about how the network must deal with specific types of traffic, and then use a compiler to transform these rules into technology-level instructions. Existing PPL compilers [SLX01][Guv03] perform validations on policies to detect conflicts such as contradicting rules for the same network segment. However, these compilers only support a subset of the language, and cannot detect conflicts related to traffic priority, neither user–groups administration.

In contrast to configuration verifiers, *Policy–Based Management* is a tool to simplify network administration and support the creation of error-free configurations. The essential concept is that managers write a set of *well–formed* policies (at business-level) in an unambiguous language, that are later translated into operational parameters (at technological-level) for each entity in the corresponding network [Str03].

[a]http://alloy.mit.edu/alloy/

The network community has been struggling for years about how to manage networks, supporting business demands and preventing configuration errors. Some approaches are focused on configuring and monitoring each network device using centralized applications and management protocols such as SNMP. Other approaches, such as VeriFlow [Khu+13], are aimed at checking the configuration of each network device and detecting conflicts, inconsistencies and bugs. In contrast, *Policy-based Network Management* focuses on the specification of the intended behavior of a network and the automatic configuration of each one of its elements [Ste+99]. It simplifies the administration and supports the creation of error–free configurations. In Policy-based Management, the essential concept is the *well–form* policy. This policy is written by managers in an unambiguous language, and later translated into operational parameters (configuration scripts) for each entity in the corresponding network [Str03].

Alloy is a first-order formal language used to model and explore solutions for a formulation based on predicates. It has been used to analyze inconsistencies in rules and policies in multiple domains. For instance, access control mechanisms such as RBAC were verified to find inconsistency and contradictions using Alloy [PSS11]. Recently, it was used to model general network behavior to find security violations [MLCD14]. Using Alloy, we can translate firewall rules into a relational model and determine if the set of firewall policies are valid over a given network topology. Moreover, it finds examples (or counterexamples) of traffic flows that satisfy (or violate) the policy.

The administration of network demands automated tools to simplify the network configuration. Managers call for mechanisms to check and validate network configurations that guarantee the achievement of its goals. Moreover, a forecast engine would be helpful to test configurations before being applied. Policy-based Management could be an important method to achieve this goal.

5.2 Firewall model, rules and conflicts

Traditional firewall middlebox is placed in the route of flow paths for filtering traffic. The decision of a firewall corresponds to the first matching in the listed rules; thus, rules are sorted. At the end of the list there is an implicit (or explicit) *denying* action that blocks all the packets that do not match any previous rule. Suppose a small network with two subnets, 140.192.37.0/24 for clients, 161.120.33.40 for the server (HTTP, FTP, and DNS), and an Internet gateway. A typical rule list that describes this example is shown in table 5.1. The infrastructure under the SDN paradigm used for this example is shown in figure 5.1. The rule list blocks the HHTP service (port 80) to the host 140.192.37.20, but allows it accesses to the rest of the subnet and Internet. FTP service is denied for 140.192.37.30 but available for other addresses. Similarly, DNS service is allowed for all addresses.

5.2.1 Filters and headers

We model the packet header as a set of fields $h = \{h_1, h_2, \ldots, h_l\}$ that represents addressable elements on a packet. A field h_i defines a specific value or a range on its domain. A filter f

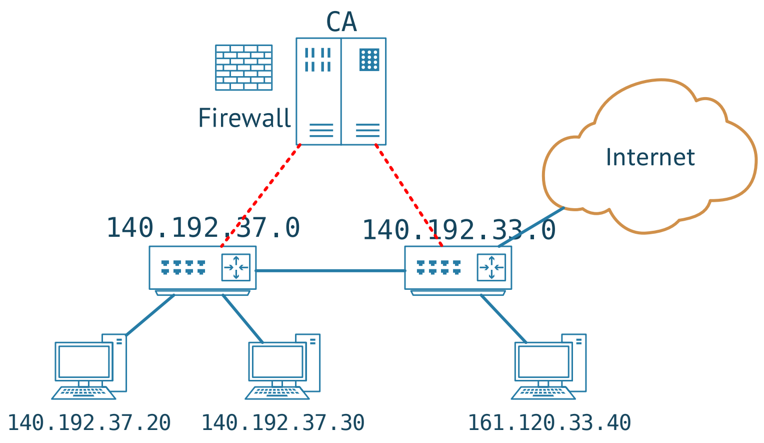


Figure 5.1: Example scenario is adapted to SDN paradigm. Before, there was a typical firewall in the middle of three networks and two routers. Now, the scenario has three switches, a controller server, and firewall application.

Table 5.1: Example of firewall rules extracted from [ASH04a].

No	Proto	Src Addr	Src Port	Dst Addr	Dst Port	Action
1	tcp	140.192.37.20	any	*.*.*.*	80	deny
2	tcp	140.192.37.*	any	*.*.*.*	80	accept
3	tcp	*.*.*.*	any	161.120.33.40	80	accept
4	tcp	140.192.37.*	any	161.120.33.40	80	deny
5	tcp	140.192.37.30	any	*.*.*.*	21	deny
6	tcp	140.192.37.*	any	*.*.*.*	21	accept
7	tcp	140.192.37.*	any	161.120.33.40	21	accept
8	tcp	*.*.*.*	any	*.*.*.*	any	deny
9	udp	140.192.37.*	any	161.120.33.40	53	accept
10	udp	*.*.*.*	any	161.120.33.40	53	accept
11	udp	140.192.37.*	any	161.120.35.*	any	accept
12	udp	*.*.*.*	any	*.*.*.*	any	deny

is a set of packet–header fields $f = \{f_1, f_2, \ldots, f_l\}$ where each element f_i represents values, boundaries or range of the element h_i. Both, filter and header must have the same format and be represented in the same domain. For example, $h = (proto, a_s, p_s, a_d, p_d)$ represents a packet header, where *proto* is the protocol (commonly TCP or UDP), a_s and a_d are source and destination IP addresses, p_s and p_d are source and destination ports.

A rule is defined by the tuple $r = (i, f, action)$, where i is an index that orders and prioritizes a set of rules, f is the filter, and an $action \in \mathrm{A} = \{\mathrm{Allow}, \mathrm{Deny}\}$ which is the operation executed by the firewall if the header matches the filter. The set R contains all possible rules.

In addition, the notation $index : \mathrm{R} \to \mathbb{N}$ is the function that returns the priority of a rule r; $\mathrm{filter} : \mathrm{R} \to \mathrm{F}$ returns the filter applied by r; and, $\mathrm{action} : \mathrm{R} \to \mathrm{A}$ gives the action executed by r. Other useful function is $match : (\mathrm{H}, \mathrm{F}) \to \{\mathrm{TRUE}, \mathrm{FALSE}\}$ that denotes if a header h

matches the filter f if:

$$\text{match}(h, f) = \bigwedge_{i=1}^{l} h_i \in f_i \tag{5.1}$$

5.2.2 Filter fields and relations

Since filter elements are comparable to each other, four relations are defined: subset, superset, equality, and disjunction, $R_{field} = \{\subset, \supset, =, \ominus\}$. Given two filters f^a and f^b, the relations over field's values are denoted as follows:

Table 5.2: Relations for packet headers fields.

$f_i^a \subset f_i^b$	values of f_i^a are included into f_i^b
$f_i^a \supset f_i^b$	values of f_i^b are included into f_i^a
$f_i^a = f_i^b$	elements of f_i^a are the same as f_i^b
$f_i^a \ominus f_i^b$	$f_i^a \cap f_i^b = \emptyset$, they are completely disjoint.

Definition 5.1 (Quantifier function)**.** *Quantifies the number of fields in f^a and f^b that satisfy the relation r. $Q : (f^a, f^b, rel) \to \mathbb{N}, rel \in R_{field}$.*

This quantifier function describes how many elements of two filters have a given relation.

5.2.3 Filter relations

Our model uses the relation set cardinality to get information about two filters. The number of valid relations over filter fields in the relations set $R_{filter} := \{equivalent, inclusive, correlated, disjoint\}$ allows to identify if two filters are inclusive, correlated, disjoint, or equivalent.

Two filters f^a and f^b are *correlated* if all fields on f^a are equals, subset, or superset of f^b. All fields on f^a are equals, subset, or superset of f^b.

$$Q(f^a, f^b, \subset) \geq 1$$
$$Q(f^a, f^b, \supset) \geq 1$$
$$\textstyle\sum_{rel \in \{=, \subset, \supset\}} Q(f^a, f^b, rel) = l$$

Two filters f^a and f^b are said *disjoint* if no field on a filter has a relation different of disjoint. $\forall i : f_i^a \ominus f_i^b$. Two filters f^a and f^b are equivalent if they exactly match the same header. $\forall i : f_i^a = f_i^b$. A filter f^a is said *inclusive* in f^b if a header that matches the first filter also matches the second one. $\forall i : f_i^a = f_i^b \vee f_i^a \subset f_i^b$.

Definition 5.2 (Filter-Relation function)**.** *Denotes the relation between two filters f^a and f^b, $f_{RFilter} : (f^a, f^b) \to R_{filter}$.*

5.2.4 Conflicts in firewall rules

There are two kinds of firewall conflicts: rule conflict and policy disagreement. A rule conflict refers to rules that one contradicts the other. Policy disagreement are rules that may be consistent but contradict the security policy. Particularly, if a set of rules contravene previous rules it says a *rule conflict.* On the other hand, if a set of rules dispute the policy, it is called a *policy violation.* In general, a firewall policy has a blacklist and a whitelist to enforce a goal.

According to Al–Shaer [AS14] and Hamed [ASH04a], there are four kinds of rule conflicts after misconfiguration.

Shadowing : a rule r_y is shadowed if there is a previous rule r_x that matches the same header but has different action. e.g. table 5.1 shows a rule set, where the rule 4 is shadowed by the rule 3.

Correlation : two rules r_x and r_y are correlated if some headers that match r_x also match r_y, but those rules have different actions. e.g. Rules 1 and 3 are correlated.

Redundancy : two rules are redundant if both perform the same action over the same packet header. e.g. rules 6 and 7 are redundant.

Generalization : a rule that matches r_x is a particular case of another matching r_y, but they perform different actions. e.g. rule 2 is a generalization of rule 1.

Although generalization is not always considered a configuration failure, it is used to exclude a header subset from a more general action, but it is considered an error prone activity. Sometimes administrators need to create specific exceptions, e.g. allowing a route from a specific terminal within a group and denying the other, they use generalization rules to allow a route and deny for every else.

Note two interesting details: in the shadowing model, the relation between (r_y, r_x) as *inclusive* means that the filters of r_y are subsets of the filter (r_x). In redundant policies indexation is important; r_y is included into r_x means that some headers will match the rule r_y after non–matching r_x headers.

A policy violation is a more interesting case of conflict. A set of firewall rules can be consistent, free of rule conflicts, but without fulfill the policy. In SDN, a datapath can change the packet header hence a corrupted application can include packet modification to circumvent the security policy. Then the verification not only has to examine the set of rules searching for conflicting rules but also the security policy should be contrasted against the set of firewall rules and determine if the set of rules achieve (or not contradicts) the policy.

5.3 FireWell: Firewall-rules Well-formed

FireWell is a framework based on predicate logic and able to detect and prevent conflicts in firewall rules over a SDN environment. FireWell connects to the controller and gets the network information[b]. The obtained information includes topology, firewall–rule set, and flow-table from devices. Then, FireWell maps forwarding rules to *well-defined* logical expressions, which are evaluated to verify satisfiability. Finally, FireWell interprets the SATsolver results. If the

[b]Connection with the controller by its RESTlet interface, http://restlet.com/

resulting predicate satisfies the configuration, the firewall–rule set does not have conflicts nor inconsistencies.

5.3.1 Model in Predicate Relational Language

FireWell is based on a formal model of policies, topologies (nodes and links), and forwarding rules. Then the formal model is written in Alloy which is a first-order relational language, and a tool to explore and find an instance that satisfies the formulation [Jac03]. As it was used in previous chapter § 3.3.1, it allows us to analyze inconsistencies in rules and policies in multiple domains [PSS11].

Remember that Alloy model comprises *facts*, and *assertions*. Facts are general assumptions always evaluated true, and assertions are expressions that will be checked. Also Alloy uses a constraint–solver to check the assertions within a scope, an upper bound of the number of elements of each type (set) included in the model. If the assertion is not satisfied, Alloy analyzer returns a counterexample. Using Alloy, we can transform specifications from Floodlight to a relational model and determine if the set of rules can be implemented in a concrete network topology without conflicts.

Network abstraction

Network topology is described as a set of nodes, and the set of links is a relation of nodes. The security policy is written as facts and assertions to validate the absence of specific conflicts. FireWell uses the same network topology abstraction defined in § 3.3.2. There, *node* is the base signature, and the set of *links* in the topology is a relation, denoted by $links : node \rightarrow node$.

Firewall Rule Model

Following the definition created in § 5.2.1, five header fields are created to formalize the filter: source and destination addresses, source and destination Ports, and Protocol. Hence, four signatures are created to describe a firewall rule: Protocol, Address, Port, and Action. The *Element* definition is a sequence that describes the index number.

Firewall conflicts model

The model implementation of relations between header fields is showed in § 5.2.2. This model is able to return data about two types of relations between filters; namely, the number of fields that satisfy the relation (the quantifier function) and the kind of relation of relation between two filters (the filter–relation functions). See definitions on definition 5.1 and definition 5.2. Some additional functions are $index(r)$ that returns the priority of r; $filter(r)$ that returns the filter applied by r; and $action(r)$ that gives the action executed by r. Now, we use the logical relations showed in § 5.2.4 to model rule conflicts as follows.

Shadowing r_x shadows r_y if $index(r_x) < index(r_y)$, $R_{filter}(filter(r_y, r_x)) \in \{equivalent, inclusive\}$, and $action(r_x) \neq action(r_y)$.

Correlation r_x correlates r_y if $index(r_x) < index(r_y)$, $R_{filter}(r_x, r_y) = correlated$, and $action(r_x) \neq action(r_y)$.

Redundancy r_x and r_y are redundant if $index(r_x) < index(r_y)$, $R_{filter}(r_y, r_x) \in \{equivalent, inclusive\}$, but $action(r_x) = action(r_y)$.

Generalization r_y is a generalization of r_x if $index(r_x) < index(r_y)$, $R_{filter}(r_x, r_y) \in \{equivalent, inclusive\}$, and $action(r_x) \neq action(r_y)$.

5.3.2 Finding conflicting rules

Verification by SATisfiability identifies the set of conflicting rules in this model. Using the SAT problem definition, as presented in § 2.2.2, the problem of detecting conflicts is reformulated as a SAT problem where rules, flows and constraints are written as propositional logic formulae. Moreover, the model defines a semantic function, like those described in § 3.1.6, to identify the minimal set of rules that makes the model satisfiable.

Conflict semantics

In order to define firewall functionality as a set of rules and then identify conflicts or violations, we follow the guidelines of Harel and Rumpe [HR04] to specify a modeling language L describing the *syntactic domain* $\mathcal{L}_L$, the *semantic domain* $\mathcal{S}_L$ and the *semantic function* $\mathcal{M}_L : \mathcal{L}_L \to \mathcal{S}_L$(traditionally written $[\![\cdot]\!]_L$). The semantic function over the complete set of rules $[\![\mathrm{R}]\!] = \Omega$ is the set of conflicts of all rules in the firewall configuration. A set of rules ρ is valid if its semantic function does not conflict and satisfies security policy $[\![\rho]\!] = \emptyset$.

Definition 5.3 (Conflict Semantic Function)**.** *A conflict semantic $[\![\rho]\!]$ of a set of rules $\rho \subseteq \mathrm{R}$, is the set of conflicts for a packet whose header matches ρ.*

$[\![\mathrm{R}]\!] = \Omega$ is the semantic of all firewall policies and produces the set of all conflicts in the firewall operation.

- The semantic function $[\![\rho]\!] | \rho \subseteq \mathrm{R}$ is the set of conflicts in the firewall configuration.
- A set of policies ρ is valid if it does not conflict and satisfies security policy $\Omega = \emptyset$.

Definition 5.4 (Rule Conflict)**.** *A rule conflict occurs when a set of rules presents any of the conflicts shown in § 5.3.1.*

Conflicts in rules are detected by the semantic function. Given two rules r_x and r_y, they do not conflict if $[\![r_x \cup r_y]\!]$ is empty, otherwise they are conflicting. The rule conflict has also a quantifier $[\![R]\!] \neq \emptyset$ i.e. when $\exists \rho \in R, Q(\rho) \neq 0$.

The set of rules ρ that creates conflicts is obtained using the minimal diagnosis function which returns the minimal set that produces conflicts.

Definition 5.5 (Minimal diagnosis)**.** *Given a set of policies $\rho \subseteq R$ such that $[\![R \setminus \rho]\!] = \emptyset$, the minimal set ρ is the minimal diagnosis.*

Definition 5.6 (Consistency)**.** *The relation of policies without any conflict. In other words, it is the ability of a firewall to support a set of policies which are free of misconfiguration nor ambiguity.*

Now we need a *verifier* able to find the semantic function $[\![\rho]\!]$, verify if that set is empty, and calculate the minimal diagnosis of that set. Then, we use FireWell to obtain a set of tuples that satisfy the policies (exactly the semantic function). If FireWell does not find any element (the set is empty), the policy set is valid.

For a set of rules ρ and a topology $\mathcal{G}$, the solution of the conjunctive SATisfiability problem S that satisfies the semantic function $[\![\rho, \mathcal{G}]\!] = \omega$ is the set of rules that creates conflicts. FireWell uses this model based on predicate logic to analyze propositional logic formulae and checks its satisfiability.

5.3.3 FireWell implementation

FireWell is a tool to validate, detect and prevent conflicts on firewall policies in SDN. FireWell is an independent application and can run remotely to the controller. It serves and consumes information with the controller. FireWell is configured to run in parallel with Alloy version 4 and the UNSAT solver of Kodkod[c]. This section shows the FireWell architecture and details its implementation that translates firewall policies, from SDN controller, for validating and detecting conflicts.

FireWell Architecture

As shown in figure 5.2 FireWell is an external application that serves and consumes information to/from the controller. FireWell has two interfaces: the first one is the RESTlet which establishes connection with the controller to get and install rules into the SDN firewall application; the second one is the interface with Alloy which parses rules on files that later are processed by the solver and its outputs are received through the same interface. FireWell also has two modules to process structures taken from the controller: the network topology parser and the firewall rule transformer. It comprises a module to parse structure into Alloy language, a module that abstracts the topology, a module that handles firewall rules, a module that creates trees of data–fields, and I/O interfaces with the RESTlet service and the interpreter.

The process through FireWell is as follows:

1) FireWell gets controller information, such as network topology, and the set of rules from the SDN firewall application.

2) FireWell builds its own representation of firewall rules.

    i) The module of topology creates the FireWell representation of nodes and links. This data structure is inspired in the formalization done in § 3.1.1.

    ii) Firewall rules are transformed into a set of fields trees to reduce the number of variables. This tree–data transformation allows to create sets of *symbols* that later are defined into predicate logic.

[c]This work uses the same methodology presented in chapter 3.

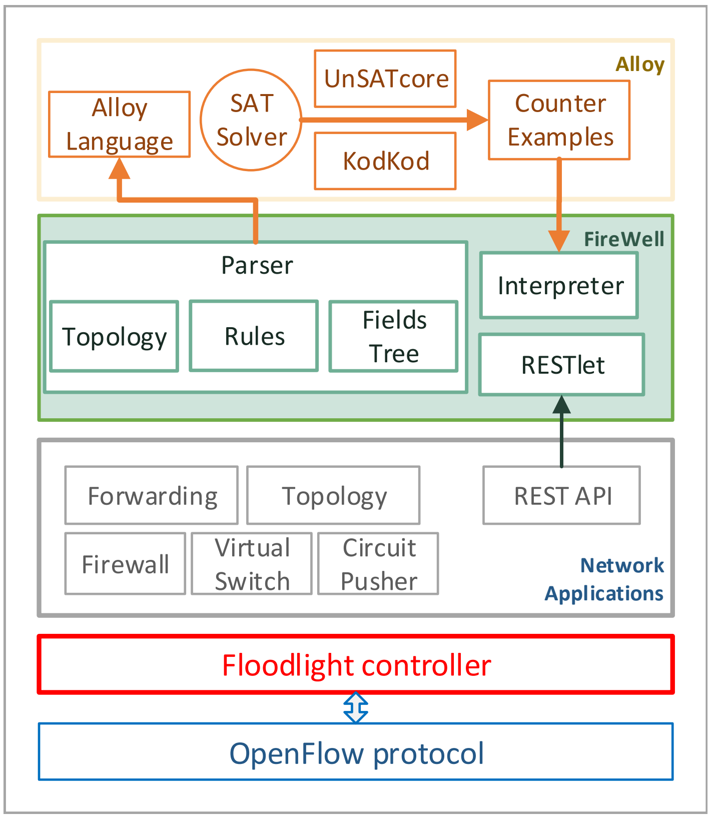


Figure 5.2: FireWell architecture. It gets topology information from Floodlight, creates the model and exchanges formulation with Alloy.

   iii) FireWell uses its *Parser* module to translate its rules into *well-formed* expressions.

3) Once rules are in logic expressions, FireWell writes an *.als* file which later will be load into Alloy. After that, the .als file is processed with the Alloy solver.

4) After the SAT solver with unsatisfiability procedure is executed, the list of counter examples, if they exist, are interpreted to identify firewall rule conflict or nodes which violate the security policy by counterexamples. Recall that a counterexample is an abstraction of conflicting rule or a policy violation.

5) Conflicts are reported to the network administrator. This report includes the firewall rule, its number, and the counterexample explanation, the set of contradicting rules or the policy circumvention.

RESTlet interface and Model Parser

FireWell, at first instance, has a RESTlet that gets information from the Floodlight controller. This information includes topology nodes and links, and the set of rules in the firewall application. Then, FireWell resolves nodes, links, and firewall rules, and builds its own representation on JAVA–data structures. Network topology is also represented on its structure and creates forwarding tables for each device with information available through the Floodlight controller.

Parsing the model has three steps. First, fundamental elements such as nodes, protocols, addresses, ports and actions are created as sets. The *parser* module writes rules, topology, paths, and flows into sets and relational logic expressions. In the second phase the parser writes trees for addresses, a sequence which defines priority, and rule instantiation as show in figure 5.3. Firewall rules are inspected. Fields from all rules are analyzed; FireWell parser identifies relations over fields and builds data–trees. This procedure maps variables and rule definitions into sets to reduce the amount of variables. Additionally, following the same model presented in § 5.3.1, FireWell creates the functions to classify filter–field relations, and rule conflicts. The identifier of each rule is represented as a sequence of elements used to illustrate priority. At the third phase, parser establishes the scope to run the model, and write the unsatisfiable cases. This output is recorded to a *.als* file which can also be run from the Alloy interface. Figure B.1 shows an example of the result the firewall model translated into Alloy language.

Floodlight [Big12] is a NOS and controller system used by FireWell because it supports upper applications and protocols, and translates their instructions to forwarding devices. At the lower level, OpenFlow [McK+08b] is the communication protocol that binds Floodlight in a centralized controller to forwarding devices, establishes standard interfaces, and executes applications, algorithms and protocols on the switches.

Tree optimization

An attribute of FireWell is the ability to reduce the number of variables that later will be evaluated in Alloy. A naive way to create sets and identify relations between their elements by creating all possible elements in a set, and then, calculate their relations. For example, a firewall rule can have IP ranges or subnets in the form 192.168.10.0/24 and then FireWell identifies field relations ($=, \subset, \supset, \ominus$) with another field, e.g. 192.168.10.0/25 and creates a tree of field relationships.

In this manner, FireWell creates a tree of relations between fields. In the case of IP addresses, the root of the tree is the wildcard ($*. * . * .*$) equivalent, because all addresses are subsets of the root. Then, FireWell analyses other policy fields, establishes relations and creates a tree. Additionally, each node in the tree has a reference to the policy. Thus, analyzing different walks over the tree, dependencies between rules can be found. Figure 5.3 shows the address tree created by FireWell from the example of table 5.1.

Interpreter

FireWell uses the Alloy–Java–API interface to process the model, and receives the solver output. An unsatisfiable result is an abstraction of a conflict. The solver, in this case the UNSAT, executes verification clauses, denoted by *check-assert* into Alloy model. Results from the SAT solver are interpreted by FireWell to identify counter examples found by the solver. The interpreter decodes the output and identifies the set of fields, rules, and nodes that are received in the counter example and identifies the conflict.

FireWell interprets the result from the SAT solver and shows if the set of firewall policies contains conflicts. Once the solver reports a counter example is found, the interpreter executes

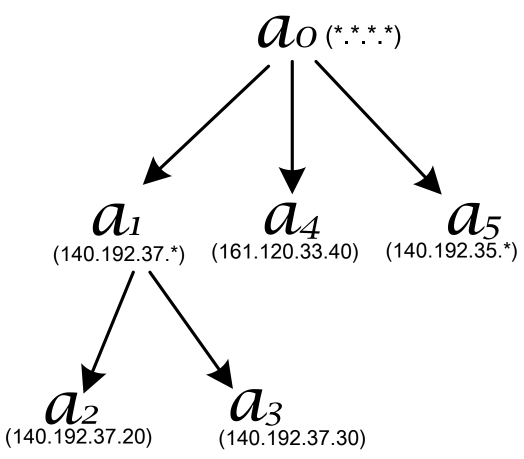


Figure 5.3: Tree of addresses created from the example of table 5.1.

a routine to identify the set of policies that generates the conflict. Finally, rules in conflict are shown to the console. FireWell is able to identify conflict fields and this information could give more tools to the administrator to identify the inconsistent rule.

5.3.4 Mapping firewall rules into Alloy model

The FireWell parser interprets firewall rules and produces Alloy instances. There are two ways in which FireWell reads firewall rules. The first one is loading a file with rules in text format as shown in table 5.1; the second one is grabbing rules through the Floodlight REST API. Once the rules are in the FireWell structure, it creates a map into Alloy language. The mapping process comprises three steps: 1) Create sets of fields, 2) create the topology, and 3) build firewall rules.

Field Sets and the Address Tree Structure

In order to write a rule, five sets are defined: Index, Address, Port, Protocol, and Action. The Index set is used to sort rules so it describes the priority field. Addresses, ports and protocol describe the datagram and identify the traffic flow. Then, these sets are mapped into relational model. The most demanding set is the addressing set because overlapping is a common operation with addresses. FireWell represents this set as a tree because it reduces the amount of variables that later will be evaluated in Alloy. More variables in the model create larger predicates which are more complex to evaluate and finding counterexamples take extra time. FireWell creates a tree of relations between fields. In case of IP addresses, the root of the tree is the wildcard $(*.*.*.*)$ because all addresses are subsets of the root. Then, FireWell analyses other policy fields and establishes relations and creates a tree. Additionally, node–structure has a reference to the policy, using the index as ID. Thus, analyzing different walks over the tree, dependencies between rules can be found. Figure 5.3 shows the tree created from the example in the Table 5.1. Note that Alloy interprets addresses just as elements of a set instead of IP values. The figure 5.3 is the relational representation of this example in Alloy language.

Listing 5.1: Abstract definition of the addresses tree for the example.

```
fact TreeCreation{
    AddressTree.root = a0        // *.*.*.* equivalent
    a1 + a4 + a5 in a0.children
    a2 + a3 in a1.children
}
```

Firewall rules

Rules are written in Alloy based on the rule model shown in § 5.3.1. Wildcard address ($*.*.*.*$) is established as the zero element $a0$ of the tree, and the rest of the addresses (or range) are specific elements in the *Address* set. The Alloy definition is depicted in listing 5.2. Note that all attributes of the rule are relations to an element of each set.

Listing 5.2: Abstract definition of a firewall rule.

```
abstract sig rule{
    ID: one Element,
    proto: one Protocol,
    srcAddr: one Address,
    srcPort: one Port,
    dstAddr: one Address,
    dstPort: one Port,
    action: one Action
}
```

For example, the instance of the first rule from the example table 5.1 is coded in listing 5.3.

Listing 5.3: Abstract definition of first rule for the example.

```
//1,tcp,140.192.37.20,any,*.*.*.*,80,deny
one sig r1 extends rule{}
{
    id = i1 and proto = TCP and srcAddr = a2
    srcPort = p0 and dstAddr = a0
    dstPort = p80 and action = deny
}
```

Conflicts

FireWell model uses the *assert* declaration of Alloy language for conflict instances. Assert instruction creates a set of constraints over the instances. For example, shadowing conflict formulation is shown in listing 5.4 following the model described in § 5.3.1. The remainder conflicts for firewall rules are detailed on appendix B.1.

Listing 5.4: Shadowing conflict definition.

```
/* a previous rule matches all packets that
* matched by this rule */

assert Shadowing {
    no x,y: rule {
        // index(X) < index(Y)
        isBefore[x.id,y.id]
        // X covers X
        (equals[x,y] or inclusiveMatch[y,x])
        x.action ≠ y.action
    }
}

check Shadowing
```

The shadowing conflict detection compares the index of each rule and calculates the inclusive match over two rules. If the action is different, the inclusive relation is held, and the index, then a shadowing conflict occurs.

Network instances

After the definitions, the parser creates a translation of policies into Alloy language. This translation starts by declaring the topology, paths, target flows, and policies. Network topology and paths are declared as appears in § 3.3.2. Also, flows and policies have the same structure.

5.4 Experiment and Results

This section presents a test case on FireWell to prevent conflicts in aggregative mode when policies are configured on the SDN controller.

To evaluate the feasibility of our proposal, we set up a test topology, our FireWell framework, and the network controller for SDN. The experimental set up shown in [ASH04a] is adapted to the SDN environment. The previous experiment has a typical firewall in the middle of three networks and two routers. Now, the present scenario has three switches, and the firewall application running on the controller server. The modified topology is emulated by Mininet [LHM10] which allows us to create realistic virtual networks, connect a SDN controller, access to switch configuration, and run custom applications.

The network operating system (NOS) used in this experiment is Floodlight which is an open SDN controller written in Java that supports OpenFlow as the protocol to communicate controller and switches. The controller supports upper applications and protocols, and installs or modifies the *forwarding tables* which define how packets will be delivered across the network. Controller translates upper instructions to *forwarding rules*. Each entry on the forwarding table describes the *action* to be executed for packets that match a *predicate*; those actions

Table 5.3: Example of firewall rules for the first three rules.

No	Action	Rule	Description
0	ALLOW	dl-type:ARP	Data link protocol
0	ALLOW	nw-proto:ICMP	Network protocol
1	DENY	nw-proto:TCP, nw-src:140.192.37.20, tp-dst:80	Host cannot get web
2	ALLOW	nw-proto:TCP, nw-src:140.192.37.*, tp-dst:80	Subnet gets web
3	ALLOW	nw-proto:TCP, nw-dst:161.120.33.40, tp-dst:80	Web server input
3	ALLOW	nw-proto:TCP, nw-src:161.120.33.40, tp-src:80	Web server output

are forward, drop or redirect the packet. At the lower level, OpenFlow [McK+08b] is a communication protocol that binds the NOS in a centralized controller to the forwarding devices.

Firewall function application of the Floodlight controller is tested by its RESTlet interface to install and remove firewall rules. For example, Table 5.3 includes the rules inserted to firewall application for the first four rules given in the example. Note that rule 0 is necessary to process address resolution protocol. Rule 1 and 2 deny web requests from this specific address and allows the rest of subnet. Rule 3 allow web traffic to/from the web the server.

FireWell parser module tests four sets of firewall rules: 100, 1000, 10000, and 100000. Figure 5.4 shows the parsing time for each set. It shows that the translation is fairly lineal; even though, the complexity of the address insertion into tree structure is not lineal. On the other hand, writing the Alloy model is more expensive, thus slower. Experiments show that writing a model with 100.000 rules takes more than 130 seconds. For the example, the parsing module can translate, insert into the tree, and build the Alloy model in less than three minutes for a set of 100.000 firewall rules.

The entire model, for the example, consists of 12 firewall rules, 6 nodes, 4 ports, and 6 addresses. After the Alloy analysis, the model produces 14389 variables, and 43266 clauses that are evaluated, and finds the list conflict (counterexamples) in less than 356 ms. It shows that solving the Alloy model is expensive; although, it depends on the codification done by the parser.

Table 5.4: Analysis of time using the Alloy API.

Num of rules	12	20	30	40
Time (ms)	1183	1224	11933	44607

Other experiments show that the analysis over 40 firewall rules takes more than 40s, see

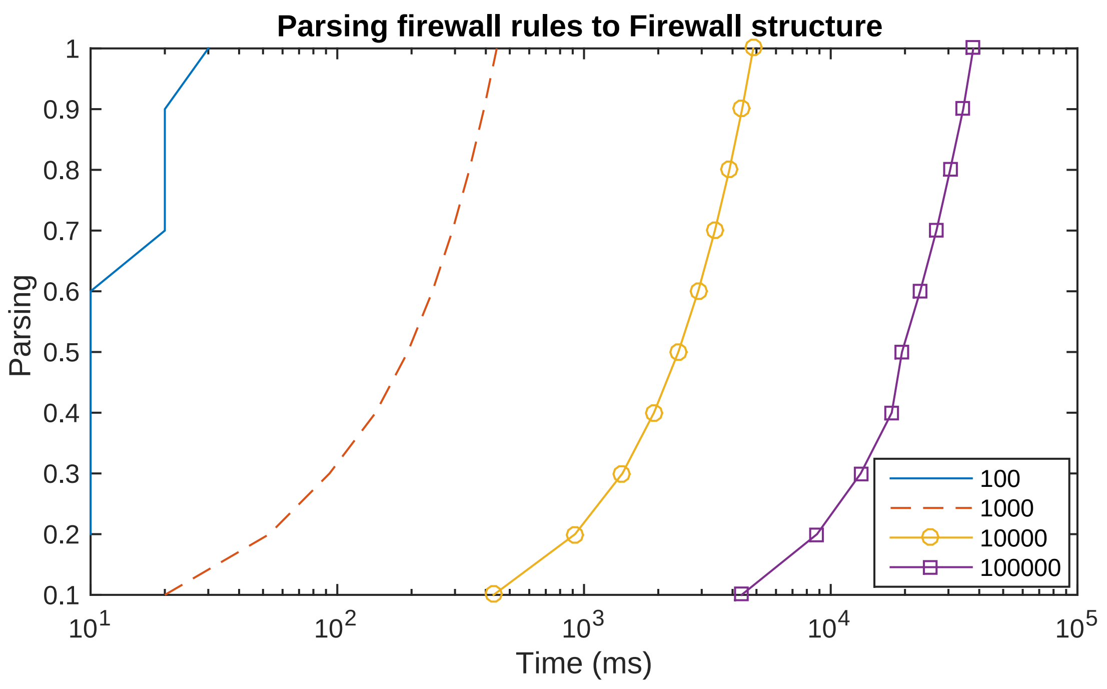


Figure 5.4: Required time to parse firewall rules to Alloy model.

Table 5.4, for the US-carrier topology taken from the Internet Topology Zoo[d]. We infer that the evaluation time does not depend exclusively on the amount of rules, but also the topology complexity. We expect to analyze similar number of rules on different topologies to test this hypothesis. A more complex model could include other forwarding rules in devices distributed across the network in addition to firewall rules.

Table 5.5 shows FireWell effectiveness. Every conflict detected in previous approaches [AS14] is also resolved with FireWell plus those related with routing and packet modification such as address spoofing.

Table 5.5: Conflicts detected in the example firewall-rule set.

Conflict	Rule Pair
Shadowing	(r2,r4) (r3,r4)
Correlation	(r2,r4) (r1,r3) (r3,r4) (r5,37)
Generalization	(r1,r2) (r5,r6) (r2 r3 r6 r7,r8) (r9 r10 r11,r12)
Redundant	(r6,r7)

[d]http://www.topology-zoo.org/gallery.html

```
Shadowing r3 r4
Shadowing r2 r4
Correlated r2 r4
Correlated r1 r3
Correlated r3 r4
Correlated r5 r7
Generalization r11 r12
Generalization r6 r8
Generalization r10 r12
Generalization r5 r6
Generalization r9 r12
Generalization r1 r2
Generalization r7 r8
Generalization r3 r8
Generalization r2 r8
Redundant r6 r7
------------ ------------
Total Time: 2299ms.
```

5.4.1 Advantages of translating to Alloy

Existing tools use algorithms that first expand all the elements in a subset definition, then runs one-by-one algorithm to find overlapping, and finally determine if there are conflicting conditions in the common fields. In contrast, FireWell maps firewall policies into a relational logic model. This translation offers some advantages over the previous work:

- *Extended support for high-level policy definition.* FireWell can adapt the input of firewall policies. With Alloy, policies are abstractions which can be easily transformed as predicates.
- *Aggregate functionality* Alloy is able to run from previous results, once an analysis is done, FireWell can use previous executions and add new signatures (policies), and execute only with those with potential conflict.
- *Support for additional types of actions* Besides denying or permitting a type of traffic through a path. These policies can be written in Alloy and support additional constraints.
- *Checking firewall application in double way* Firewall rules can be checked to find inconsistencies in a set of firewall rules, but there is not possible to check how firewall–forwarding rules affect other network functionalities. With SDN all applications insert forwarding rules in datapaths, and with Alloy we can check if the firewall behavior conflicts with the topology or with the rules installed by another network application.

5.5 Summary and Conclusion

This chapter presents FireWell, a firewall policy analyzer supported on Alloy to detect conflicts in policies. First, it reads policy configuration directly from the Floodlight controller, and builds abstract representation of policies. Our tool exploits relational logic to explore configuration conflicts in firewalls, and reasons about defined constraints in a simpler and more complete way than the existing solutions. Currently, we are focused on evaluating alternatives to optimize the conflict detection performance. Mainly, we are considering *model slicing* techniques where the tool analyses subsets of the network instead of all the elements. In addition, we are working on an implementation that uses KodKod, the internal library used by Alloy that provides support for partial instances to optimize processing.

FireWell creates a variable scope for the symbolic representation of addresses, ports, and protocols. Ranges are also symbolic, which are represented by a single variable rather than an array of variables. This reduces the number of variables and simplifies the evaluation. The translation made by the *parser* is fundamental for the performance of the SAT-solver and general behavior. We use Alloy to create better translation of complex policies into CNF, as compared to other existing tools. Optimizing the translation in Alloy will allow us to contemplate other network features such as security, routing, QoS, or network resilience. We understand that having Boolean quantifiers, such as universal ($\forall$) and existence ($\exists$), degrades the performance, but they are necessary to test more complex properties like security or resilience. For example, creating symbolic multipath and evaluating properties over the multipath is a Boolean formula that requires logical quantifiers instead of calculating *all possible* elements before performing the validation.

FireWell is a project to create well–defined firewall policies on SDN. As main contributions, this chapter and its related paper [MLCD15] 1) identifies firewall misconfiguration for SDN networks, 2) shows that firewall rules can be translated into a model written in Alloy, and 3) introduces our implementation that solves the model to detect and prevent conflicts on these rules. Using FireWell, a network administrator could prevent the introduction of conflicting rules into the SDN infrastructure. FireWell extracts firewall rules from the SDN by the REST interface of Floodlight controller, detects the conflicting rules (if they exist), and determines if a new rule will conflict with the existing ones.

Chapter 6

Verification of Policies in Multiple Domains

A problem well put is half solved.

John Dewey

This chapter proposes a mechanism to audit network policies in multiple domains called AudIt. This approach allows the network administrators to validate if the network policies are enforced by an external domain. In summary, this chapter introduces the foreign controller verification problem. Also, multi–domain policies in programmable networks are defined. A mechanism to gather information from external SDN domains is detailed, and a validation engine that uses gathered information to check if a network policy is enforced by the external domain is designed.

In the SDN architecture, the control plane remains on a centralized server and makes decisions about how the traffic is processed. The controller is responsible for managing connections, addressing, and routing protocols. In summary, it manages the behavior of its own network domain. The entire network is an aggregation of several domains and each has its own network controller. Forwarding rules must be implemented through several datapaths on multiple domains. For instance, the network traffic is delivered using an internal network such as LANs, and an external network such as WANs or Internet. Only the domain controller has access to the rules used by each datapath in its network domain. Thus, neither a datapath nor a controller can access information about rules from *foreign* domains.

Current SDN is good for dealing with policies in a single domain. However, the network administrator cannot observe how an external network, out of its domain, handles the traffic. The administrator is not able to enforce nor monitor network policies on multi–domain scenarios. This problem is defined as the *policy enforcement on multi–domain.* If the network administrator needs to enforce or monitor a network policy, she needs to grab information of how her traffic is handled by a foreign domain. She can define applications in her own network domain but cannot do it in the foreign domains. Unfortunately, she cannot check if external domains enforce a network policy because she cannot determine how the traffic is delivered in those external networks.

This situation can be especially critical in network security policies. For example, crucial traffic that is delivered through external networks can be duplicated or redirected to other network machines using a simple application in the external domain SDN controllers. Since network administrators cannot get access to the rules in the external networks, they are unable to detect these situations or validate if their policy is achieved.

For this reason, we present the problem of auditing own policies at external domains. Then, we present a model to illustrate network topology, paths, forwarding rules and policies in § 6.2. Later, we introduce the AudIt protocol which can gather information from an external SDN domains and validate if an own policy is enforced by the external domain. Finally, we describe AudIt design, functionality, and present an example.

6.1 Auditing Policies in Multi–Domain Networks

A *Network policy* is a set of conditions, constraints, and settings about how a specific type of traffic must be managed by a network. Policy may also include authorized users and hosts to create connections, and the circumstances under which they can or cannot connect. Network policies are the accurate and unambiguous way to specify the traffic behavior. Review table 2.4 for general definitions.

Initially, Stone *et al.* proposed a path-based policy language (PPL) that abstracts topological (physical) paths and flows to check network properties [SLX01] such as invariants § 2.4.2. With programmable networks as SDN, new network policy abstractions are under development, therefore the challenges in policy checking open an interesting field for research. High–level declarative languages were proposed to represent network policies with more expressiveness. Declarative languages such as FML [Hin+09b; Hin+09a] express network policies in terms of flows. For general purposes, Hinrichs developed a declarative language called Flow-based Management Language (FML) to describe network policies and configuration in a high-level and declarative approach[Hin+09a]. FML is based on flows, and checks the first packet of every flow against the policy. FML identifies a network flow by: *source* and *target* for users, hosts, and access points, in addition to protocols and requests

A flow is the specification of a traffic, sometimes is called a *session*, that contains common attributes such as source, destination, protocol, but also can specify more granular characteristics as duration, valid time, users, data format and so on § 3.1.4. Then those policies are processed using DATALOG to find matching flows. Other languages were designed for SDN are Merlin and NetCore. Merlin [Sou+13] is a framework to write network policies for SDN. NetCore [MFHW12] is a language for describing forwarding rules and it is integrated with another framework called Frenetic[a] [Fos+11]. These languages allow network administrators to define policies in a single-domain networks. They did not contemplate checking policy enforcement on a third-domain.

[a]Frenetic is a project from Cornell and Princeton universities. `http://www.frenetic-lang.org/`

In contrast to previous approaches, AudIt offers the ability to write and check network policies; it is unified with the controller, extracts forwarding data, and checks it. AudIt also uses the flow specification, as it is described in § 3.1.4, and verifies if the set of flows is valid for the given topology. Moreover, AudIt reports inconsistencies at level of flows in addition to instructions at the level of hardware implementations.

For example, suppose that a network policy defines *only computers assigned to members of IT department can get access to database servers.* The *policy* can be written as:

$$\textit{allow}\,(\texttt{src},\texttt{target}) \;\mid\; \texttt{src} \in \textit{IT} \wedge \texttt{target} \in \textit{DataBase} \tag{6.1}$$

$$\textit{deny}\,(\texttt{src},\texttt{target}) \;\mid\; \texttt{src} \notin \textit{IT} \wedge \texttt{target} \in \textit{DataBase} \tag{6.2}$$

Expression (6.1) means that the network must allow flows from IT to database servers. Since policies must be closed, (6.2) denies any flow from other machines to database servers. In SDN networks, policies are enforced by its domain controller which rules the behavior of every datapath in its domain. So, the question is: How can a network administrator monitor that this policy is enforced?

6.1.1 Policies in Multi-domain Networks

Consider a multiple–domain network where each domain is managed by its own controller. This example scenario is depicted in figure 6.1; the domain A, left cloud, is ruled by its *controller* CA, and operates the IT department and its users. The foreign domain B, right cloud, is managed by the controller CB and connects the database servers. the network administrator supervises her own controller CA, and may install forwarding rules on devices S1, S2, and S3 to deal with traffic generated by the IT department. However, network controller CA cannot access the rules of datapaths in the domain B neither compel controller CB to install required forwarding actions into the devices S4, S5, and S6.

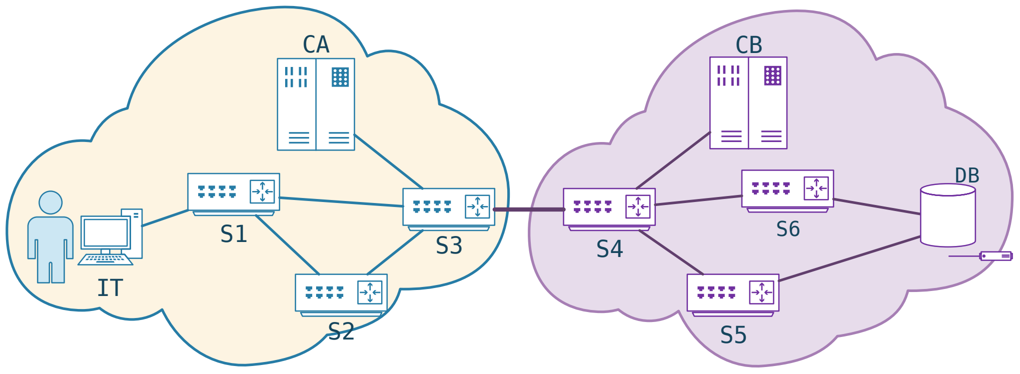


Figure 6.1: Domain A may send a policy to be implemented in domain B, but there is no guarantee that B implements the policy correctly.

A *multi-domain policy* must be enforced in both domains, its own and foreign domains. CA controller may share the policy with CB controller, and expects that CB correctly implements the policy in its devices. However, there is no certainty that delivered traffic in the external domain B obeys any policy defined by A. Then, the administrator of domain A cannot enforce policies related to delivering traffic to database servers, because the domain B is external.

6.1.2 Challenges in multi-domain networks

Each network controller oversees configuring switching devices on its domain. Figure 6.2 shows the required configuration installed on devices in the path of domain A that process the flow until it reaches the next domain. *Network configuration* is the set of rules that implements the policy. In this case, the configuration conducts the *flow traffic* from IT department to database servers. This traffic arrives to switch S1, then is forwarded to switch S3, and finally it is forwarded to domain B interface, switch S4. Clearly, a controller does not know and cannot handle implemented configurations in external domains. However, administrators want to know if their policies are enforced in external domains. Because of database servers are located in an external network, for instance it could be hosted by another company, the above policy redirects flows from the IT department to external servers but denies the flows originated from sources other than IT department. Usually, companies rely on external networks such as WANs and Internet to deliver their network *flows*.

S1 flowtable

Match	Action
Src = IT $\wedge$ Dst = DB	$\langle$ Fwd S3 $\rangle$

S3 flowtable

Match	Action
Src = IT $\wedge$ Dst = DB	$\langle$ Fwd S4 $\rangle$

Figure 6.2: Flowtables for devices in domain A. Note that this policy implementation is closed and avoids connections from different sources of IT department.

Since the configuration of a network device is protected information, it is only accessed by its own domain controller, and the administrator wants to check if the external controller applies a policy on its domain. The main challenge is: how to detect if a policy is enforced by an external domain? and how to audit the policy enforcement without revealing risky information?

6.1.3 Policies in Programmable Networks

In the SDN environment there are some languages to describe network policies. For example, NetCore is a high-level declarative language that describes the desired behavior of the network but does not define the implementation of that behavior at datapath level. With NetCore language is possible to express packet forwarding policies for SDN networks [MFHW12]. Another declarative language is Merlin [Sou+13] but it is based on logical predicates and regular expressions which can be solved using linear programming to determine forwarding paths.

Verification of SDN configurations is focused on checking network properties that follow a rule. For instance, VeryFlow [Khu+13] creates network–wide invariants and checks them against rules. FatTire [RCGF13] writes the policies by regular expressions and validates them. Other works try to find *conflicting rules*, i.e. rules that contradict earlier ones. In such a way FortNOX [Por+12] checks new flow rules against a flow–constraint set, and authenticates the source of rules by digital signatures. It implements the *alias set rule reduction* algorithm to identify rules conflict and role-based authentication to authenticate OF applications. Another illustration is NetPlumber [Kaz+13], which searches if a candidate rule introduces network misconfigurations or policy violations. It executes a procedure called Header Space Analysis (HSA) over dependency graphs to find conflicts. These approaches examine the forwarding tables from each network device and check if they conform with the specified policy. However, none of these approaches support the validation of policies in external domains.

Policy implementation into flow-rules

Network applications — or functionality— run on a controller and define the general behavior or *policies* by installing specific configurations on each switching device. Regularly, those programs use OpenFlow (OF) [Ope12] to communicate controllers and forwarding devices, and install, modify, or get *flow-rules* that specify how a device deals with specific traffic. A flow–rule is a pair `<match,action>` on the device's *flowtable.* A flow–rule defines which `action` is performed once a packet header matches the `match` pattern.

OF defines a set of messages to control the internal information on each device, and rules used to process a flow. In summary, OF messages can add, modify, and query rules from a device's flowtable. Actions also include: dropping a packet (`DROP`), forwarding a packet to a specific port (`FWD`), or report the set of installed rules (`STATUS`). The rule-set is *closed*, and the packet is reported to the controller if its header does not match any rule.

6.2 Topology and Policy Models

This section presents the model that describes the physical topology, paths, and flows. Also, this model describes network operations such as flows, policies, and conflicts. After the model is set up, the *relation of correspondence* is used to write the model in predicate logic and other Alloy expressions.

6.2.1 Network topology and paths

The model for network G and path $p \in P$ used in this chapter is based on the definition of § 3.1.1. Also, it includes three network invariants such as the connection to all nodes, the self–loop restriction, and the links consistency. However, there are some differences from the model shown in the previous chapter. Nodes in this model do not represent a datapath, a node denotes a *device port.* Then, under this abstraction, node transitions represent physical links in the topology and forwarding rules. At this point, it is appropriated to describe the *transitive closure.* This characteristic is needed to tackle the reachability property when describing a path. A binary relation R is *transitive* if it contains tuples in the way $a \rightarrow b$ and $b \rightarrow c$, and also contains $a \rightarrow c$. This relation is noted as R^+ and contains R. Let the function $\mathsf{links}^+(n)$ be the set of all nodes that can be reached from n. A path has no loops if $\nexists n \in N_p | n \in \mathsf{links}^+(n)$. Also for convenience, we denote a path as a node sequence as $\langle s, n_1, n_2, \ldots, t \rangle$. The wildcard symbol $(*)$ is included in this model to denote any unspecified node or sequence of nodes. For example, the path $p = \langle A, *, C \rangle$ is the path that starts at node A and ends at C.

6.2.2 Traffic flows

Flow is a fundamental abstraction for the model. It is similar to *communication session* supported by a set of paths and device configurations. Traffic flow defines network parameters needed to create a competent communication channel. A flow provides enough detail to describe a set of feasible sessions, and provides a way to group and manage these sessions.

Definition 6.1 (Traffic flow)**.** *A traffic flow is a sequence of packet–header constraints* $f = (f_1, \ldots, f_n)$*. Each term* f_i *is a restriction over a traffic characteristic, strongly related to filters on packet fields.*

Due to flow is defined as a set of traffic constraints, this model indicates that header–fields match those constraints. The used definition allows us to construct flexible and composed communication flows. A term of flow involves transport-layer protocol, source / destination at third layer, and some application fields. Also, this model uses a set of operators over these packet fields to define the flow. For instance, $flow_a = \{\texttt{protocol = TCP}, \texttt{src\_ip = 192.168.5.10}, \texttt{dst\_ip = 192.168.7.10}\}$ details a traffic flow between those IP addresses and TCP as transport protocol. Note that this flow only defines the traffic in one way. It means that the other direction is not included in this definition. Note that this flow definition is only associated to communication characteristics and packet fields, but not to the set of paths that supports the flow. Paths were considered on § 3.1.1.

6.2.3 Policy conflicts and semantics

In order to define the set of paths that implement a flow and then identify policy conflicts and violation, we use modeling language L to describe a *syntactic domain* $\mathcal{L}_L$, a *semantic domain* $\mathcal{S}_L$ and a *semantic function* $\mathcal{M}_L : \mathcal{L}_L \to \mathcal{S}_L$. We use the tradition schema as it is shown in § 3.1.6.

As we define in § 3.1.5, a policy is a set of rules, and a rule is a tuple $\pi = (p, f, C, \alpha)$ where f is the target flow composed of packet–field values, p is a path that supports the flow, C is a set of conditions, over the flow f or path p, and α is an action, regularly $\{permit, deny\}$. Then, a policy is a set of rules that achieves a management procedure. Forwarding rule is the action that a node executes to forward a packet into a datapath. For example, forwarding rules are stored in the *iptable* application to configure the firewall in Linux kernel. Those rules are described in terms of flows, by the previous definition.

The *semantic* policy is the result of the semantic function $[\![\pi, G]\!]$ which determines a set of configurations, ω in this model. Then, ω implements the flow over a path on the topology G and complies the policy π. As was shown in § 3.1.6, the semantic of all policies

$[\![\Pi]\!] = \Omega$ produces the set of all network implementation. The complete network configuration is denoted by Ω. The semantic function $[\![\pi, G]\!]$ of a policy contains the sets of paths, flow definitions, conditions and the network G that satisfy the policy π. Obviously, the policy π is valid in a network G, if $[\![\pi, G]\!]$ is not empty.

Essentially, a policy resolves whether allowing the flow f over the set P concludes on a specific action α. The semantics over the policy produces a configuration network that allows or denies the traffic flow according to the policy. For example, a network administrator wants to apply the policy: *Ana is a user with profile of IT member, who is in the subnetwork 192.168.5.*/24 (S1), is allowed to access the database at subnetwork 192.168.7.*/24 (S6) and port 1521, and her traffic must go through the router S3.*

Now the manager has to detail the policy. In order to do that she solves the following steps:

1. path `S1_S6` := `<S1,*,S3,*,S6>`,
2. transport protocol: = `TCP`,
3. port number: = 1521;
4. the conditions `user = Ana`, and `Ana ∈ IT member`;
5. finally, the policy decision: `permit`

In this way, finding ω means finding the configuration set and instructions that implement the paths and the policy π. Note that IP addresses, user groups, traffic class and protocols should be modeled as *sets*. On the other hand, ordered items, such as *time*, are modeled as *sequences* to be able to compare them using $\leq$ and $\geq$ operators.

$$\begin{aligned}\omega = \mathit{impl}(p : \mathit{path} | (\texttt{S1\_S6}) \wedge \texttt{protocol} = \texttt{TCP} \wedge \texttt{port} = \texttt{1521} \\ \wedge\, \texttt{Ana} : \mathit{user} \in \texttt{IT member})\end{aligned} \tag{6.3}$$

Equation (6.3) represents the policy as a conjunctive normal form *predicate* (CNF), and logically solve it. Later, it allows us to check a formal solution using a model finder as Alloy [Jac06], compare solutions, and find inconsistencies.

A policy conflict occurs when a set of policies are not implemented by any configuration or there are inconsistencies that prevent the generation of any of them. See definition 3.8. Essentially, if the semantics of a policy is empty, it means that there is not configuration set that satisfies the policy. Given two valid policies π_1 and π_2, they are not conflicting in the network G if $[\![\pi_1 \cup \pi_2, G]\!]$ is not empty. That is, two policies are not conflicting if there is a set of paths, flows, and restrictions in the network G that satisfies both policies. In contrast, we say that two policies π_1 and π_2 are conflicting in G if $|[\![\pi_1 \cup \pi_2, G]\!]| = \emptyset$.

On the other hand, the semantics over the minimum set of policies without conflicts is the *minimal diagnosis*. It is the littlest configuration applicable and functional from a set of policies. See definition 3.9. Where a set of policies $\pi \subseteq \Pi$ such that $[\![\Pi \setminus \pi]\!] \neq \emptyset$, the minimal set π is the minimal diagnosis.

After the model is proposed, a *verification engine* is needed to calculate the semantic function $[\![\pi, G]\!]$ and verify if the result set is empty, or the minimal diagnosis of that set. Similar methodology and tools are also used for validating paths on network infrastructure [MLCD14], and we show how to use the minimal diagnosis to detect and prevent firewall–rule conflicts on software–defined networking [MLCD15].

6.3 Checking multi-domain policies with AudIt

AudIt is an auditing tool that uses the extension for OpenSwitch and OpenFlow protocols and the domain–controller platform to validate network policies on a foreign domain. Our proposal creates a language definition and transformation to audit network policies. It uses Alloy translation to obtain a set of tuples that satisfy the policies (exactly the semantic function). If Alloy does not find any element (the set is empty), the policy set is said invalid or conflicting.

AudIt comprises three fundamental modules: 1) an extension to the OpenFlow protocol to enable external auditing, 2) the AudIt interface for network devices that gathers information about the actions performed in external domains to carry the flows of interest, and 3) a validation engine that runs into the internal network controller and detects policy violations.

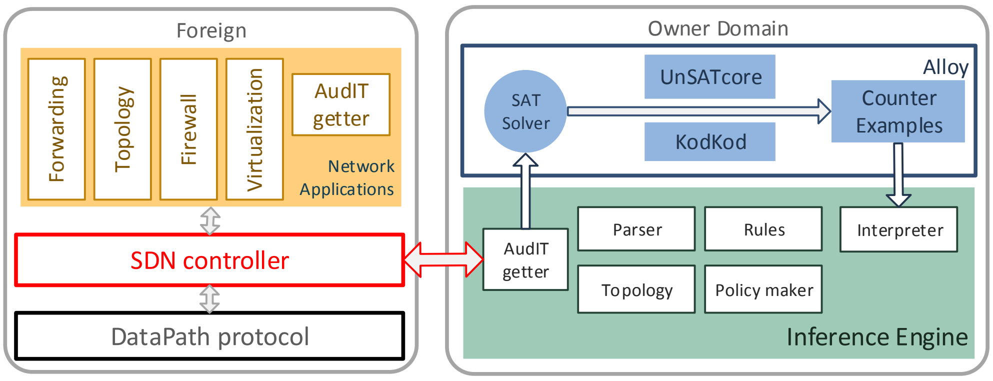


Figure 6.3: AudIt architecture

AudIt architecture is depicted on figure 6.3. AudIt works as a validation protocol that allows a controller to gather auditing information from external domains and validate the origin policy. It has two phases: gathering network information and validation process. First, the controller, in the origin domain, gets information from the foreign domain which send back *audit packets*. Datapath of the foreign domain produces those audit packets and route them back through the network as *regular* traffic. Then, when auditor packet reaches devices in the external domain, these network devices report a subset of its own flowtable to the controller in the origin domain. Finally, the controller in the origin domain processes the gathered flow tables to obtain all the processing rules related to the flows of interest, and executes the *inference engine* that checks if the external domain is accomplishing the security policy.

6.3.1 AudIt: the protocol extension

OpenFlow specifies *control messages* between controller and forwarding devices. Control messages can perform three important tasks: 1) *modify–state* by adding or deleting flow tables in the device, 2) *collect-statistics* by reading counters and device statistics, and 3) *managing* groups of flow tables. Controller is also able to request device's *status*, where the device reports the flowtable to the controller. AudIt uses regular controller primitives to request information from the flowtable on devices from the external domain.

Information gathering

Once the controller enables AudIt on each network device, and the audit packet arrives, the device invokes `OFPMP_TABLE_FEATURES` and the header match to filter a subset of the rule table that matches the header. Thereby, it extracts a set of all the *related flowtable entries* (RFE).

$$\texttt{RFE} = \{e | e.\texttt{src} \odot \pi.\texttt{src} \cup e.\texttt{target} \odot \pi.\texttt{target}\} \tag{6.4}$$

A Related Flowtable Entries (RFE) e is a forwarding rule that match some fields with the policy abstraction. π is the policy and $\odot$ is the match relation. For example, equation (6.4) shows a RFE that matches two fields, source and destination addresses. The RFE set for the example in figure 6.1 contains the IP addresses and masks in the IT–database scenario depicted on expressions 6.1 and 6.2.

AudIt message

Figure 6.4 shows the structure of an AudIt message. It has the same flow header of the audited traffic in order to be routed through the same path; moreover, it includes origin controller identifier, controller authentication data, the list of fields and rules to be filtered by the device, and other AudIt settings.

Flow header	Origin Controller ID	AudIt settings	Controller Signature
List of fields		List of policies	

Figure 6.4: Structure of an AudIt packet. The list of policies are constraints over packet fields.

6.3.2 AudIt protocol

Figure 6.5 shows the proposed protocol that allows controllers to enable AudIt protocol, gather information from foreign devices, and check network policies.

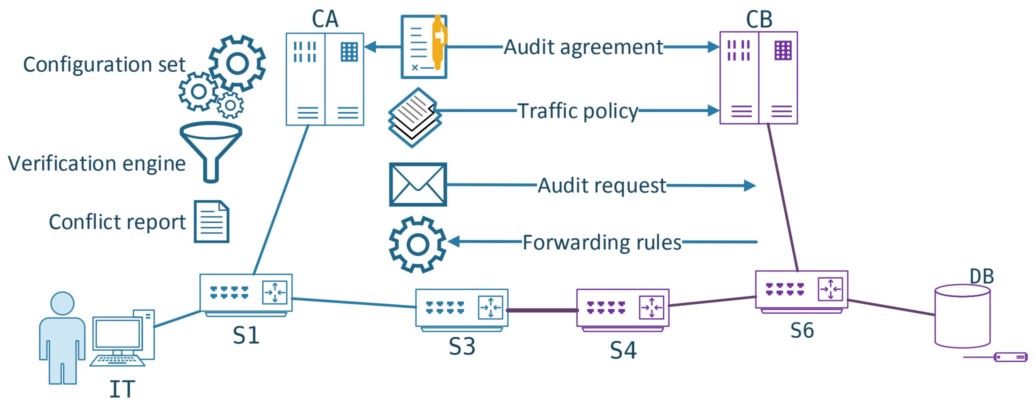


Figure 6.5: AudIt protocol execution. Devices from domain B report packet rules to CA, then A verifies traffic policy and generates an auditing report.

1. Involved domains subscribe an *audit agreement* that specifies the permission to create, send and process AudIt packets. Then, all implicated domains update their modules so that it can recognizes the audit request and overwrites `AUDIT_ENABLE` variable.

2. Origin domain A shares the traffic policy over IT's traffic with B. The network policy described in expressions 6.1 - 6.2 which state *DataBase is only accessed from IT department*. External-domain controller CB enforces the security policy in its network, translates the policy into rules applicable to its infrastructure.

3. Origin controller creates an AudIt packet. This packet contains all packet fields from the regular flow traffic. This procedure requests information about how the traffic is delivered. Thus, foreign datapaths process the AudIt packet as they process regular data flow, and use an interface to return the Related Flowtable Entries (RFE).

4. Foreign devices reply the AudIt packet with the RFE list. The list of entries from its flowtable.

5. At the origin domain, the controller of A executes the validation engine, determines if there is a subset of rules that violates the policy, and writes a conflict report.

6.3.3 Multi-domain Policy Checking

With the purpose of policy checking, origin controller gets the rules —forwarding rules— related with the traffic policy that comes from the external domain. Then, it validates the set of RFE against the policy to identify violations. Figure 6.6 shows the RFE set that CA gathers from domain B of the example shown in figure 6.1. It is a list of rules related with the traffic policy defined in expressions 6.1 and 6.2. Then, the validation engine determines if this subset of rules violates the policy. Similar policy-rule validation engine is found in NetPlumber [Kaz+13].

S4 flowtable

Match	Action
Src = IT ∧ Dst = DB	⟨ Fwd S6 ⟩

S6 flowtable

Match	Action
Src = IT ∧ Dst = DB	⟨ Fwd DB ⟩

Figure 6.6: Related flowtables entries from devices in domain B.

6.3.4 Inference Engine based on SAT

The developed *inference engine* is able to check implementation procedures against network policy.

Network Topology

Topology is defined as a set of *nodes*. *Links* is a closure relation with arity two over the node set. We model device ports as nodes, and forwarding rules and datapaths as links, hence paths are the result of the closure relation over nodes. Figure 6.7 shows how the topology is represented in terms of device ports. A forwarding rule, the simplest instruction that redirects a packet from one port to another, it is represented as part of the path. Under this perspective, the configuration is part of the topology. Forwarding rules are shown in the figure as dotted lines. Our model handles forwarding rules as *soft–links*. These soft-links are considered as regular topology once the model is built.

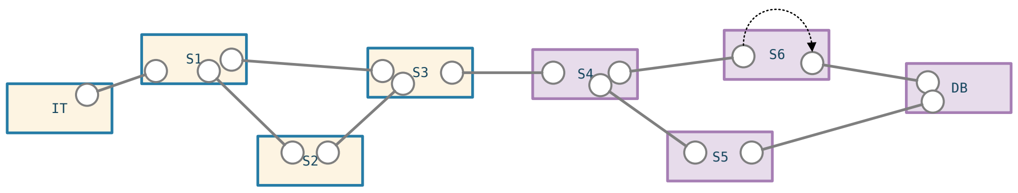


Figure 6.7: Representation of a network topology based on ports from the original deployment. Blocks are network devices and circles are ports. Forwarding rules are depicted as dotted lines that connect two ports.

An optimization opportunity arises because forwarding rules create soft–links which are interpreted as part of the topology. If the traffic policy is too much specific, the resulting topology is a disconnected graph, even if some paths are reduced or eliminated from the model. This abstraction of nodes as device ports, and soft-links can reduce the complexity, and hence the evaluation time.

Traffic flow

Flow is depicted as the list of constraints over packet fields, traffic movement sense, and other topological considerations. For example, the flowtable described in Figure 6.2 is the interpretation of constraints, source and destination addresses, over fields of a packet header. Note that flowtable also denotes the *soft-link* between two ports generated by the forwarding rule. Nevertheless, these soft–links are part of the topology just as wired links do. In other words, the model does not discern one from another. Communication details such as protocols or port numbers are considered sets if these elements are part of the packet header. Since our approach uses set theory notions, ordering is not considered in this model, because of this it cannot have policies with arithmetic conditionals. For example, the expression *if the port number is greater than 1024, then ...* is invalid in our approach.

As part of the methodology of this thesis, the inference machine is implemented on Alloy [Jac02]. It is fed with external-domain information gathered by the audit procedure or through services that shares the forwarding information.

6.4 Experimental results

The proposed AudIt protocol is implemented and tested using the Floodlight controller [b]. Test cases are divided into two groups: information gathering, and violation inferring. Controller runs on a server and deploys a test–network using mininet [c], which operates as external domain and implements the example topology used by Sethi in [SNM13]. From another terminal, which operates as owner domain, AudIt interface is executed and extracts traffic information from the controller. AudIt implementation creates a topology representation, a policy inventory, and a configuration repository. Thereafter, the inference engine is executed. AudIt writes, policies, configurations and topology as Alloy instructions and executes the satisfiability solver KodKod as done in previous chapters §§ 3.4 and 5.4.

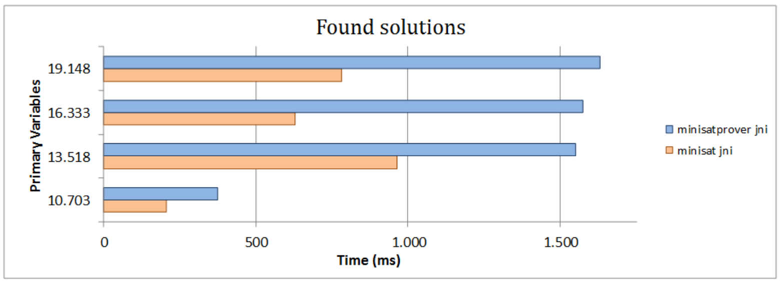


Figure 6.8: Solutions from Alloy implementation. The same implementation is evaluated using two solvers: minisat and minisatprover with minimal unsatisfiable core.

Figure 6.8 shows two evaluations over the same topology and set of policies. FatTree topology described in [SNM13] is used to test our approach (figure 6.9). AudIt takes less than a second using the minisat solver, which only finds if an instance accomplishes the set of policies. On the other hand, if the network administrator wants to determine the set of policies violated by the external domain, she executes AudIt with the minisatprover option and could take up to 1.5s. These measures are lower than the values reported on [SNM13], for the same Fat Tree topology composed of 20 switches, 16 hosts, and 48 links. We test forwarding and reachability on an Intel i5 at 3.0 GHz, with 3.74 GB of RAM. With the intention of showing how state explosion and variable affect the performance, we test AudIt for 930K, 1.5M, 2.2M and 2.8M of states, which are represented on primary variables shown in figure 6.8.

[b]`http://www.projectfloodlight.org/floodlight/`
[c]`mininet.org`

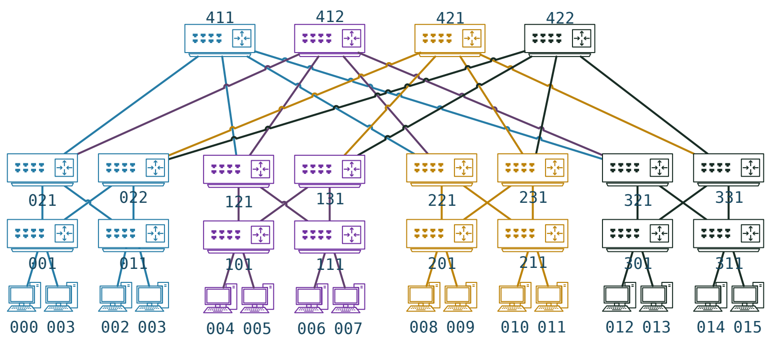


Figure 6.9: FatTree topology for test.

6.5 Summary and Conclusions

In contrast to previous works shown in § 6.1, our work proposes to express network policies as predicates but use a SAT solver and a model finder to evaluate predicates, find inconsistencies and detect policy violations. AudIt uses Alloy [Jac02] to describe the network topology, policies and network traffic. Mirzaei *et al.* proposed using Alloy to verify network properties in [Mir+16]. They model internal states of a network and OpenFlow switches.

Our approach presented OpenFlow AudIt, a mechanism that checks if foreign domains are enforcing multi-domain network policies. AudIt helps to overcome policy-checking limitations of the SDN architecture. It comprises (1) an extension to the OpenFlow protocol to enable external auditing, (2) an Audit protocol to gather information about rules applied to specific network *flows*, and (3) a validation engine that uses flow information and determines if the external network is enforcing specific traffic policies. Additionally, AudIt can identify policy violations. It informs the network configuration, the rule and the flow that infringes the policy and its identifier. In general terms, AudIt allows network administrators to gather information from external domains and determine if network policies are enforced in multiple domains.

However, AudIt does not have complete information about the network as opposed to NetPlumber [Kaz+13]. Moreover, AudIt requires the deployment of our OpenFlow extensions into the network devices in those external domains. Commercial products (i.e. switches from companies such as IBM or HP) do not support the deployment of new extensions without a firmware update. We expect that future experimental implementation shows the benefits of AudIt and can be a foundation to introduce multi-domain policy validation into the standard.

Letting external domains gather information about network flow processing may represent a potential security risk for external controllers. In addition, controllers in external domains may include programs that hide information or mimic policy enforcement. Future work focuses on evaluating security risks on our experimental implementation in order to determine which additional mechanisms are required to ensure safe auditing of multi-domain policies.

Chapter 7

Conclusion and Future Work

This chapter summarizes the main contributions of this work and presents potential directions for future research.

7.1 Conclusions

The purpose of this thesis is to study verification of network policies in Software–Defined Networking oriented to Policy–Based Management. For that end, this thesis proposes modeling network policies by predicate–relational logic and uses a solver to find solutions and counter examples for the model.

In the research process, verification of topology invariants was the initial approach. Different techniques for verifying network policies were studied including constraint programming and optimization techniques. However, we were focused on closed policies. The objective was to verify if that a policy is achieved by a network configuration, but also if the configuration is the minimum required to fulfill the policy. This condition forces the use of formal methods to guarantee the closed policy–configuration relationship. We identified that this problem can be solved if it is expressed as a SAT problem, we studied structures to represent policies as predicates, and applied a solver. The main aspects considered in this thesis are summarized in the following paragraphs.

Topology and policy model A model for topology and policy was proposed. We developed a structure to model topologies based on predicates, constraints and axioms. In similar way, policies are defined, with sets to describe packet–header fields and filters to model the restrictions. Fundamental network invariants such as reachability, no–loop paths, and connectivity were checked. Checking those characteristics reassures the effectiveness of the proposed model.

SDN model as Interdependent Networks This thesis presented a SDN representation as Interdependent Network. This model presents a possible network behavior after verification failures. This model also describe graph properties and measures an SDN scenario of failures, attacks, to explain the network behavior when the data or the control plane experiment failures if the network invariants are wrong.

Firewall and filter rules model After essential characteristics for topology were verified, a natural step was checking rules for firewall and access control. A predicate–relational model was designed for set of packet–header fields, set relationship and operators, and filters per field. Then, firewall conflicts were modeled as logical relations over fields, and the model was prepared to identify inconsistencies on SDN rules of firewall.

Datapath configuration Vs. Closed Policy Another goal was to identify if a different network domain is enforcing a given policy. This identification was achieved by analyzing from configuration to policies. A module was designed to grab forwarding information from datapaths and later contrast it against the policy. This mechanism can audit network policies on a foreign domain and reduces neglected activities of an external service provider.

Conceptual framework of verification for networking Additionally, this thesis presented a review on mechanisms of verification applied to networks and SDN. The aim was to offer a conceptual framework to apply verification instruments from other disciplines to networking software. Logical formulation and solvers for verification such as SAT problem are fit to be use in the networking software domain.

7.2 Discussion

This thesis decides using transformations to a SAT problem to deal with the quantifier in predicate formalism. Network policies treated by this thesis, require the use of logical quantifiers (QBF) to compose *closed* policies. However, there are difficulties in applying SAT for the verification of temporal properties. For the time being, there is no efficient way to express temporal properties on propositional logic. However, some approaches use bounds to simulate *steps* in the protocol [Tre09].

The SAT problem presented in this work is a NP–complete. Hence, its runtime grows exponentially with the number of variables in the graph. This exponential growth shows difficulties when the problem scales. This means that, in the worst case, it is necessary to calculate all possible combinations of variables to evaluate the satisfiability of a formula. However, SAT solvers have progressed in recent years and have developed efficient–search methods. Our approach uses solvers that implements variations of the DPLL algorithm which reduce the number of variables to efficiently evaluate clauses. In the last years, new approaches have enhanced the solver efficiency by clause elimination, branching steps after local–exhaustive search, learned–clauses, and random restarting and assignation [MZ09].

An advantage of using model checking for networking is that it explores all possible states until it finds an error state. Another advantage is the counter–example finder. Common

Model–Checking tools generate counter examples which highlight the conditions where the formula is evaluated as false and violates constraints, policies or configurations.

Efficiency improvements facilitate using SAT for industrial applications. Now, SAT solvers are more common, and multiple tools are developed to verify and test. Their use for software verification and debugging is the motivation to use them for this thesis. However, the challenge is how to deal with the scale.

7.3 Contributions

The following table summarizes the contributions of this thesis which are related with the research goal described in the Introduction chapter. This thesis presents four contributions, each one maps a objective and complements the research project.

Table 7.1: Summary of contributions of this doctoral thesis.

Objective	**Contribution**
To create a model in predicate–relational logic of the network infrastructure and validate network functionality	We create a formal lightweight model for the network and validate of topology invariants such as reachability, no–loops, blackholes, avoidances and waypoints.
To represent the SDN as an interdependent network and analyze the impact of targeted attacks	We model a typical AS network with SDN, the control and data planes, as an interdependent network and evaluate the robustness. This evaluation shows that network functionality is lost after failures.
To represent SDN functionalities governed by policies, such as firewall and auditing, and evaluate the functionalities with a lightweight formal method.	We formulate a model for firewall and auditing functionalities to evaluate rule shadowing, redundancy, correlation, generalization, spuriousness and non–bypassing.
To implement a validation process for network policies and test the solution by a verification–case	We build a validation mechanism based on policies, predicates and relational transformations in the lightweight formal method written in Alloy. Then, the model search for inconsistencies between specification and implementation. The generated report shows those policies that cannot be applied on the infrastructure.

7.4 Future work

This thesis opens new and exciting options for future research. We want to highlight the following:

More network functions SDN and NFV are network functions that were brought to software. This thesis focuses on just three features: network invariants, firewall rules, and path–configuration–policy coherence. Checking correctness and verifying other network functionalities. Failure recognition and recovery process are examples of network applications that can be studied.

Other failure consequences This thesis studies how failures (targeted attacks) are propagated on an interdependent network as SDN. However, we use a ER model to model topologies. An interesting behavior to study is modeling the network failures under other physical topologies beyond ASs. For example, studying other topologies as datacenter, campus, or Internet providers networks could improve the design of resilient mechanisms for SDN.

Scale?, Cloud may be the answer Formal methods are being used for checking multiple kinds of systems, especially those related with hardware and software on critical systems. The main concern is the scale of the model because it is a NP problem. However, this issue could be handled by the cloud. SAT evaluation can be accomplished by parallel computing. A key point here is the construction of a structure to store a policy and how to divide the research space for simultaneous search.

Cost quantification Associated to parallel search of satisfiability, calculating the cost of finding SAT solutions and counter examples would be welcomed. Evaluating complexity, formally or empirically, is a critical issue, but estimating the economic cost, on cloud services for example, is a realistic measure of how this kind of approaches can be implemented as practical solutions.

Other logic representations Dynamic representation is useful, if not mandatory, especially for security policies. Using temporal logic, for example Temporal Logic of Actions (TLA), it is possible to model concurrent systems such a network protocol. On the other hand, modal logic allows to model (un)frequent events such as those occurring in a IDS. Currently, we are focused on evaluating alternatives to optimize the conflict detection performance. Mainly, we are considering *model slicing* techniques where the compiler analyses subsets of the network instead of all the elements. In addition, we are working on an implementation that uses KodKod, the internal library used by Alloy, in order to provide support for partial instances to optimize the process. Besides PPL, there are other network policy languages focused on concerns such as fault tolerance and security such as FML and Merlin [Sou+13; Hin+09b]. Future work is planned to support these languages and detect conflicts among policies aimed at dealing with different concerns. Future approaches could include temporal analysis of policies if a policy triggers an action, and those actions activate other policies. These sequences of actions might be observed on changes over routing rules as those analyzed by Reitblatt [Rei+12]

Bibliography

[MLCD16] F. Maldonado-Lopez, E. Calle, and Y. Donoso. “Checking Multi-domain Policies in SDN”. In: *Int. J. of Computers, Communication and Control* 11.3 (June 2016), pp. 393–405.

[MLD13] F. A. Maldonado-Lopez and Y. Donoso. “Reliable Critical Infrastructure: Multiple Failures for Multicast using Multi-Objective Approach”. In: *Int. J. of Computers, Communication and Control* 8 (2013), pp. 79–86.

[RCM16] D. F. Rueda, E. Calle, and J. L. Marzo. “Robustness Comparison of 15 Real Telecommunication Networks: Structural and Centrality Measurements”. In: *Journal of Network and Systems Management* (2016), pp. 1–21.

[MLCD15] F. A. Maldonado-Lopez, E. Calle, and Y. Donoso. “Detection and prevention of firewall-rule conflicts on software-defined networking”. In: *Reliable Networks Design and Modeling (RNDM), 2015 7th International Workshop on.* Oct. 2015, pp. 259–265.

[MLCD14] F. Maldonado-Lopez, J. Chavarriaga, and Y. Donoso. “Detecting Network Policy Conflicts Using Alloy”. In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z.* Ed. by Y. Ait Ameur and K.-D. Schewe. Vol. 8477. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 314–317.

[MLD12] F. A. Maldonado-Lopez and Y. Donoso. “Multicast Session Protection Planner - Tool to plan and deploy protection infrastructure: a SPEA approach”. In: *5 Congrés Internacional de l’Associació Catalana d’Intel·ligencia Artificial.* Ed. by D. Riaño, E. Onaindia, and M. Cazorla. Vol. 248. Frontiers in Artificial Intelligence and Applications. 978-1-61499-138-0, 2012, pp. 191–200.

[MLCD11] F. A. Maldonado-Lopez, J. Corchuelo, and Y. Donoso. “Unavailability and cost minimization in a parallel-series system using multi-objective evolutionary algorithms”. In: *Proceedings of the 2011 international conference on applied, numerical and computational mathematics, and Proceedings of the 2011 international conference on Computers, digital communications and computing.* ICANCM’11/ICDCC’11. Barcelona, Spain: World Scientific, Engineering Academy, and Society (WSEAS), 2011, pp. 33–38.

[ASH03] E. Al-Shaer and H. Hamed. “Firewall Policy Advisor for anomaly discovery and rule editing”. In: *Integrated Network Management, 2003. IFIP/IEEE Eighth International Symposium on.* 2003, pp. 17–30.

[SLX01] G. Stone, B. Lundy, and G. Xie. “Network Policy Languages: A survey and a new approach”. In: *IEEE Network* 15.1 (2001), pp. 10 –21.

[McK+08b] N. McKeown et al. “OpenFlow: enabling innovation in campus networks”. In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Mar. 2008), pp. 69–74.

[Mar+14] J. Martins et al. “ClickOS and the Art of Network Function Virtualization”. In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation.* NSDI’14. Seattle, WA: USENIX Association, 2014, pp. 459–473.

[OGP03] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. “Why Do Internet Services Fail, and What Can Be Done About It?” In: *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4.* USITS’03. Seattle, WA: USENIX Association, 2003, pp. 1–1.

[OM08] K. O’Callaghan and S. Mariappanadar. “Restoring Service after an Unplanned IT Outage”. In: *IT Professional* 10.3 (2008), pp. 40–45.

[BP01] A. B. Brown and D. A. Patterson. “To Err is Human”. In: *Proceedings of the First Workshop on Evaluating and Architecting System dependabilitY (EASY ’01.* 2001.

[Ker04] Z. Kerravala. *As the Value of Enterprise Networks Escalates, So Does the Need for Configuration Management.* Tech. rep. Yankee Group, 2004.

[Jun] *What’s Behind Network Downtime?* Tech. rep. Juniper Networks, 2008.

[Woo10] A. Wool. “Trends in Firewall Configuration Errors: Measuring the Holes in Swiss Cheese”. In: *Internet Computing, IEEE* 14.4 (July 2010), pp. 58–65.

[Abr96] J.-R. Abrial. *The B-book: Assigning Programs to Meanings.* New York, NY, USA: Cambridge University Press, 1996.

[MWA02] R. Mahajan, D. Wetherall, and T. Anderson. “Understanding BGP Misconfiguration”. In: *SIGCOMM Comput. Commun. Rev.* 32.4 (Aug. 2002), pp. 3–16.

[MZ09] S. Malik and L. Zhang. “Boolean Satisfiability from Theoretical Hardness to Practical Success”. In: *Commun. ACM* 52.8 (Aug. 2009), pp. 76–82.

[PW10] M. Pătraşcu and R. Williams. “On the Possibility of Faster SAT Algorithms”. In: *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms.* SODA ’10. Austin, Texas: Society for Industrial and Applied Mathematics, 2010, pp. 1065–1075.

[Jac03] D. Jackson. “Alloy: A logical modelling language”. In: *ZB* 3 (2003), p. 1.

[VN11] S. J. Vaughan-Nichols. “OpenFlow: The Next Generation of the Network?” In: *Computer* 44.8 (2011), pp. 13–15.

[Koh+00] E. Kohler et al. “The click modular router”. In: *ACM Trans. Comput. Syst.* 18.3 (Aug. 2000), pp. 263–297.

[Fea+04] N. Feamster et al. “The case for separating routing from routers”. In: *Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture.* FDNA ’04. Portland, Oregon, USA: ACM, 2004, pp. 5–12.

[Cae+05] M. Caesar et al. "Design and implementation of a Routing Control Platform". In: *Proc. Networked Systems Design and Implementation.* 2005, pp. 15–28.

[Gre+05] A. Greenberg et al. "A clean slate 4D approach to network control and management". In: *SIGCOMM Comput. Commun. Rev.* 35.5 (Oct. 2005), pp. 41–54.

[Cas+06] M. Casado et al. "SANE: A Protection Architecture for Enterprise Networks". In: *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15.* USENIX-SS'06. Vancouver, B.C., Canada: USENIX Association, 2006, pp. 1–15.

[Cas+07] M. Casado et al. "Ethane: taking control of the enterprise". In: *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications.* SIGCOMM '07. Kyoto, Japan: ACM, 2007, pp. 1–12.

[Cas+09] M. Casado et al. "Rethinking enterprise network control". In: *IEEE/ACM Trans. Netw.* 17.4 (Aug. 2009), pp. 1270–1283.

[McK+08a] N. McKeown et al. *OpenFlow: Enabling Innovation in Campus Networks.* White paper. Stanford University et al., 2008.

[Ope11] Open Networking Foundation. *OpenFlow Switch Specification.* Tech. rep. ONF Open Networking Foundation, 2011.

[Ope12] Open Networking Foundation. *OpenFlow Switch Specification.* Tech. rep. ONF Open Networking Foundation, 2012.

[Ope13] Open Networking Foundation. *OpenFlow Switch Specification.* Tech. rep. ONF Open Networking Foundation, 2013.

[Cur+11] A. R. Curtis et al. "DevoFlow: Scaling Flow Management for High-performance Networks". In: *SIGCOMM Comput. Commun. Rev.* 41.4 (Aug. 2011), pp. 254–265.

[NSBT14] X.-N. Nguyen, D. Saucez, C. Barakat, and T. Turletti. "Optimizing Rules Placement in OpenFlow Networks: Trading Routing for Better Efficiency". In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking.* HotSDN '14. Chicago, Illinois, USA: ACM, 2014, pp. 127–132.

[Mal+04] D. A. Maltz et al. "Routing design in operational networks: a look from the inside". In: *SIGCOMM Comput. Commun. Rev.* 34.4 (Aug. 2004), pp. 27–40.

[Gud+08] N. Gude et al. "NOX: towards an operating system for networks". In: *SIGCOMM Comput. Commun. Rev.* 38.3 (July 2008), pp. 105–110.

[Kim+12] H. Kim et al. *Lithium: Event-Driven Network Control.* Tech. rep. Georgia Institute of Technology and Yale University, 2012.

[Eri13] D. Erickson. "The Beacon OpenFlow Controller". In: *HotSDN.* ACM. 2013.

[Big12] Big Switch Networks. *Project Floodlight. [Online] http://www.projectfloodlight.org/floodlight.* 2012.

[Kop+10] T. Koponen et al. "Onix: a distributed control platform for large-scale production networks". In: *Proceedings of the 9th USENIX conference on*

Operating systems design and implementation. OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–6.

[TG10] A. Tootoonchian and Y. Ganjali. "HyperFlow: A Distributed Control Plane for OpenFlow". In: *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking.* INM/WREN'10. San Jose, CA: USENIX Association, 2010, pp. 3–3.

[CCN11] Z. Cai, A. L. Cox, and T. S. E. Ng. *Maestro: Balancing Fairness, Latency and Throughput in the OpenFlow Control Plane.* Tech. rep. Rice University, 2011.

[LPSY14] B. Lee, S. H. Park, J. Shin, and S. Yang. "IRIS: The Openflow-based Recursive SDN controller". In: *Advanced Communication Technology (ICACT), 2014 16th International Conference on.* Feb. 2014, pp. 1227–1231.

[LHM10] B. Lantz, B. Heller, and N. McKeown. "A Network in a Laptop: Rapid Prototyping for Software-defined Networks". In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks.* Hotnets-IX. Monterey, California: ACM, 2010, 19:1–19:6.

[Fos+11] N. Foster et al. "Frenetic: a network programming language". In: *SIGPLAN Not.* 46.9 (Sept. 2011), pp. 279–291.

[MFHW12] C. Monsanto, N. Foster, R. Harrison, and D. Walker. "A Compiler and Run-time System for Network Programming Languages". In: *SIGPLAN Not.* 47.1 (Jan. 2012), pp. 217–230.

[Mon+13] C. Monsanto et al. "Composing Software Defined Networks". In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI).* 2013, pp. 1–14.

[Sou+13] R. Soulé et al. "Managing the Network with Merlin". In: *ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets'13).* 2013.

[And+14] C. J. Anderson et al. "NetKAT: Semantic Foundations for Networks". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL '14. San Diego, California, USA: ACM, 2014, pp. 113–126.

[Voe+13] A. Voellmy et al. "Maple: Simplifying SDN Programming Using Algorithmic Policies". In: *SIGCOMM Comput. Commun. Rev.* 43.4 (Aug. 2013), pp. 87–98.

[NFSK14] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi. "Tierless Programming and Reasoning for Software-Defined Networks". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14).* Seattle, WA: USENIX Association, Apr. 2014, pp. 519–531.

[WLSF11] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. "OFRewind: Enabling Record and Replay Troubleshooting for Networks". In: *Proceedings of the 2011 USENIX conference on USENIX annual technical conference.* USENIXATC'11. Portland, OR: USENIX Association, 2011, pp. 29–29.

[Han+12] N. Handigol et al. "Where is the debugger for my software-defined network?" In: *Proceedings of the first workshop on Hot topics in software defined networks - HotSDN '12.* New York, New York, USA: ACM Press, Aug. 2012, p. 55.

[DSB14] R. Durairajan, J. Sommers, and P. Barford. “Controller-agnostic SDN Debugging”. In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. CoNEXT ’14. Sydney, Australia: ACM, 2014, pp. 227–234.

[Fos+13] N. Foster et al. “Languages for software-defined networks”. In: *Communications Magazine, IEEE* 51.2 (2013), pp. 128–134.

[Han+14] N. Handigol et al. “I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks”. In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 71–85.

[Das+11] S. Das et al. “Application-aware aggregation and traffic engineering in a converged packet-circuit network”. In: *Optical Fiber Communication Conference and Exposition (OFC/NFOEC), 2011 and the National Fiber Optic Engineers Conference*. 2011, pp. 1–3.

[WNS12] G. Wang, T. E. Ng, and A. Shaikh. “Programming Your Network at Run-time for Big Data Applications”. In: *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. HotSDN ’12. Helsinki, Finland: ACM, 2012, pp. 103–108.

[WSY11] K. C. Webb, A. C. Snoeren, and K. Yocum. “Topology Switching for Data Center Networks”. In: *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*. Hot-ICE’11. Boston, MA: USENIX Association, 2011, pp. 14–14.

[She+09] R. Sherwood et al. “Flowvisor: A network virtualization layer”. In: *OpenFlow Switch Consortium, Tech. Rep* (2009).

[Kop+14] T. Koponen et al. “Network Virtualization in Multi-tenant Datacenters”. In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI’14. Seattle, WA: USENIX Association, 2014, pp. 203–216.

[WBR11] R. Wang, D. Butnariu, and J. Rexford. “OpenFlow-based Server Load Balancing Gone Wild”. In: *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*. Hot-ICE’11. Boston, MA: USENIX Association, 2011, pp. 12–12.

[AF+10] M. Al-Fares et al. “Hedera: dynamic flow scheduling for data center networks”. In: *Proceedings of the 7th USENIX conference on Networked systems design and implementation*. NSDI’10. San Jose, California: USENIX Association, 2010, pp. 19–19.

[AKL13] S. Agarwal, M. Kodialam, and T. Lakshman. “Traffic engineering in software defined networks”. In: *INFOCOM, 2013 Proceedings IEEE*. 2013, pp. 2211–2219.

[Rot+12] C. E. Rothenberg et al. “Revisiting Routing Control Platforms with the Eyes and Muscles of Software-defined Networking”. In: *Proceedings of the First*

Workshop on Hot Topics in Software Defined Networks. HotSDN '12. Helsinki, Finland: ACM, 2012, pp. 13–18.

[SSZMF12] L. Suresh, J. Schulz-Zander, R. Merz, and A. Feldmann. "Demo: programming enterprise WLANs with odin". In: *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication.* SIGCOMM '12. Helsinki, Finland: ACM, 2012, pp. 279–280.

[FDFE14] J. François, L. Dolberg, O. Festor, and T. Engel. "Network Security Through Software Defined Networking: A Survey". In: *Proceedings of the Conference on Principles, Systems and Applications of IP Telecommunications.* IPTComm '14. Chicago, Illinois: ACM, 2014, 6:1–6:8.

[AX15] I. Alsmadi and D. Xu. "Security of Software Defined Networks: A survey". In: *Computers & Security* (2015).

[Shi+13] S. Shin et al. "FRESCO: Modular Composable Security Services for Software-Defined Networks". In: *NDSS.* The Internet Society, 2013, pp. 1–16.

[MKK11] S. A. Mehdi, J. Khalid, and S. A. Khayam. "Revisiting traffic anomaly detection using software defined networking". In: *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection.* RAID'11. Menlo Park, CA: Springer-Verlag, 2011, pp. 161–180.

[Kem+12] J. Kempf et al. "Scalable Fault Management for OpenFlow". In: *Communications (ICC), 2012 IEEE International Conference on.* 2012, pp. 6606–6610.

[Wic+15] J. A. Wickboldt et al. "Software-defined networking: management requirements and challenges". In: *IEEE Communications Magazine* 53.1 (2015), pp. 278–285.

[Dix+13] A. Dixit et al. "Towards an Elastic Distributed SDN Controller". In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking.* HotSDN '13. Hong Kong, China: ACM, 2013, pp. 7–12.

[HSM12] B. Heller, R. Sherwood, and N. McKeown. "The controller placement problem". In: *Proceedings of the first workshop on Hot topics in software defined networks.* HotSDN '12. Helsinki, Finland: ACM, 2012, pp. 7–12.

[WH13] M. Wasserman and S. Hartman. *Security Analysis of the Open Networking Foundation (ONF) OpenFlow Switch Specification.* Tech. rep. IETF, 2013.

[Feh13] S. Fehr. "Flexible networks for better security". In: *Network Security* 2013.3 (2013), pp. 17 –20.

[LSK09] D. Liginlal, I. Sim, and L. Khansa. "How significant is human error as a cause of privacy breaches? An empirical study and a framework for error management". In: *Computers & Security* 28 (2009), pp. 215 –228.

[QH15] J. Qadir and O. Hasan. "Applying Formal Methods to Networking: Theory, Techniques, and Applications". In: *Communications Surveys Tutorials, IEEE* 17.1 (2015), pp. 256–291.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Ed. by Springer. Springer, 1999.

[CE82] E. M. Clarke and E. A. Emerson. “Logics of Programs: Workshop, Yorktown Heights, New York, May 1981”. In: ed. by D. Kozen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982. Chap. Design and synthesis of synchronization skeletons using branching time temporal logic, pp. 52–71.

[Cla08] E. M. Clarke. “25 Years of Model Checking: History, Achievements, Perspectives”. In: ed. by O. Grumberg and H. Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. Chap. The Birth of Model Checking, pp. 1–26.

[ASA11] E. Al-Shaer and M. Alsaleh. “ConfigChecker: A tool for comprehensive security configuration analytics”. In: *Configuration Analytics and Automation (SAFECONFIG), 2011 4th Symposium on.* 2011, pp. 1–2.

[Rei+12] M. Reitblatt et al. “Abstractions for network update”. In: *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication.* SIGCOMM ’12. Helsinki, Finland: ACM, 2012, pp. 323–334.

[KKPB07] M. Karsten, S. Keshav, S. Prasad, and M. Beg. “An Axiomatic Basis for Communication”. In: *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications.* SIGCOMM ’07. Kyoto, Japan: ACM, 2007, pp. 217–228.

[Jac02] D. Jackson. “Alloy: A Lightweight Object Modelling Notation”. In: *ACM Trans. Softw. Eng. Methodol.* 11.2 (Apr. 2002), pp. 256–290.

[KT06] J. Kleinberg and É. Tardos. *Algorithm Design.* Alternative Etext Formats. Pearson/Addison-Wesley, 2006.

[GPFW96] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah. “Algorithms for the Satisfiability (SAT) Problem: A Survey”. In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science.* American Mathematical Society, 1996, pp. 19–152.

[Gan] “SAT-Based Scalable Formal Verification Solutions”. In: Boston, MA: Springer US, 2007. Chap. SAT-Based Verification Framework, pp. 247–261.

[Mos+01] M. W. Moskewicz et al. “Chaff: Engineering an Efficient SAT Solver”. In: *Proceedings of the 38th Annual Design Automation Conference.* DAC ’01. Las Vegas, Nevada, USA: ACM, 2001, pp. 530–535.

[SE05] N. Sörensson and N. Een. *MiniSat v1.13 - A SAT solver with conflict-clause minimization.* Tech. rep. Chalmers University of Technology, Sweden, 2005.

[DM06] B. Dutertre and L. de Moura. *Integrating Simplex with DPPL(T).* Tech. rep. SRI-CSL-06-01. Computer Science Laboratory, SRI International, May 2006.

[Bry86] R. E. Bryant. “Graph-Based Algorithms for Boolean Function Manipulation”. In: *IEEE Transactions on Computers* C-35.8 (1986), pp. 677–691.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. “Tools and Algorithms for the Construction and Analysis of Systems: 5th International Conference, TACAS'99 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99 Amsterdam, The Netherlands, March 22–28, 1999 Proceedings". In: ed. by W. R. Cleaveland. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. Chap. Symbolic Model Checking without BDDs, pp. 193–207.

[BEL75] R. S. Boyer, B. Elspas, and K. N. Levitt. "SELECT-a formal system for testing and debugging programs by symbolic execution". In: *ACM SIGPLAN Notices - International Conference on Reliable Software* (1975).

[BUZC11] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. "Parallel Symbolic Execution for Automated Real-world Software Testing". In: *Proceedings of the Sixth Conference on Computer Systems.* EuroSys '11. Salzburg, Austria: ACM, 2011, pp. 183–198.

[McG12] R. McGeer. "Verification of switching network properties using satisfiability". In: *Communications (ICC), 2012 IEEE International Conference on.* 2012, pp. 6638–6644.

[Hol97] G. J. Holzmann. "The model checker SPIN". In: *IEEE Transactions on Software Engineering* 23.5 (1997), pp. 279–295.

[CCGR99] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. "Computer Aided Verification: 11th International Conference, CAV'99 Trento, Italy, July 6–10, 1999 Proceedings". In: ed. by N. Halbwachs and D. Peled. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. Chap. NuSMV: A New Symbolic Model Verifier, pp. 495–499.

[End75] A. Endres. "An Analysis of Errors and Their Causes in System Programs". In: *SIGPLAN Not.* 10.6 (Apr. 1975), pp. 327–336.

[Can+12] M. Canini et al. "A NICE way to test OpenFlow applications". In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation.* NSDI'12. San Jose, CA: USENIX Association, 2012, pp. 10–10.

[ZKVM14] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. "Automatic Test Packet Generation". In: *IEEE/ACM Transactions on Networking* 22.2 (Apr. 2014), pp. 554–566.

[FB05] N. Feamster and H. Balakrishnan. "Detecting BGP configuration faults with static analysis". In: *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2.* NSDI'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 43–56.

[YCZ10] Z. Yin, M. Caesar, and Y. Zhou. "Towards Understanding Bugs in Open Source Router Software". In: *SIGCOMM Comput. Commun. Rev.* 40.3 (June 2010), pp. 34–40.

[Xie+05] G. Xie et al. "On static reachability analysis of IP networks". In: *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE.* Vol. 3. 2005, 2170–2183 vol. 3.

[XA07] Y. Xie and A. Aiken. “Saturn: A Scalable Framework for Error Detection Using Boolean Satisfiability”. In: *ACM Trans. Program. Lang. Syst.* 29.3 (May 2007).

[GS05] T. G. Griffin and J. L. Sobrinho. “Metarouting”. In: *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications.* SIGCOMM ’05. Philadelphia, Pennsylvania, USA: ACM, 2005, pp. 1–12.

[Nar05] S. Narain. “Network Configuration Management via Model Finding”. In: *Proceedings of the 19th Conference on Large Installation System Administration Conference - Volume 19.* LISA ’05. San Diego, CA: USENIX Association, 2005, pp. 15–15.

[Mai+11] H. Mai et al. “Debugging the data plane with Anteater”. In: *SIGCOMM Comput. Commun. Rev.* 41.4 (Aug. 2011), pp. 290–301.

[KVM12] P. Kazemian, G. Varghese, and N. McKeown. “Header Space Analysis: Static Checking for Networks”. In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12).* San Jose, CA: USENIX, 2012, pp. 113–126.

[NHW12] S. Natarajan, X. Huang, and T. Wolf. “Efficient conflict detection in flow-based virtualized networks”. In: *Computing, Networking and Communications (ICNC), 2012 International Conference on.* 2012, pp. 690–696.

[FS14] S. K. Fayaz and V. Sekar. “Testing Stateful and Dynamic Data Planes with FlowTest”. In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking.* HotSDN ’14. Chicago, Illinois, USA: ACM, 2014, pp. 79–84.

[SNM13] D. Sethi, S. Narayana, and S. Malik. “Abstractions for model checking SDN controllers”. In: *Formal Methods in Computer-Aided Design (FMCAD), 2013.* 2013, pp. 145–148.

[GRF13] A. Guha, M. Reitblatt, and N. Foster. “Machine-verified Network Controllers”. In: *SIGPLAN Not.* 48.6 (June 2013), pp. 483–494.

[Mir+16] S. Mirzaei et al. “Using Alloy to Formally Model and Reason About an OpenFlow Network Switch”. In: *CoRR* abs/1604.00060 (2016).

[Nar13] S. Narain. “ConfigAssure: A Science of Configuration”. In: *Military Communications Conference, MILCOM 2013 - 2013 IEEE.* 2013, pp. 1497–1498.

[Kim+15] H. Kim et al. “Kinetic: Verifiable Dynamic Network Control”. In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15).* Oakland, CA: USENIX Association, May 2015, pp. 59–72.

[Kuz+12] M. Kuzniar et al. “A SOFT way for openflow switch interoperability testing”. In: *Proceedings of the 8th international conference on Emerging networking experiments and technologies.* CoNEXT ’12. Nice, France: ACM, 2012, pp. 265–276.

[KZCG12] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. "VeriFlow: verifying network-wide invariants in real time". In: *Proceedings of the first workshop on Hot topics in software defined networks.* HotSDN '12. Helsinki, Finland: ACM, 2012, pp. 49–54.

[Khu+13] A. Khurshid et al. "VeriFlow: Verifying Network-Wide Invariants in Real Time". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI).* 2013.

[Kaz+13] P. Kazemanian et al. "Real time Network Policy Checking Using Header Space Analysis". In: *Proceeding on Network System Design and Implementation (NSDI).* USENIX Association, 2013.

[Hin+09b] T. L. Hinrichs et al. "Practical Declarative Network Management". In: *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking.* WREN '09. Barcelona, Spain: ACM, 2009, pp. 1–10.

[LBGV13] N. Lopes, N. Bjorner, P. Godefroid, and G. Varghese. *Network Verification in the Light of Program Verification.* Tech. rep. 2013.

[YL16] H. Yang and S. S. Lam. "Real-Time Verification of Network Properties Using Atomic Predicates". In: *IEEE/ACM Transactions on Networking* 24.2 (2016), pp. 887–900.

[Woo04] A. Wool. "A quantitative study of firewall configuration errors". In: *Computer* 37.6 (2004), pp. 62–67.

[ASH04b] E. Al-Shaer and H. Hamed. "Modeling and Management of Firewall Policies". In: *Network and Service Management, IEEE Transactions on* 1.1 (2004), pp. 2–10.

[ASH04a] E. Al-Shaer and H. Hamed. "Discovery of policy anomalies in distributed firewalls". In: *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies.* Vol. 4. 2004, 2605–2616 vol.4.

[Yua+06] L. Yuan et al. "FIREMAN: a toolkit for firewall modeling and analysis". In: *Security and Privacy, 2006 IEEE Symposium on.* 2006, 15 pp.–213.

[GL07] M. G. Gouda and A. X. Liu. "Structured firewall design". In: *Computer Networks* 51.4 (2007), pp. 1106 –1120.

[Liu08] A. Liu. "Formal Verification of Firewall Policies". In: *Communications, 2008. ICC '08. IEEE International Conference on.* 2008, pp. 1494–1498.

[JS09] A. Jeffrey and T. Samak. "Model Checking Firewall Policy Configurations". In: *Policies for Distributed Systems and Networks, 2009. POLICY 2009. IEEE International Symposium on.* 2009, pp. 60–67.

[ZMMN12] S. Zhang, A. Mahmoud, S. Malik, and S. Narain. "Verification and synthesis of firewalls using SAT and QBF". In: *Network Protocols (ICNP), 2012 20th IEEE International Conference on.* 2012, pp. 1–6.

[Son+13] S. Son et al. "Model checking invariant security properties in OpenFlow". In: *Communications (ICC), 2013 IEEE International Conference on.* June 2013, pp. 1974–1979.

[ASMEAE09] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. Elbadawi. “Network configuration in a box: towards end-to-end verification of network reachability and security”. In: *Network Protocols, 2009. ICNP 2009. 17th IEEE International Conference on.* 2009, pp. 123–132.

[HHAZ14] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao. “FLOWGUARD: Building Robust Firewalls for Software-defined Networks”. In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking.* HotSDN ’14. Chicago, Illinois, USA: ACM, 2014, pp. 97–102.

[Ste+99] M. Stevens et al. *Policy Framework.* Tech. rep. Internet-Draft. IETF, 1999.

[Wes+01] A. Westerinen et al. *Terminology for Policy-Based Management.* Tech. rep. RFC 3198. IETF, 2001.

[BA07] R. Boutaba and I. Aib. “Policy-based Management: A Historical Perspective”. In: *Journal of Network and Systems Management* 15.4 (2007), pp. 447–480.

[Str03] J. Strassner. *Policy-Based Network Management: Solutions for the Next Generation (The Morgan Kaufmann Series in Networking).* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[Dur+00] D. Durham et al. *The COPS (Common Open Policy Service) Protocol.* Tech. rep. RFC 2748. IETF, 2000.

[Cla89] D. Clark. *Policy Routing in Internet Protocols.* Tech. rep. RFC 1102. IETF, 1989.

[LBN99] J. Lobo, B. Bhatia, and S. Naqvi. “A policy description language”. In: *Proc. of AAAI.* 1999, pp. 291–298.

[DDLS01] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. “Policies for Distributed Systems and Networks: International Workshop, POLICY 2001 Bristol, UK, January 29–31, 2001 Proceedings”. In: ed. by M. Sloman, E. C. Lupu, and J. Lobo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. Chap. The Ponder Policy Specification Language, pp. 18–38.

[GM03] S. Godik and T. Moses. *eXtensible Access Control Markup Language (XACML).* Tech. rep. OASIS, 2003.

[Loo+06] B. T. Loo et al. “Declarative Networking: Language, Execution and Optimization”. In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data.* SIGMOD ’06. Chicago, IL, USA: ACM, 2006, pp. 97–108.

[Guv03] A. Guven. “Speeding Up a Path-Based Policy Language Compiler”. MA thesis. Monterrey, California: Naval Postgraduate School, 2003.

[AWY08] R. Alimi, Y. Wang, and Y. R. Yang. “Shadow Configuration As a Network Management Primitive”. In: *SIGCOMM Comput. Commun. Rev.* 38.4 (Aug. 2008), pp. 111–122.

[CLN03] J. Chomicki, J. Lobo, and S. Naqvi. “Conflict resolution using logic programming”. In: *Knowledge and Data Engineering, IEEE Transactions on* 15.1 (2003), pp. 244–249.

[Mog+13] J. C. Mogul et al. “Corybantic: Towards the Modular Composition of SDN Control Programs”. In: *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks.* HotNets-XII. College Park, Maryland: ACM, 2013, 1:1–1:7.

[Sun+14] P. Sun et al. “A Network-state Management Service”. In: *SIGCOMM Comput. Commun. Rev.* 44.4 (Aug. 2014), pp. 563–574.

[AuY+14] A. AuYoung et al. “Democratic Resolution of Resource Conflicts Between SDN Control Programs”. In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies.* CoNEXT ’14. Sydney, Australia: ACM, 2014, pp. 391–402.

[MDTW14] R. Majumdar, S. Deep Tetali, and Z. Wang. “Kuai: A model checker for software-defined networks”. In: *Formal Methods in Computer-Aided Design (FMCAD), 2014.* 2014, pp. 163–170.

[Dil96] D. L. Dill. “The Murphi Verification System”. In: *Proceedings of the 8th International Conference on Computer Aided Verification.* CAV ’96. London, UK, UK: Springer-Verlag, 1996, pp. 390–393.

[KCZ13] P. Kazemian, M Change, and H Zheng. “Real Time Network Policy Checking Using Header Space Analysis”. In: *USENIX Symposium on Networked Systems Design and Implementation* (2013), pp. 1–13.

[Hin+09a] T. L. Hinrichs et al. *Expressing and Enforcing Flow-Based Network Security Policies.* Tech. rep. University of Chicago, 2009.

[BRA10] J. R. Ballard, I. Rae, and A. Akella. “Extensible and scalable network monitoring using OpenSAFE”. In: *Proceedings of the 2010 internet network management conference on Research on enterprise networking.* INM/WREN’10. San Jose, CA: USENIX Association, 2010, pp. 8 –11.

[RFRW11] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. “Consistent updates for software-defined networks: change you can believe in!” In: *Proceedings of the 10th ACM Workshop on Hot Topics in Networks.* HotNets-X. Cambridge, Massachusetts: ACM, 2011, pp. 71–76.

[KRRW12] N. Kang, J. Reich, J. Rexford, and D. Walker. “Policy transformation in software defined networks”. In: *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication.* SIGCOMM ’12. Helsinki, Finland: ACM, 2012, pp. 309–310.

[YTG13] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali. “On scalability of software-defined networking”. In: *Communications Magazine, IEEE* 51.2 (2013), pp. 136–141.

[Qaz+13] Z. A. Qazi et al. “SIMPLE-fying Middlebox Policy Enforcement Using SDN”. In: *SIGCOMM Comput. Commun. Rev.* 43.4 (Aug. 2013), pp. 27–38.

[VKF12] A. Voellmy, H. Kim, and N. Feamster. “Procera: a language for high-level reactive network control”. In: *Proceedings of the first workshop on Hot topics in software defined networks.* HotSDN ’12. Helsinki, Finland: ACM, 2012, pp. 43–48.

[KF13] H. Kim and N. Feamster. “Improving network management with software defined networking”. In: *Communications Magazine, IEEE* 51.2 (2013), pp. 114–119.

[NRFC09] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark. “Resonance: Dynamic Access Control for Enterprise Networks”. In: *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking.* WREN ’09. Barcelona, Spain: ACM, 2009, pp. 11–18.

[Fea+10] N. Feamster et al. “Decoupling policy from configuration in campus and enterprise networks”. In: *Local and Metropolitan Area Networks (LANMAN), 2010 17th IEEE Workshop on.* 2010, pp. 1–6.

[ASAH10] E. Al-Shaer and S. Al-Haj. “FlowChecker: configuration analysis and verification of federated openflow infrastructures”. In: *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration.* SafeConfig ’10. Chicago, Illinois, USA: ACM, 2010, pp. 37–44.

[Kan+13] M. Kang et al. “Formal Modeling and Verification of SDN-OpenFlow”. In: *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013.* 2013, pp. 481–482.

[SPNR13] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. “SymNet: Static Checking for Stateful Networks”. In: *Proceedings of the 2013 Workshop on Hot Topics in Middleboxes and Network Function Virtualization.* HotMiddlebox ’13. Santa Barbara, California, USA: ACM, 2013, pp. 31–36.

[Lew09] T. G. Lewis. *Network Science: Theory and Practice.* John Wiley & Sons, Inc., 2009.

[Die10] R. Diestel. *Graph Theory.* Springer-Verlag Berlin Heidelberg, 2010.

[HR04] D. Harel and B. Rumpe. “Meaningful Modeling: What’s the Semantics of “Semantics”?” In: *Computer* 37.10 (2004), pp. 64–72.

[Jac06] D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[TCJ08] E. Torlak, F. S.-H. Chang, and D. Jackson. “FM 2008: Formal Methods: 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008 Proceedings”. In: ed. by J. Cuellar, T. Maibaum, and K. Sere. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. Chap. Finding Minimal Unsatisfiable Cores of Declarative Specifications, pp. 326–341.

[Sto00] G. Stone. “A Path-Based Network Policy Language”. PhD thesis. Monterrey, California: Naval Postgraduate School, 2000.

[PSS11] D. Power, M. Slaymaker, and A. Simpson. “Automatic Conformance Checking of Role-Based Access Control Policies via Alloy”. In: *Engineering Secure Software and Systems: Third International Symposium, ESSoS 2011, Madrid, Spain, February 9-10, 2011. Proceedings.* Ed. by Ú. Erlingsson, R. Wieringa, and N. Zannone. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 15–28.

[KD12] K. Kant and C. Deccio. “Handbook on Securing Cyber-Physical Critical Infrastructure”. In: ed. by S. Das, K. Kant, and N. Zhang. Morgan Kaufmann, 2012. Chap. Security and Robustness in the Internet Infrastructure.

[APS14] M. Avalle, A. Pironti, and R. Sisto. “Formal verification of security protocol implementations: a survey”. In: *Formal Aspects of Computing* 26.1 (2014), pp. 99–123.

[AS14] E. Al-Shaer. “Classification and Discovery of Firewalls Policy Anomalies”. In: *Automated Firewall Analytics.* Springer International Publishing, 2014, pp. 1–24.

[RCGF13] M. Reitblatt, M. Canini, A. Guha, and N. Foster. “FatTire: declarative fault tolerance for software-defined networks”. In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking.* HotSDN ’13. Hong Kong, China: ACM, 2013, pp. 109–114.

[Por+12] P. Porras et al. “A security enforcement kernel for OpenFlow networks”. In: *Proceedings of the first workshop on Hot topics in software defined networks.* HotSDN ’12. Helsinki, Finland: ACM, 2012, pp. 121–126.

[SH15] S. Scott-Hayward. “Design and deployment of secure, robust, and resilient SDN controllers”. In: *Network Softwarization (NetSoft), 2015 1st IEEE Conference on.* 2015, pp. 1–5.

[EK13] W. Ellens and R. E. Kooij. “Graph measures and network robustness”. In: *CoRR* abs/1311.5064 (2013).

[Seg11] J. Segovia. “Robustness against Large-Scale Failures in Communications Networks”. PhD thesis. University of Girona, 2011.

[LHKA12] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. “F10: A Fault-Tolerant Engineered Network”. In: *USENIX NSDI 2013.* 2012, pp. 399–412.

[Cal+10] E. Calle et al. “A multiple failure propagation model in GMPLS-based networks”. In: *Network, IEEE* 24.6 (2010), pp. 17–22.

[CLM04] P. Crucitti, V. Latora, and M. Marchiori. “Model for cascading failures in complex networks”. In: *Phys. Rev. E* 69 (4 2004), p. 045104.

[SQZ14] D. H. Shin, D. Qian, and J. Zhang. “Cascading effects in interdependent networks”. In: *IEEE Network* 28.4 (June 2014), pp. 82–87.

[Ste+11a] J. P. G. Sterbenz et al. “Modelling and analysis of network resilience”. In: *2011 Third International Conference on Communication Systems and Networks (COMSNETS 2011).* 2011, pp. 1–10.

[SSYS10] A. Sydney, C. Scoglio, M. Youssef, and P. Schumm. “Characterising the Robustness of Complex Networks”. In: *Int. J. Internet Technol. Secur. Syst.* 2.3/4 (Dec. 2010), pp. 291–320.

[Sch+11] C. M. Schneider et al. “Mitigation of malicious attacks on networks”. In: *Proceedings of the National Academy of Sciences* 108.10 (2011), pp. 3838–3841.

[BRSBJ15] I. Bachmann, P. Reyes, A. Silva, and J. Bustos-Jimenez. “Miuz: measuring the impact of disconnecting a node”. In: *2015 34th International Conference of the Chilean Computer Science Society (SCCC).* 2015, pp. 1–6.

[HKYH02] P. Holme, B. J. Kim, C. N. Yoon, and S. K. Han. “Attack vulnerability of complex networks”. In: *Phys. Rev. E* 65 (5 2002), p. 056109.

[SSSK08] A. Sydney, C. M. Scoglio, P. Schumm, and R. E. Kooij. “ELASTICITY: Topological Characterization of Robustness in Complex Networks”. In: *CoRR* (2008).

[YKS11] M. Youssef, R. Kooij, and C. Scoglio. “Viral conductance: Quantifying the robustness of networks with respect to spread of epidemics”. In: *Journal of Computational Science* 2.3 (2011), pp. 286 –298.

[Mie12] P. V. Mieghem. “The viral conductance of a network”. In: *Computer Communications* 35.12 (2012), pp. 1494 –1506.

[Ste+11b] J. Sterbenz et al. “Evaluation of network resilience, survivability, and disruption tolerance: analysis, topology generation, simulation, and experimentation”. In: *Telecommunication Systems* (2011), pp. 1–32.

[Wu95] T.-H. Wu. “Emerging technologies for fiber network survivability”. In: *Communications Magazine, IEEE* 33.2 (1995), pp. 58 –59, 62–74.

[Tor92] D. Torrieri. “Algorithms for finding an optimal set of short disjoint paths in a communication network”. In: *Communications, IEEE Transactions on* 40.11 (1992), pp. 1698–1702.

[HH07] A. Haider and R. Harris. “Recovery techniques in next generation networks”. In: *Communications Surveys Tutorials, IEEE* 9.3 (2007), pp. 2–17.

[JOK03] B. G. Jozsa, D. Orincsay, and A. Kern. “Surviving multiple network failures using shared backup path protection”. In: *Computers and Communication, 2003. (ISCC 2003). Proceedings. Eighth IEEE International Symposium on.* 2003, 1333–1340 vol.2.

[Sch03] D. Schupke. “Multiple failure survivability in WDM networks with p-cycles”. In: *Circuits and Systems, 2003. ISCAS ’03. Proceedings of the 2003 International Symposium on.* Vol. 3. 2003, III–866–III–869 vol.3.

[KAJ09] M. Kiaei, C. Assi, and B. Jaumard. “A Survey on the p-Cycle Protection Method”. In: *Communications Surveys Tutorials, IEEE* 11.3 (Apr. 2009), pp. 53 –70.

[LCCM10] Z. Liang, W. A. Chaovalitwongse, M. Cha, and S. B. Moon. “Redundant multicast routing in multilayer networks with shared risk resource groups: Complexity, models and algorithms”. In: *Computers & Operations Research* 37.10 (2010), pp. 1731 –1739.

[Pat+02] D. Patterson et al. *Recovery Oriented Computing (ROC): Motivation, Definition, Techniques,* tech. rep. Berkeley, CA, USA: University of California at Berkeley, 2002.

[PSB11] E. Palkopoulou, D. Schupke, and T. Bauschert. “Shared backup router resources: realizing virtualized network resilience”. In: *Communications Magazine, IEEE* 49.5 (2011), pp. 140–146.

[RSM03] S. Ramamurthy, L. Sahasrabuddhe, and B. Mukherjee. “Survivable WDM mesh networks”. In: *Lightwave Technology, Journal of* 21.4 (Apr. 2003), pp. 870 –883.

[Sha+11] S. Sharma et al. “Enabling fast failure recovery in OpenFlow networks”. In: *Design of Reliable Communication Networks (DRCN), 2011 8th International Workshop on the.* 2011, pp. 164–171.

[Ku13] M. Kuźniar et al. “Automatic failure recovery for software-defined networks”. In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking.* HotSDN ’13. Hong Kong, China: ACM, 2013, pp. 159–160.

[PBH10] R. Parshani, S. V. Buldyrev, and S. Havlin. “Interdependent Networks: Reducing the Coupling Strength Leads to a Change from a First to Second Order Percolation Transition”. In: *Physical Review Letters* (2010).

[Bul+10] S. V. Buldyrev et al. “Catastrophic cascade of failures in interdependent networks”. In: *Nature* 464 (Apr. 2010), pp. 1025–1028.

[MMCM12] M. Manzano, J. L. Marzo, E. Calle, and A. Manolovay. “Robustness analysis of real network topologies under multiple failure scenarios”. In: *Networks and Optical Communications (NOC), 2012 17th European Conference on.* June 2012, pp. 1–6.

[IKSW13] S. Iyer, T. Killingback, B. Sundaram, and Z. Wang. “Attack Robustness and Centrality of Complex Networks”. In: *PLoS ONE* (2013).

[CD15] S. Chattopadhyay and H. Dai. “Towards Optimal Link Patterns for Robustness of Interdependent Networks against Cascading Failures”. In: *2015 IEEE Global Communications Conference (GLOBECOM).* 2015, pp. 1–6.

[Hua+11] X. Huang et al. “Robustness of interdependent networks under targeted attack”. In: *Phys. Rev. E* 83 (6 2011), p. 065101.

[Ken+14] D. Y. Kenett et al. “Network of Interdependent Networks: Overview of Theory and Applications”. In: *Networks of Networks: The Last Frontier of Complexity.* Ed. by G. D’Agostino and A. Scala. Cham: Springer International Publishing, 2014, pp. 3–36.

[DDTL16] R. Du, G. Dong, L. Tian, and R. Liu. “Targeted attack on networks coupled by connectivity and dependency links”. In: *Physica A: Statistical Mechanics and its Applications* 450 (2016), pp. 687 –699.

[Tre09] J. Treur. “Past–future Separation and Normal Forms in Temporal Predicate Logic Specifications”. In: *J. Algorithms* 64.2-3 (Apr. 2009), pp. 106–124.

Appendix A

PPL interpreter for Alloy

A.1 Checking Network Invariants

The appendices work exactly the same way as chapters, they are numbered with letters rather than numbers though.

```
one sig ALPHA, BRAVO, CHARLIE, DELTA, ECHO extends node{}

one sig exampleTopology extends topology{
nodes = ALPHA + BRAVO + CHARLIE + DELTA + ECHO
links = ALPHA -> BRAVO + ALPHA -> DELTA + BRAVO -> CHARLIE +
BRAVO -> ALPHA + DELTA -> ALPHA + CHARLIE -> BRAVO +
CHARLIE -> DELTA + BRAVO -> ECHO + DELTA -> ECHO +
DELTA -> CHARLIE + ECHO -> BRAVO + ECHO -> DELTA
}
```

A.2 Specification Languages for Policies

Ponder is a language for specification of security policies. The example is a policy that grants four actions to network administrator over switches of its domain.

Table A.1: Syntax and example of Ponder policy [DDLS01].

Syntax	**inst** (**auth+** — **auth-**) policyName "{" **subject** [¡type¿] domain-Scope-Expression ; **target** [¡type¿] domain-Scope-Expression ; **action** action-list ; [**when** constraint-Expression ;] "}"

Example	**inst auth+** switchPolicyOps {
	subject /NetworkAdmin;
	target ¡PolicyT¿ /Nregion/switches;
	action load(), remove(), enable(), disable() ; }

Another language for specifying network-security policies is XACML [GM03] which is a language for specifying access-control. This is XML-based and works as request/response language. A XACML policy contains subjects, targets, resources and actions. Figure A.1 shows the basic syntax and a small example. Target is the condition to satisfy by the subject and the resources, and actions —or effects at rule level— are the policy output. It is a request/response language because operates by queries where the policy server respond to a request with a allow or deny and the set of successive actions —in some cases it returns indeterminate or not applicable. Obviously, this language is implemented as the IETF Policy Framework [Ste+99] with PDP, PEPs and policy repositories.

```
<Subject>
<AttributeAttributeId=”subject:subject-id”
DataType=”data-type:rfc822Name”>
<AttributeValue>bs@simpsons.com</AttributeValue>
</Attribute>
</Subject>

<Resource>
<AttributeAttributeId=”resource:ufs-path”
DataType=”XMLSchema#anyURI”>
<AttributeValue>/medico/record/patient/BartSimpson</AttributeValue>
</Attribute>
</Resource>

<Action><AttributeAttributeId=”action:action-id”
DataType=”XMLSchema#string”>
<AttributeValue>read</AttributeValue>
</Attribute>
</Action>
```

Figure A.1: Example of policy description in XACML language.

Ethane With Ethane, administrator defines fine–grained policies which are declared over high–level names, include the path, and the network binds the packet and the source. Ethane created a language called *Pol-ETH* which followed the AC paradigm.

Table A.2: Syntax and example of Pol–ETH policy [Cas+07].

Syntax	¡namespace¿ := ¡namespace_name¿ = ”[” [¡element¿] ”]” ¡rule¿ := ”[”[hsrc—hdst— apsrc—apdst]=**in**(¡element¿)”]:”¡action¿ ¡action¿ :=[allow—deny—outbound-only—waypoint(¡element¿)] ;

Example	#Goups
	phones = [”gphone”,”rphone”];
	computers = [”private”,”server”];
	#Rules
	# Do not allow phones and private computers to communicate
	[(hsrc=in(”phones”)∧(hdst=in(”computers”))] : deny;
	[(hsrc=in(”computers”)∧(hdst=in(”phones”))] : deny;

FSL is the flow-based security language (FSL) [Hin+09a]. FSL was one of the first policy languages for SDN, it was designed for NOX [Gud+08]. With FSL is possible to specify policies of access controls, isolation, and communication paths.

OpenSAFE expresses routing policies for monitoring using the ALARMS language which

Table A.3: Syntax and example of policy in FSL [Hin+09a], later FML [Hin+09b] language.

Syntax	[**allow—deny**] (Us, Hs, As, Ut, Ht, At, Prot, Req)
Example	$allow(Flow) \Leftarrow Prot = arp$ $allow(Flow) \Leftarrow H_t = auth\_srvr \wedge Prot = http \wedge A_s = patio$ $deny(Flow) \Leftarrow U_s = unknown$

describes a path as inputs, selections, filters and sinks.ALARM Here the input is the span port; selection is the traffic with port number 443; the traffic goes through decrypts and counter filters, and finally is sinks into TCP dump.

Table A.4: Syntax and example of ALARM policy[Hin+09b].

Syntax	(input,selections,filters,sinks)
Example	`span port, port=443, (decryption, counter), TCP dump`

SIMPLE [Qaz+13] which is a policy-enforcement layer for middleboxes like firewalls, IDSs or proxies, with layers L4 to L7 —higher-layers— functionality, into L2 and L3 functions of SDN. SIMPLE expressed traffic as dataflows and allowed integration with legacy middleboxes. The traffic is the result of function composition, and the management is presented an Integer linear problem (ILP) formulation with datapath constraints, sequences of datapaths, and load balancing.

Table A.5: Syntax and example of a SIMPLE policy [Qaz+13]

Syntax	`class=<external ,web>`
	`<src= internal prefix , dst= external prefixes ,`
	`srcport=*, dstport=80,proto=TCP>`

Example	`class=<external ,web>`
	`<src= internal prefix , dst= external prefixes ,`
	`srcport=*, dstport=80,proto=TCP>`

Flowexp language

Listing A.1: Syntax of policies in Flowexp language.

```
Constraint → TRUE | FALSE | !Constraint
| (Constraint | Constraint)
| (Constraint & Constraint)
| PathConstraint | HeaderConstraint ;
PathConstraint → list( Pathlet );
Pathlet → Port Specifier [p ∈ {Pi}]
| Table Specifier [t ∈ {Ti}]
| Skip Next Hop [.]
| Skip Zero or More Hops [.*]
| Beginning of Path [^]
(Source/Sink node)
| End of Path [ $ ]
(Probe node);
HeaderConstraint  → $H_{received} \cap H_{constraint} \neq \phi$
| $H_{received} \subset H_{constraint}$
| $H_{received} == H_{constraint}$ ;
```

Frenetic Whit is a network administrator composes policies in Frenetic, then the compiler translates policies into stream queries and transformations.

Listing A.2: Syntax of policies in Frenetic language.

```
Query        q ::= Select(a)*
Where(pat)*
GroupBy([h_1,...,h_n])*
SplitWhen([h_1, \ldots , h_n])*
Every(n)*
Limit(n)
Agregates  a ::= packets | sizes | counts
Header       h ::= inport | vlan | srcamc | dstmac |
ethtype | srcip | dstip | protocol |
srcport | dstnport | switch
Patterns   pat ::= true_p() | h_pat(n) |
and_pat([pat_1,...,pat_n]) |
or_pat([pat_1,...,pat_n]) |
diff_pat(pat_1,pat_2) | not_pat(pat)
```

Appendix B

Conflict detection

B.1 Detecting other types of conflicts

```
abstract sig path {
nodes : some node,
links : node -> node,
source : one nodes,
target : one nodes
} {
no n : nodes | n in n.^(links) // no loops
target in source.^(links)
source.^(links) in nodes
}

pred isValid ( p : path, t : topology ) {
p.nodes in t.nodes
p.links in t.links
}
```

Target flows Target is the traffic category that a policy conducts. We model target traffic with groups and sequences. Sequence structures are ordered series of elements. First, we define elements with a relation to the next element. The concept of next element allows us to maintain a ordering property. Always, a root or *zero* element is defined to identify where the sequence starts, and a loop-free predicate is mandatory to guarantee ordering consistency.

```
abstract sig Element { next: lone Element}
one sig root extends Element{}
assert consistency{ e:Element in root.^(next)}
```

Groups are defined under set theory abstraction. First, we define an abstract signature of an element, then we define the group and the relation that *contains* the elements within. Also, we declare fact (axiom) that guarantees an element only belongs to a group

```
abstract sig E{}
abstract sig Group{contains : some E}

fact oneIdOneGroup{
all e : E  | no disj  g, g' : Group |
e in (g.contains & g'.contains)
}
```

Conflicts involving complext paths This is an example:

```
// define path xx1
//  { <Alpha, *, Charlie, *, Echo>};
pred xx1 ( p : path ) {
isValid[ p, topologyOne ]       // is a valid path
p.source = Alpha                // starts in Alpha
p.target = Echo                 // ends in Echo

Charlie in p.nodes              // Charlie is a node
Charlie in p.source.^(p.links)  // Charlie is in the path
}

// define path xx2
//  { <Alpha, *, Charlie, *, Echo, *, Delta>};
pred xx2 ( p : path ) {
isValid[ p, topologyOne ]       // is a valid path
p.source = Alpha                // starts in Alpha
p.target = Delta                 // ends in Delta

Charlie in p.nodes              // Charlie is a node
Charlie in p.source.^(p.links)  // Charlie is in the path
Echo in p.nodes                 // Echo is a node
Echo in Charlie.^(p.links)      // Echo is in the path after Charlie
}

// define path xx3
//  { <*, Alpha, *, Charlie, *>};
pred xx2 ( p : path ) {
isValid[ p, topologyOne ]       // is a valid path
Alpha in p.nodes                    // Alpha is a node
Alpha in p.source.^(p.links)        // Alpha is in the path
Charlie in p.nodes                  // Charlie is a node
Charlie in Alpha.^(p.links)         // Charlie is in the path after Alpha
}
```

Conflicts involving IP addresses This is an example:

```
// policy1 smith {<Alpha,*,Charlie>}
//       {sourceIp=10.*.*.*} {*} {deny}
// policy2 smith {<Alpha,*,Charlie>}
//       {sourceIp=10.10.1.*} {*} {permit}
```

Conflicts involving Groups and Users This is an example:

```
// -- NOTE: "John" user is part of
//       the "Students" group
// policy1 smith {<Alpha,*,Charlie>}
//       {group=students} {*} {deny}
// policy2 smith {<Alpha,*,Charlie>}
//       {user=john} {*} {permit}
```

Conflicts involving Actions in Policies This is an example:

```
// -- NOTE: Both policies set different values
//       for priority
// policy1 smith {<Alpha,*,Charlie>}
//       {traffic_class=video} {*} {priority:=1}
// policy2 smith {<Alpha,*,Charlie>}
//       {traffic_class=video} {*} {priority:=2}
```

Conflicts involving Policy Maker priorities This is an example:

```
// -- NOTE: smith and john has different policy
//       maker priorities ?
// policy1 smith {<Alpha,*,Charlie>}
//       {traffic_class=video} {*} {deny}
// policy2 john {<Alpha,*,Charlie>}
//       {traffic_class=video} {*} {permit}
```

```
one sig a0, a1, a2, a3, a4, a5 extends Address{}

fact TreeCreation{
AddressTree.root = a0      // *.*.*.* equivalent
a1 + a4 + a5 in a0.children
a2 + a3 in a1.children
}

abstract sig rule{
id:       one Element,
proto:    one Protocol,
srcAddr: one Address,
srcPort: one Port,
dstAddr: one Address,
dstPort: one Port,
action:   one Action
}

//1,tcp,140.192.37.20,any,*.*.*.*,80,deny
one sig r1 extends rule{}
{
id = i1
proto = TCP
srcAddr = a2
srcPort = p0
dstAddr = a0
dstPort = p80
action = deny
}

/*  SHADOWING
* a previous rule matches all packets that match this rule //
*/
assert Shadowing {
no x,y: rule {
isBefore[x.id,y.id]                    // X < Y
(equals[x,y] or inclusiveMatch[y,x]) // X covers X
x.action ≠ y.action
}
}
check Shadowing
```

Figure B.1: Alloy representation of addresses, address tree, firewall rule definition, the example of the first rule from Table 5.1, and the canonical form to find shadowed policies.

Appendix C

Alloy Syntax

```
sigDecl ::= [abstract] [mult] sig name,+ { decl,* }
decl ::= [disj] name,+ : [disj] expr
mult ::= lone | some | one
block ::= { expr* }
factDecl ::= fact [name] block
predDecl ::= pred [qualName] name [paraDecls] block
funDecl ::= fun [qualName] name [paraDecls] : expr { }
paraDecls ::= ( decl,* ) | [ decl,* ]

mult ::= lone | some | one
decl ::= [disj] name,
factDecl ::=
predDecl ::=
funDecl ::= fun [qualName] name [paraDecls]
expr ::= const | qualName
const ::= number | none | univ | iden
unOp ::= ∨ | or | ∧ | and | ≤> | iff | implies
compOP ::= in | = | < | > | =< | ≥
```

Appendix D

Parsing topology

The topology is parsed as port representation. Each node in the topology is a node in the physical topology.

We recreate the physical topology as a set of nodes.

Algorithm 1 Topology parser algorithm

```
1: procedure TopologyParser($t : topology$)
2:     $nodeList \leftarrow t.nodes$                    ▷ Get all nodes from the topology.
3:     $linkList \leftarrow t.links$                    ▷ Get all links from the topology.
4:     for all $n : node \in Nodes$ do
5:         $p \leftarrow n.ports$
6:         for all $p : port \in n.ports$ do
7:             $nodelList \leftarrow n.name, p.number$
8:         end for
9:         for all $r : rule \in n.rules$ do
10:            $linkList.Add(getLinksFromNode(n))$
11:        end for
12:    end for
13: end procedure

14: procedure getLinksFromNode($n : node$)
15:     for all $r : rule \in n.rules$ do
16:         $in \leftarrow r.inport$
17:         $out \leftarrow r.outport$
18:         $linkList.Add(in, out)$
19:     end for
20: end procedure
```

For each node, the parsed topology, has rule list. For each rule we create a link at our representation.

Packet matching Packet fields are read by the datapath.

Protocol	**Constant**
IP	eht_type = 0x0800
IPv6	eht_type = 0x86dd
UDP	ip_proto = 0x11
TCP	ip_proto = 0x06
SCTP	ip_proto = 0x84