Model based testing techniques for software defined networks

Asma Berriri

université PARIS-SACLAY

TELECOM SudParis

NNT : 2019SACLL017

Thèse de doctorat

Model Based Testing Techniques for Software Defined Networks

Thèse de doctorat de l'Université Paris-Saclay
préparée à Télécom SudParis

École doctorale n°580 Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat: Réseaux, Information et Communications

Thèse présentée et soutenue à Evry, le 22 Octobre 2019, par

Asma BERRIRI

Composition du Jury :

Pierre SENS Professeur, Université Paris 6	Président
Yacine GHAMRI-DOUDANE Professeur, La Rochelle Université	Rapporteur
Antoine ROLLET Maître de Conférences, HDR, Université de Bordeaux	Rapporteur
Nikolai KOSMATOV Chercheur, HDR, CEA LIST Saclay	Examinateur
Djamal ZEGHLACHE Professeur, Télécom SudParis	Directeur de thèse
Natalia KUSHIK Maitre de Conférence, Télécom SudPAris	Co-encadrante

Abstract

Having gained momentum from its concept of decoupling the traffic control from the underlying traffic transmission, Software Defined Networking (SDN) is a new networking paradigm that is progressing rapidly addressing some of the long-standing challenges in computer networks. Since they are valuable and crucial for networking, SDN architectures are subject to be widely deployed and are expected to have the greatest impact in the near future. The emergence of SDN architectures raises a set of fundamental questions about how to guarantee their correctness. Although their goal is to simplify the management of networks, the challenge is that the SDN software architecture itself is a complex and multi-component system which is failure-prone. Therefore, assuring the correct functional behaviour of such architectures and related SDN components is a task of paramount importance, yet, decidedly challenging.

How to achieve this task, however, has only been intensively investigated using formal verification, with little attention paid to model based testing methods. Furthermore, the relevance of models and the efficiency of model based testing have been demonstrated for software engineering and particularly for network protocols. Thus, the creation of efficient and reusable model based testing approaches becomes an important stage before the deployment of virtual networks and related components. The problem addressed in this thesis relates to the use of *formal models* for guaranteeing the correct functional behaviour of SDN architectures and their corresponding components. Formal and effective test generation approaches are in the primary focus of the thesis. In addition, automation of the test process is targeted as it can considerably cut the efforts and cost of testing.

The main contributions of the thesis relates to model based techniques for deriving high quality *test suites*. Firstly, a method relying on *graph enumeration* is proposed for the functional testing of SDN architectures. Secondly, a method based on *logic circuit* is developed for testing the forwarding functionality of an SDN switch. Further on, the latter method is extended to test an application of an SDN controller. Additionally, a technique based on an *extended finite state machine* is introduced for testing the switch-to-controller communication. As the quality of a test suite is usually measured by its *fault coverage*, the proposed testing methods introduce different *fault models* and seek for test suites with *guaranteed fault coverage* that can be stated as sufficient conditions for a test suite *completeness / exhaustiveness*.

Résumé en français

Les réseaux logiciels (connus sous l'appellation : Software Defined Networking, SDN), qui s'appuient sur le paradigme de séparation du plan de contrôle et du plan d'acheminement, ont fortement progressé ces dernières années pour permettre la programmabilité des réseaux et faciliter leur gestion. Reconnu aujourd'hui comme des architectures logicielles pilotées par des applications, offrant plus de programmabilité, de flexibilité et de simplification des infrastructures, les réseaux logiciels sont de plus en plus largement adoptés et graduellement déployés par l'ensemble des fournisseurs. Néanmoins, l'émergence de ce type d'architectures pose un ensemble de questions fondamentales sur la manière de garantir leur correct fonctionnement. L'architecture logicielle SDN est elle-même un système complexe à plusieurs composants vulnérables aux erreurs. Il est essentiel d'en assurer le bon fonctionnement avant déploiement et intégration dans les infrastructures.

Dans la littérature, la manière de réaliser cette tâche n'a été étudiée de manière approfondie qu'à l'aide de vérification formelle. Les méthodes de tests s'appuyant sur des modèles n'ont guère retenu l'attention de la communauté scientifique bien que leur pertinence et l'efficacité des tests associés ont été largement démontrés dans le domaine du développement logiciel. La création d'approches de test efficaces et réutilisables basées sur des modèles nous semble une approche appropriée avant tout déploiement de réseaux virtuels et de leurs composants. Le problème abordé dans cette thèse concerne l'utilisation de modèles formels pour garantir un comportement fonctionnel correct des architectures SDN ainsi que de leurs composants. Des approches formelles, structurées et efficaces de génération de tests sont les principales contributions de la thèse. En outre, l'automatisation du processus de test est mise en relief car elle peut en réduire considérablement les efforts et le coût.

La première contribution consiste en une méthode reposant sur l'énumération de graphes et qui vise le test fonctionnel des architectures SDN. En second lieu, une méthode basée sur un circuit logique est développée pour tester la fonctionnalité de transmission d'un commutateur SDN. Plus loin, cette dernière méthode est étendue pour tester une application d'un contrôleur SDN. De plus, une technique basée sur une machine à états finis étendus est introduite pour tester la communication commutateur-contrôleur.

Comme la qualité d'une suite de tests est généralement mesurée par sa couverture de fautes, les méthodes de test proposées introduisent différents modèles de fautes et génèrent des suites de tests avec une couverture de fautes guarantie.

Acknowledgments

This thesis would not have been possible without the guidance of Professor Djamal ZEGHLACHE, whose broad vision and farsightedness have helped me sail my way through this research work. The thing I find most amazing about Djamal is his foresight. Providing me scientific support and professional advice, he has been a person who embodies characteristics that I can only aim to model myself after. I offer my sincerest gratitude for making me aware of the new perspectives that have found their way into this work, as well as enlightened my mind.

I would like to offer my deepest thanks to Doctor Natalia KUSHIK. She has made this endeavor academically possible, shaped research, and instilled in me the quality to cultivate my own potential. Her knowledge and suggestions have proven to be invaluable and have contributed profoundly to the results presented in this thesis. It has been an exceptional privilege to work with her. I dearly appreciate her personality, her work ethics, and the positive energy that she always emits, as well as how it affects and impacts those around her. These all are rare virtues. I would like to express my sincerest gratitude towards her. Thank you so much, Natalia!

I am sincerely grateful to Professor Yacine GHAMRI-DOUDANE, Doctor Antoine ROLLET, Professor Pierre SENS and Doctor Nikolai KOSMATOV for sparing the time to read and examine this thesis.

My sincere thanks go to Fondation Mines Télécom and Carnot Télécom & Société Numérique for the financial grant within the Futur & Ruptures (Future and Disruptive Innovation) programme.

My genuine appreciation goes out to my co-authors, Professor Nina YEVTUSHENKO and Doctor Jorge LOPEZ for their invaluable, precise, and diligent contribution to improve this work.

In addition, I had the honor of discussing the topics related to this thesis with researchers Igor BURDONOV and Alexander KOSSATCHEV. It is my privilege to extend my thanks to them for inspiring me with their interest in my research.

Deep respect I express to my work colleagues at Télécom SudParis.

To my parents, Mustapha and Mennana, for accepting nothing less than excellence from me. To my beloved brothers, Ghassen and Houssem, for always believing in me and encouraging me to follow my dreams.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Contents

1.1 Motivation

Computer networks and Internet structure usually consist of different network devices such as routers, switches and different types of middle-boxes. For managing and configuring such network devices, a set of specific and predefined command lines based on embedded operating system is usually used. Thus, traditional networks are essentially hardware-based and suffer from significant shortcomings regarding research and innovations, reliability, flexibility and manageability. For example, it can be argued that managing a large number of network devices is a big challenge and is prone to many errors. The emergence of new technologies, such as cloud and virtualization, generated the need for networks with higher accessibility and dynamic management. For solving the problems and limitations of traditional networks, the concept of Software Defined Networking (SDN), that separates control and data planes, was introduced along with an associated interaction protocol between the two planes known as OpenFlow (OF) [102]. OF is an emerging standard for SDN that ensures a clear separation of the data and control planes and provides central programmable control and management of the network using an SDN controller. Although SDN and OpenFlow started as academic experiments [102], they gradually gained the approval of the IT and telecommunications industries that have since adopted and started using and integrating the SDN paradigm into their cloud and network infrastructures.

The emerging field of SDN has enabled network deployment and service upgrade on software time scales which has huge benefits in the network domain. This is because in the future,

the network operators will not compete on the basis of network coverage alone but on the basis of features and services. The initial impact of SDN was seen in the datacenters. As early as 2012, Google had their full scaled datacenter running as an SDN based architecture. SDN is now all set to integrate the wireless domain too. With SDN, network administrators can adopt many new technologies and applications rapidly on hardware-independent network architectures regardless of multiple vendor-dependent protocols. The architecture involves SDN controller(s) residing in the control plane while the forwarding element(s) (switches) and hosts make the data plane. The SDN architecture allows end-users (e.g., network administrators, operators, etc.) to specify *requested paths* (e.g., network policies, services) that should be implemented as routes or simply *implemented paths* followed by traffic in the data plane.

Consequently, the impact of SDN on the networking domain will be enormous. Kreutz et al. [84] conclude that SDN is established as a key technology in the future of networking systems.

Although SDN goal is to simplify the management of networks, several challenges arise. Firstly, the software SDN architecture itself is a complex multi-component system, operating in heterogeneous and failure-prone environments. Additionally, the requirements defining these architectures and their related components are also complex and evolving. Therefore, it becomes of the highest priority to further raise the quality of such architectures and components before their wide deployment.

SDN architectures and their components are built on software and as a result, unlike traditional networking systems, they have become increasingly sophisticated. Such large and complex software more likely contain bugs that may disrupt the network functioning and corrupt its operation. A recent study on the hazards in SDN-based Google network architectures [57] reported that software bugs contributed to more than 33% of the high impact failures documented in postmortem reports, which they attribute mainly to a high speed of network evolution, and the need to keep up with the growing user traffic and demand for new features and services. Another large-scale study by Microsoft [94] on root causes of end-users impacting incidents in their production networks reports similar results. It shows that software bugs contributed to 36% of critical outages, being major problem, way ahead of hardware failures and human errors. To summarize, in terms of virtualized networking systems, increased complexity and higher customer expectations of quality impose thorough testing before any deployment. Indeed, the software nature of such complex networks makes them error prone, so the process of defect detection plays a crucial role. The typical testing process applied to SDN is nonetheless human-intensive and is as such usually unproductive and often inadequately realized. More importantly, this testing process does not provide any assurance about the quality of the tests. Research on test techniques that guarantee the fault coverage is therefore essential and required to foster the adoption and deployment of SDN based solutions and systems.

The requirements that must be fulfilled by the SDN architectures and their components are extremely complex and evolving very fast with the support of open source developments and communities. For example, in the OF protocol requirements [113], just the flow entries installation command ($Flow\_Mod$) is more than two pages long [113]. Thus, these requirements can be subject to ambiguities. Further on, these requirements are expressed in informal languages which can cause different interpretations by developers. Therefore, all of these factors would contribute to increasingly higher likelihood of implementations exhibiting diverging behaviours from their requirements. Consequently, informal reasoning does not lead to proving the correctness of such architectures/components. Under these circumstances, the introduction of formal testing techniques in SDN domain becomes necessary and obvious and is the focus of this thesis work.

Although, a number of valuable efforts in the context of formal verification and testing SDN

architectures and their components already exist (see Chapter 3), there is still a lot of space to improve the situation. In fact, prior research focusing on assuring the correctness/consistency of SDN architectures/components has resulted in techniques that belong either to *verification* or *testing*. The techniques of the former can ensure the respect of a given policy in the data plane and can help checking configuration errors and problematic controller programs in the control plane. Yet, as they check whether a *model* of the SDN architecture/component satisfies a given set of properties, they can only guarantee that the *properties* hold for the *model* and hence some implementation faults can still escape this check as no test cases are applied to the implementations. The techniques of the latter alleviate this challenge by targeting the implementation under test (IUT). However, they are either performed for checking the paths / networks implemented in the data plane rather than checking the functionality of a given critical SDN component, or they do not provide any guarantee about the test effectiveness. This brings to the picture *model based testing* methods where test generation is based on the model of a system under test (SUT). This line of work in the context of SDN in particular has not matured yet. Not only the proposed approaches are rare but also they mostly focus on testing the correct packets pipeline processing when it comes to testing the data plane for example rather than assuring the correct functioning of the switch as an integral component of the SDN architecture.

In summary, we feel or contend that model based testing is one of the most convenient testing method which helps in detecting errors and bugs and can assure the proper functioning of SDN architectures and their components. Indeed, model based testing allows the creation of consistent, reusable, and well-documented models on the one hand and the derivation of test cases with guaranteed fault coverage on the other hand. This is an important stage in the testing process of SDN.

1.2 Problem Statement / Research Questions

The goal of this thesis is to check that the implementations of an SDN architecture and corresponding components conform to their requirements. Due to their phenomenal success, SDN implementations are becoming increasingly complex, with such features as accepting complex inputs (end-user requests), packets processing and interaction with a logically centralized controller. In the quest for conformance, the task of guaranteeing their correctness is becoming ever more challenging. Further on, it is not unusual that an entire SDN architecture might exhibit a behaviour such that the requests are correctly implemented in the underlying data plane while the SDN components of such architecture (e.g., switch, controller) are not implemented and/or operating correctly (hidden bugs).

The main research issue this thesis is concerned with, relates to *assuring the compliance of Software Defined Networking architectures and their components with respect to their specifications by means of model based testing*.

The first critical *challenge* is to guarantee the consistency between the high level requested paths and the configurations' implementations of these requests in the data plane. In other words, given an SDN architecture as the system under test, i.e., the SDN controller translating end-user requested paths into flow rules and the SDN switches implementing these flow rules in the data plane to correctly forward traffic to hosts, what inputs should be applied to the controller and what outputs should be observed at the data plane level such that conclusions about the correctness of the whole architecture can be drawn, i.e., whether the SDN architecture is functioning as expected/desired.

Resolving this first challenge will increase the confidence in reliable SDN architectures

deployment without which trust in their implementations and operation can not be established or built.

However, resolving the first challenge does not guarantee the correctness of the SDN components forming the architecture. Moreover, should a bug be discovered by testing the entire architecture, it certainly becomes of interest to localize its cause (possible root cause) or responsible SDN components. Indeed, the functional correct behaviour of SDN components should not be taken for granted. Therefore, a second critical *challenge* that needs to be addressed concerns guaranteeing the correct behaviour of crucial SDN components, particularly the switch and the controller.

The switch exposes two interfaces, one to perform packet processing in the data plane and the second to communicate with a controller that instructs it how to process these packets. Therefore, two major *challenges* arise. Firstly, given the switch specified as a set of configurations to forward packets in the data plane, how the correctness of its forwarding functionality can be guaranteed. Secondly, given the switch in its communication with the controller as the system under test. The SUT takes as input OF messages from the controller and outputs replies to the controller as specified by the OF requirements. The correctness of such interaction needs to be assured.

The controller in an SDN architecture is also a core SDN component responsible for making decisions on managing switches in the underlying data plane. Therefore, ascertain the correct implementation of the controller is crucial. To this end, it is important to guarantee the correct behaviour of its modules / applications. One critical module considered in this work is the one responsible for translating end-users requests, specifying two devices between which a link should be implemented, to corresponding configurations. The SUT in this case has a one-direction communication with a given application from which it receives a request. It has also a one-direction communication with a given switch in the data plane. The *challenge* is to guarantee that the controller module under investigation assigns correctly the ports of the network devices as specified by the request.

Based on the previous analysis, we focus on the following arising research questions:

1. ***RQ1*** How to assure the correct behaviour of *SDN architectures*?

2. ***RQ2*** How to assure the correct *forwarding behaviour* of an SDN-enabled switch in the data plane ?

3. ***RQ3*** How to assure the correct behaviour of an SDN switch in its *interaction and interfacing with a controller*?

4. ***RQ4*** How to assure the correct behaviour of the *module/application of an SDN controller* responsible for translating, for a given switch in the data plane, requests into corresponding ports of the switch of interest?

To tackle these questions and challenges, in the upcoming section, the structure of this work will be related to the solutions proposed by this thesis for addressing each key question.

This thesis develops novel testing techniques for guaranteeing the correct behaviour of SDN architectures and their components. The techniques combine model based testing and mutation analysis. The proposed approaches allow testing the actual implementations rather than checking some network properties by means of formal verification as done by most of the state of the art.

1.3 Contributions and Structure of the Thesis

To address the aforestated research questions, we opt for *model based testing* because it systematically generates from the model a collection of tests (test suites) that, when run against the SUT, will provide sufficient confidence that it behaves as the model predicted it would. The difference between model based testing methods and verification methods (massively used in the state of the art; Chapter 3) is basically about the stimulation of the system under test vs. the checking of the model. Now, the complexity of SDN architectures and related components is not low, which results in verification methods being hard to apply to such systems. Model based testing on the other hand scales much better and has been used to test large systems.

Although model based testing requires more up-front effort in building the model, it offers substantial advantages over traditional software testing methods. Firstly, once a model is built, it is easier to generate and re-generate test cases than it is with hand-generated test cases. Besides, the quality of the generated test cases is high in comparison to other testing techniques. This helps in detecting subtle errors. Indeed, model based testing techniques strive to automatically generate test cases that are able to reveal whether any modelled fault has been implemented. As a result, these techniques guarantee a fault coverage of the model and are able, and have shown, to produce high quality test suites.

The result of this thesis can be divided into four main facets and contributions:

1. ***Model based approach for testing the functional behaviour of entire SDN architectures*;**

2. ***Model based approach for testing the forwarding functionality of the switch as a critical SDN component*;**

3. ***Model based approach for testing the switch in its interaction with the controller*;**

4. ***Model based approach for testing a module/application of the controller. Specifically, the controller application responsible for translating requested paths into pairs of ports of a given switch*.**

The thesis is organized as follows. This chapter gives an overview and motivation of the research topics of this thesis. It introduces the problems that the work is dealing with, its objectives, contributions and structure.

Next, Chapter 2 has a foundation nature and includes the background on Software Defined Networking architectures and their components and interfaces, verification and testing concepts, model based testing, mutation analysis and black and white box testing approaches. Additionally, while introducing these foundations and concepts, we point out which chapter(s) they are used in and how they relate to the thesis work.

Chapter 3 examines the related work on verification and testing techniques with respect to SDN architectures and their components. For that purpose, a taxonomy is provided. Herewith, a link to the current chapter is done. The literature investigation has elicited a set of limitations and concerns and a set of research directions that the work of this thesis follows. The observed lack of model based testing techniques applied to SDN in the current literature and ensuing analysis, has led to the model oriented testing methodologies adopted and proposed in this thesis.

The four following chapters relate to these proposed testing approaches.

Chapter 4 characterizes a novel model based technique for testing entire SDN architectures taking into account potential interoperability issues in the controller-to-switch communication. The technique aims to ensure that requests expressed by end-users are correctly

implemented in the data plane. The approach relies on *graph/path enumeration*. The chapter relates to the first *challenge* of answering the question of how to guarantee the correct functioning of such architectures. In particular, a *fault model* is introduced where the *fault domain* contains potential implementations of *virtual paths* requested by an end-user. Afterwards, approaches for test generation under *black box* and *white box* testing assumptions are proposed. To guarantee the *fault coverage*, the conditions are proven such that when under both testing assumptions, a *complete* test suite with respect to such fault model can be derived. Additionally, Chapter 4 provides an experimental evaluation of the proposed approach. A discussion on the obtained results is then given so as to support the effectiveness of the presented testing method. Results show that the derived test suites were able to detect a number of functional inconsistencies in the considered SDN architectures.

Now that we have provided a novel testing technique guaranteeing to check for the correctness of the entire SDN architecture, the question that automatically arises is how about the correctness of its components. Further ahead, one might want to identify which exact component is not working as expected. This is addressed in detail in Chapters 5, 6, and 7.

In particular, the second part of the thesis proposes novel model based techniques for testing critical SDN components. At first, the thesis looks at testing the switch as a crucial component of SDN architectures in two aspects, then a module of the controller is considered.

The *forwarding functionality* of the switch modelled and analyzed as a *'stateless'* system without considering its interaction with the controller is investigated in Chapter 5. For this purpose, the chapter proposes a *logic circuit* based testing approach. An appropriate *fault model* is introduced and a logic synthesis algorithm is presented. Some mutation operators over the derived switch specification are introduced, then logic circuit based approach and related SAT solving are utilized for detecting equivalent mutants. Further on, both *active* and *passive* testing strategies are explored. Finally, the chapter demonstrates the effectiveness of the approach using experimental evaluation. The results show for example that test suites derived based on traditional logic circuit fault models have a high fault coverage for SDN-enabled switch faults. This piece of work relates to the *second challenge*.

Despite the effectiveness potential of the solution presented in Chapter 5 in detecting implementation forwarding errors, it does not cover the behaviour of the switch in its interaction with the controller. This is tackled in Chapter 6. The chapter proposes an *extended finite state machine* based test generation strategy for testing the functional behaviour of a switch in its communication with an SDN controller with respect to requirements described in the OpenFlow specification. A part of the requirements is formalized, then based on the derived model, an appropriate *fault model* is introduced and a test generation method for deriving exhaustive test suites with respect to such fault model is presented. To demonstrate the effectiveness of the proposed approach, an experimental evaluation is performed which aims at the assessment of the derived test suites *fault coverage* on one hand and at the execution of the derived tests against an OF implementation under test, on the other hand. The conducted evaluation shows the effectiveness of the approach; besides, experiments reveal several implementation faults and specification ambiguities when a switch implementation is tested. By that, the *third challenge* is covered.

Chapter 7 utilizes the results of Chapter 5 and addresses the further question of the outlined challenges. The model based approach presented and discussed in Chapter 5 is adapted and adjusted to tackle one module of the controller, particularly the one responsible for translating end-user requests into corresponding pairs of ports of a switch of interest. First, the problem is described. Then, the formalization of the specification, the test generation approach, and the test execution strategy follow subsequently. This relates to the *fourth challenge*.

Chapter 8 completes this work with a summary and outlook. The proposed testing ap-

proaches' capabilities and limitations are reviewed, the general trends of the quality assurance for SDN architectures and their components are recalled and influences of the contributions of this thesis are outlined.

1.4 Dissertation Roadmap

In this section, the dependencies between chapters throughout this thesis are outlined.

The detailed discussion on SDN, verification and testing in Chapter 2 serves mostly as a foundation for the considered topics.

Chapter 3 includes a review of the state of the art work on verification and testing techniques for SDN architectures and SDN components yielding to the position of our contributions in the field.

Chapters 4, 5, 6 and 7 contain the central achievements of the presented work. Additionally, the reader can refer to the summaries given at the end of each chapter, which provide the essential overview of the subjects and results on their contents.

The concept of the *Logic Circuit* based approach which is presented in Chapter 5 is used as the basis for Chapter 7.

Author's Publications & Talks

The work presented in this thesis is original work undertaken between October 2016 and September 2019 at SAMOVAR/CNRS, Télécom SudParis / Université Paris Saclay. It has been financed by the "Futur & Ruptures" (Future and Disruptive Innovation) programme grant awarded to the author by "Fondation Mines-Télécom" and "le Carnot Télécom & Société Numérique". The work resulted in the following publications.

Conferences

[1] Asma Berriri, Jorge López, Natalia Kushik, Nina Yevtushenko, and Djamal Zeghlache. "Towards Model based Testing for Software Defined Networks." In: *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE, Funchal, Madeira, Portugal, March 23-24*. 2018, Pages 440–446.

[2] Jorge López, Natalia Kushik, Asma Berriri, Nina Yevtushenko, and Djamal Zeghlache. "Test Derivation for SDN-Enabled Switches: A Logic Circuit Based Approach." In: *Proceedings of the IFIP International Conference on Testing Software and Systems*. Springer. 2018, Pages 69–84.

Archives

[1] Asma Berriri, Natalia Kushik, and Zeghlache Djamal. "Extended Finite State Machine based Test Generation for an OpenFlow Switch." working paper or preprint. 2019. URL: https://hal.archives-ouvertes.fr/hal-02262841.

Some of the research leading to this thesis has appeared previously in the following.

Journals

[1] Asma Berriri, Natalia Kushik, and Djamal Zeghlache. "On using finite state models for optimizing and testing SDN controller components." *Russian Physics Journal* 59.8/2 (2016), Pages 5–7.

Portions of this work have been already presented in the following.

Participation in Seminars and Conferences

[1] Asma Berriri. *Formal Approaches for Testing in Software Defined Networks. Poster and presentation*. Journée doctorants Samovar 2018, Paris, France. 2018. URL: http://samovar.telecom-sudparis.eu/spip.php?article1177.

[2] Asma Berriri. *Formal Approaches for Testing in Software Defined Networks. Poster presentation.* Visite HCERES - évaluation laboratoire Samovar les 4 et 5 décembre 2018, Paris, France. 2018. URL: http://samovar.telecom-sudparis.eu/spip.php?article1158.

[3] Asma Berriri. *Formal Approaches for Verification and Testing in Software Defined Networks. Presentation*. The 4th GDR RSD and ASF Winter School on Distributed Systems and Networks 2019,Pleynet, Sept Laux, France. 2019. URL: https://sites.google.com/site/rsdwinterschool/program-2019.

[4] Asma Berriri. *Formal Approaches for Verification and Testing in Virtual Networks. Presentation*. Méthodes de Test pour la Vérification et la Validation (MTV2) du GdR GPL du CNRS, ENSIIE, Paris, France. 2018. URL: http://logimas.mics.centralesupelec.fr/wp-content/uploads/2018/12/MTV2-A.Berriri-final-extended.pdf.

[5] Asma Berriri. *Testing and Verification for Software Defined Networks. Presentation*. The 7th Halmstad Summer School on Testing, HSST 2017 in cooperation with TOCSYC Network, Halmstad University, Sweden, June 12-15th. 2017. URL: http://ceres.hh.se/mediawiki/HSST_2017.

[6] Asma Berriri. *Towards Testing and Verification in SDN. Poster presentation*. Journée Futur & Ruptures, février 2017, IMT, Télécom ParisTech, Paris, France. 2017. URL: https://www.imt.fr/journee-futur-ruptures-jeudi-2-fevrier-2017-a-limt/.

[7] Asma Berriri. *Towards Testing and Verification in Software Defined Networks. Poster and presentation*. Journée doctorants Samovar 2017, Paris, France. 2017. URL: http://samovar.telecom-sudparis.eu/spip.php?article1063.

[8] DigiCosme Spring School on Formal Methods and Machine Learning. *ForMaL*. 2019. URL: https://formal-paris-saclay.fr.

[9] GT LTP Langages Types et Preuves du GdR GPL du CNRS. *ENSIIE, Paris, France*. 2018. URL: http://web4.ensiie.fr/~guillaume.burel/ltp/journee_2018.html.

2 Background

Contents

In this chapter, the fundamentals and basic background information, on which we base our work of testing SDN architectures/components, are provided. Firstly, in Section 2.1, the SDN paradigm concepts, architecture, and components are presented. The necessary theoretical background on the terminology and semantic of verification and testing used in the thesis is covered in Section 2.2. Armed with these basics, we dig deeper into notions related to model based testing in Section 2.3 with a brief insight into how these concepts will be used in our contributions. Further on, in Section 2.4, the concepts of mutation analysis are introduced. An overview of the black and white box testing approaches in Section 2.5 completes the theoretical basics.

2.1 Software Defined Networking

In this section, we provide a general description of SDN and give an overview of the components and interfaces.

SDN is an emerging networking paradigm that is now growing in usage and popularity, progressing rapidly and addressing some of the long-standing challenges in computer networking. SDN platforms are subject to be widely used and deployed. Recently, they are deployed into several core and data center networks. This paradigm brings a major concept, namely it decouples the data control of the network from the data transmission. It moves the control logic into a logically centralized component called controller. In contrast to the traditional network architectures, the separation of roles in an SDN architecture is the key to achieving flexibility and to making it easier to introduce new concepts in networking. One can certainly observe that the abstraction offered by the SDN architecture provides wider flexibility on developing and implementing new network functionalities and simplifies the configuration and management of modern networks suggesting the opportunity for more innovations.

2.1.1 Overview of the SDN Architecture and SDN Interfaces

The foundation of SDN is proposed by the standardization organization called Open Network Foundation (ONF) [142]. In an SDN architecture, a logically centralized control function (the *controller*) translates the applications' requirements and applies control instructions over the *forwarding devices* (the *switches*) in the data plane, while providing relevant information up to the SDN *applications* [102]. The forwarding devices in the data plane then reroute data packets to other forwarding devices and to *hosts* according to these control instructions [132], [54], more specifically, forwarding and filtering *rules* [134].

An SDN architecture is composed of three layers, i.e., (i) the network applications, (ii) the control plane composed of one or multiple controllers, and (iii) the data plane composed of the forwarding devices and hosts [142]. The interaction between these layers is performed through Application Programming Interfaces (APIs). The SDN controller has mainly two APIs. The southbound API responsible for collecting network status and updating forwarding rules in the forwarding devices. The northbound API such as the 'Representational State Transfer' (REST) API handles interaction with the application layer, i.e., receiving requests/ policies described in high level languages from SDN applications and providing a synchronized global view. It enables direct expression of network behaviour and requirements. Northbound API presents a programmable API to network control and management applications. The southbound API allows the exchange of control messages between the controller and the SDN forwarding devices. This interface dictates the format of the exchanged control messages. Multiple southbound interfaces exist such as OpenFlow (OF) [102], ForCES [44], and POF [143]. The OpenFlow protocol is the most deployed SDN protocol as the southbound interface [152]. Multiple OpenFlow protocol versions exist including versions 1.0, 1.3, 1.5. During the thesis, we used the stable releases of OpenFlow at the time (versions 1.0 and 1.3). All OF versions use the same structure of SDN rules, with some action and match field additions in each version.

An example of an SDN architecture is depicted in Figure 2.1.

SDN network architectures allow end-user (e.g., a network administrator) requests/requirements representing network policies to be specified and implemented in the data plane. Network applications can issue the requests on the shape of data paths that have to be implemented between pairs of sources and destinations through the network architecture. The controller then computes the appropriate flow rules and pushes them to the switches. A switch

acts as a forwarding device receiving and sending network packets in accordance with the configured rules. It also sends events such as traffic statistics, network changes and acknowledgments to the controller. An end-user request might impose for example the traversal of a given sequence of switches. An example of a request is 'traffic from hosts in local area network $Subnet_1$ to the internet must traverse the switch S_1 and one of the switches S_6 and S_7 (Figure 2.1).

In this thesis, we assume that the network architecture is *functioning correctly* if the network policies (requests) defined by the network applications and translated by the controller are correctly implemented by network devices in the data plane. In Chapter 4, we propose a *model based testing technique* to guarantee such correctness.

In the following, we detail the layers of the SDN architecture.

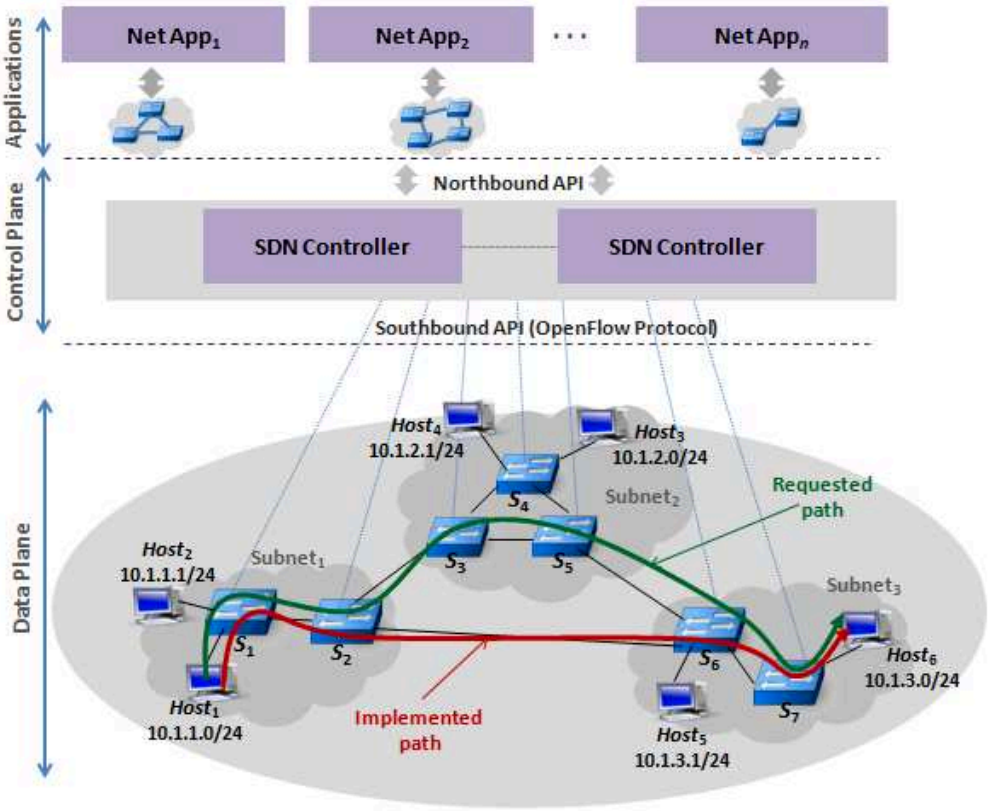


Figure 2.1 – SDN layered architecture.

2.1.2 Application Layer

As illustrated in Figure 2.1, the application layer resides above the control layer. Through the northbound API, SDN applications can conveniently access a global network view and can implement different strategies to configure the underlying physical infrastructure (data plane) using a high level language. The application layer mainly consists of the end-user network applications or functions that consume the SDN network services. Examples of such applications include network visualization, load balancing and firewalls applications. Based on the network configuration requirements and specific needs, a network administrator can program new network applications (new network functionalities) in standard programming languages.

2.1.3 Control Plane

The control layer bridges the application layer and the data plane. It consists of a set of software based SDN controllers providing a consolidated control functionality through open APIs [59]. This layer supervises the network forwarding behaviour.

The controller sets up all forwarding devices in the data plane, maintains topology information, and monitors the overall status of the entire architecture [102]. It updates the flow table by adding and removing rules using protocols such as OpenFlow [102]. Each forwarding device has a set of flow tables with rules. A rule has three parts: the matching condition to a specific flow; the action to be applied to this flow, and counter to track the rule occurrence for management purposes.

The Controller presents two behaviours, namely reactive and proactive. In the reactive mode, the first packet of flow received by a forwarding element triggers the controller to insert rules in each forwarding element of the data plane. In fact, the controller listens to switches passively and configures routes on-demand (by installing the corresponding rules). It receives messages of connected hosts from the switches. Upon receiving a $Packet\_In$ message from the switch, the controller looks for the destination host location and sets the path by sending $Flow\_Mod$ messages to affected switches in the path. In the proactive mode, the controller pre-populates the flow tables in each forwarding element.

All functions of the control plane are performed by the controller. It has full network topology information and the location of hosts. When a forwarding element receives a packet for which there is no matching rule in its flow tables, it forwards it (using $Packet\_In$) to the controller asking for the action to take upon this new flow. The controller can define the port that the flow must be forwarded to or take other actions, such as dropping the packet. The controller must set the entire path by sending $Flow\_Mod$ messages to all switches from the source to the destination.

The SDN controller allows the applications to communicate with the SDN forwarding devices, and creates the global view of the network. The controller is also able to monitor all the network forwarding elements regularly. It then informs the network applications of the network changes using the northbound interface. Then, the network applications manage and implement policies in the network devices using the northbound interface.

Throughout the thesis, we consider architectures with controllers' deployments logically representing a single controller. However, the proposed approaches can easily be extended with an architecture with multiple controllers.

2.1.4 Data Plane

The data plane consists of forwarding devices and hosts. The forwarding devices include physical and virtual switches which are interconnected between each other and with hosts. An example of an SDN switch is the software OpenvSwitch (OVS) [121].

The Switch

A switch exposes two interfaces allowing its interaction with packets in the data plane on one hand (forwarding functionality) and its interaction with the controller on the other hand. A switch does not have any built intelligence and relies on the controller to give it a set of rules to know how to treat/forward incoming packets. These rules are then saved in the switch flow tables [113]. When a packet arrives to the forwarding device, it is matched against rules in the flow tables. The action is triggered if the matching is satisfied and then, the counter is updated. If the packet does not match any entry in the flow tables, it is sent to the controller

over a secure channel to ask for an action. Packets are matched against all rules based on some prioritization scheme. The flow table could have a priority field associated with each rule. Higher number indicates that the rule should be processed before.

An SDN forwarding rule (also called a flow entry) is composed of three parts [113]

- Matching fields: packet header values to match the incoming packets in addition to the input port. We refer to the matching fields as *matching parameters*.
- Actions: set of instructions to apply to the matching packet such as forward to specific output port, flood, drop, send to controller or modify packet headers.
- Locations / priorities: to control the rule hierarchy.

The Switch in its Data Plane Interface (Forwarding Functionality) The forwarding rules are grouped in different flow tables and are considered to be the *configurations* of switches with respect to packets and application flow management.

As an example of rules installed in a switch, consider the set of rules defined in Table 2.1. The table includes the following *matching parameters*:

- *Flow Table*, a virtual partition for the installed rules;
- *Priority*, the order attributed to the rule to be applied with respect to other rules in the flow table;
- *Input Port* (In_port), the ingress port of the incoming packets;
- *Ethernet Type* (Eth_type), the type of traffic carried by the Ethernet datagram;
- *Source and Destination IP Addresses* respectively (IP_source, IP_dest), define the IP protocol source and destination addresses;
- *Output ports* ($Output$) defines the set of ports to which a matching packet should be forwarded.

Flow Table	Priority	In_port	Eth_type	IP_source	IP_dest	Output
0	500	$*$	ARP (0x806)	$*$	$*$	Port 1
0	500	1	ARP (0x806)	$*$	$*$	**"All"**
1	501	1	IP (0x800)	$10.0.0.1/32$	$10.0.0.2/32$	Port 2
1	501	2	IP (0x800)	$10.0.0.2/32$	$10.0.0.1/32$	Port 1

Table 2.1 – Example of rules installed in a switch

For example, the third rule in Table 2.1 is specified in the flow table 1 with the priority 501. When the packets having the source IP address $10.0.0.1$ and destination IP address $10.0.0.2$ arrive to Port 1 of the switch, these packets have to be forwarded through the (output) Port 2 of the switch.

Chapter 5 answers the question of *how to guarantee the forwarding functionality of a switch specified as a set of configurations* in the data plane.

The Switch in its Southbound Interface

The OF specification [113] describes the behaviour of the switch and its communication with the controller. It specifies OF messages handling via the southbound API. Examples of messages received/sent by the switch include OFPT_HELLO message for connection establishment; OFPT_FEATURE for advertising the supported capabilities; FLOW_MOD for handling modification of rules in the switch; OFPT_BARRIER to get the information about when a given

command is applied; OFPT_MULTIPART for reporting statistics and OFPT_ECHO for sensing the liveness.

Before any messages can be exchanged, the connection establishment process takes place implying OF version and capability negotiation. Both ends of the connection exchange HELLO messages immediately after the lower layer (TCP/TLS) connection establishment. Afterwards, to be aware of the capabilities of the switch, FEATURE is exchanged. In case this message is not received by the controller and after a timeout, the latter disconnects the switch. Once the connection is successfully established, different messages can be exchanged, e.g., ECHO, FLOW_MOD, BARRIER and MULTIPART. The actions of rules installed by the FLOW_MOD messages are defined by the OF requirements and include for example modification of IP and VLAN values.

If an OF implementation complies with the requirements, the exchange of these messages should be performed correctly with the specified parameters. In Chapter 6, a *model based approach is proposed to test the switch in its communication with the controller*.

2.2 Verification and Testing

It is necessary to evaluate or judge the 'correctness' of an SDN architecture and its related SDN components, i.e., whether the SDN architetcure/component meets its requirements and specifications and whether it fulfills its intended purpose.

In the following subsections, basic concepts related to verification and testing employed throughout the thesis are briefly introduced.

2.2.1 Verification

Verification is a process that involves mathematical proof showing that a system satisfies a set of desired properties. Formally, given a system M, verification aims at the creation of a set of properties P that are iteratively checked during phases of development of M to determine whether or not the behaviour of M meets the set of properties P [32, 46].

In the context of SDN, the network verification problem can be formulated as follows. *Given an abstract model of an SDN component(s) (or a composition of those)* $Model(N)$ *and a set of network properties* P *expressed in a given logic formulae, determine whether* $Model(N)$ *satisfies* P.

We distinguish the different verification methods applied to SDN based on the mathematical formalism used in the reasoning process during the verification. In this work, under verification we understand a process that does not require any stimulation of the system under verification.

Examples of network properties (referred to as invariants as well) to be verified include:

- Reachability [43] is concerned with whether the network always successfully delivers packets to the intended end hosts. A definition of reachability property can be for example that a packet pkt can get from the source host $Host_1$ to the end host $Host_6$ in Figure 2.1.
- Forwarding Loop [97] occurs if the same packet returns to a location that it has visited before. There are several possible definitions of this property, e.g., returning to the same location with exactly the same header, or returning to the same location with a possibly different header. The former case indicates the presence of an infinite loop, since this packet will repeatedly return to this location. The latter case may also be undesirable since there is usually no reason for a packet to return to the same location.

- Black-holes [141] means that packets are dropped because there is no destination configured on one of the forwarding devices they traverse.

2.2.2 Testing

In software development methods, *testing* occupies a central position of ensuring software quality. In order to judge the correctness of a system under test, one should observe or monitor for each test execution what the system does, how it does it, and perhaps when it does it. *Active testing* is defined when a system under test (SUT) is stimulated by appropriate inputs, i.e., test sequences / cases, and the conclusion about its correctness is made based on the observations of its output responses. Testing techniques applied to SDN architectures/component (s) are based on stimulating the architecture/component(s) (or composition of those) under test by test cases and observing their reactions with the intent of finding errors. *Passive testing* is defined when one just monitors the SUT and observes that the behaviour is correct or incorrect without stimulating the SUT. Debugging / troubleshooting is defined as stimulating the SUT by appropriate inputs and observing its reactions in the objective of localizing errors [10], [107].

Random testing [10] generates test cases in a (uniformly) random way with negligible effort. A more evolved form, referred to herewith as 'semi-random' consists of 'controlling' the way random test cases are generated. For example, starting with randomly generated inputs and repeatedly modifying them, more or less at random, to produce new inputs. This increases the probability of inputs found in this way being 'interesting'.

2.3 Model Based Testing

Model based testing has received increasing attention due to its ability to improve productivity, by automating test planning, generation, and execution. In model based testing, test cases/sequences forming a *test suite* are generated from an abstract model, which captures the desired behaviour of the system. Then, the test cases/sequences are executed against a real implementation of the system and the conformance of the implementation to the specification is checked by comparing the observed outputs with the ones specified by the model, for some suitable definition of conformance. The specification can be a formal model of the system and might also be defined by a set of (end-user) requirements that should be correctly implemented.

The central artifact of model based testing is the *model*. It serves as an abstraction of the system under test (SUT), manageable by the test engineers. In this context, the primary idea behind a model based method is the benefit of deriving a specification for a system that might cover its functional behaviour. The model/specification may be utilized as the basis for automating parts of the testing process and can lead to the generation of more efficient and effective test cases/sequences.

A large number of possibilities is present with respect to how to model the SUT. For example *logic circuits* or state based models such as Finite State Machines, *Extended Finite State Machines*, Input/Output Transition Systems, etc., might be considered. For state based models, most notations for test modeling are based on *states* and their identification.

In the following subsections, we provide a glance insight into preparatory ingredients for the author contributions. In subsection 2.3.1, we give an introductory overview of a formal representation of an SDN architecture that supports our first contribution in Chapter 4. In subsections 2.3.2 and 2.3.3, we introduce logic circuit and extended finite state machine as

models that we use in Chapter 5 and 6 respectively to support the proposed test generation techniques. In subsection 2.3.4, definitions related to the notion of fault models are provided.

2.3.1 Formal Representation of an SDN Architecture

SDN architectures satisfy end-user requests/requirements by forwarding data in a given data plane. At the level of a switch in the data plane, forwarding decisions are defined by rules. To implement desired requirements during forwarding, network administrators/operators define requests to be implemented in the data plane. For instance, they impose some policies to be applied to the flows. The 'specification' in this case is defined by a set of (end-user) requirements that should be correctly implemented.

In the thesis, unless the context is explicitly indicated, we refer to the data plane as the *Resource Network Connectivity Topology* ($RNCT$[1]) for which a formal definition is given in Chapter 4. An $RNCT$ depicts the SDN components in the resource connectivity network. An informal description of a path in an $RNCT$ is depicted in Definition 2.1. A more formal definition is provided in Chapter 4.

Figure 2.1 presents an example of a network topology consisting of one controller, seven switches and six hosts. Each switch is connected to the SDN controller. S_1 is connected to hosts $Host_1$ and $Host_2$, S_4 is connected to $Host_3$ and $Host_4$, S_6 is connected to $Host_5$ and S_7 is connected to $Host_6$.

Definition 2.1.
A *path* in an $RNCT$ between two given hosts is a sequence of network devices that starts and ends with hosts and all other intermediary devices are switches.

Based on the topology of Figure. 2.1, the corresponding $RNCT$ and some examples of its paths are illustrated in Figure. 2.2. In this example, the $RNCT$ is a network with seven switches and six hosts.

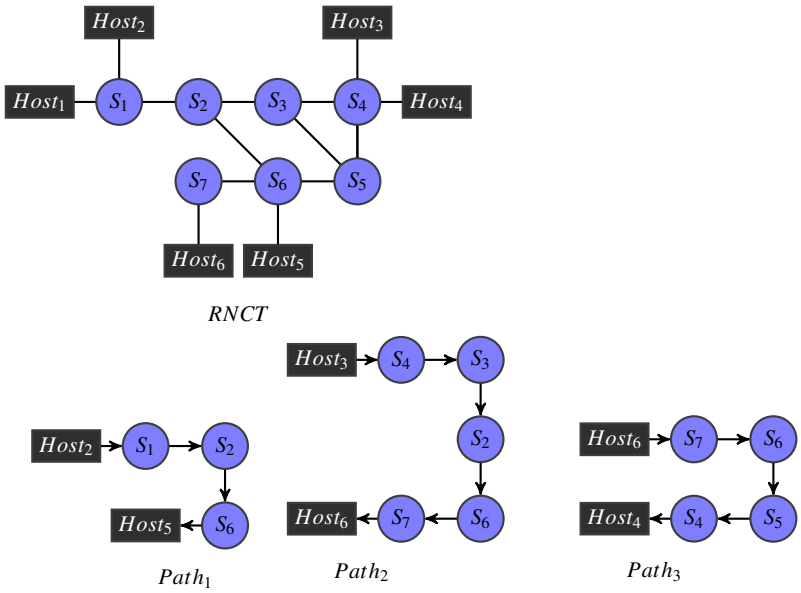


Figure 2.2 – $RNCT$ of the network topology in Figure 2.1 and examples of its paths

The presence of potential errors/bugs in the SDN architecture certainly breaks the intended

[1]Hosting infrastructures can be physical or virtualized.

network functions. The errors might be due to the inconsistencies between end-users' (e.g., network administrators) logical requests and the actual flow-level implementations. Figure 2.1 shows a misbehaviour of an SDN architecture consisting in implementing a different request (marked in red in the data plane) than the desired/specified one (marked in green). Chapter 4 tackles this problematic and proposes a model based testing approach aiming to the detection of such misbehaviours.

2.3.2 Logic Circuit

The specification of a system can be represented by a *logic circuit* as the underlying model in model based testing. We propose this model for the testing approach developed in Chapter 5.

A sequential logic circuit consists of combinational logic and memory elements, namely latches. A *combinational circuit* is composed of logic gates (AND, OR, etc.); each logic gate implements a Boolean function. Unlike sequential logic circuits whose outputs are dependant on both their present inputs and their previous output, which gives them memory, i.e., state, the outputs of combinational logic circuits are only determined by the logic function of their current input, logic vectors of '0' or '1', at any given instant in time. Thus a combinational circuit is memoryless.

Definition 2.2.
A logic circuit, representing the system specification, is said to be *Complete* (or completely specified) if the output is defined for every possible input vector, otherwise, it is said to be *Partial*.

There are different formats of logic circuit representation. In this work, we consider the Berkley Logic Interchange Format (BLIF) [18]. In this format, a combinational circuit is described by the corresponding look-up table (LUT). An LUT contains a set of input/output Boolean vectors describing the circuit's behaviour. The LUT table of the partial specification, portrayed in Figure 2.3, is shown in Table 2.2.

Figure 2.4 shows an example of the BLIF file for the logic circuit of Figure 2.3. The names of external inputs and outputs are listed in the file. Then, following those declarations, the truth tables for each of the gates with their inputs and outputs are listed. For example, element $n6$ is a function of two arguments $x0$ and $x2$.

A logic circuit can be modelled as an AND-INVERTER Graph (AIG). In fact, a Boolean network is a directed acyclic graph with nodes representing logic gates and directed edges representing wires connecting the gates. AIG is a combinational Boolean network composed of two-input AND-gates and inverters [106]. In an AIG, each node has at most two incoming edges. A node with no incoming edges is a primary input. Primary outputs are represented using special output nodes. Each internal node in the AIG represents a two-input AND function.

Example
Figure 2.3 illustrates an example of a partial specification (logic circuit) designed as an AIG.

2.3.3 Extended Finite State Machine

A state based model is one of the most powerful ways to represent a system Sys where a number of stimuli (inputs) is received by Sys and actions (outputs) are produced by Sys. For example, the specification of a system can be represented by a state based model such as

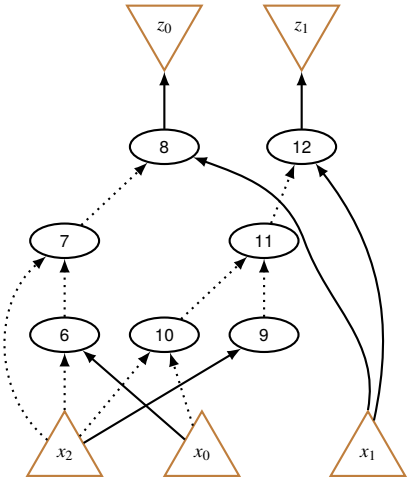


Figure 2.3 – An example of a partial logic circuit specification designed as an AIG (c_{ex})

$x0, x1, x2$	$z0, z1$
0 1 0	0 1
0 1 1	1 0
1 1 1	1 1
1 1 0	1 0

Table 2.2 – The look-up table (LUT) of the specification c_{ex} illustrated in Figure 2.3

```
. model cirex
. inputs x0 x1 x2
. outputs z0 z1
. names x0 x2 n6
10 1
. names x2 n6 n7
00 1
. names x1 n7 z0
10 1
. names x0 x2 n9
11 1
. names x0 x2 n10
00 1
. names n9 n10 n11
00 1
. names x1 n11 z1
10 1
. end
```

Figure 2.4 – A BLIF description of C_{ex} shown in Figure 2.3

Finite State Machine (FSM) or Extended Finite State Machine (EFSM) as the underlying model while testing.

These models are used to describe behaviours of sequential systems where outputs depend on inputs and the current state. This is to be opposed to combinational behaviours where the output is only dependent on the set of inputs as described earlier with combinational logic circuits in Subsection 2.3.2.

In this context, classical FSM for example can be used. An FSM is a transition system with a finite number of inputs, outputs, states and transitions each labeled by an input/output pair [48]. FSMs are widely used in various application domains, such as modeling and testing communication protocols, and other reactive systems.

States, transitions, inputs, and outputs are the building blocks of an FSM. The collection of states represents all the possible situations in which the FSM may be. The model goes through a sequence of transitions to reach a certain state. A state is some kind of a memory that represents the current state of the model. From a software point of view, a state can be a set of specific values for a collection of variables. A transition is an allowable two-state sequence that results in an output and must specify a starting state and a final state (of the transition). A transition usually means a change in the value for state variables. An input triggers a transition.

Definition 2.3.
Formally, an *FSM* [48] is a quintuple (S, I, O, h_S, S_{in}), where

- S is a finite set of states with the set $S_{in} \subseteq S$ of initial states;
- I is a finite non-empty set of inputs;
- O is a finite non-empty set of outputs;
- $h_S \subseteq S \times I \times O \times S$ is a transition or behaviour relation, where a 4-tuple $(s, i, o, s') \in h_S$ is

a transition.

If $|S_{in}| = 1$, then the machine is *initialized*, otherwise it is *non-initialized*. In this work, we consider initialized machines.

In spite of their good expressiveness, FSMs are not powerful enough to model in a succinct way practical systems. For example, systems which contain variables and where their operations depend on the variable values. The EFSM model extends the classical FSM model with input and output parameters, context variables, update functions and predicates defined over context variables and input parameters. It is more adequate to model complex reactive systems.

The contribution of Chapter 6 proposes an EFSM to model the switch-to-controller communication and investigates the problem of deriving input test sequences based on such model.

In the remaining of this subsection, we give formal definitions related to EFSM and a simple illustrative example.

Let X and Y be finite sets of inputs and outputs; IN_p, OUT_p and C_v be finite disjoint sets of input/output parameters and context variables respectively. Some inputs (outputs) are related to subsets of parameters. For $x \in X$, let $IN_{px} \subseteq IN_p$ be the set of input parameters of x and let $D_{IN_{px}}$ be the set of input vectors, each component of an input vector corresponds to an input parameter associated with x. The set of output parameters and vectors are similarly defined. Let D_{C_v} be the set of context vectors $\mathbf{v}$. Given an input x and a (possibly empty) set of input vectors, a parameterized input is a tuple $(x, \mathbf{px})$ where $\mathbf{px}$ is an input parameter vector. A sequence of parameterized inputs is called a parameterized input sequence. Parameterized outputs and their sequences are defined similarly.

Definition 2.4.
An *EFSM* [118] $\mathcal{S}$ over X, Y, IN_{px}, OUT_{py}, C_v, $D_{IN_{px}}$, $D_{OUT_{py}}$ and D_{C_v} is a pair (S,T) of a finite set of states S and a finite set of transitions T between states in S, such that each transition $t \in T$ is a tuple $t = (s, x, P, op, y, up, s\prime)$, where

- $s, s\prime \in S$ are the initial and final states of the transition, respectively;
- $x \in X$ is the input of the transition;
- $y \in Y$ is the output of the transition;
- P, op and up are functions, defined over input parameters and context variables
  - $P : D_{IN_{px}} \times D_{C_v} \rightarrow \{True, False\}$ is the predicate of the transition;
  - $op : D_{IN_{px}} \times D_{C_v} \rightarrow D_{OUT_{py}}$ is the output parameter function of the transition;
  - $up : D_{IN_{px}} \times D_{C_v} \rightarrow C_v$ is the context update function of the transition.

If a transition t has a predicate, the latter must be satisfied in order for t to be enabled. A configuration of $\mathcal{S}$ is a pair $(s, \mathbf{v})$.

Definition 2.5.
An EFSM $\mathcal{S}$ [118] is

- *Deterministic* if any two transitions outgoing from the same state with the same input have mutually exclusive predicates;
- *Complete* if for each pair $(s, x) \in S \times X$, there exists at least one transition at state s with the input x, otherwise $\mathcal{S}$ is called *partial*;

- *Initially connected* if each state of S is reachable from the initial state.

In this thesis, we consider deterministic, initialized, initially connected but not necessarily complete EFSMs.

Example

We illustrate the notion of an EFSM and how it operates through a simple example. Consider an EFSM given in Figure 2.5 which is defined over state set $S = \{State_1, State_2, State_3\}$, inputs a and b, i.e., $X = \{a, b\}$, where b is non-parameterized and a is parameterized with an integer parameter k with value $a.k$, outputs 0 and 1, i.e., $Y = \{0, 1\}$. The set of context variables is $C_v = \{w\}$. In this example, we assume that D_{C_v} (domain of the variable w) is the set of all non-negative integers. The set of input parameters is IN_p and we assume that the domain $D_{IN_{pa}}$ (domain of the input parameter k) is the set of all non-negative integers, while $D_{IN_{pb}} = \varnothing$ as b is non-parameterized. The EFSM of this example is deterministic, initialized ($State_1$ is the initial state), initially connected and partial.

For a parameterized input, for example a, let $a(0)$ denote the fact that the EFSM receives the input a with the parameter value $a.k = 0$. The machine has six transitions. For example, it has $t_1 = (State_1, a, 1 \leq a.k \leq 5, -, 0, w := w + 1, State_2)$ with states $State_1$ and $State_2$ as start and final states, respectively, the predicate $1 \leq a.k \leq 5$, and variable update function $w := w + 1$. Assume that $(State_1, w = 0)$ is a current configuration of the EFSM and the machine receives a parameterized input $a(k)$, then the machine checks the predicates of outgoing transitions from $State_1$ that are satisfied for the current configuration under the input a with parameter value $a.k$. If the received value $a.k = 3$, then the machine checks predicate $1 \leq a.k \leq 5$. As $1 \leq a.k \leq 5$ holds, the transition t_1 is executed according to the context update function $w := w + 1$ with output 0, and the machine moves from $State_1$ to the final state $State_2$ as specified by t_1. In fact, the machine moves from configuration $(State_1, w = 0)$ to configuration $(State_2, w = 1)$.

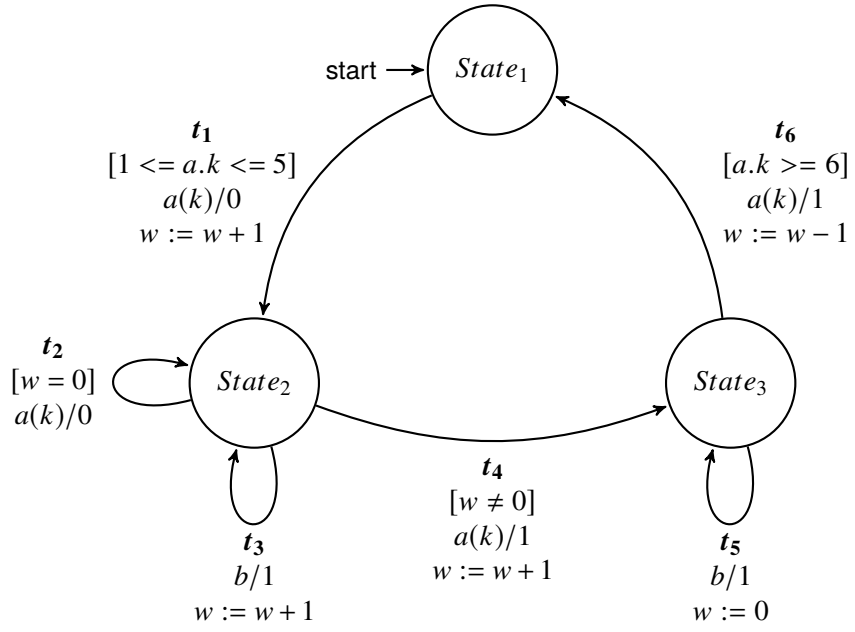


Figure 2.5 – Example of an EFSM

2.3.4 Fault Models

As the quality of a test suite is usually measured by its fault coverage, i.e., the types and number of faults that can be detected by the test suite, the proposed testing methods in this thesis introduce different fault models and seek for test suites with guaranteed fault coverage that can be stated as (necessary and) sufficient conditions [119] for a test suite *exhaustiveness* / *completeness* (Definition 2.6 below).

The main motivation for using fault models is to have tests which can detect, i.e., cover certain types of implementation faults. Such tests offer a 'guarantee' for the test quality in terms of fault coverage.

In a usual way, a fault model is defined as a tuple $\langle \mathcal{S}, @, \mathcal{FD} \rangle$ [120] where $\mathcal{S}$ is the specification, $@$ is the conformance relation and $\mathcal{FD}$ is the fault domain. In general, the specification is a formal model of the system, however, it might also be defined by a set of (end-user) requirements that should be correctly implemented.

The relation $@$ defines the conformance of a given implementation I to the specification $\mathcal{S}$. If the specification is complete (completely specified) then the conformance relation can be chosen to be equivalence and can be represented for example by the equality. If the specification is partial, then the conformance relation can be represented for example by the quasi-equivalence denoted as $\simeq$. The fault domain $\mathcal{FD}$ is a set of implementations. In model based testing, the specification model can be altered (mutated) in order to model a fault in the implementation. $\mathcal{FD}$ can be defined to contain the resulting mutants. Checking that a mutant from $\mathcal{FD}$ is not equivalent to the specification means to guarantee that the implementation does not implement any of the incorrect behaviours. The conformance relation $@$ partitions the set $\mathcal{FD}$ into conforming and nonconforming implementations (mutants).

As usual, an implementation $I \in \mathcal{FD}$ is called conforming if $I @ \mathcal{S}$; otherwise, I is a nonconforming implementation. Given the specification $\mathcal{S}$, a test case is a finite input sequence of $\mathcal{S}$. An implementation under test passes a test case/sequence if the output response of the implementation to the test case/sequence is contained in the set of output responses of the specification $\mathcal{S}$ to the test case; otherwise, the implementation under test fails the test case. As usual, a test suite is a finite set of test cases. An implementation under test passes (fails) a test suite if the implementation passes each test case (or fails some test case). If an implementation fails a test suite then we say that the implementation can be detected with the test suite.

Definition 2.6.

- A test suite TS is said to be *exhaustive* w.r.t. the fault model $\langle \mathcal{S}, @, \mathcal{FD} \rangle$ if each implementation $I \in \mathcal{FD}$ such that $I \not@ \mathcal{S}$ can be detected with TS.
- A test suite TS is said to be *sound* w.r.t. the fault model $\langle \mathcal{S}, @, \mathcal{FD} \rangle$ if each implementation $I \in \mathcal{FD}$ such that $I @ \mathcal{S}$ passes TS.
- A test suite TS is said to be *complete* w.r.t. the fault model $\langle \mathcal{S}, @, \mathcal{FD} \rangle$ if TS is exhaustive and sound.

2.4 Mutation Analysis

Mutation analysis is a powerful approach for both evaluating test suites' effectiveness and supporting test generation [111]. The principle idea is to inject 'artificial' faults, called mutations,

into the code or the specification model yielding mutants. It allows to measure test effectiveness based on the number of detected mutants. Researchers have proven that detecting mutants results in finding real faults [74]. In particular, this has been shown as well for model based mutation [3]. Indeed, it has been demonstrated that specification model mutants lead to tests that are able to reveal implementation faults that were neither found by manual tests, nor by industrial tools [3]. Moreover, model based mutation's power is to identify faults related to missing functionality and misinterpreted specifications.

In model based testing, mutants are introduced based on model transformation operators that alter the specification. When the faults injected in the specification model to obtain corresponding mutants are defined by a user such as an expert, a test engineer, etc.; the resulting mutants are referred to as *user-defined* mutants. There are two kinds of mutants, first-order mutants when the specification and the mutant models differ by a single model transformation, and higher-order mutants, derived from the specification model after multiple transformations. When a mutant is detected by a test sequence (case), it is said to be *killed*. Otherwise, it is said to be *survived*. To measure the adequacy of testing and assess the *fault coverage* of test suites, a standard metric called *mutation score* is used. It is defined as the ratio of mutants killed by the test suite under assessment to the total number of unique mutants [10]. This ratio gives an evaluation of the fault revealing power of a test suite [111]. To calculate the mutation score, one has to execute the whole test suite against every selected mutant.

In the thesis, we make use of mutation analysis. In chapter 4, the fault coverage of the derived test suites is evaluated based on a code mutation. In Chapters 5 and 6, the established models are mutated in order to support test generation, and experimental evaluations based on mutation score metric are conducted in both chapters in order to prove the effectiveness of the proposed testing approaches.

2.5 Black Box and White Box Testing

In testing, the black box approach is a technique for test case generation where test cases/sequences are constructed according to information derived from the specification or requirements without requiring knowledge of the internals of the system. In other words, a system under test (SUT) is treated as a black box, i.e., we do not have any knowledge about the internal structure of the SUT. Only information about what inputs does the SUT expect and what are the specified outputs is available, without knowledge of how the SUT derives those results. This means Black Box testers do not have access to the source code and are oblivious of the SUT architecture. Note that the requirements might be user-defined. For example, in Chapter 4, while making use of this approach, the test cases are constructed based on the requests of an end user (a network administrator/operator).

A Black Box tester typically interacts with an SUT through interfaces by providing inputs (points of control) and examining outputs (points of observation) without knowing where and how the inputs were operated upon. In Black Box testing, the SUT is exercised over a range of inputs and the outputs are observed for testing correctness.

White box testing refers to the technique of testing an SUT with knowledge of the internals of the system. White box testers have access to the source code and are aware of the system architecture. A white box tester typically analyzes source code, derives test cases from knowledge about the source code, and finally targets specific code paths to achieve a certain level of code coverage. A white box tester with access to details about the SUT can readily craft efficient test cases that exercise specific parts of the SUT for example. This allows the tester to examine for example parts of a system that are 'suspicious', rarely tested or pointed out as

‘doubtful’ by an ‘expert’/‘knowledgeable’ user.

Black box and white box testing approaches both choose test cases that investigate a particular characteristic of the system, however in white box testing, test cases can be generated to test some implementation specific aspects of the system.

2.6 Chapter Conclusions

The aim of this chapter has been to discuss the backgrounds of SDN architectures and their components on one hand and the basics of testing related notions on the other hand. Herein, the details on SDN, verification and testing, model based testing, mutation analysis, and black and white box testing approaches have been presented.

3

State Of The Art

"Science arose from poetry. . . when times change the two can meet again on a higher level as friends."

– Johann Wolfgang von Goethe

Contents

3.1 Introduction

The process of verification/testing of an SDN architecture/component involves checking whether the latter behaves in the way it was designed to behave. In this thesis, formal approaches for testing SDN architectures/components are proposed. With this in mind, the first step is to investigate the literature works that have contributed to developing techniques for SDN verification and testing. This chapter reviews this field of research. At first, the current introduction refers the reader to other related surveys in the field. Afterwards, the state of the art solutions are presented with emphasis on their mode of operation, what objective they have addressed and what solutions they have proposed.

It is worth mentioning that the literature study in this chapter does not include SDN security and fault tolerance works. Indeed security and fault tolerance can be considered as an aspect of overall network correctness or lead to network errors. However, we believe their related techniques need independent reasoning and examination.

The general principles of SDN have been covered in several surveys that appeared as early as 2012, e.g., [110, 84, 69, 167, 5]. However, in these works, the verification and testing techniques applied to SDN have been briefly overviewed. A layered description of existing work in this area has been covered in [156]. In the same line, some tools for SDN testing and debugging have been covered in [65], [107], [55], [127], [110], and [84]. A brief overview of the challenges related to SDN correctness has been provided in [72]. An insight into various mathematical tools and verification methods used in the analysis of SDN has been covered in [56]. A brief introduction of the works on the same topic has been presented in [124]. To the best of our knowledge, the topic area of verification and testing SDN architectures and their components has so far only been covered in detail by one survey. Li et al. [92] recently have focused on the application of formal verification and testing methods to both traditional and SDN architectures.

We note that in the majority of the aforementioned efforts, SDN verification and testing techniques have either been barely discussed as part of the challenges SDN brings or have been a part of a more wide study about networking in general. In this chapter, we present a comprehensive survey of the research relating to this topic that has been carried out to date. We analyze and summarize different solutions and categorize them w.r.t. the techniques they involve.

The remainder of this chapter is structured as follows. Section 3.2 presents the first group of existing solutions related to SDN verification. Section 3.3 is devoted to the group of existing SDN testing techniques. Each of the aforementioned sections introduces the technique of interest, investigates its application to SDN, illustrates it with an example and finally derives some general conclusions. The Subsections 3.2.3 and 3.3.6 recapitulate the verification and testing efforts and identify their limitations respectively. Finally, Section 3.4 summarizes the analysis and concludes the chapter.

3.2 Verification Techniques for SDN

In this section, we summarize and classify existing SDN verification techniques. Subsequently, we present two main categories namely Off-line and Run-time. To illustrate the underlying mechanism behind a given technique, a simple example is usually provided based on the network topology of Figure 3.1.

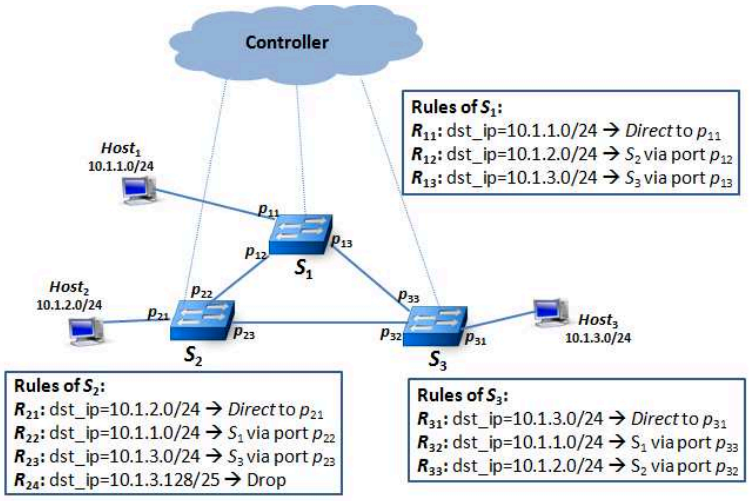


Figure 3.1 – An example of an SDN network topology

3.2.1 Off-Line SDN Verification Techniques

These techniques are the most popular for checking the SDN correctness. They verify a fixed configuration of the network by making the assumption that the forwarding behaviour remains the same as far as the controller does not explicitly instruct new rules, update or remove rules in the switch. In general, off-line verification techniques consider the global behaviour of the SDN network topology as a snapshot of network state which they analyze and then the predefined network properties are checked.

3.2.1.1 SAT Solving

SDN verification problem can be reduced to a satisfiability (SAT) problem and solved by SAT solvers. SAT expresses the problem as Boolean expressions using propositional logic. The interesting question here is: how does this technique reduce an SDN network verification problem to a SAT one?

SAT solving has been applied to the data plane where in fact the forwarding rules in each switch are represented by Boolean expressions. Next, the property to check (e.g., reachability) is expressed a Boolean expression as well (see example). Note that usually a counterfactual reasoning is used to enable a form of negation of a property in order to prove it holds. Once the SDN verification problem is formulated using Boolean expressions, deciding the satisfiability of such expressions, i.e., determining if there exists an assignment (or prove there does not exist) of the Boolean variables that makes the expression logically True, allows to conclude whether the property holds. This is done by feeding the resulting expression as input to a SAT solver. In case the property is violated, a counterexample is returned. For example, to express the data plane verification problem as a SAT one, the matching part of each rule can be encoded as a Boolean expression in the following way. Consider the matching part of a given rule denoted as $(p_1 \in V_1 \wedge p_2 \in V_2 \wedge \ldots \wedge p_n \in V_n)$ where $p_1, p_2, \ldots, p_n$ refer to the variables representing the packet fields (e.g., MAC address, IP address, port number) and $V_1, V_2, \ldots, V_n$ refer to the intervals wherein these variables can take values. For example, $dst_{ip} \in 10.1.3.0/24$ means that the IP destination address is in the subnet $10.1.3.0/24$. This matching part is represented by a Boolean expression $exp_n = var_1 \wedge var_2 \wedge \ldots \wedge var_n$ such that $p_i \in V_i$ is mapped to var_i that takes the value True if the value of p_i is indeed in the interval V_i and False otherwise. The expression exp_n, for a given assignment of truth values for the variables var_i, results in one

of the Boolean value True or False.

SAT solving technique has been first proposed by Mai et al. [97] who have shown how network properties can be translated into SAT instances which are checked using a SAT solver for detecting potential problems / issues in the data plane, in particular violations of properties such as absence of routing loops and black-holes. The tool implementing this technique is called Anteater. The experiments have shown that, for example for checking three standard network invariants in a campus network, Anteater spends two hours. Then, McGeer [101] has extended this idea to consider a network of OF switches as a network of Boolean expressions. The network properties have been reduced to logic expressions over the variables of this network. In the same line of applying this technique to the data plane, Zhang et al. [173] have also formulated the verification problem as a SAT problem. Reachability and loops are the properties checked in this work.

Example

Consider the network topology of Figure 3.1 and the SDN verification problem is to check reachability between S_1 and S_3. This example is inspired by the paper of Mai et al. [97]. The matching part of each rule in each switch is encoded as a Boolean expression (in our example, for simplicity, each Boolean expression includes one Boolean variable, indeed here we use only the destination IP addresses). R_{12} of switch S_1 that forwards packets to S_2 is represented as the Boolean expression $exp = var_1$ where var_1 is a Boolean variable that represents $dst_{ip} \in 10.1.3.0/24$. All the other rules are encoded similarly.

- The Boolean expression that represents the forwarding between a host and a switch or between a switch and a host is $exp_{Host_iS_i} = exp_{S_iHost_i} = var_{Host_iS_i}$ where $var_{Host_iS_i}$ is a Boolean variable that represents $dst_{ip} \in 0.0.0.0/24$.

- The Boolean expression that represents the forwarding between two switches (e.g., S_1 and S_2) is $exp_{S_1S_2} = var_{S_1S_2}$ where $var_{S_1S_2}$ is a Boolean variable that represents $dst_{ip} \in 10.1.2.0/24$. The Boolean expressions that represent the forwarding between S_1 and S_3, S_2 and S_1, S_2 and S_3, S_3 and S_1, S_3 and S_2 are encoded similarly.

The resulting Boolean expression that expresses one possible path from $Host_1$ to $Host_3$ is then $var_{Host_1S_1} \wedge var_{S_1S_2} \wedge var_{S_2S_3} \wedge var_{S_3Host_3}$, another path can be expressed by $var_{Host_1S_1} \wedge var_{S_1S_3} \wedge var_{S_3Host_3}$ and consequently to check the reachability between $Host_1$ to $Host_3$, it suffices to feed the Boolean expression depicted in Equation 3.1 to a SAT solver that will try to find an assignment (if it exists) that makes Equation 3.1 evaluate to True.

$$\begin{aligned} exp_{Host_1Host_3} = &(var_{Host_1S_1} \wedge var_{S_1S_2} \wedge var_{S_2S_3} \wedge var_{S_3Host_3}) \vee \\ &(var_{Host_1S_1} \wedge var_{S_1S_3} \wedge var_{S_3Host_3}) \end{aligned} \tag{3.1}$$

In this simple example, the Boolean expression depicted in Equation 3.1 is satisfied, hence the property holds.

SAT solving technique determines whether a Boolean formula expressing SDN components and properties is satisfiable. It has mostly been applied to the data plane layer and has been used in checking properties such as reachability and loops.

3.2.1.2 Symbolic Execution and SMT Solvers

As any verification technique, symbolic execution [83, 21] checks whether certain properties can be violated by the system. This is done by simultaneously exploring multiple paths that the system could take under different inputs. The system can take symbolic (rather than concrete) input values. The execution is performed using a symbolic execution engine. For each explored path, the engine maintains two important pieces of information: a logic formula that describes the conditions satisfied by the branches taken along that path, and a symbolic variable, say $store$ that maps variables to symbolic expressions or values. The execution of a 'branch' updates the formula, while 'assignments' update the symbolic variable $store$. A satisfiability modulo theories (SMT) solver is then used to verify whether there are any violations of the property along each explored path and if the path is feasible, i.e., if its formula can be satisfied by some assignment of concrete values. In the following, we investigate how such a technique is applied to solve an SDN network verification problem.

In fact, symbolic execution technique models the forwarding behaviour of an SDN architecture (or component) by describing the behaviour of each forwarding table (and hence rules). The packet header fields (e.g., IP address, MAC address) are expressed symbolically where each variable representing a field is symbolic, i.e., assigned a set of values that is specified by an associated constraint. To symbolically execute the whole model, the symbolic engine follows multiple branches on each encountered rule. It exercises all possible paths and records the constraints of the symbolic input on each path. The constraints define which values of the input vector lead to this path. Then, the path constraints are solved using a constraint solver determining if the property is violated. For example, when switch S_1 of the network in Figure 3.1 receives a packet, it is handled according to its table. However because the packet fields do not have a concrete value, no specific rule is applied to it, instead, the approach branches out taking into account all possibilities of actions of S_1. S_1 has three rules, namely: R_{11} that forwards packets to $Host_1$, R_{12} that forwards packets to S_2 and R_{13} that forwards packets to S_3. In this case four branches are considered; the first one is when R_{11} is applied, the second is when R_{12} is applied, the third one is when R_{13} is applied and the last one is when S_1 sends $Packet\_Out$ message to the controller without applying any rule. Suppose that $Host_1$ wants to send a packet pkt to $Host_2$, this can be expressed by $dst_{ip}(pkt) == ip(Host_2)$ (a). When pkt arrives at S_1, the expression from the applied rule in the first branch is $dst_{ip}(pkt) == ip(Host_1)$ (b). Constraint solvers are used to solve these expressions along each path. In this example, (a) and (b) are not satisfied simultaneously so the path that takes the first branch is not feasible.

This technique has first been introduced by Dobrescu et al. [43] where the pipeline processing of a switch is modelled as a tree that consists of subtrees, one per table, and the property to verify is crash-freedom. The model describing the behaviour of a switch is composed of models describing the behaviour of each of its tables. A table model takes as input a symbolic vector representing the fields of a network packet and outputs either the same vector or a modified version of it, which in return is fed to the next table in the pipeline. A symbolic vector is a vector of symbolic variables. Dobrescu et al. have extended this work in [42] where they have added more properties to be checked. Panda et al. [114] have applied this approach to the data plane as well. In their work, network properties such as reachability and isolation have been expressed using formulae that imply constraints on data flow within the model, then, the SMT solver Z3 [37] has been used to check if the specified properties hold. Wu et al. [153] have followed the same approach to generate a model of the switch from its source code, the resulting model can then be used to check a set of network properties. Yakuwa et al. [158] have applied this approach to the SDN network modelled as a transition system and the SMT solver Yices has been used to execute the paths and solve the constraints.

In the same line, Kazemian et al. [77] have proposed to simulate the network model using

'symbolic' variables instead of concrete packet field values at the inputs of the network model (the space of all possible packet headers localized at all possible input ports in the network called 'the network space'). The switches are modelled as transfer functions which map header $head$ arriving on port $port$ to $(header, outputport)$. During this process, expressions based on the initial symbolic variables and the functionality of each of the switches are derived. The set of expressions represents implicitly the set of states that are reachable by the set of packets with an appropriate set of inputs. Therefore, this allows the behaviour of the network (in a specific state) to be verified with a single execution step, simultaneously under all possible input packets. This process is called 'Header Space Analysis' (HSA). Now for a given property such as reachability, HSA computes it from source $Host_1$ to destination $Host_n$ via switches $S_1, S_2, \ldots, S_{n-1}$ as follows. First, a header space region at $Host_1$ is created representing the set of all possible packets $Host_1$ can send: a symbolic representation of packet fields. Next, switch S_i's transfer function is applied to the representation of the packet to generate a set of regions at its output ports, which in turn are fed to S_{i+1}'s switch transfer function. The process continues until a subset of the flows that left $Host_1$ reaches $Host_n$. Wang et al. [150] have applied the HSA approach for checking SDN firewall applications.

Canini et al. [26, 25] have modelled an SDN controller event handler application as a transition system. At each state, the system exposes a set of possible transitions, each of which evolves the system from one state to another. To check the application correctness, the approach checks after traversal of each transition that predefined network properties (specified as Python code snippets, e.g., loops and black-holes) hold in the current state. To accomplish this, the resulting transition based model is subject to checking using symbolic execution where the latter involves symbolic packets (defined as a group of symbolic integer variables that each represents a header field). The algorithm that checks if the property holds in each state of the model runs as follows. At any given state, each transition (modelling the event handler application) is 'symbolically' executed. This allows the set of packets that exercise all code paths in the event handler to be identified. For every feasible path, the symbolic execution engine finds an equivalence class of packets that enable such a path. For each equivalence class, one 'concrete' packet is chosen which will enable the next transition to the next state of the controller application under verification. Therefore from each state, there are as many transitions as the number of equivalence classes. In summary, while checking the properties, the state space is exploded by 'symbolically' executing the transitions. Experimental results of the work have shown that this approach (implemented as a tool called NICE) can be effective only for relatively small networks due to potential state explosion.

Example

This example is motivated by Dobrescu et al. [43]. Let us suppose that the switch S_1 of Figure 3.1 is composed of two tables T_1 and T_2. The models of these tables and their corresponding trees are shown on the left part of Figure 3.2. The resulting tree is shown on the right side. In the first step of the verification of S_1, each table is verified individually. In the second step, the table is checked as a part of the pipeline. The verification in the first step is done as follows. Each table model is executed with symbolic input (Pkt in this example) and the segments that may cause the property to be violated are marked as 'suspect'. At the end of this step, a constraint and a symbolic variable $store$ are obtained for every feasible segment, e.g., for seg_1 the constraint is $c_1 = (Pkt < 0)$ and the symbolic variable $store$ is $sym_1(Pkt) = Pkt_{out} \leftarrow 0$, for seg_3, the constraint is $c_3(Pkt) = (Pkt < 0)$ and the symbolic $store$ is $sym_3(Pkt) = crash$. Segment seg_3 causes a crash, it is marked as suspect. The verification in the second step is done as follows. Each path p_i (a sequence of segments seg_i) that includes at least one suspect segment is checked. In this case, the paths that include the suspect segment are p_1

and p_4. The resulting constraint at p_1 is $c_{p_1} = (c_1(Pkt) \wedge c_3(sym_1(Pkt))) = (Pkt < 0) \wedge (0 < 0)$ which is always False and thus p_1 is not feasible, similarly for p_4. However, all feasible paths consist of non-suspect segments (never crash), therefore the property of crash-freedom holds in switch S_1.

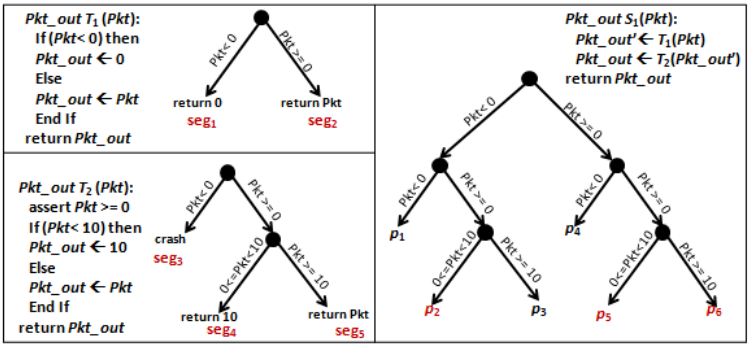


Figure 3.2 – A symbolic execution encoding of the switch S_1 of the topology in Figure 3.1

Example adapted from Dobrescu et al. [42]

Not only has symbolic execution technique been applied to the data plane layer but also to the control plane (e.g., Canini et al. [26, 25]) and to the whole architecture as well (e.g., Yakuwa et al. [158]). Symbolic execution technique has been employed to check properties such as reachability, absence of loops, black-holes and crash-freedom.

3.2.1.3 Model Checking Over Temporal Logic

Model checking over Temporal Logic is a verification technique that provides an algorithmic means of determining whether an abstract model (representing, for example, a hardware or software design) satisfies a formal specification expressed as a temporal logic formula. Moreover, if the property does not hold, the method identifies a counterexample execution that shows the source of the problem [31]. In the following, we discuss how such technique is adopted to solve an SDN verification problem.

In fact, the SDN architecture/component in this case is represented as a finite state model and the properties to be verified (e.g., reachability, absence of black-holes, absence of loops, OF rule consistency, etc.) as temporal logic formulae, then a checking if the properties hold for the resulting model is performed. To illustrate this technique, let us use Computation Tree Logic (CTL) to model an SDN architecture as a transition system where a state of the network can be specified by the pair $(pkt, switch)$ such that pkt denotes the packet and $switch$ denotes its location. The transitions between the different states can be specified by the rules installed in each switch of the topology. For example, the CTL formula $dst_{ip}(pkt) = 10.1.3.0 \& switch = S_1$ is satisfied by all packets with IP destination address $10.1.3.0$ which are located in the switch S_1. The formula $AF(pkt, switch = S_3)$ states the formula $(pkt, switch = S_3)$ holds at some points in the future. In this case, packets can always reach S_3 from their current location. The formula $(pkt, switch = S_2) \implies AF(pkt, switch = S_1)$ states that all packets at switch S_2 must eventually be forwarded to S_1. The different rules in each switch of the network as well as the property can be expressed using CTL in this way and then be fed to a model checker. If the model violates the property, the model checker returns a counterexample.

For checking the forwarding behaviour of a switch or the whole data plane, the rules have been modelled as a state based system and the properties have been expressed either in LTL as done by Peresini et al. [116] or in CTL as performed by Gutz et al. [61] and Al-Shaer et al.

[136, 135]. Model checkers such as SPIN [67], JPF [149], SMV, and NuSMV [28] have been used.

For checking the control plane, certain applications of the controller have been modelled either as a finite transition system or a timed automata (TA) [9] as performed by Croft et al. [34]. Properties have been expressed for example in LTL as done by Skowyra et al. [140], or CTL as done by Kim et al. [82]. Also model checkers such as SPIN, NuSMV and Alloy (Nelson et al. [108, 109]) have been used.

For checking the whole SDN architecture, different models have been proposed. A transition system has been adopted by Majumdar et al. [98] and Skowyra et al. [141]. Properties such as reachability and black-holes have been expressed using LTL. For example, $G\Phi$ is a formula where Φ is a formula over a set of atomic propositions and G is the operator of LTL that means 'globally'. Majumdar et al. have developed a model checker called 'Kuai' to solve the verification problem. A network of finite timed automata (parallel composition of timed automata) [8] has been used by both Kang et al. [76, 75] and Podymov et al. [123]. The properties have been expressed using TCTL logic, i.e., CTL augmented which allows considering several possible future from a state of the system. VERSA [30] and UPPAAL have been employed. Albert et al. [4] have encoded an SDN architecture into a specific language called Abstract Behavioral Specification [73], then model checking has been used to check properties including rules consistency and absence of loops. Zakharov et al. [165] have proposed a finite automata based model to capture the behaviour of an SDN architecture. The reachability property has been specified using temporal logic. Shkarupylo et al. [139] have modelled an SDN architecture using the Temporal Logic of Action (TLA) formalism [90]. The reachability property has been verified. A TLA model checker has then been used in two different checking modes namely breadth-first search (BFS) and depth-first search (DFS). Yuzhuang et al. [164] have composed 'already' verified SDN components (separately) and have formulated the network properties such as reachability and deadlock to verify them on the resulting model. The result is subject to be verified by a model checker.

Example

We consider the network topology in Figure 3.1 and the property to check is reachability from $Host_1$ to $Host_3$. We provide a simple example showing the reduction of the problem of the verification of this topology into a model checking problem. The example is depicted by Equation 3.2 (using CTL). The initial state represents all possible packets are at host $Host_1$. The forwarding rules on switches moving packets between ports can be viewed as state transitions. For example, the first formula states that all packets with IP destination address $10.1.2.0$ which are located in the switch S_1 must eventually be forwarded to S_2. The model and property can be input to a model checker (e.g., SMV). If the latter returns pass, the property holds on the model. If the model violates the property, a counterexample is returned. In this simple example, the model satisfies the property.

$$
\begin{array}{ll}
(pkt, Host_1)\ (Initial\ state) & \text{Can packets from } Host_1 \text{ reach } Host_3 \text{ ?} \\
(pkt, Host_1) \implies (pkt, S_1) & \textbf{Check:} \\
(dst_{ip}(pkt) = 10.1.2.0\ \&\ switch = S_1) \implies AF(pkt, switch = S_2) & EF(pkt, S_3)? \\
(dst_{ip}(pkt) = 10.1.3.0\ \&\ switch = S_1) \implies AF(pkt, switch = S_3) & E \text{ means that } \exists \text{ at least one path starting from} \\
(dst_{ip}(pkt) = 10.1.1.0\ \&\ switch = S_2) \implies AF(pkt, switch = S_1) & \text{the current state where the property holds.} \\
(dst_{ip}(pkt) = 10.1.3.0\ \&\ switch = S_2) \implies AF(pkt, switch = S_3) & F \text{ means that the property eventually has to} \\
(dst_{ip}(pkt) = 10.1.1.0\ \&\ switch = S_3) \implies AF(pkt, switch = S_1) & \text{hold (somewhere on the subsequent path).} \\
(dst_{ip}(pkt) = 10.1.2.0\ \&\ switch = S_3) \implies AF(pkt, switch = S_2) &
\end{array}
\tag{3.2}
$$

Model checking technique has been applied to the data plane, control plane and the whole architecture. A variety of properties including reachability, loops and black-holes have been checked using such technique.

3.2.1.4 Deductive Verification and Theorem Provers

Similar to model checking, theorem proving technique expresses the properties to be verified as logic formulae, then axioms and inference rules serve to derive new formulae from existing ones. The technique checks whether the property is valid with the axiom and derivation rules [36]. How this technique is applied to solve an SDN verification problem will be our main focus in this subsection.

We provide a simple example of theorem proving encoding illustrating how this technique can be applied to an SDN architecture/component in the example below.

A first group of works applies this technique to SDN by expressing both the SDN architecture and the property to be checked using a set of formulae in a given logic. Then the relation between these two entities is verified as a proof using a theorem prover. Ball et al. [14] have modelled the controller events (e.g., the receipt of a packet from a switch) and switch events (e.g., executing a rule and forwarding an incoming packet to a certain port (or dropping it)) as well as desired properties using first-order logic, and then have implemented classical Floyd-Hoare-Dijkstra deductive verification. The tool implementing their work is called VeriCon [14]. Examples of properties include the absence of black-holes. Anderson et al. [11] have described network applications as functions using packet histories. Then the Coq proof assistant [15] has been used to prove their correctness. The checked properties are reachability and traffic isolation. Attiogbe [12] has proposed an approach to build correct SDN components from the refinement of a global formal model of an SDN architecture, using the decomposition of the global model into the target components. The whole SDN architecture has been viewed as a discrete event system and modelled using set theory. Then, Rodin [47] prover has been used to establish model consistency. Examples of checked properties include 'the data packets received by any switch are sent by the controller or by the other switches'. Rodin supports LTL, and CTL for expressing properties with the standard modal and temporal operators.

For data plane verification, a similar approach has been developed by Chen et al. [27]. The approach uses a declarative language to describe the OF functionalities. Network properties such as reachability have been expressed using LTL and decomposed into global and local properties. The verification steps include specifying the global and local properties, generating lemmas for proving that local properties are satisfied given the global ones.

A second group of works employs theorem proving technique by developing 'new releases' of the SDN component to be verified. In this way, the SDN component under verification is verified in advance 'once and for all' before deployment. For example, McGeer [100] has proposed a new protocol (based on the OF protocol) that is proved. The author has demonstrated formally the correctness of this protocol based on theorems and lemmas. In particular, he has proven that only a single set of rules is present on a switch at any time. Guha, Reitblatt and Foster [60] have also developed a verified SDN controller in the Coq proof assistant [15] and have proven it correct against a formal specification and a detailed model of an SDN architecture. Gordon Stewart [144] has built upon Guha et al. work by providing a suite of tools for verifying properties that are input to Guha et al.'s verified controller.

Example

To illustrate an example of the theorem proving technique operating on the SDN topology of Figure 3.1, we use the propositional logic to model the forwarding behaviour (motivated

by Ball et al. [14]) and we prove that the reachability property from $Host_1$ to $Host_3$ holds (Equation (u)). Let S_i denote a switch, p_{ik} a port of S_i (k is the number of ports in S_i) and $Host_i$ a host. $link(S_i, p_{ij}, Host_i)$ denotes that host $Host_i$ is directly connected to switch S_i via port p_{ij}. $link(S_i, p_{ik}, p_{jk}, S_j)$ denotes that port p_{ik} of switch S_i is directly connected to port p_{jk} of switch S_j. We propose to use reductio ad absurdum strategy for proving. Some of the axioms are depicted as follows:

$$link(S_1, p_{11}, Host_1) \quad \text{(a)} \qquad link(S_1, p_{12}, p_{22}, S_2) \quad \text{(d)}$$
$$link(S_2, p_{21}, Host_2) \quad \text{(b)} \qquad link(S_2, p_{23}, p_{32}, S_3) \quad \text{(e)}$$
$$link(S_3, p_{31}, Host_3) \quad \text{(c)} \qquad link(S_1, p_{13}, p_{33}, S_3) \quad \text{(f)}$$

$$S_1.forwardTab(dst_{ip}(pkt) = 10.1.1.0, p_{11}) \quad \text{(g)} \qquad S_3.forwardTab(dst_{ip}(pkt) = 10.1.1.0, p_{31}) \quad \text{(m)}$$
$$S_1.forwardTab(dst_{ip}(pkt) = 10.1.2.0, p_{12}) \quad \text{(h)} \qquad S_3.forwardTab(dst_{ip}(pkt) = 10.1.2.0, p_{32}) \quad \text{(n)}$$
$$S_1.forwardTab(dst_{ip}(pkt) = 10.1.3.0, p_{13}) \quad \text{(i)} \qquad S_3.forwardTab(dst_{ip}(pkt) = 10.1.3.0, p_{32}) \quad \text{(o)}$$

$$link(S_1, p_{13}, p_{33}, S_3) \wedge S_1.send(dst_{ip}(pkt) = 10.1.3.0, p_{13}) \implies S_3.receive(dst_{ip}(pkt) = 10.1.3.0, p_{33}) \quad \text{(p)}$$
$$link(S_3, p_{31}, Host_3) \wedge S_3.send(dst_{ip}(pkt) = 10.1.3.0, p_{31}) \implies Host_3.receive(dst_{ip}(pkt) = 10.1.3.0, p_{31}) \quad \text{(q)}$$
$$link(Host_1, p_{11}, S_1) \wedge Host_1.send(dst_{ip}(pkt) = 10.1.3.0, p_{11}) \implies S_1.receive(dst_{ip}(pkt) = 10.1.3.0, p_{11}) \quad \text{(r)}$$
$$S_1.receive(dst_{ip}(pkt) = 10.1.3.0, p_{11}) \wedge S_1.forwardTab(dst_{ip}(pkt) = 10.1.3.0, p_{13}) \implies S_1.send(dst_{ip}(pkt) = 10.1.3.0, p_{13}) \quad \text{(s)}$$
$$S_3.receive(dst_{ip}(pkt) = 10.1.3.0, p_{33}) \wedge S_3.forwardTab(dst_{ip}(pkt) = 10.1.3.0, p_{31}) \implies S_3.send(dst_{ip}(pkt) = 10.1.3.0, p_{31}) \quad \text{(t)} \tag{3.3}$$

In order to prove the formula in Equation (u), we prove the contrapositive, that is the formula in Equation (v).

$$Host_1.send(dst_{ip}(pkt) = 10.1.3.0, p_{11}) \implies Host_3.receive(dst_{ip}(pkt) = 10.1.3.0, p_{31}) \quad \text{(u)}$$

$$\neg Host_3.receive(dst_{ip}(pkt) = 10.1.3.0, p_{31}) \implies \neg Host_1.send(dst_{ip}(pkt) = 10.1.3.0, p_{11}) \quad \text{(v)}$$

To perform such a proof, we use the resolution inference rule until either we can derive Equation (v) or we cannot apply the inference rule anymore.

First the implications in Equations (p), (q), (r), (s), and (t) can be written as follows:

$$\neg link(S_1, p_{13}, p_{33}, S_3) \vee \neg S_1.send(dst_{ip}(pkt) = 10.1.3.0, p_{13}) \vee S_3.receive(dst_{ip}(pkt) = 10.1.3.0, p_{33}) \quad \text{(aa)}$$
$$\neg link(S_3, p_{31}, Host_3) \vee \neg S_3.send(dst_{ip}(pkt) = 10.1.3.0, p_{31}) \vee Host_3.receive(dst_{ip}(pkt) = 10.1.3.0, p_{31}) \quad \text{(bb)}$$
$$\neg link(Host_1, p_{11}, S_1) \vee \neg Host_1.send(dst_{ip}(pkt) = 10.1.3.0, p_{11}) \vee S_1.receive(dst_{ip}(pkt) = 10.1.3.0, p_{11}) \quad \text{(cc)}$$
$$\neg S_1.receive(dst_{ip}(pkt) = 10.1.3.0, p_{11}) \vee \neg S_1.forwardTab(dst_{ip}(pkt) = 10.1.3.0, p_{13}) \vee S_1.send(dst_{ip}(pkt) = 10.1.3.0, p_{13}) \quad \text{(dd)}$$
$$\neg S_3.receive(dst_{ip}(pkt) = 10.1.3.0, p_{33}) \vee \neg S_3.forwardTab(dst_{ip}(pkt) = 10.1.3.0, p_{31}) \vee S_3.send(dst_{ip}(pkt) = 10.1.3.0, p_{31}) \quad \text{(ee)} \tag{3.4}$$

We can apply resolution inference rule to Equations (aa) and (f), and by resolving away $link(S_1, p_{13}, p_{33}, S_3)$ and $\neg link(S_1, p_{13}, p_{33}, S_3)$, we get Equation (aaa).

$$\neg S_1.send(dst_{ip}(pkt) = 10.1.3.0, p_{13}) \vee S_3.receive(dst_{ip}(pkt) = 10.1.3.0, p_{33}) \quad \text{(aaa)}$$

By applying also resolution inference rule to Equations (bb) and (c), we get Equation (bbb).

$$\neg S_3.send(dst_{ip}(pkt) = 10.1.3.0, p_{31}) \vee Host_3.receive(dst_{ip}(pkt) = 10.1.3.0, p_{31})$$
(bbb)

Similarly for Equations (cc) and (a), we get Equation (ccc).

$$\neg Host_1.send(dst_{ip}(pkt) = 10.1.3.0, p_{11}) \vee S_1.receive(dst_{ip}(pkt) = 10.1.3.0, p_{11})$$
(ccc)

Also the resolution rule applied to Equations (bbb) and (ee), we get Equation (ddd).

$$Host_3.receive(dst_{ip}(pkt) = 10.1.3.0, p_{31}) \vee \neg S_3.receive(dst_{ip}(pkt) = 10.1.3.0, p_{33}) \vee$$
$$\neg S_3.forwardTab(dst_{ip}(pkt) = 10.1.3.0, p_{31})$$
(ddd)

Similarly for Equations (ddd) and (m), we get Equation (eee).

$$Host_3.receive(dst_{ip}(pkt) = 10.1.3.0, p_{31}) \vee \neg S_3.receive(dst_{ip}(pkt) = 10.1.3.0, p_{33})$$
(eee)

And finally for Equations (eee) and (aaa), we get Equation (fff) which is equivalent to Equation (v) and thus the property is proven.

$$Host_3.receive(dst_{ip}(pkt) = 10.1.3.0, p_{31}) \vee \neg S_1.send(dst_{ip}(pkt) = 10.1.3.0, p_{13})$$
(fff)

Theorem proving technique has been applied to both the data plane and control plane layers. It can be used in checking a variety of properties such as reachability and the absence of black-holes. When applied to SDN, interactive theorem provers (e.g., Coq) that demand explicit user guidance in the proof have been utilized.

The next section examines and categorizes some of the run-time verification methods that have been applied to SDN architectures/components.

3.2.2 Run-Time SDN Verification

Run-time SDN verification seeks to verify the properties of the network in the presence of arbitrary updates from the controller. That is when rules' insertion, modification, and deletion are performed by the controller and thus the network changes over time. The solutions developed to solve this issue either extend the off-line verification or rely on traversal of dependency graphs modelling the SDN network topology.

3.2.2.1 Application of 'Off-line SDN Verification Techniques' Online

Several researchers have applied 'Offline SDN verification techniques' presented earlier in Subsection 3.2.1 and have added a solution to sketch the motion of the rules' updates in time in order to verify the properties guaranteeing that every packet traversing the network is processed by exactly one consistent global network configuration/rule. Depending on the proposed solutions to handle the updates, we classify the literature contributions using this technique into three groups. The first group encloses the works proposing to revise the model after each update and then apply off-line verification again. The second group encloses the works proposing a modelling language that incorporate the dynamic of the updates. Finally, the last group encompasses the works proposing the construction of a graph modelling the network, the revision of this graph for each update and the application of one of the off-line verification technique on the graph to check the property of interest.

In the first group, model checking and theorem proving techniques have been used to check properties of static network configurations before and after the updates. To this end,

a 'monitor' listens for incoming updates. Once a change happens, the model is updated accordingly and the off-line technique is run again over it. For example, Hussein et al. [71] have used the model checker UPPAAL in this way to verify that no violations would occur, had the rule update been installed in the switch. Examples of verified properties include loops and the time delay for a controller to update a switch versus the switch to forward a packet. Sethi et al. [133] have also used this method where a network model and the property have been specified using propositional logic formulae. The model has been incrementally constructed based on an abstraction that consists in focusing on the behaviour of the network in presence of one packet and abstracting away all the rest. At a given time, all the packets (except one) in the data plane are considered as the ones that trigger updates to the network state as they are forwarded as events to the controller. Only one packet at a time is then to be forwarded in the data plane. The switch table can then be abstracted to contain only rules about this packet and only updates corresponding to these rules are incorporated in the model. Murphi model checker [41] has been run over the resulting model. A combination of model checking and theorem proving techniques has been used in this first group as well, as related in the work by Reitblatt et al. [125]. In this case, a mathematical model that sketches the behaviour of the SDN architecture has been formalized and proven in the Coq proof assistant. Then, properties such as loops have been expressed using CTL. To show that the model still satisfies the desired properties after each update, NuSMV [28] model checker has been run over the updated model.

The second group of works proposes to handle dynamic changes by adding semantics to the used modelling language. This offers the ability to push new rules without modifying the checking engine. After the model is established, symbolic execution/SMT for example, as an off-line technique, can be applied. This has been done by Lopes et al. [95] where a given language (Datalog) has been augmented to allow specifying network invariants and model the forwarding behaviour of the changing network. Their solution has been implemented as an extension of the SMT solver Z3 where a number of optimizations have been added. Another work in this group has been proposed by Reitblatt et al. [126]. To guarantee that every packet exclusively uses either the old rule or the new rule and not some combination of the two, an abstraction mechanism has been incorporated in the modelling language, and then off-line model checking has been applied. The main idea behind the abstraction consists in stamping each packet with a version number at its ingress switch indicating which rule set should be applied.

The third group of works suggests the construction of a graph modelling the network topology, the revision of the graph upon a rule update, and the application for example of symbolic execution/SMT by simulating the resulting graph over symbolic packets. This has been done by Kazemian et al. [78] as an extension of their previous work on off-line symbolic execution technique [77]. A 'plumbing graph' which captures all possible paths of packets through the data plane has been constructed. Nodes in the graph correspond to the rules and directed edges represent the next hop dependency of these rules. A rule is represented as a tuple $\langle match, action \rangle$. A rule R_{11} has a next hop dependency to rule R_{21} if (a) there is a physical link from R_{11}'s switch to rule R_{21}'s switch; and (b) the domain of R_{21} has an intersection with range of R_{11}. The domain of a rule is the set of headers that match on the rule and the range is the region created by the action transformation on the rule's domain. A directed edge from rule R_{11} to R_{22} has a filter which is the intersection of the range of R_{11} and the domain of R_{22}. When a flow passes through a directed edge, it is filtered by the corresponding filter. A filter represents all packet headers at the output of R_{11} that can be processed by R_{22}. In response to an update, nodes are added/updated in the plumbing graph. The work by Shelly et al. [138] also fits in this third group. They have built upon the work of kazemian et al. [78].

The property to be checked is the capability of the controller to restore reachability in the presence of a link failure provided that the physical graph of the network is still connected. The approach is based on computing failure scenarios (a scenario represents a set of links to fail simultaneously) that maintain the reachability. For every link e, it checks if the network is still connected without e, then, if yes, fails the physical link e and see if forwarding is still possible between all nodes; if not, it reports the link failure. For this purpose, a monitor is interposed in the southbound interface. It computes the failure scenarios and schedules them for execution while monitoring the network. After a failure scenario is executed, the verification engine (simulation of the plumbing graph by symbolic packets [78]) verifies that the reachability still holds.

Off-line verification techniques presented earlier have been applied to determine whether the behaviour of the SDN architecture/component under verification satisfies the desired properties in the presence of updates. This approach has been applied to both data and control planes. It has checked for example reachability, loops and rules' consistency properties.

3.2.2.2 Dependency Graph Traversal

This technique reduces the SDN verification problem to the traversal of a 'dependency' graph modelling the architecture/component. A dependency graph is a directed graph where vertices may represent SDN components (e.g., switches) or rules in a switch, and edges represent the dependency relation between these entities (components, rules). To capture the 'live' network activity, the dependency graph is constructed using a monitor placed between the controller and the data plane. Two main solutions have been proposed depending on how this graph is built. In the first solution, the graph is pre-built, then, after each update, the graph is revised correspondingly, and the verification process is relaunched. The second solution builds the dependency graph on the fly. We divide the works using this technique into two groups correspondingly.

In the first group, Zeng et al. [170] for example have proposed a network policy checker called 'Libra'. To capture changes in the rules, the checker uses parallel processing to record network event streams from the controller. A dependency graph has been constructed based on all rules and recorded events. The nodes represent $(local\_switch, remote\_switch)$ pairs and the edges represent the forwarding relationship. Then a MapReduce [38] checker has been used in order to check network properties based on the pre-built graph. In fact, Libra partitions the graph into several sub-graphs corresponding to each subnet, then the properties have been checked on those sub-graphs in parallel, using a graph library. Another example of work in this first group has been related by El-Hassany et al. [64] where a happens-before graph that captures the ordering of events (e.g., OF messages, packets) has been pre-built. The violation of a property has been defined as the result of events concurrency (race) error. Filters have been defined that can query the graph to check for properties violations. For example, a 'commutativity' filter detects whether changing the order of two events affects the network state. Example of a detected violation concerns races occurring when the controller installs a set of rules and then sends a packet matching these rules without waiting for them to be committed first. In the same line, Zhang et al. [175] have proposed a 'Quantitative Forwarding Graph' (QFG) that represents how packets are forwarded. In a QFG, each node is denoted as a tuple (H, D, S, G), representing any packet in the packet header space H arriving at a network device D, when the network device is at a particular state S with performance G. An edge pointing from one node to another means the modification of a packet. Whenever the rules are updated, it is easier to find the affected QFG nodes as well as their dependencies, thus, the verification can be limited to only those affected flows. To check reachability for

example, a BFS is run on the QFG.Horn et al. [68] have constructed a directed edge-labelled graph from the rules based on equivalence classes ECs of packets (EC are defined as the set of packets that are treated the same across the data plane). The edges in this case are labelled by the matching/action part of a rule. An insertion of a new rule for example results in the creation of a new edge labelled with its matching part, and existing edges are updated correspondingly. The verification of reachability is also performed by a traversal of the graph.

The second group of work constructs the dependency graph on the fly while the updates are happening and checks if the property holds by a traversal of such graph. In this context, khurshid et al. [80, 81] have used a multidimensional prefix tree (trie) structure from which a dependency graph is generated each time an update from the controller is perceived. A trie is an ordered tree that associates the set of packets matched by a rule with the rule itself. Each level in the trie corresponds to a specific rule's field. Each node has three branches representing the possible values that the rule can match (0, 1, and * (wildcard)). The trie can be seen as a composition of several sub-tries, each corresponding to a packet header field. A path from the trie's root to a leaf of one of the bottom most sub-tries thus represents the set of packets that a rule matches. Each leaf stores the rules that match that set of packets, and the devices at which they are located. When a new rule is to be inserted, a traversal of the trie is performed to find all the rules that intersect it. These rules collectively define a set of packets that could be affected by the incoming rule. This set will generate equivalent classes ECs of packets. By traversing the trie for each EC, a dependency graph is generated where a node represents an EC at a particular device, and a directed edge represents a forwarding decision for a particular (EC, device) pair. To check reachability for example, a traversal of the resulting dependency graph (e.g., using DFS) is effected. Yang et al. [159] have modelled each rule by a predicate and an action. The variables of the predicate are packet header fields. The data plane is modelled as a directed graph where nodes represent switches and each directed edge represents the link from the output port of one switch to the input port of another. Each input port is guarded by a list of rules predicates and each output port is guarded by the predicate of the matching rule in that node followed by a list of rules predicates of the next node. Any packet can pass through if a predicate is True. Reachability in presence of an update has been checked by constructing a tree on the fly. The reachability tree from a given port to all other ports is computed by performing a DFS on the graph . Beckett et al. [16] have extended the work of Khurshid et al. [80, 81] to verify the network not only in the presence of updates but also if the verification conditions change during the verification process, that is, when the verification is suspended temporarily to add some changes and then is resumed. For example, the programmer may want to suspend the verification temporarily to add a set of hosts to the topology. Checking if the property holds in this case is performed using the same graph based verification solution of [80, 81] augmented with the traversal of a new data structure, namely a tree representation of regular expressions. In fact, the matching part of a rule as well as the property to be checked are modelled as regular expressions describing the possible paths that packets matching the condition may traverse. The regular expression formulae are then represented as a tree. A node represents a 'forall' quantifier in the formula whose children correspond to set elements, and whose result is the conjunction of the results of each of its children. A leaf node represents a concrete property that is checked by the solution proposed in [80, 81]. A truth value stored at each node represent the validity of the corresponding sub formula. Afterwards, the result is propagated up to the parent of the node.

To capture the dynamic of rules updates, the technique based on dependency graph traversal models an SDN network topology as a graph that is updated accordingly and the verification problem is reduced to the traversal of such graph. The technique has been applied to the data plane and can be used to check mainly reachability, loops and rules' consistency properties.

3.2.3 Summary and Conclusions about Verification Techniques

Fig. 3.3 summarizes the taxonomy of the different verification techniques used in the literature for SDN.

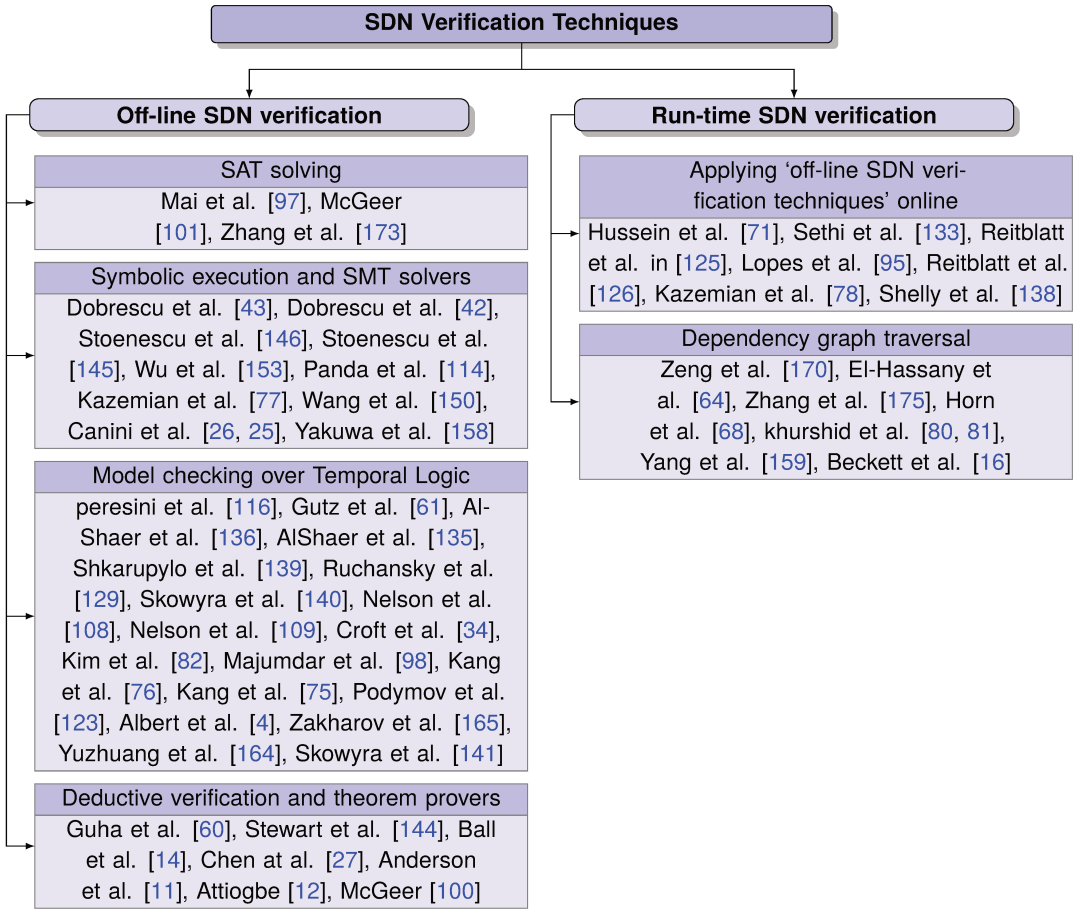


Figure 3.3 – SDN verification techniques taxonomy

Table 3.1 compares the verification techniques w.r.t. the properties they can check. All the techniques check reachability and loops.

Table. 3.2 compares the verification techniques w.r.t. what components these techniques have been applied to. The main observation is that the whole data plane is by far the most popular component to be verified. We also note that only model checking and symbolic execution have been applied to the whole architecture.

According to our analysis, some conclusions can be drawn. One can observe the large usage of formal verification techniques for checking the consistency/correctness of SDN architectures/components. In fact, verification techniques can ensure the respect of a given policy in the data plane and can help checking configuration errors and problematic controller programs in the control plane. Yet, there are two strong grounds of restrictions to them. First, as they check whether a *model* of the SDN architecture/component satisfies a given set of properties, they can only guarantee that the *properties* hold for the *model* and hence some implementation faults can still escape this check as no test cases are applied to the implementations. Secondly, they usually aim at verifying the model of a single SDN component and ignore the correctness of the entire SDN architecture, e.g., whether the architecture's implementation violates the network policies/requests. Therefore, testing techniques that stimulate

Technique	Reachability	Loops	Black-holes	Traffic isolation	Rules' consistency
Off-Line SDN Verification Techniques					
SAT Solving	✓	✓	✗	✗	✗
Symbolic Execution and SMT Solvers	✓	✓	✓	✗	✗
Model Checking over Temporal Logic	✓	✓	✓	✗	✗
Deductive Verification and Theorem Provers	✓	✓	✓	✗	✗
Run-Time SDN Verification Techniques					
Application of 'Off-line SDN Verification' Online	✓	✓	✗	✗	✓
Dependency Graph Traversal	✓	✓	✗	✗	✓

Table 3.1 – Comparison of SDN verification techniques w.r.t. checked properties

Verification Technique	Controller	Switch	Data plane	Whole SDN architecture
Off-Line SDN Verification Techniques				
SAT Solving	✗	✗	✓	✗
Symbolic Execution and SMT Solvers	✓	✗	✓	✓
Model checking over Temporal Logic	✓	✗	✓	✓
Deductive Verification and Theorem Provers	✓	✗	✓	✗
Run-Time SDN Verification Techniques				
Application of 'Off-line SDN Verification' Online	✗	✗	✓	✗
Dependency Graph Traversal	✗	✗	✓	✗

Table 3.2 – Comparison of verification techniques applied to various SDN components

the SDN components' implementation are necessary to explore.In the next section, we investigate and categorize testing techniques applied to SDN.

3.3 Testing Techniques for SDN

This section encompasses the literature techniques that involve the system under examination which is stimulated by inputs (note the contrast with the verification techniques presented earlier which do not require any stimulation of the system). We categorize these techniques depending on their underlying test generation mechanisms.

3.3.1 Log Analysis for Test Generation

This technique works around a buggy log and aims at automatically identifying/replaying the sequence of events (test sequence), that have generated an observed problem. This sequence

is then applied to the SUT, thus allowing to localize the cause of the problem. For example, Scott et al. [131, 130] have applied this technique to the control plane. Given a log L (of the controller) that contains a bug, the objective is to eliminate events from L that are not causally related to the bug. The result is a minimal sequence of events that when executed reproduces the bug. L is represented as a sequence of external (e.g., link failures) and internal (e.g., OF messages) events. The 'minimization' procedure contains two phases. Phase (a) consists of searching through subsequences of the logged external events of L. Phase (b) consists of deciding when to inject external events for each subsequence such that, during replay, an invariant violation is provoked again. Phase (a) is based on delta debugging algorithm [166], a divide-and-conquer algorithm which, from an input sequence of events, denoted as seq, iteratively divide seq in two. Then, taking into consideration the invariant that was violated by the execution of seq, if a subsequence successfully triggers the invariant violation, the other subsequences are ignored and the algorithm keeps refining that one until it finds a minimal one. Phase (b) is based on delaying event delivery to make sure that the replayed sequence of events obeys to the original 'happens-before' order. Note that this technique can create a shortened log without making assumptions about the language or instrumentation of the controller under test. However, it is not guaranteed to always find such minimal sequence due to partial visibility of internal events and non-determinism. Moreover, the minimization process is not always possible when considering extremely large traces.

Example

Figure 3.4 shows an example of the operation of the log analysis for test generation technique. A trace of events of a controller (POX) buggy module is shown in Listing 3.4a. We can see that a single 'SwitchFailure' event is enough to cause the controller to crash. The derived minimal causal sequence from the buggy trace is shown in Listing 3.4b, (only parts of the log and the derived sequence are shown). This sequence can then be used as a test case to stimulate a controller under test with the intention of finding bugs/errors.


```
...
{"class": "ControlMessageReceive", "dependent_labels": [], "dpid": 3, "timeout_disallowed": false, "time": [1359230214, 746195], "fingerprint": ["ControlMessageReceive", {"class": "
        ofp_features_request"}, 3, ["127.0.0.1", 6633]], "controller_id": ["127.0.0.1", 6633], "label": "i17"}
{"class": "ControlMessageReceive", "dependent_labels": [], "dpid": 4, "timeout_disallowed": false, "time": [1359230214, 746482], "fingerprint": ["ControlMessageReceive", {"class": "
        ofp_hello"}, 4, ["127.0.0.1", 6633]], "controller_id": ["127.0.0.1", 6633], "label": "i18"}
{"class": "ControlMessageReceive", "dependent_labels": [], "dpid": 2, "timeout_disallowed": false, "time": [1359230214, 746816], "fingerprint": ["ControlMessageReceive", {"class": "
        ofp_features_request"}, 2, ["127.0.0.1", 6633]], "controller_id": ["127.0.0.1", 6633], "label": "i19"}
{"class": "ControlMessageReceive", "dependent_labels": [], "dpid": 1, "timeout_disallowed": false, "time": [1359230214, 747156], "fingerprint": ["ControlMessageReceive", {"class": "
        ofp_features_request"}, 1, ["127.0.0.1", 6633]], "controller_id": ["127.0.0.1", 6633], "label": "i20"}
{"dependent_labels": [], "class": "SwitchFailure", "dpid": 4, "label": "e21", "time": [1359230214, 747398]}
{"class": "ControlMessageReceive", "dependent_labels": [], "dpid": 1, "timeout_disallowed": false, "time": [1359230214, 848838], "fingerprint": ["ControlMessageReceive", {"class": "
        ofp_barrier_request"}, 1, ["127.0.0.1", 6633]], "controller_id": ["127.0.0.1", 6633], "label": "i24"}
...
```


(a) Part of the original buggy log of POX controller


```
...
{"class": "ControlMessageReceive", "dependent_labels": [], "dpid": 1, "timeout_disallowed": false, "time": [1359230214, 747156], "fingerprint": ["ControlMessageReceive", {"class": "
        ofp_features_request"}, 1, ["127.0.0.1", 6633]], "controller_id": ["127.0.0.1", 6633], "label": "i20"}
{"dependent_labels": ["e89"], "class": "SwitchFailure", "dpid": 1, "label": "e16", "time": [1356996714, 125807]}
...
```


(b) Part of the derived test sequence

Figure 3.4 – Example showing a test case derived using the 'log analysis for test generation' technique

3.3.2 'Specific' Packets for Test Generation

In order to localize the network failures, this technique generates specific packets (e.g., with modified headers) and applies them against the SUT (data plane) in order to perform path tracing capable of tracking packet trajectories/histories in the SDN data plane.

Handigol et al. [62] have introduced packet histories. A packet history is (1) the route a packet takes through a network, (2) the switch modifications of the header at each hop. This tracking mechanism can help administrators to diagnose a network problem. The packet histories are obtained by means of Packet History Filters (PHFs), a regular-expression-like language created to process and filter postcards. Example of PHFs is to match packets that start at $Host_1$ or go through S_2 or experience a loop. Postcards are records created whenever a packet passes by a switch. A postcard is formed mainly by the switch id and the output ports.To generate a postcard, each packet entering the switch is duplicated and processed to generate packet histories.

A mechanism of tagging packets (e.g., with VLAN IDs) and installing temporarily rules to catch them to identify their paths has been proposed in several works. For example, this has been performed by Avanesov et al. [13] and Csikor et al. [35] to accelerate the identification of root causes of performance degradation in the data plane. Tseng et al. [147] have presented a similar mechanism as well. This mechanism allows to detect for example network loops and black-holes. Zhang et al. [172] have also used this mechanism. A 'path table' data structure has been employed to record forwarding paths between any two ports in the network. The approach first generates the path table based on the network topology and the installed rules. Then, when a test packet traverses through the network, each switch generates the tag of the packet.

Aljaedi et al. [6] have proposed to rewrite the packets' headers so to include ingress port numbers of the switches traversed and, thus, obtaining the followed path. Then probe packets have been injected in the data plane. Zhang et al. [171] have proposed to dedicate unused bits in a packet's header to carry a path identifier across the followed path. Agarwal et al. [2] have proposed an SDN 'traceroute' tool that reserves some bits of the packet headers exclusively for its use. It injects probe packets to identify network paths. Wang et al. [151] have proposed 'sTrace' that works in a similar way.

Other works have taken advantage of usual debugger actions such as breakpoint and backtrace. This is related for example in contributions by Handigol et al. [62] and Durairajan et al. [45]. The approach attempts to reconstruct the sequence of events origin of a faulty behaviour. A breakpoint is a filter defined on packet headers that applies to one or more switches. A backtrace is constructed using postcards explained above [62]. When a packet triggers a breakpoint along its path, a backtrace shows the sequence of forwarding actions seen by that packet.

Wu et al. [154] have proposed an approach that relies on the concept of provenance causality [24] (from the databases) to help find a causal explanation of a network problem. The provenance of a packet that has been derived via a certain rule R consists of the packet that triggered R and the rule R itself. The provenance is tracked based on a provenance graph representing the changes to the packets (using history of packets).

3.3.3 (Semi)-Random Test Generation

This approach stimulates the SUT by inputs that are randomly or semi-randomly generated. For example, the same inputs (OF messages randomly generated) are applied to several controllers and conclusions about their correct behaviour are drawn by establishing a consensus on their outputs (e.g., if all outputs from all the controller responses to the same input are the

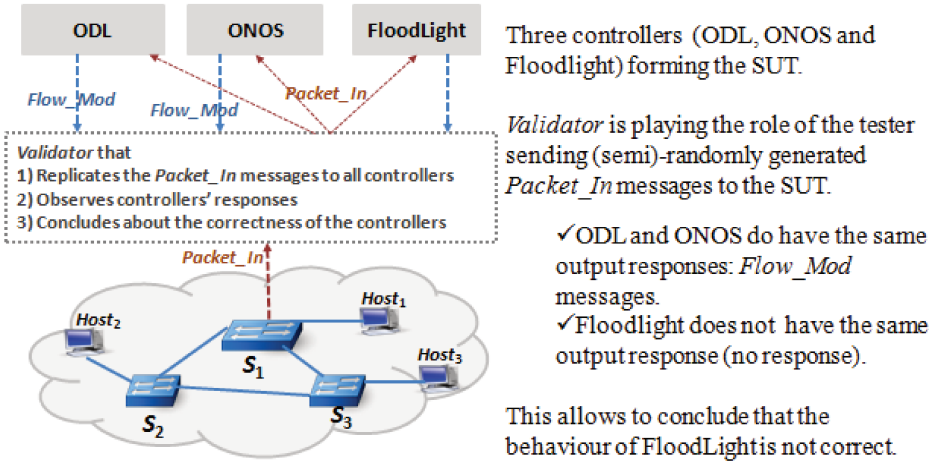


Figure 3.5 – Example illustrating the test execution of tests (semi)-randomly generated

same). This has been performed for example by Zhang et al. [174] and Mahajan et al. [96]. The approach is based on the full consensus on the messages received from different controllers by comparing their outputs' reactions to the same applied input. The approach consists in (i) intercepting the $Packet\_In$ messages, (ii) replicating them to a set of different controllers (iii) comparing their responses to reach consensus on every message. The test generation approach is reduced to replication of OF events/messages. This is achieved for example by the validator (that will serve as a tester stimulating the controllers with inputs) that intercepts the $Packet\_In$ messages coming from a switch and replicating these messages to the other controllers. If the outputs of all the controllers under test are the same, then their behaviour is concluded to be 'correct', otherwise the controllers with the different responses are considered to contain a bug. Similarly, Shalimov et al. [137] have performed the comparison of different SDN controllers w.r.t. the number of failures/faults during a long period of test. A failure/fault means that the controller sends a session termination or does not provide a reply to a given request within a timeout. A test bed is presented that contains a number of OF switches and hosts connected to a controller. The switches are playing the role of the tester here. The test sequences in this case are formed by $Packet\_In$ messages sent by five switches.A similar approach has been proposed by Lin et al. [93] where $Packet\_In$ traffic has been randomly generated to test the controller under test.

(Semi-) random generation has also been employed for testing the switch as the SUT, in works by for example Rotsos et al. [128], Kuzniar et al. [88, 86], and Wundsam et al. [155]. In all cases, a series of experiments on some of the OF switch implementations (including OVS and hardware switches) has been conducted. In the work of Rotsos et al., test packet generation has been assured by the 'pktgen' (the linux packet generator module [112]), and an extension of the design of the NetFPGA Stanford Packet Generator [33]. Kuzniar et al. [86, 88] have built upon the work of Canini et al. [26, 25] and semi-randomly generated packets, i.e., random packets but in a controlled order) have been applied to the SUT. Wundsam et al. [155] have proposed to record and apply a subset of randomly generated packets to the data plane. The traffic is recorded, partitioned and then different subsets of traffic are applied many times to the SUT to reproduce the failure.

Example

Figure 3.5 illustrates an example of test execution and decision making based on the full consensus on the responses received from the controllers.

3.3.4 Verification for Test Generation

This technique involves the use of the verification techniques described earlier in Section 3.2 to serve for deriving the test cases that are used to stimulate the SUT. We group works proposing such a testing technique depending on the underlying verification method.

In the first group of works, SAT solving as the underlying verification technique is used to generate probe packets in order to check data plane rules and detect forwarding faults. It is assumed that a faulty rule does not act on some or all packets that it should process. The cause could be either 'rule missing' faults or 'rule priority' faults. The generation of test packets is done as follows. Based on a set of rules expressed as Boolean expressions, a set of constraints (Boolean expressions over packet header fields) that a probe packet should satisfy is created. Then a SAT solver is used to find a satisfying assignment of the packet header fields (the constraints). These packets are then converted into 'real' probe packets using packet generation libraries. To detect faults related to missing rules, Perevsini, kuzniar et al. [117, 87] have used the aforesaid technique. Further, Bu et al. have observed that without testing a rule priority order, i.e., testing only rule existence, cannot guarantee forwarding correctness. Similarly, their technique [23] has reduced the generation of probe packets problem into a SAT problem taking into account the priorities of the rules.

The second group of works relies on symbolic execution as the underlying verification method to generate test cases. This has been performed considering the switch in its OF interface and the switch in its data plane interface as SUTs. Kuzniar et al. [89] have considered the switch-to-controller communication as the SUT. The objective is to find whether there exists any sequence of inputs (OF messages) under which one switch implementation behaves differently than another switch implementation which is considered as the 'reference' or 'golden' one. To this end, (1) sequences of inputs that cover all possible executions for each implementation are constructed. Then, (2) an intersection of input subspaces belonging to different implementations is computed to finally find the common input sequence that causes the different implementations to produce different outputs (inconsistencies). Phase (1) relies on symbolic execution technique that is run on each implementation and the outcomes of such execution include a list of path conditions each of which summarizes the input constraints (over fields of OF messages) that must hold during the execution of a given path. For each implementation, the path conditions that share the same output results are grouped and a constraint solver is used to determine the input subspaces that satisfy the conjunction of the grouped path conditions. The solver then computes an intersection of input subspaces belonging to different implementations hence providing inputs that cause different behaviours of different implementations.

When considering the switch in its data plane interface as the SUT, symbolic execution has been used to generate test packets based on a model of the rules. A symbolic packet (symbolic variables represent the packet fields) is injected at the desired source port, and this packet is then propagated through the (rules) model. The output is a series of paths, where each path places a number of constraints on the header fields of the injected packet. At each port reached by the symbolic packet, the values and constraints on the header variables can be inspected to discover which packets are allowed, what input packets can reach the output, and how the packets look like at the output, on all the execution paths that reach that port. A solver is then used to solve the constraints at each path and generate concrete values for all the header fields, resulting in a concrete packet which is then injected into the network and outputs are then observed. This technique has been used by Zeng et al. [169, 168] where an Automatic Test Packet Generation (ATPG) framework has been proposed. The method has also been used by Stoenescu et al. [146, 145] and Fayaz et al. [53, 52].

3.3.5 Model Based Testing

This technique relies on a formal model built to support the test generation activity. As *models* play a crucial role in this technique, a major testing challenge moves to the modelling the SUT itself [119]. According to whether the adopted model is a *finite transition system* or a *tree*, we partition the state of the art efforts using this technique into two groups. We further partition the works in the first group according to whether it is about a single finite state model or a composition of those.

The first subgroup uses a single model, for example applied to the data plane (a switch or a set of those in its data plane interface), Lebrun et al. [91] have formalized each rule in the network and each network administrator/operator request as an automaton. The test generation relies on mapping each formalized rule to test packets that match the conditions expressed by the body of the rule. The resulting test packets are then injected into the data plane, the path followed by each test packet is tracked. Then depending on whether the string representing the data paths followed by each test packet is accepted or not by the corresponding automaton, conclusion about whether the request is satisfied in the data plane is drawn. Alsmadi et al. [7] have modelled one module of the controller behaviour, namely an SDN firewall, with function nets (simplified high-level Petri nets), a modelling formalism in the MISTA tool [157]. The latter tool has been used to generate test cases from the model.

The second subgroup models the SDN component(s) as a composition of finite state models. For example, Fayaz et al. [51] have modelled the switches as a transition based system where each state represents a state of the switch, then a model has been established as logical preconditions that carry over from one switch model to the next. For example, if the policy is to ensure that switch S_1 processes the traffic before switch S_2 then the 'traffic being processed by S_1' is a precondition to enter the state model of S_2. Test sequences generation can be formulated as a problem of identifying a sequence of transitions that causes the set of network switches to transition from their current states to a desired goal state. Yao et al. [160] have modelled the pipeline processing in a switch as a pipeline of a finite set of state transition systems (machines) grouped in different 'stages' that communicate between each other through a finite set of channels, a finite set of shared variables is defined at the level of the global model and not in the machines. Global initial and final states are defined for the entire model. In such model, a table in the OF switch is mapped to a 'stage' and each flow entry of that table is mapped to a different state machine. The test generation technique is based on three phases as follows. From the model, a high level abstraction description represented by a graph is extracted. Then the test sequences are formed by finding and linking paths inside the component machines starting from the global initial state and ending in the global final state. The final phase of test generation consists of turning the test sequences into Testing and Test Control Notation version 3 (TTCN-3) to be finally executed on the switch implementations under test. Similar to [160], Zhang et al. [176] have modelled the pipeline processing inside the switch as well, they have used also a state based model with variables. They have stated that the test generation is based on Dijkstra algorithm but without giving further details. For testing SDN applications (e.g., MAC-Learning), Yao et al. [161, 162] have applied their previous work [160]. They have described the behaviour of the applications by the combination of a number of state based machines and an abstract topology. First, model checking has been used to verify the model against design faults. Second a test generation approach from the model has been proposed based on partial composition and symmetry simplification.

The second group models the behaviour of an SDN component(s) as a *tree based structure*. For example, Zhao et al. [178, 177] have modelled the set of rules inside each switch as a multi-rooted tree where each level of the tree represents a table in the switch. Since all the packets entering the switch must be matched against the first flow table then the root nodes

of the tree represent all the rules in the first flow table. Second, flow tables are sequentially numbered, and the pipeline processing can only go forward and not backward, therefore it is a directed acyclic graph. Third, the GOTO_TABLE instruction of a given rule directs packets to another table with a larger sequence number, therefore nodes are unidirectionally connected by that kind of instructions. It is assumed that rules can be sequentially numbered and each rule is mapped to one vertex and one edge set. Given this tree model, the probe generation problem is reduced to finding the minimum vertex sets to cover all the rules in the tree and an algorithm based on DFS is developed to this end. These inputs are used to stimulate four switch implementations (Open VSwitches residing in two Pica8 P5101 switches) and are capable of detecting bugs such as black-holes.

3.3.6 Summary and Conclusions about Testing Techniques

Figure 3.6 summarizes the taxonomy of the different techniques used in the literature and categorized under *Testing* umbrella. Table 3.3 compares the testing techniques w.r.t. what components these techniques have been applied to. The main observation is that the whole data plane is by far the most popular component that has been tested.

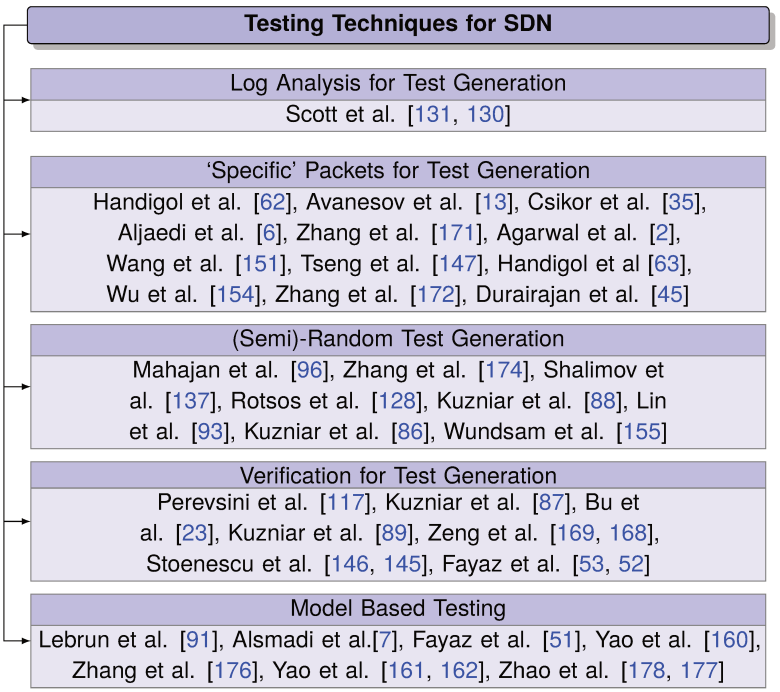


Figure 3.6 – SDN testing techniques taxonomy

Testing Technique	Controller	Switch	Data plane	Whole SDN architecture
Log Analysis for Test Generation	✗	✗	✓	✗
'Specific' Packets Generation	✗	✗	✓	✗
'Semi' Random Testing	✓	✓	✓	✗
Verification for Test Generation	✗	✓	✓	✗
Model Based Testing	✗	✓	✗	✗

Table 3.3 – Comparison of testing techniques applied to various SDN components

Where testing in SDN environments is concerned, researchers have investigated several strategies for generating tests. In particular, the following test generation strategies have been studied: analysis of the logs, special modification of packets, (semi-) random generation, verification, and model based testing. We notice however that model based techniques have not been widely applied to SDN, in particular when a switch - controller or data plane - controller composition represents the system under test.

3.4 Chapter Conclusions

In this chapter related work on SDN verification and testing has been introduced and its analysis has been performed. At first, a taxonomy has been elaborated, dividing the body of work into two groups, namely *verification techniques* and *testing techniques*. Further, the first group has been divided into *off-line* and *run-time* subgroups. The second group has been further divided into five subgroups according to the underlying test generation method, namely, *log analysis*, *'specific' packet*, *(semi-)random*, *verification* and *model based*. Additionally, for each group, a summary pointing out the corresponding limitations has been provided. For additional research on related work, the reader has been suggested to refer to the surveys given in the introduction of this chapter.

Our state of the art analysis shows that in the last decade, SDN verification techniques, both off-line and at run-time, have been largely elaborated and applied for guaranteeing the consistency/correctness of SDN components. Existing verification approaches mostly concern the consistency of the configurations and the validation of the OpenFlow rules. However, they are in two points limited when applied to SDN. First, as they do not target bugs/errors in the (different) implementations; and secondly, as they usually aim at verifying a single SDN component and ignore the correctness of the entire SDN architecture, e.g., whether the SDN implementation violates the network operator policies. Moreover, these verification methods face the challenge of complexity and scalability when applied to complex architectures such as SDN. Further, these techniques perform their analysis on a model of the SDN system and do not stimulate the implementations which limit their capacity to detect some subtle bugs.

Furthermore, the research challenges in SDN testing are still not appropriately tackled because they do neither cover all functional aspects of the SDN components nor study the functional behaviour of the compositions, such as switch - controller, data plane - controller, etc.. Indeed, our analysis shows that existing testing techniques have not been widely studied and SDN testing in general is still a challenging hot research subject that needs more elaborations.

This motivates our first contribution introduced in Chapter 4 to provide novel formal testing solutions to comprehensively test the SDN architecture as a whole.

In addition, the idea of checking the implementations of SDN components (e.g., SDN-enabled switch, controller) and finding some inconsistencies by applying conformance testing is barely explored in the literature. For example, for testing the switch, the few research work that focus on this area (mainly [160] and [176]), consider the pipeline processing inside the switch as the system under test and do not consider the switch-to-controller link. Moreover, model based testing has not been applied to the controller.

This sets another gap in the picture and thus highlights our next contributions (Chapters 5, 6 and 7).

4

Model Based Testing for SDN Architectures: A Graph / Path Enumeration based Approach

"The true subject matter of the tester is not testing, but the design of test cases."
– Paul Ammann, Jeff Offutt

Contents

The previous chapter shows that the current state of knowledge in the research field of SDN verification and testing is rich but focused primarily on formal verification and testing based on either (semi-) random test generation or testing techniques based on verification.

Further, prior research on SDN testing and particularly model based testing has resulted in techniques that either model very specific SDN components or their composition (e.g., data plane). These approaches, however, have not tested the entire SDN architecture. Even if SDN components are well tested in isolation, their composition can still face interoperability issues. Therefore, there is a need of methods and techniques for checking the entire SDN architecture functionality.

This chapter provides the first contribution of the thesis and shows how such contribution covers a significantly less explored area in SDN testing research, namely model based testing.

4.1 Introduction

It is crucial to have reliable network architectures, and this requirement does not change with SDN. With the introduction of greater programmability, the chances of software faults (or bugs) increase.

The first contribution of the thesis proposes a novel model based testing technique based on *graph enumeration*. The technique is aimed at testing functional aspects of SDN architectures (we do not consider non-functional SDN issues, such as security, trust, etc.). We focus on so called *active testing* when a system under test (SUT) is stimulated by appropriate inputs, i.e., test sequences / cases, and the conclusion about its correctness is made based on observations of its related outputs.

In particular, we define a *fault model* where the *fault domain* contains potential implementations of virtual paths (representing requests) requested by a user, i.e., the wrongly and correctly implemented paths allowed with respect to the underlying resource connectivity graph. To guarantee the *fault coverage*, we prove the conditions when under black box and white box testing assumptions a *complete* test suite with respect to such fault model can be derived.

The problem statement is described in Section 4.2 of this chapter. We present a formal model for an SDN architecture as a whole in Section 4.3, and follow it with a description of the method used for traffic generation and observation in Section 4.4. Section 4.5 introduces an appropriate fault model. Section 4.6 describes in detail the black and white box approaches adopted in this work. Finally, the experimental evaluation of the proposed approach is provided in Section 4.7.

4.2 Problem Statement

A critical challenge in SDN architectures is to ensure the consistency between high level network requirements' definitions (paths) and low level configurations' implementations. In other words, how would a network administrator/operator know the network requirements and policies defined at the control plane are correctly implemented by network devices in the data plane infrastructure.

Given the SDN architecture as the system under test, i.e., the SDN controller translating end-user requests into flow rules and the SDN switches and hosts implementing these flow rules in the data plane, we are interested in addressing what inputs should be applied to the controller and what to observe in the data plane level so that we can draw conclusions about the correctness, i.e., whether the SDN architecture is functioning as expected/desired.

Figure 4.1 shows the system under test we are interested in. The figure illustrates how in the applications layer, end-users define their policies / requests in form of paths. Suppose a network policy specifying that traffic from the source host $Host_1$ to the destination host $Host_6$ should pass through the switches S_1, S_2, S_3, S_5, S_6 and S_7. The requested policies are then fed as inputs to the SUT. In the data plane level, Figure 4.1 portrays two paths (from $Host_1$ to $Host_6$) implementing the same aforementioned request. In this example, the path depicted in dotted line and highlighted in green is the 'correctly' implemented one. The path highlighted in solid pink is the 'wrongly' implemented one.

The research question being addressed in this chapter can then be formulated as follows. *How to ensure that the implemented requests in the data plane conform to the defined/expected requests?*

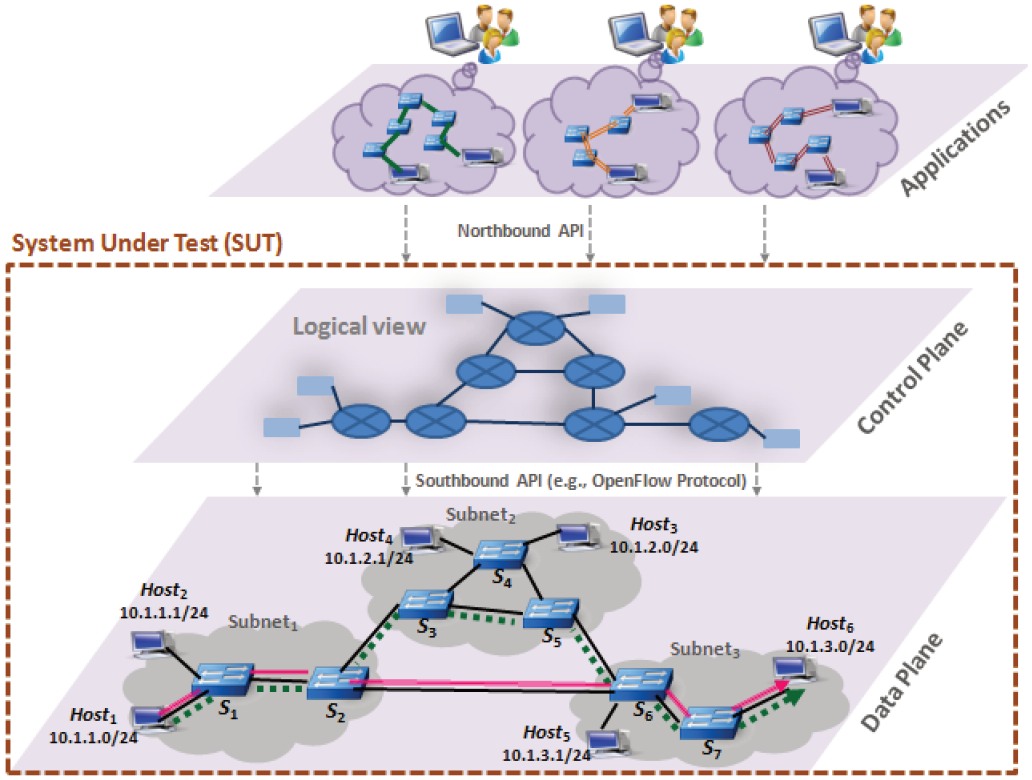


Figure 4.1 – Topology showing an SDN architecture as the SUT

The architecture shows an example of *requested* Vs *implemented* paths in the data plane layer

4.3 Formal Modelling for an SDN Architecture

To tackle the aforementioned problematic and assure the functional correctness of SDN architectures, the major contribution of this chapter is a model based testing technique for testing such complex and composite systems with respect to end-user defined requirements.

The technique relies on appropriate *graph / path enumeration*. It proposes two different approaches for generating test suites for the architecture under test, namely a black box and a white box approaches. Moreover, to ensure the fault coverage of the derived test suites, a corresponding *fault model* is proposed.

The *observed outputs* are generated upon applying a corresponding test suite to the SUT and (automatic) traffic generation allows to make the necessary observations, such as correct or wrong configurations, correct or wrong flow tables, etc..

The topological structure of the SDN architecture represents the network components such as controllers, switches and hosts while the traffic flows are generated by the dynamic features, namely the packets. To capture this structure, our approach represents the data plane as the *Resource Network Connectivity Topology* ($RNCT$), depicting the SDN components in the resource network. Definition 4.1 formalizes this notion.

Definition 4.1.

An $RNCT$ is represented by an undirected (network links are assumed to be bidirectional) and k-colored graph $G = (V, E, c)$, where

- V is the set of nodes representing network components (switches, hosts, etc.);
- E is the set of edges of the graph representing connections between two nodes (links) in the $RNCT$. Edges are unordered pairs $(x, y) \mid x, y \in V$;
- c is a coloring function $c : V \mapsto \mathbb{N} \cup \{0\}$ such that given a node in the network, a corresponding color is assigned to it as a hashed integer. Note that the colors of adjacent nodes can be the same, differently from the common graph coloring functions.

Every node a of the graph G (can be a host or a switch) has a set of ports which can be input as well as output and each such port corresponds to some edge at the node a and vice versa, each edge at the node a is associated with a corresponding port. Therefore, there is one-to-one correspondence between edges at the node a and the set of its ports. Since G has neither multiple edges nor self loops there is one-to-one correspondence between the set of ports of a and the set of neighbour nodes of a.

Example

As an example, the previously depicted model ($RNCT$) can accurately represent the data plane shown in Figure 4.1 by the binary-colored graph depicted in Equation 4.1.

$$RNCT = (V, E, c), where:$$

$$V = \{S_1, S_2, S_3, S_4, S_5, S_6, S_7, Host_1, Host_2, Host_3, Host_4, Host_5, Host_6\}$$

$$E = \left\{ \begin{array}{l} (Host_1, S_1), (Host_2, S_1), (Host_3, S_4), (Host_4, S_4), (Host_5, S_6), \\ (Host_6, S_7), (S_1, S_2), (S_2, S_3), (S_2, S_6), (S_3, S_4), (S_3, S_5), (S_4, S_5), \\ (S_5, S_6), (S_6, S_7) \end{array} \right\} \tag{4.1}$$

$$c(v) = \begin{cases} 1, & if\ v = S_1 \vee v = S_2 \vee v = S_3 \vee v = S_4 \vee v = S_5 \vee v = S_6 \vee v = S_7 \\ 0, & otherwise \end{cases}$$

Note that in the above example, '1' represents a *switch color*, and '0' represents a *host color*, correspondingly.

In particular, Figure 4.1 presents an example of a network topology consisting of one controller, seven switches and six hosts. Each switch is connected to the SDN controller. The switch S_1 is connected to the hosts $Host_1$ and $Host_2$, the switch S_4 is connected to the hosts $Host_3$ and $Host_4$, the switch S_6 is connected to the host $Host_5$, and the switch S_7 is connected to the host $Host_6$.

We model the end-user requests by *virtual paths* in the $RNCT$. Definition 4.2 specifies the notion of a virtual path.

Definition 4.2.
A *virtual path* (simply a path) in an $RNCT$ G is a sequence of directed edges whose head and tail nodes are hosts and all other intermediary nodes are switches.

Issuing a forwarding rule to a switch creates a *virtual link* from and to other component(-s) adjacent to the switch if the rule forwards traffic to a given port. For example, assume that for switch S_1 shown in Figure 4.1, $Host_1$ is connected to port 1 of S_1 and S_2 is connected to port 2 of S_1. The rule R_1 shown in Equation 4.2 issued by the controller and installed in S_1 creates a virtual link $Host_1 \rightarrow S_1 \rightarrow S_2$ if the destination MAC address is equal to 01:80:c2:00:00:00.

$$\begin{aligned} R_1 : table = 0, priority = 99, in\_port = 1, \\ dl\_dst = 01:80:c2:00:00:00, actions : output = 1 \end{aligned} \tag{4.2}$$

More formally, the application of a forwarding rule creates a virtual link $e \in E^*$ as a sequence of *directed* edges from the $RNCT$ edges.

The reason for the specific Definition 4.2 for paths is that for testing purposes, observing traffic generated from one host to another is how the resulting configuration is collected as an 'output', we develop this part in the next subsection. We assume the hosts in the $RNCT$ do not act as switches or relays of network packets, furthermore, we assume switches do not act as hosts in the $RNCT$. Note that at a physical level, the previous cases are possible, however, the $RNCT$ model must not consider such possibilities. The forwarding rules used to control the traffic in the $RNCT$ construct a virtual partial path (link) or a set of those.

We note that edges cannot be duplicated in a path, otherwise, infinite loops can be potentially formed. Furthermore, in this chapter we consider that nodes cannot be duplicated in a path as well, studying this possibility is left for future work.

4.4 Traffic Generation and Observation

Checking the output reaction of the SDN architecture can be performed through a network traffic initiation.

As we aim to check that the paths are implemented correctly in the data plane, we focus on specific traffic generation. For generating traffic, we propose to use the Internet Control Message Protocol (ICMP) echo request / echo reply packets through the known ping utility.

Ping is one of the most common tools used as an administrator utility which can identify if a targeted host in the data plane is reachable or not. Ping operates by generating and sending ICMP packets, or echo requests, to the destination host and wait for an ICMP response, or an echo reply. The experiments using this mechanism are provided in Section 4.7.

We however note the existence of different approaches for automatic traffic generation (see, for example [169, 91, 52]).

The ICMP request / reply is performed for each pair of hosts that correspond to the *head* and *tail* nodes of the paths. Later, the passing traffic is inspected at all node interfaces via a simple network sniffer. The network traffic of all switches can be obtained in different ways, starting from a simple Unix-like sniffer as the tcpdump utility and finishing with non-software-based (physical / vendor) switches via protocols such as NetFlow [29] or sFlow [122].

Example

As an example, consider the requested path in Equation 4.3 with respect to the $RNCT$ in Figure. 4.1 (path highlighted in green in the figure).

$$(Host_1, S_1)\,(S_1, S_2)\,(S_2, S_3)\,(S_3, S_5)\,(S_5, S_6)\,(S_6, S_7)\,(S_7, Host_6) \tag{4.3}$$

The corresponding traffic generation (through ICMP echo request) and the traffic observation are illustrated in Figure 4.2. We depict the messages and a timestamp when the message is observed. In Figure 4.2, 't1' denotes the first time instance after traffic generation started. If the network flow follows the requested path, i.e., the expected output response is observed during the traffic generation, then we consider that the applied test case has *passed*, otherwise the test case has *failed*.

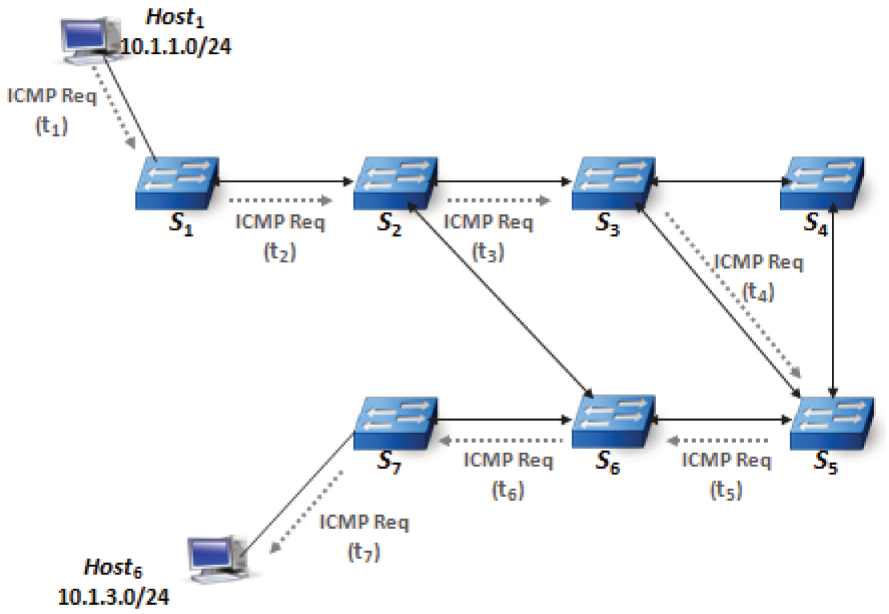


Figure 4.2 – Traffic generation and flow observation w.r.t. the $RNCT$ of Figure 4.1

4.5 Introducing a Fault Model

Given the SDN architecture in Figure. 4.1, we propose to test the entire SDN architecture, including the controller(-s), switches, and connections between them. Namely, we propose to derive an application that 'is responsible' for test generation and execution. In other words, a tester is foreseen that sends specific requests to the SDN controller asking for different paths to be implemented in the $RNCT$.

The test generation architecture is illustrated in Figure. 4.4, where the application layer is executing only the tester. According to our assumptions, the inputs that need to be generated

by the orchestrator in order to guarantee that the SDN architecture is functioning properly are paths limited by the $RNCT$.

We assume that the SDN architecture is *functioning correctly* when each requested path and only it is created. In fact, mostly connectivity issues are tested and we consider the specification, in this study, defined by a set of (end-user) requirements that should be correctly implemented. In this case, we propose a fault model (similar to [50]) represented as a pair $\langle @, \mathcal{FD} \rangle$ where $@$ is a conformance relation (between what is requested and what is really implemented). We define $@$ as the equality. $\mathcal{FD}$ is the fault domain defined as a set of potential implementations.

Another issue is about the fault domain $\mathcal{FD}$ of the fault model. According to Definition 4.2, the following types of faults can be considered:

1. A requested edge can be directed to a wrong node;
2. Additional edges can appear;
3. Some edges can disappear.

Thus, a fault domain $\mathcal{FD}$ contains all possible paths of the $RNCT$.

Definition 4.3.
A *test case* is a path of the $RNCT$ and a *test suite* is a finite set of paths. As usual, a test suite is *complete* with respect to the fault model $\langle =, \mathcal{FD} \rangle$ if any difference between a requested and implemented path can be detected. In other words, each correct implementation $I_1 \in FD$ passes a *complete* test suite while each faulty implementation $I_2 \in FD$ (with respect to the equality relationship) fails such a test suite.

4.6 Black Box and White Box Testing Approaches relying on Path Enumeration

After the fault model is defined, usual testing approaches can be used for deriving test suites with the *guaranteed fault coverage*. Below, we discuss how 'black' and 'white' box test derivation approaches can be employed for this purpose.

4.6.1 Black Box Testing Approach

As the set of all paths of the $RCNT$ is finite, the simplest way to construct a complete test suite with respect to the fault model $\langle =, \mathcal{FD} \rangle$ is to consider the set of all such paths.

Proposition 4.4.
The set of all $RCNT$ paths is a *complete* test suite with respect to the fault model $\langle =, \mathcal{FD} \rangle$.

In general, this test suite is rather long especially when the number of switches and hosts in the data plane is large, therefore we propose an approach for reducing its length based on *equivalence classes* of paths.

4.6.1.1 Equivalence Classes of Paths

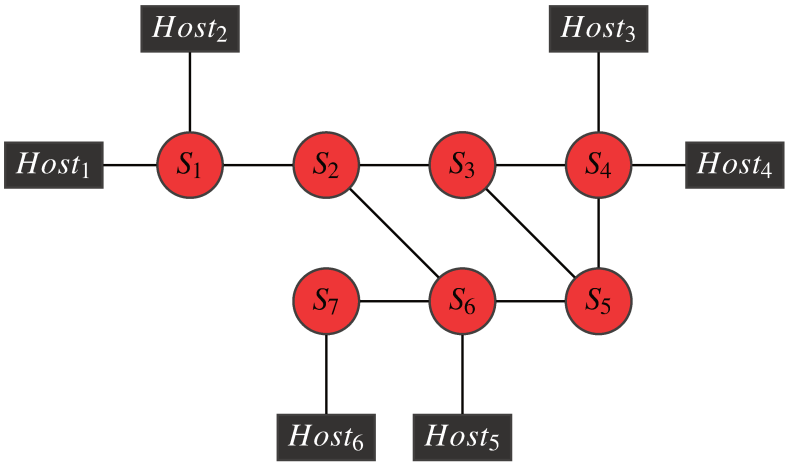
We make the assumption that each node in the network processes inputs independently of the previous node, i.e., the node where packets come from. Definition 4.5 specifies when two paths are *equivalent*.

Definition 4.5.
Two paths are considered (i, j)-*equivalent* if both paths have a directed edge from node i to node j. That is, all the packets that should be directed from i to j are either processed correctly, i.e., are sent from node i to node j, or are processed wrongly, i.e., are sent anywhere except to the j-th node.

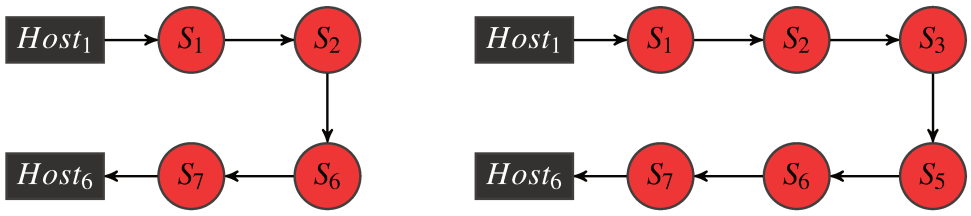
Example

In Figure 4.3, the paths $P1$ and $P2$ shown in Equation 4.4 are considered to be *equivalent* with respect to the edge $(Host_1, S_1)$ as well as with respect to the edge $(S_7, Host_6)$.

$$\begin{aligned} P1 &= (Host_1, S_1)\,(S_1, S_2)\,(S_2, S_6)\,(S_6, S_7)\,(S_7, Host_6) \\ P2 &= (Host_1, S_1)\,(S_1, S_2)\,(S_2, S_3)\,(S_3, S_5)\,(S_5, S_6)\,(S_6, S_7)\,(S_7, Host_6) \end{aligned} \tag{4.4}$$



$RNCT$



$P1$ $P2$

Figure 4.3 – $RNCT$ of the network topology in Figure 4.1 and examples of two *equivalent* paths

Proposition 4.6.
The set of paths that contains a path of each (i, j)-*equivalent* class where (i, j) is an edge in the $RCNT$, is a *complete* test suite with respect to the fault model $\langle =, \mathcal{FD} \rangle$.

Proof.
Indeed, a complete test suite has at least one request where a packet should be sent from

node i to node j. If the packet is processed correctly (according to the monitoring results), then due to the testing assumptions, we conclude that each packet directed from node i to node j will indeed be sent to the j-th node. □

Example

By direct inspection, one can confirm that the number of all paths for the $RNCT$ example in Figure 4.3 equals 90. However, the proposed *equivalence* classes approach allows to reduce this test suite down to 30 paths only.

In order to cover all *equivalence* classes in an optimal way, an optimization problem should be stated and solved. One option is to consider the Boolean (weighted) matrix and solve the corresponding covering problem [148] for which many libraries and scalable software solutions are developed.

If the node processes a packet depending on where it comes from then *equivalence* classes could be considered with respect to path subsequences of length $l \geq 2$. Given a sequence γ of $RNCT$ edges of length l between node i and node j, two paths are considered γ-*equivalent* if they both contain γ. A test suite is *complete* if it has at least one path of each *equivalence* class. A minimal cover of a corresponding Boolean matrix can also be used for optimal test generation. However, such test suite minimization problem is left for future work.

4.6.2 White Box Testing Approach

In some cases, mainly for reducing testing complexity, it may be desired not to generate test suites with respect to all possible edges in the $RNCT$. The complexity can be reduced if a set of *critical edges* that need to be tested first can be defined; for example, critical edges that include critical network services/requests.

For this reason, we propose Algorithm 1 that generates a test suite with guaranteed fault coverage with respect to a set of critical edges, i.e., if a fault occurs at a given critical edge, it is detected. The algorithm is based on generating a test case tc as shown in Equation 4.5 that traverses a critical edge (v_i, v_j) for all critical edges $E' \subseteq E$.

$$tc = (v_1, v_2)\ldots(v_i, v_j)\,(v_j, v_{j+1})\ldots(v_{n-1}, v_n) \tag{4.5}$$

We consider in the fault domain $\mathcal{FD}$ implementations that can potentially contain three types of faults that need to be detected, namely

1. An edge is directed to a wrong node, i.e., from the edge of interest $e = (v_i, v_j)$ to $e' = (v_i, v_{j'})$ where $j \neq j'$.

2. An edge $e = (v_i, v_j)$ is deleted.

3. A non-existing edge $e = (v_i, v_k)$ is created for a critical edge (v_i, v_j) where $j \neq k$.

As potential faults are enumerated explicitly, Algorithm 1 returns the test suite under the white box testing assumption.

By construction, Proposition 4.7 holds.

Proposition 4.7.
Algorithm 1 returns a *complete* test suite with respect to the fault model $\langle =, \mathcal{FD} \rangle$ where $\mathcal{FD}$ has each path with a critical edge.

Algorithm 1: White Box Test Suite Generation for an SDN Architecture

Input : $RNCT = (V, E, c)$, a binary-colored graph where hosts are "0" colored.
$E' \subseteq E$, a set of *critical* edges.

Output : A complete test suite TS_w w.r.t. the explicitly enumerated edges of interest.

1 $TS_w \leftarrow \varnothing$

2 **foreach** $f = (v_i, v_j) \in E'$ **do**

3 Find (backtrack) $p_b = (v_1, v_2) \ldots (v_i, v_j)$, the shortest sequence of edges such that $c(v_1) = 0$, i.e., the shortest sequence that starts in a host and finishes at the node v_j.

▷ Note that if $c(v_i) = 0$ then $p_b = (v_i, v_j)$

4 Find (forwardtrack) $p_f = (v_i, v_j) \ldots (v_{n-1}, v_n)$, the shortest sequence of edges such that $c(v_n) = 0$, i.e., the shortest sequence that finishes in a host and starts at node v_j.

▷ Note that if $c(v_j) = 0$ then $p_f = (v_i, v_j)$

5 $TS_w \leftarrow TS_w \cup \{p_b p_f\}$

6 **return** TS_w

Despite its conceptual simplicity, Algorithm 1 is successful at deriving a *complete* test suite with respect to the defined fault model which has each path with a critical edge.

Its underlying principle is to construct a test case $tc \in TS_w$ in the following way. For each critical edge (v_i, v_j) in the $RNCT$, firstly, it uses *backtracking* to find the shortest sequence of edges starting in a given host and finishing at the node v_j (line 3). This is performed in an incremental fashion where candidate edges are first appended to the potential tc solution, one edge at a time, then those edges that fail to satisfy the constraints of the problem (i.e., starting node is a host, ending node is the node v_j) at any point of time are removed and the search continues until a shortest sequence is found. Secondly, it uses *forward-tracking* to find the shortest sequence of edges starting at the node v_j and finishing at a given host (line 4). This is performed in a similar way as backtracking but using inverted constraints.

Note that a test minimization can be performed similar to the black box testing approach, via solving a covering problem. In this case, a minimal set of paths that cover all critical edges can be identified. We intend to investigate this direction in future work as well.

4.7 Experimental Evaluation for Testing SDN Architectures

To evaluate and demonstrate the effectiveness of our proposed approach, we aim to perform experiments showing how it is possible to certify that a given SDN architecture is functioning correctly.

The Mininet [105] SDN emulator, is chosen to emulate real-world SDN network environments for our evaluation and proof of concept experimentation. Indeed, Mininet is usually utilized to emulate SDN architectures, and represents an accurate means to mimic networks effectively [70]. Mininet provides an easy way to emulate and prototype SDN networks using Open vSwitch (OVS) [121] switches.

We chose Open Network Operating System (ONOS) [17] and OpenDaylight (ODL) [104] controllers. Both controllers are open source. They manage network traffic by handling and processing network events through different APIs. They also offer developers the possibility to implement their network applications and to run them on the controller. Both also offer to SDN applications many services that handle and process network events, parse packets or interact with the network switches in the data plane through southbound API such as OpenFlow.

4.7.1 Experimental Set Up

Experiments are carried out in a virtualized environment as portrayed in Figure 4.4. The environment consists of two virtual machines. The first machine (virtual machine #1) is emulating the network data plane by nodes which run software switches, in our setup Open vSwitch version 2.0.2 is used. Here, the topology consists of the different nodes tied (stitched) together and with the SDN controller.

The emulated SDN network is connected to a CentOS 7 virtual machine (#2) with 8 cores and 12GB of RAM running alternatively an instance of one of the real controllers used here. In fact, to prove the validity of our approach, real SDN controllers were utilized for the experiments, especially an ODL Boron-S3, and an ONOS version 1.10.4.

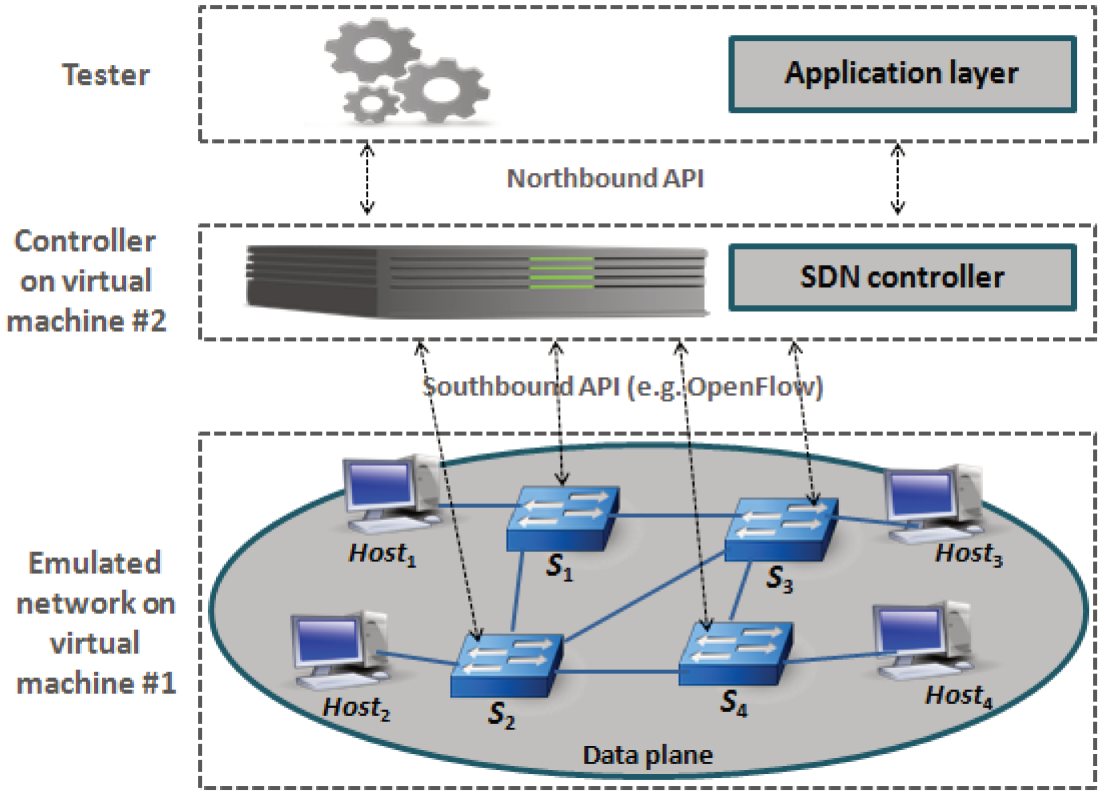


Figure 4.4 – Testbed framework for an SDN architecture analysis

The machine used for the experimental evaluation is an Ubuntu 14.04 LTS virtual machine running on VirtualBox version 5.1.14 r112924 (Qt5.6.2) for Mac OS X, with 2GB of RAM and 1 core of a 2.3 GHz Intel Core i5. The emulated networking environment in Mininet is used here to provide a well controlled but realistic testbed.

As depicted in Figure 4.4, four OpenFlow switches steer network traffic between various hosts.

The tester (run in the application layer) sends specific requests to the SDN controller asking for different paths to be implemented in the $RNCT$.

To be able to observe output reactions, traffic generation and observation is based on the ICMP echo request / echo reply packets through the ping utility as explained in Section 4.4.

4.7.2 Results and Evaluation

4.7.2.1 Results

Considering Proposition 4.6, we have derived a test suite TS for (i,j)-equivalence classes, including each edge (i,j) at least once.

When the test suite has been executed against the SDN architecture composed by the experimental set up and the ODL controller, all tests have failed. The test suite has been executed by requesting the proper flow instantiation via the ODL REST interface. In fact, none of the requested paths of the test suite were implemented. The controller have given positive replies (HTTP 201 - created) to the creation of all individual flow entries, however, none have been installed in the Open vSwitches. Positively replying to a request for flow creation and not implementing it in the data plane indicates an incorrect functionality, independently of any potential misconfigurations.

On the other hand, when the test suite was executed against the SDN architecture with the experimental set up using the ONOS controller, all tests successfully passed and the requested paths were correctly implemented in the data plane.

4.7.2.2 Mutation Testing

The test suite TS is indeed a complete test suite with respect to the presented fault model $\langle =, FD \rangle$, and therefore, its execution against an SDN architecture (implementation) provides a guarantee regarding the correct functioning of that SDN architecture, if the test suite passes it.

To further evaluate the fault detection effectiveness of each test sequence α in the derived test suite TS (test sequence 'power' / effectiveness), we have deliberately introduced a bug in the ONOS controller to provide positive replies to the creation of flow entries which are not installed on the devices.

To obtain a faulty implementation with the previously described bug, a single statement has been deleted from the ONOS controller's source code. Note that statement deletion is often considered in mutation testing [40] when the test suite quality is estimated. Particularly, in the ONOS implementation, the statement has been deleted in the FlowsWebResource.java file. As ONOS compiles with regression tests, a special compilation process ignoring such tests has been executed. Given the modified code source of the ONOS controller, the fault coverage of each test sequence in the test suite has been assessed.

As a result, all the test sequences have failed when being executed against the modified SDN architecture. This demonstrates that each test case in the test suite is 'capable' of detecting the introduced bug by its own. Therefore, these preliminary experiments also showcase the power of the obtained test cases and motivate to consider the test minimization in the future.

4.7.2.3 Discussion

Although the experiments were not performed for realistic SDN architectures, some conclusions can be drawn.

Firstly, the SDN architecture composed by the experimental set up together with the ONOS controller *is guaranteed* to be free of the faults considered in the defined fault domain $\mathcal{FD}$. In particular, it is guaranteed to be free of the wrong redirection, edge deletion and edge creation faults.

Further on, the obtained test suite is proven to be effective. Moreover, it has proven to detect other types of bugs (for example, a single statement deletion). However, an interesting direction for future work could be to investigate the relationship between the single statement deletion fault and the errors listed above (the wrong redirection, edge deletion and edge creation faults).

Secondly, the SDN architecture composed by the experimental set up together with the ODL controller seems not to be free of the bugs under consideration. Note that even if the second conclusion may be considered as trivial, the proposed approach can be seen as a helpful mechanism to 'certify' the correct functionality of a given SDN architecture under certain conditions and assumptions.

4.8 Chapter Conclusions

In this chapter, we have focused on model based testing techniques for checking the functionality of SDN architectures.

As the inputs of the SDN architectures are user-defined requests / policies, we have proposed to represent them by *paths*, then formal approaches for effective test generation for such non-trivial inputs have been presented. Namely, specific *graph / path enumeration* techniques under black box and white box testing assumptions have been discussed for testing an SDN architecture.

To formally prove the fault coverage, a *fault model* has been defined where the fault domain contains different implementations of the requested paths. Further on, conditions have been established, under which a *complete* test suite with respect to the defined fault model can be derived.

The experimental results have shown that the proposed approaches can detect implementation bugs in the SDN architecture under test. Moreover, the derived test suites have been proven to be efficient ('powerful') as each sequence of the test suite of interest has been capable of detecting by itself an artificially introduced bug in the case of the architecture with the ONOS set up.

It is worth noting that, at the time the author has focused on developing new formal approaches for testing SDN components as portrayed in Chapters 5, 6, and 7, an interesting direction of the work started in this chapter has been explored by the team involved in this work and led by Natalia Kushik.

Namely, more equivalent classes for testing have been explored and discussed by Yevtushenko et al. [163]. Their work has extended the contribution of this chapter by proposing several fault models (under different testing assumptions) with respect to the underlying $RNCT$. Some conditions for deriving a complete test suite as well as the complexity upper bounds with respect to the defined fault models have also been established. The effectiveness of their proposals has been planned to be investigated experimentally in future works.

The next chapter proposes a model based technique for testing a critical component of the SDN architecture, namely an SDN-enabled switch. More precisely, as a first step, we focus on testing the forwarding functionality of the switch modelled and analyzed as a stateless system without considering its interaction with the controller. Appropriate logic circuits are proposed to model the switch behaviour, an adequate fault model is introduced and then both active and

passive testing approaches are proposed.

5

Test Derivation for SDN-enabled Switches: A Logic Circuit based Approach

Contents

In the previous chapter, a testing method has been presented to guarantee the correct functional behaviour of an entire SDN architecture. Unfortunately, this does not automatically imply that the SDN components forming the architecture under test are functionally correct. The presented method thereby would not permit concluding on the faulty components, had it detected a misbehaviour in the whole architecture. For this reason, we now turn the attention to testing some critical SDN components forming the whole architecture. As a first step, we focus on the switch; firstly we propose to check its forwarding functionality in this chapter, and then we aim at testing its interaction with a controller in the next Chapter 6. Later on, we propose to examine another critical component, namely the controller in Chapter 7.

This chapter proposes a model based test generation technique that relies on *Logic Circuits* to test the forwarding functional behaviour of the SDN-enabled forwarding elements analyzed and modelled as 'stateless' systems.

5.1 Introduction

Network configuration and management are notoriously difficult due to the size and complexity of the network and it has been shown that 62% of the network downtime is caused by configuration errors [79]. An important consequence of the SDN principles is the separation of concerns introduced between the definition of network policies, their implementation in the data plane, and the forwarding of traffic. Although this separation is key to the desired flexibility, the risk of misconfigurations is even more exacerbated.

The behaviour of an SDN-enabled switch in the data plane strongly depends on the set of rules it has to enforce. With appropriate rules, an SDN-enabled switch can act like a Layer-2 switch or a router for example. Moreover, several network applications can be implemented, e.g., monitoring, accounting, traffic shaping, routing, access control, and load balancing [103]. Therefore, to ascertain the correct implementation of these applications and operators policies, it is important to guarantee the correct forwarding behaviour of the switch based on these rules.

We focus on testing SDN-enabled switches that act as *forwarding devices* receiving and sending network packets in accordance with a set of configured rules. In modern networks, such devices are predominantly implemented in software, and therefore different software bugs can induce different functional faults.

In this chapter, we propose a *logic circuit* (model) based technique for testing the forwarding functionality of an SDN-enabled switch. In particular, we propose to model the switch behaviour as a corresponding logic circuit in order to take advantage of various scalable manipulations over such circuits as well as to benefit from well-established techniques for their testing.

We contend that both active and passive testing can take advantage of such representation. In particular, we estimate the usefulness of logic circuit based fault models for detecting bugs and misconfigurations in the SDN-enabled switch implementations. To this end, we propose an *active testing approach* detailed in Section 5.7.1. Firstly, we introduce potential mutations over the switch rules and then we discover which of these mutations can be effectively detected using the proposed logic circuit based approach. Furthermore, we discuss how Boolean Satisfiability (SAT) solvers can be utilized for detecting equivalent mutants. Afterwards, we present a *passive testing approach* relying on a scalable solution for the switch monitoring on the basis of logic circuits and related operations in Subsection 5.7.2.

Section 5.2 of this chapter motivates and states the problem. Section 5.3 presents the formal representation of an SDN-enabled switch and includes necessary notations and definitions. A *fault model* is introduced in Section 5.4. Section 5.6 presents a specification model

for an SDN-enabled switch. We present two logic circuit based approaches for testing the switch in Section 5.7, namely, an active testing approach presented in Subsection 5.7.1 and a passive testing approach described in Subsection 5.7.2. Preliminary experimental results for a set of switch rules are presented in Section 5.8. Finally, section 5.9 concludes the chapter.

5.2 Problem Statement

The continuous SDN architectures innovation introduces a need for testing its components. Moreover, developing testing methods to guarantee that the entire architecture functions correctly as achieved in Chapter 4 is not enough. Indeed, an SDN component implemented with best available hardware and software features still may not be ready to be deployed unless it undergoes a testing phase. Further on, if the testing method presented in Chapter 4 reveals some faults, it is important to be able to localize them by pointing out the component causing the faulty behaviour (e.g., if it is the switch or the controller).

To achieve compliant and functional SDN components, effort must be put into all parts constituting the architecture. One of the critical component is the SDN-enabled switch.

While testing high level network functionality, the functional correct behaviour of an SDN-enabled switch might be taken for granted. To ensure correct architecture operation, all switches must work correctly. In other words, it may take just one buggy switch to cause problems in the form of incorrect forwarding for example. If failures start occurring in SDN architectures, the hard-earned ability to innovate in the networking space will be severely hampered by mistrust.

Motivating Example

For example, while using the ONOS controller [17] and Open vSwitch (OVS) [121] we have detected a potential overflow with respect to the switch port numbers. Namely, any request with an output port number which is greater than or equal to 2^{16} produces inconsistent results. Each of such requests gets the assigned port number modulus 2^{16}. The OpenFlow switch specification [113] states that the maximal physical and logical port number is 4294967040 (0xffffff00). Therefore, one can conclude there is a bug in the OVS implementation[1]. Such software bugs lead to the incorrect packet processing, i.e., the specification given as a set of rules for the switch is not respected. This type of issues only raise the importance of detecting such bugs. Thus the behaviour of such switch is critical to the correct functioning of the whole architecture and its correctness must be proven so as to avoid failures.

The research question we are addressing in this chapter can be formulated as follows. *Given the switch specified as a set of configurations to steer packets in the data plane, how to guarantee the correctness of its forwarding functionality?*

5.3 Formal Representation of an SDN Switch and Notations

We define the switch state as the collection of all the extracted rules stored in the switch. While the controller can change the rules in the switches, in this chapter, we consider a snapshot of the switch state in the network.

In other words, in this chapter we are interested in testing the switch in its data plane interface only and we check if its implementation complies with the set of rules defining its

[1]Version 2.0.2 used with the ONOS controller version 1.10.4.

behaviour, i.e., we assume that the switch is stateless. In the next chapter, we examine the system under test considering the switch in interaction with an SDN controller.

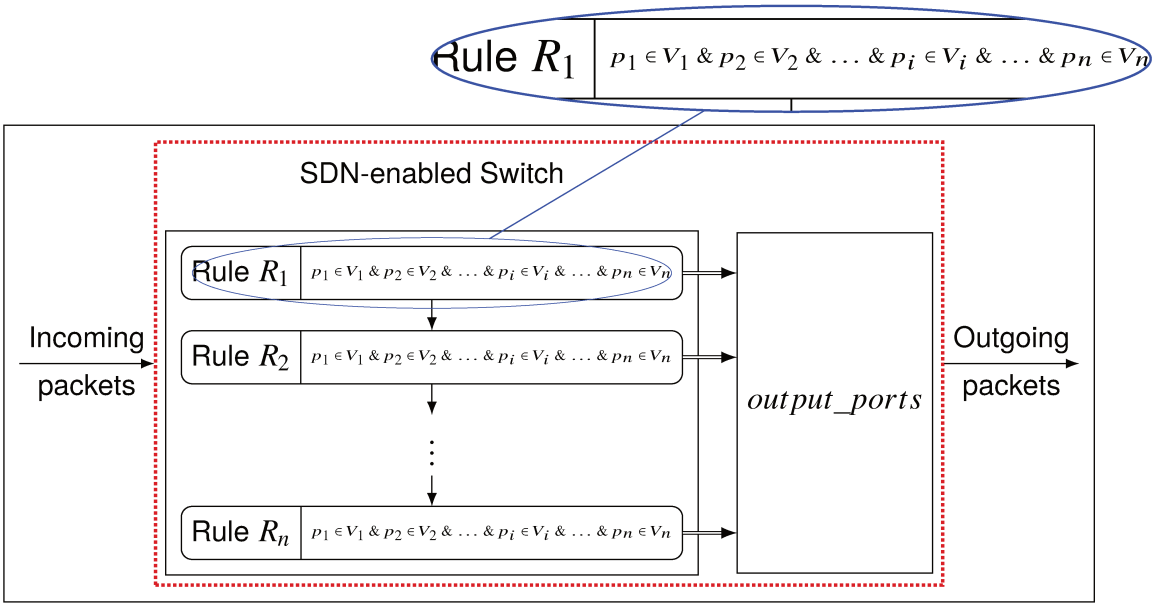


Figure 5.1 – Topology showing an SDN-enabled switch as the SUT

Definition 5.1.
A *rule* R as a part of the switch configuration is defined by the implication described in Equation 5.1.

$$R = (p_1 \in V_1 \,\&\, p_2 \in V_2 \,\&\, \ldots \,\&\, p_i \in V_i \,\&\, \ldots \,\&\, p_n \in V_n) \implies output\_ports = \{o_1, o_2, \ldots, o_m\} \tag{5.1}$$

Where

- p_i refers to an input parameter (e.g., IP_source, IP_dest, In_port, etc.);
- o_i refers to an output port;
- The sets V_1, V_2, ..., V_n define a range or an interval for each switch parameter p_1, p_2, ..., p_n, correspondingly[2].

Definition 5.2.

(I) We denote as Π_{p_i} the *projection operator* characterized with a parameter p_i such that for a given rule R (defined in Equation 5.1), Π_{p_i} is specified by Equation 5.2.

$$\Pi_{p_i} = V_i \tag{5.2}$$

(II) Similarly, we denote by Π_{out} the *output projection* of R defined in Equation 5.3.

$$\Pi_{out} = \{o_1, o_2, \ldots, o_m\} \tag{5.3}$$

[2]The defined intervals are assumed to contain integers, without loss of generality.

Definition 5.3.

(I) An *output mutant* for the rule R is defined in Equation 5.4.

$$\begin{aligned}&(p_1 \in V_1 \;\&\; p_2 \in V_2 \;\&\; \ldots \;\&\; p_i \in V_i \;\&\; \ldots \;\&\; p_n \in V_n)\\ &\qquad \Longrightarrow output\_ports = \{o'_1, \ldots, o'_{m'}\}\\ &\text{such that} \quad \{o'_1, \ldots, o'_{m'}\} \neq \{o_1, \ldots, o_m\}\end{aligned} \tag{5.4}$$

(II) A *parameter value mutant* for the rule R is defined by Equation 5.5

$$\begin{aligned}&(p_1 \in V_1 \;\&\; p_2 \in V_2 \;\&\; \ldots \;\&\; p_i \in V'_i \;\&\; \ldots \;\&\; p_n \in V_n)\\ &\qquad \Longrightarrow output\_ports = \{o_1, o_2, \ldots, o_m\},\\ &\text{such that} \quad V'_i \neq V_i\end{aligned} \tag{5.5}$$

Example

In the example illustrated in Table 2.1 in Chapter 2, the number of parameters is $n = 6$, and $|V_i| = 1,\ i \in \{1, \ldots, 6\}$. The number of output ports m for each rule R in Table 2.1 is 3.

5.4 Introducing a Fault Model

We assume that the switch implementation has no faults if each packet is processed exactly in the way the switch configuration requires. Moreover, if for a given packet pkt there is no rule R in the switch configuration such that the matching part of R shown in Equation 5.6 contains the necessary *preamble*, the packet pkt is simply dropped by the switch, i.e., should not be forwarded anywhere.

$$(p_1 \in V_1 \;\&\; p_2 \in V_2 \;\&\; \ldots \;\&\; p_n \in V_n) \tag{5.6}$$

Note however that a switch can output the packet to 'consult' with the SDN controller about the action applied to an 'unknown' packet [103]. Nevertheless, the controller might alter the rules in the switch configuration as a result. In this chapter, we assume this is a different *specification* and we propose a different approach for testing the switch-to-controller communication in Chapter 6.

We introduce a fault model that has three items, namely $\mathcal{FM} = \langle \mathcal{S}, =, \mathcal{FD} \rangle$ where $\mathcal{S}$, the specification, is the set of switch rules, i.e., the rule forwarding configuration of the switch (referred along the chapter simply as switch configuration); $=$ is the conformance relation represented by the equality, and $\mathcal{FD}$ is the fault domain where the potential switch implementations are explicitly enumerated.

As usual, we are interested in deriving *exhaustive* test suites, such that $\forall \mathcal{I} \in \mathcal{FD}, \mathcal{I} \neq \mathcal{S}$, is detected by the test suite.

We also note that the system specification in this case can be *complete* (completely specified) or *partial* as defined in Definition 5.4. We further discuss how completeness and partiality of $\mathcal{S}$ affect the exhaustiveness of the test suites derived using well-known logic circuit based fault models.

Definition 5.4.

The set $\mathcal{S}$ of switch rules is said to be $\begin{cases} \textit{Complete} \text{ if it satisfies Equation 5.7.} \\ \textit{Partial}, \text{ otherwise.} \end{cases}$

$$\begin{aligned} &\forall\ preamble\ \ (p_1 \in V_1\ \&\ p_2 \in V_2\ \&\ \ldots\ \&\ p_n \in V_n) \\ &\exists\ a\ rule\ R \in \mathcal{S} \quad \text{such that} \\ &R = (p_1 \in V_1\ \&\ p_2 \in V_2\ \&\ \ldots\ \&\ p_n \in V_n) \implies output\_ports = \{o_1, o_2, \ldots, o_m\} \end{aligned} \tag{5.7}$$

5.5 Fault models for Logic Circuits

Three types of faults can occur in the circuit implementation, namely *Single Stuck-At Faults*, *Single Bridge Faults*, and *Hardly Detectable Faults*. We refer to a circuit that contains a fault as a *mutant*.

- ***Single Stuck-At Fault (SSF)*** The most common model used for logic faults is the single stuck-at fault. It assumes that a fault in a logic gate results in one of its inputs or the output to be fixed at either a logic '0' (stuck-at-0) or at logic '1' (stuck-at-1).

  The fault of this type arises when the value of a certain gate 'gets stuck' at logic one or zero. This error type is reproduced in the corresponding BLIF file by replacing all input values of a certain gate by the symbol "dash" (-) with the corresponding output ('1' or '0').

  Recall the example given in Chapter 2 where the BLIF file for the logic circuit of Figure 2.3 is given in Listing 2.4. An example of the single stuck-at fault at which the value of the gate $z1$ is stuck at 1 is shown in Figure 5.2a;

- ***Single Bridge Fault (SBF)*** The fault of this type arises when the input of a certain gate is erroneously connected (disconnected) to (from) another gate. Such fault type is reproduced in the corresponding BLIF file by replacing the input name of a given gate with another.

  Figure 5.2b shows an example of a single bridge fault in which input $n11$ of gate $z1$ is replaced by output of gate $z0$;

- ***Hardly Detectable Fault (HDF)*** This type of fault is meant to have slight differences with respect to the overall behaviour of the original circuit. The goal is to make a single gate change its output value for a single input.

  In order to produce these mutants in the BLIF files a single output cover is modified. Within this single output cover a single input is selected. The value of the input is modified to a distinct input value. This distinct input value can be '1' or '0' if the value is don't care ('-'); '1' or '-' if the value is '0'; and finally '0' or '-' if the value is '1'.

  To illustrate HDF mutants, Figure 5.2c shows an example of the hardly detectable fault of our running example. In this example, the gate $n9$ is modified such that only when $x0$ is '1' and $x2$ is '0' then $n9$ is '1'. In the original circuit specification, $n9$ corresponds to the Boolean function $n9 = x0$ AND $x2$. In the mutant, $n9$ corresponds to the Boolean function $n9 = x0$ AND NOT $x2$.

```
. model cirex
. inputs x0 x1 x2
. outputs z0 z1
. names x0 x2 n6
10 1
. names x2 n6 n7
00 1
. names x1 n7 z0
10 1
. names x0 x2 n9
11 1
. names x0 x2 n10
00 1
. names n9 n10 n11
00 1
. names x1 n11 z1
-- 1
. end
```

(a) SSF mutant

```
. model cirex
. inputs x0 x1 x2
. outputs z0 z1
. names x0 x2 n6
10 1
. names x2 n6 n7
00 1
. names x1 n7 z0
10 1
. names x0 x2 n9
11 1
. names x0 x2 n10
00 1
. names n9 n10 n11
00 1
. names x1 z0 z1
10 1
. end
```

(b) SBF mutant

```
. model cirex
. inputs x0 x1 x2
. outputs z0 z1
. names x0 x2 n6
10 1
. names x2 n6 n7
00 1
. names x1 n7 z0
10 1
. names x0 x2 n9
10 1
. names x0 x2 n10
00 1
. names n9 n10 n11
00 1
. names x1 n11 z1
10 1
. end
```

(c) HDF mutant

Figure 5.2 – Examples of SSF, SBF, and HDF mutants of C_{ex} shown in Figure 2.4

5.6 Deriving a Logic Circuit for a Switch Specification

The specification $\mathcal{S}$ represented by a set of switch rules is not scalable for solving different problems, such as for example, searching for two rules in possibly different tables that coincide or that on the contrary, contradict each other. We therefore, propose to build a logic circuit that preserves the behaviour of $\mathcal{S}$ on one hand, but allows taking advantage of several scalable manipulations over the Boolean vectors (logic circuits) on the other hand.

Such logic circuit $\mathcal{LC}$ can be derived in different ways and in this work, we focus on the use of logic synthesis solutions from a Look-up-Table (LUT) for a system of (partially specified) Boolean functions[3]. The corresponding procedure is described in Algorithm 2.

Given a set of rules forming the specification $\mathcal{S}$, Algorithm 2 derives a logic circuit simulating the behaviour of $\mathcal{S}$ as follows.

First, it determines the set of parameters $P = \{p_1, p_2, \ldots, p_n\}$ such that at least one *preamble* of at least one rule of $\mathcal{S}$ uses each $p_i, \forall i \in \{1, 2, \ldots, n\}$ (line 1). Then, the numbers of primary inputs and outputs of the logic circuit (lines 2 and 3) are computed. The number of primary inputs is the sum $\sum_{i=1}^{n} \lceil log_2(1 + max(\bigcup_{R \in \mathcal{S}} \Pi_{p_i})) \rceil$ where $max(\bigcup_{R \in \mathcal{S}} \Pi_{p_i})$ is the maximal element in all sets for the parameter p_i where all values of p_i are non-negative. The symbol $\lceil x \rceil$ denotes the ceiling function applied to x. The number of primary outputs is $max(\bigcup_{R \in \mathcal{S}} \Pi_{out})$, which denotes the maximum output port number used in $\mathcal{S}$.

Next, an empty LUT is derived (line 4). The variables of the LUT correspond to the computed number of primary inputs. The partially specified Boolean functions of the LUT correspond to the computed number of primary outputs.

Afterwards, for each rule $R \in \mathcal{S}$, the preamble and the output ports are encoded as follows. A corresponding Boolean vector B_i of length $\lceil log_2(1 + max(\bigcup_{R \in \mathcal{S}} \Pi_{p_i})) \rceil$ encoding the *preamble* is derived (line 7). Also, for the output ports' set, a corresponding Boolean vector B_port

[3]It is intuitively right to consider Boolean representations for values transmitted in network packets as they represent data in binary strings.

Algorithm 2: Logic Circuit Derivation from a Set of Switch Rules

Input : A specification $\mathcal{S}$ represented by a set of switch rules

Output : A logic circuit $\mathcal{LC}$ simulating $\mathcal{S}$

1 Define the set of parameters $P = \{p_1, p_2, \ldots, p_n\}$ such that each parameter $p_i \in P$ is used in at least one preamble of at least one rule $R \in \mathcal{S}$.

2 Determine the number of the primary inputs for the logic circuit as $\sum_{i=1}^{n} \lceil log_2(1 + max(\bigcup_{R \in \mathcal{S}} \Pi_{p_i})) \rceil$ such that $max(\bigcup_{R \in \mathcal{S}} \Pi_{p_i})$ is the maximal element in all sets for the parameter p_i where all values of p_i are non-negative, and $\lceil x \rceil$ denotes the ceiling function applied to x.

3 The number of the primary outputs for the logic circuit equals to $max(\bigcup_{R \in \mathcal{S}} \Pi_{out})$, where $max(\bigcup_{R \in \mathcal{S}} \Pi_{out})$ denotes the maximum output port number used in $\mathcal{S}$.

4 Derive an empty $LUT\ L$ for a system of $max(\bigcup_{R \in \mathcal{S}} \Pi_{out})$ partially specified Boolean functions of $\sum_{i=1}^{n} \lceil log_2(1 + max(\bigcup_{R \in \mathcal{S}} \Pi_{p_i})) \rceil$ variables

5 **foreach** *rule* $R \in \mathcal{S}$ **do**

6 **foreach** $r = (v_1, v_2, \ldots, v_n) \in V_1 \times V_2 \times \ldots \times V_n$, *where* $V_i = \Pi_{p_i}, \forall i \in \{1, 2, \ldots, n\}$ **do**

7 Encode each v_i, $i \in \{1, 2, \ldots, n\}$ by a Boolean vector B_i of length $\lceil log_2(1 + max(\bigcup_{R \in \mathcal{S}} \Pi_{p_i})) \rceil$

8 Set the Boolean vector B_port to $(00\ldots0)$, $|B\_port| = max(\bigcup_{R \in \mathcal{S}} \Pi_{out})$

9 **foreach** *output port* $o_j \in \{o_1, \ldots, o_m\}$ **do**

10 Set o_j-th bit of B_port to 1 (the first bit starts at the rightmost position with index 1)

11 Add a new line to the LUT, i.e., set L to $L \cup \{B_1 B_2 \ldots B_n | B\_port\}$

12 Run a logic synthesis solution for deriving a logic circuit $\mathcal{LC}$ from the $LUT\ L$

13 **return** $\mathcal{LC}$

of length $|B\_port| = max(\bigcup_{R \in \mathcal{S}} \Pi_{out})$ encoding the output ports is derived (lines 8, 9 and 10). Then, for each rule in $\mathcal{S}$, these derived Boolean vectors are added to the LUT (line 11). Finally, a logic synthesis solution is run and a logic circuit $\mathcal{LC}$ simulating the behaviour of $\mathcal{S}$ is returned (lines 12 and 13).

For the running example of the set of switch rules listed in Table 2.1, the LUT derived by Algorithm 2 has four lines illustrated in Table 5.1. Note that dashes (–) denote 'don't care' terms[4].

[4]The netmasks of the IP addresses are taken into consideration by the dashes in the corresponding field.

$x_1x_2\ldots x_{89}$	$f_1f_2f_3$
01111101 00--10000000011 0----------	001
01111101 00011000000001 10----------	110
11111101 01011000000000000000101 0000000000000000000000000001 0000101 0000000000000000000000000010	010
11111101 01101000000000000000101 0000000000000000000000000001 0000101 0000000000000000000000000001	001

Table 5.1 – Look-up table for the switch running example

5.7 Active and Passive Testing Approaches

5.7.1 Active Testing Approach

5.7.1.1 Test Suite Generation

Once a logic circuit $\mathcal{LC}$ that simulates the behaviour of the switch with the rules $\mathcal{S}$ is derived, one can apply different techniques for test generation. In this work, we propose a technique in which the circuit $\mathcal{LC}$ representing the switch rules is 'altered' in order to obtain a set of mutants of different kinds. The test suite derived to kill each mutant of a particular type can later be applied to the implementation of an SDN switch. The goal of deriving such test suite is to distinguish the output of a correct implementation from an assumed incorrect implementation (mutant)[5].

In our approach, we consider three types of faults that can occur in the switch circuit implementation, namely *Single Stuck-At Faults* (*SSF*), *Single Bridge Faults* (*SBF*), and *Hardly Detectable Faults* (*HDF*) and we derive corresponding test suites. The advantage of this approach is that logic circuit testing techniques are well studied and elaborated and there exist a number of tools for such automatic test derivation. For instance, several test generation strategies against the aforementioned circuit faults have been proposed in the last decades. The interested reader can, for example, refer to [99] and [115]. In our work, we used the tool developed by Kushik et al. [85] together with the logic synthesis and verification tool called ABC by Brayton et al. [22].

Moreover, test suites derived against the *Single Stuck-At Fault mutants* are claimed to have high fault coverage with respect to other types of circuit mutants. Our approach investigates the fault coverage of the derived test suites when testing SDN-enabled switches effectively described by corresponding logic circuits.

Furthermore, in some cases, certain properties for a test suite fault coverage can be guaranteed. For example, for *SSF* mutants Propositions 5.5 and 5.6 hold.

Proposition 5.5.
If $\mathcal{S}$ is complete and Equation 5.8 is satisfied then each output fault in the rule R is detected by an *exhaustive* test suite with respect to *SSF*s.

$$\begin{aligned} &\exists\, i \in \{1,\ldots,m\} \text{ such that} \\ &\exists!\ R \in \mathcal{S}, \\ &R = ((p_1 \in V_1\ \&\ p_2 \in V_2\ \&\ \ldots\ \&\ p_n \in V_n) \implies output\_ports = \{o_1, o_2, \ldots, o_m\}) \\ &\text{and } o_i \notin \{o_1, o_2, \ldots, o_m\} \end{aligned} \tag{5.8}$$

Proof.
The completeness of the specification $\mathcal{S}$ automatically implies the completeness of the system

[5]Note that we do not focus in this work on testing unsupported ports: the port number(-s) of an implementation under test (IUT) should belong to the set of supported port numbers.

of Boolean functions implemented by the circuit $\mathcal{LC}$ (Algorithm 2). Each output fault, represented in Equation 5.9, is only detected by a test suite if this test suite includes the input pattern $B_1B_2\dots B_n$ that corresponds to the rule R.

$$\begin{aligned}&(p_1 \in V_1 \;\&\; p_2 \in V_2 \;\&\; \dots \;\&\; p_n \in V_n) \implies output\_ports = \{o'_1, o'_2, \dots, o'_{m'}\},\\ &\text{and} \quad \{o'_1, o'_2, \dots, o'_{m'}\} \neq \{o_1, o_2, \dots, o_m\}\end{aligned} \tag{5.9}$$

As the test suite TS contains a pattern that distinguishes each *SSF* mutant of the logic circuit $\mathcal{LC}$, for the output i of $\mathcal{LC}$ this pattern can only be $B_1B_2\dots B_n$, otherwise the stuck-at-one fault in the i-th output cannot be detected (due to the uniqueness of the rule R). □

Note that whenever the set $\mathcal{S}$ of switch rules is not complete, the logic circuit $\mathcal{LC}$ is derived for a system of partially specified Boolean functions. Therefore, the behaviour of the circuit over the undefined patterns can be specified in different ways. In our approach and in our experiments, we use ABC, which sets the corresponding outputs to 0. This fact allows to guarantee the fault coverage for output mutants of the rules when initially the specification $\mathcal{S}$ is not complete.

Proposition 5.6.
If for a set of rules $\mathcal{S}$, Equation 5.10 is satisfied, then each output fault in the rule R is detected by an exhaustive test suite w.r.t. *SSF*s.

$$\begin{aligned}&\exists\, i \in \{1, \dots, m\} \text{ such that}\\ &\exists!\; R \in \mathcal{S}, R = ((p_1 \in V_1 \;\&\; p_2 \in V_2 \;\&\; \dots \;\&\; p_n \in V_n) \implies output\_ports = \{o_1, o_2, \dots, o_m\})\\ &\text{and} \quad o_i \in \{o_1, o_2, \dots, o_m\}\end{aligned} \tag{5.10}$$

Proof.
Similar to Proposition 5.5, a test suite TS which detects each stuck-at-zero fault on the i-th output of $\mathcal{LC}$ must contain a pattern $B_1B_2\dots B_n$ that corresponds to the preamble $(p_1 \in V_1 \;\&\; p_2 \in V_2 \;\&\; \dots \;\&\; p_n \in V_n)$ of rule R. This exact pattern detects each output fault in the rule R. □

Corollary 5.7.
If in the set $\mathcal{S}$ of switch rules, each output port is used in at most one rule, then an exhaustive test suite w.r.t. *SSF*s is also exhaustive w.r.t. rule output mutants.

We note however, that the above statements do not necessarily hold for the parameter value mutations. Such faults can in some cases be detected by other mutants of logic circuits such as, bridges or hardly detectable faults. Nevertheless, thorough investigation of the correlation between the mutations of rules and those of logic circuit still needs to be performed. Such investigation is left for future work.

5.7.1.2 SAT Solving for Equivalent Mutant Detection

Whenever possible rule mutations are enumerated explicitly and therefore, a test suite TS is derived under the white box testing assumption aiming at killing all the mutants of certain type, the question of *equivalent mutants* automatically rises [58].
Indeed, mutations of different orders (especially second and higher) have a high probability of deriving an equivalent mutant. However, as the number of patterns can be rather high

$(2^{\sum_{i=1}^{n}\lceil log_2(1+max(\cup_{R\in S}\Pi_{p_i}))\rceil})$, applying / checking all such patterns can be a time consuming task, and thus, detecting equivalent mutants by direct (brute force) search becomes unfeasible. Correspondingly, such equivalent mutants can be effectively detected whenever two logic circuits $\mathcal{LC}$ and $\mathcal{LC}_M$ for both the specification $\mathcal{S}$ and the mutant $\mathcal{M}$ under investigation, are derived.

Indeed, the equivalence decision problem can be reduced to the well-known SAT problem. For this reason, a *miter* of two circuits can be derived. For two logic circuits $\mathcal{LC}$ and $\mathcal{LC}_M$ with the set $X = \{x_1, \ldots, x_k\}$ of inputs and the sets $O = \{o_1, \ldots, o_p\}$ and $O' = \{o'_1, \ldots, o'_p\}$ of outputs, a *miter* Mit with the set $X = \{x_1, \ldots, x_k\}$ of inputs and a single output is derived as follows. The output function of Mit is the result of a logic OR operation of the functions $f_1, \ldots, f_p$ that are implemented as the XORs of output functions $g_1, \ldots, g_p$ and $h_1, \ldots, h_p$ of the circuits $\mathcal{LC}$ and $\mathcal{LC}_M$ correspondingly, i.e., $f_j = g_j \oplus h_j$, $j \in \{1, 2, \ldots, p\}$. Circuits LC and LC_M are equivalent, and so are the sets of rules $\mathcal{S}$ and $\mathcal{M}$, if each output of the miter Mit always equals 0, i.e., when the corresponding Boolean function is UNSAT. Algorithm 3 implements this strategy to detect equivalent mutants.

Algorithm 3: Equivalence Check for a Switch Mutant

Input : A specification $\mathcal{S}$ represented by a set of switch rules and its mutant $\mathcal{M}$

Output : The verdict about the mutant equivalence or a test case killing $\mathcal{M}$

1 Run Algorithm 2 for both, specification S and its mutant $\mathcal{M}$, obtain the logic circuits $\mathcal{LC}$ and $\mathcal{LC}_M$, correspondingly

2 Construct the miter Mit on the circuits $\mathcal{LC}$ and $\mathcal{LC}_M$

3 Run a SAT solver for the Boolean function f implemented by Mit

4 **if** *UNSAT* **then**

5 **return** *the verdict 'The mutant $\mathcal{M}$ is equivalent to $\mathcal{S}$'*

6 **return** *A satisfying pattern B for the Boolean function f*

The correctness of the proposed equivalence check is established by Proposition 5.8.

Proposition 5.8.
For a given set $\mathcal{S}$ of switch rules and a given mutant $\mathcal{M}$, of this specification, Algorithm 3 returns a test case killing $\mathcal{M}$ if and only if the mutant $\mathcal{M}$ is not equivalent to $\mathcal{S}$.

Proof.
Indeed, the circuit Mit implements a constant 0 if and only if the outputs B_port coincide for all input patterns $B_1B_2 \ldots B_n$ (Algorithm 2). Thus, a satisfying pattern B for the function f returns an input $B_1B_2 \ldots B_n$ for the preamble $(p_1 \in V_1 \;\&\; p_2 \in V_2 \;\&\; \ldots \;\&\; p_n \in V_n)$ where the output ports differ. □

We note that such equivalence check can be performed over the logic circuit representations in a scalable manner. The reason is that both circuits $\mathcal{LC}$ and $\mathcal{LC}_M$ are combinational, i.e., without latches or internal memory.

For sequential circuits, the derivation of the miter as well as the the SAT problem formulation is in fact much more complex. The latter means, that if modelling a switch as a stateful system, for example when taking into account its potential communication with an SDN controller, the

solution of the equivalence check might not be scalable. More research and experiments are needed in this area, and these tasks are left for the future work.

5.7.2 Passive Testing Approach

We previously discussed the use of logic circuits for *active* test generation for an SDN-enabled switch. In fact, whenever the access to the switch is limited and its behaviour can only be observed, a logic circuit $\mathcal{LC}$ modelling the specification $\mathcal{S}$ can still be effectively utilized. The reason is that the simulation of $\mathcal{LC}$ can in some cases be much faster than the search of the particular switch rule and its further application to conclude about the expected output port(-s). In other words, the task of the switch monitoring that verifies that the packets are forwarded to the exact ports specified by $\mathcal{S}$, can be reduced to the problem of the circuit simulation[6], i.e., obtaining an output pattern for a given input one. This approach is described in Algorithm 4.

Algorithm 4: SDN-enabled Switch Monitoring

Input : The switch implementation under test $\mathcal{I}$ and $\mathcal{S}$ the corresponding specification $\mathcal{I}$ must implement

Output : Alerts for the packets processed wrongly by the given IUT

1 Run Algorithm 2 on $\mathcal{S}$, obtain a logic circuit $\mathcal{LC}$

2 **while** *working* **do**

▷ *working* is a Boolean flag to control the execution of the monitoring process

3 $packet\_observed \leftarrow input(\mathcal{I})$

▷ *input* returns the processed input of the IUT

4 Extract the Boolean vector B_packet from the $packet\_observed$ header parameters, including the encoded value for the input port of the IUT

5 $port\_observed \leftarrow output(\mathcal{I})$

▷ *output* returns the port number for the *input*

6 Encode $port\_observed$ as the Boolean vector B_port

7 **if** $port\_observed \neq sim(\mathcal{LC}, B\_packet)$

▷ *sim* is a function that simulates the circuit behaviour over a given input

8 **then**

9 $alert(packet\_observed)$

▷ Alert an incorrect processing of $packet\_observed$

Algorithm 4 takes as input the switch implementation under test $\mathcal{I}$ and $\mathcal{S}$ the corresponding specification $\mathcal{I}$ must implement. It calls Algorithm 2 to derive the corresponding $\mathcal{LC}$ from $\mathcal{S}$ (line 1). Then, it stores the header being processed of the received input packet (line 3) in the variable $packet\_observed$. Next, the algorithm extracts the different parameters of

[6]Under the assumption that the circuit simulation is correct.

the stored header leading to the construction of a Boolean vector B_packet (line 4). Then it observes and stores the output ports of the implementation under test (lines 5) in the variable $port\_observed$ and encodes the latter as a Boolean vector B_port (6). Finally, if the output ports are different from those returned by the function that simulates the circuit behaviour over a given input, then, an alerts is emitted indicating that the packets are processed wrongly by the given implementation under test (line 9).
As we will discuss in Section 5.8, in many cases, the verdict about the correct or incorrect application of a given switch rule can be made much faster when the logic circuit representation is exploited.

5.8 Experimental Evaluation for Testing an SDN-enabled Switch

As in the previous chapter, the experiments have been performed on the widely-known SDN-enabled switch, namely Open vSwitch (OVS) [121] version 2.0.2, in an emulated networking environment using Mininet.

5.8.1 Experimental Set Up

The topology of the emulated network is similar to the one shown in Figure 5.3. This topology models the data plane as a graph where all switches are connected to an SDN controller. In Figure 5.3, hosts and switches are labeled with strings starting with the word $Host$ and letter S respectively. For each edge in the graph, the corresponding port number used by the two nodes is depicted; for example, in the edge (S_2, S_3), the label '3 3' indicates that the port 3 at switch S_2 is connected to the port 3 at switch S_3.
In addition, the Ethernet MAC addresses for each of the hosts are shown above or below each host. For our experiments, without loss of generality, we have chosen the switch S_3 (depicted with a dotted pattern) as the system under test. The ONOS [17] controller version 1.10.4 has been used for all the experiments.

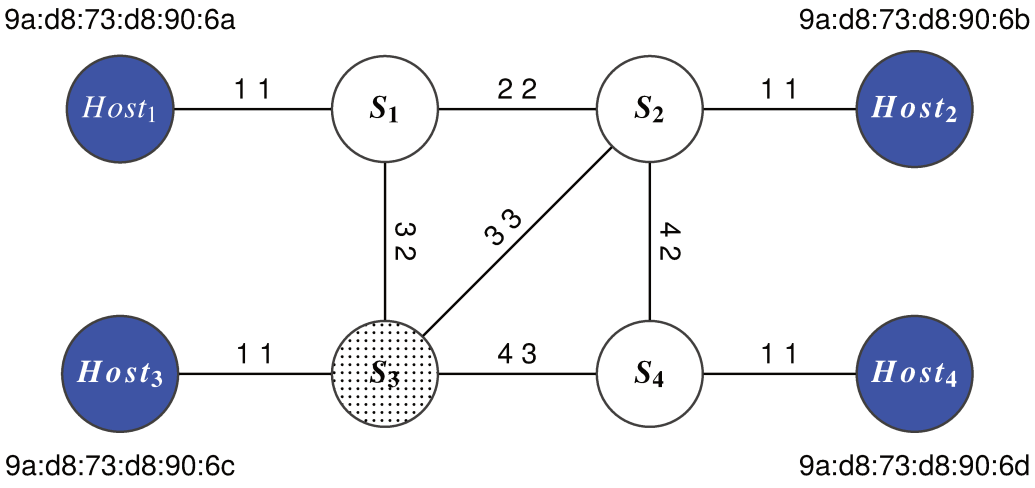


Figure 5.3 – Experimental set up topology for testing an SDN-enabled switch

The experiments have been executed under different virtual machines running under a VirtualBox Version 5.2.8 r121009 for Mac OS X 10.13.4. The characteristics of the used virtual machines are shown in Table 5.2.

ID	Operating System	CPUs	RAM
$VM1$	CentOS 6.9 2	Intel(R) Core(TM) i5-2415M CPU @ 2.30GHz processors	3GB
$VM2$	Ubuntu 14.04.4 LTS (Mininet Dist.)	Intel(R) Core(TM) i5-2415M CPU @ 2.30GHz processors	2GB

Table 5.2 – Experimental platform for an SDN switch

5.8.2 Results and Evaluation

5.8.2.1 Logic Circuit Fault Models for SDN-enabled Switch Fault Model

There exist a large number of possibilities how an SDN network can be (re) configured, and therefore how to obtain the switch rule set specification $\mathcal{S}$. In fact, as studied in Chapter 4, network applications in the application layer query the SDN controller for information of the data plane and request the implementation of different rule sets for the forwarding devices (e.g., SDN-enabled switches).
The network applications can have different goals; for example, one application can monitor the data plane in order to balance the traffic load between the network links (edges). One of the most common applications for an SDN network is the Layer 2 switching application; this application forwards Layer 2 packets (e.g., Ethernet) between hosts using the shortest paths. To provide meaningful results, we have programmed the data plane with a Layer 2 switching application. The rules have been pushed through the controller using the REST interface. After pushing the rules, communication from / to any host has been successfully achieved via the Ethernet protocol.
A logic circuit $\mathcal{LC}$ has been derived from the specification $\mathcal{S}$ using Algorithm 2. The logic synthesis tool used for deriving $\mathcal{LC}$ has been the ABC tool [22].

Example

As an example, one rule $R \in \mathcal{S}$ installed on S_1 is depicted in Equation 5.11.

$$\begin{aligned} R = ((&\text{INPUT\_PORT} \in \{1\} \;\&\; \text{ETH\_SRC} \in \{\text{9a:d8:73:d8:90:6a}\} \;\&\; \\ &\text{ETH\_DST} \in \{\text{ff:ff:ff:ff:ff:ff}\} \implies output\_ports = \{2,3\}) \end{aligned} \tag{5.11}$$

Note that after the logic synthesis of the specification, the corresponding LUT entry for the rule in Equation 5.11 is depicted in Table 5.3.

$x_1x_2\ldots x_{89}$	$f_1f_2f_3$
01100110101101100001110011110110001001000001101010111111111111111111111111111111111111111111111111111111	110

Table 5.3 – Example of a look-up table entry for the rule in Equation 5.11

Once the logic synthesis accomplished, $\mathcal{LC}$ has 99 inputs and 4 gates. The logic circuit has then been saved into the Berkeley Logic Interchange Format (BLIF). A BLIF Mutant Generator (BMG) [85] tool has been executed to generate mutants for the SSF, SBF, and HDF types. The total number of mutants is 214, where 206 mutants are all SSF mutants, 4 are randomly chosen SBF, and likewise 4 are randomly chosen HDF mutants. A distinguishing pattern has been found for each non-equivalent mutant. For each of the fault models, a test suite has

been obtained, i.e., TS_{SSF}, TS_{SBF}, and TS_{HDF}. Furthermore, the union of all three test suites has been used to obtain TS_{ACF}, a test suite for all circuit faults[7]. The original BLIF circuit representing $\mathcal{LC}$ contains a sum of products, hence, only 4 gates (the output gates).

In order to check whether the fault coverage increases with different circuit representations, the original BLIF file representing $\mathcal{LC}$ has been re-synthesized as an AND-INVERTER graph (AIG). We hereafter refer to this circuit as $\mathcal{LC}'$ modelling the specification $\mathcal{S}'$ where $\mathcal{S}$ and $\mathcal{S}'$ are functionally equivalent, ($\mathcal{LC}'$ is functionally equivalent to $\mathcal{LC}$ modelling $\mathcal{S}$). $\mathcal{LC}'$ has 99 inputs and 395 gates, and therefore, the total number of mutants is 1778, where 998 mutants are all SSF mutants, 395 are randomly chosen SBF, and likewise 395 are randomly chosen HBF mutants. The same procedure of test generation applied to $\mathcal{LC}$ has been performed on $\mathcal{LC}'$ to obtain the corresponding test suites.

Table 5.4 summarizes the numbers of generated mutants for $\mathcal{LC}$ (modelling $\mathcal{S}$) and $\mathcal{LC}'$ (modelling $\mathcal{S}'$).

Circuit	SSF	SBF	HDF	Total
$\mathcal{S}$	206	4	4	214
$\mathcal{S}'$	998	395	395	1778

Table 5.4 – Number of generated mutants

To check the fault coverage of traditional digital circuit fault models, a set $\mathcal{M}$ of 45 mutants of $\mathcal{S}$ has been generated. The set of mutants contains different (higher) order mutants. After running Algorithm 3 to remove from $\mathcal{M}$ the equivalent mutants (1 equivalent mutant has been removed), each pattern p in each of the test suites TS_{SSF}, TS_{SBF}, TS_{HDF}, TS_{ACF} has been used to simulate the behaviour of $\mathcal{S}$ and compare it to the behaviour of each mutant M in the non-equivalent mutant set, $\forall\ M \in \mathcal{M}$.

The mutation score / fault coverage[8] obtained for each of the test suites is shown in Table 5.5.

Circuit	SSF	SBF	HDF	ACF (total)
$\mathcal{S}$	79%	45%	18%	86%
$\mathcal{S}'$	95%	97%	95%	100%

Table 5.5 – Fault Coverage for traditional digital circuit fault models

As shown in Table 5.5, the fault coverage of traditional logic circuit fault models reaches 100% for a Layer 2 switching specification. Therefore, we conclude that test suites derived based on traditional logic circuit fault models have a high fault coverage for SDN-enabled switch faults.

An interesting aspect is that the fault coverage highly increases when the original circuit specification is transformed into an AIG. It is reasonable to assume that AIGs have more gates, and therefore more mutants, hence more distinguishing patterns are obtained with such representations. Thus, different functional errors in the SDN-enabled switch rules can be covered by a larger test suite when derived based on such AIGs.

[7] ACF stands for 'all circuit faults'.

[8] Calculated as the ratio of killed mutants to total number of (non-equivalent) mutants.

5.8.2.2 Using Logic Circuits for Monitoring

As discussed earlier, run-time monitoring for verifying functional properties of a given IUT $\mathcal{I}$ requires that the monitor (in this case $\mathcal{LC}$) is not slower than $\mathcal{I}$.
On one hand, an observation one can make is that it can be presumed that a combinational circuit has a constant (or near to constant) computational time for any input pattern. On the other hand, another observation is that the switch implementations such as Open vSwitch effectively work by caching the corresponding actions to be applied to a subsequent matching packet, and thus, these cached actions are applied to ulterior packets matching the rule [121]. The match look-up (and corresponding action to be applied) is performed using a set of hash tables which size is increased with each unique match [121].
Therefore, a priori and based on the intuition these observations can induce, one can make the hypothesis that the simulation of a logic circuit for the implementation of the switch behaviour should be faster than the described inherent caching mechanism.
To validate our hypothesis and prove that logic circuits are indeed suitable for run-time monitoring of SDN-enabled switches, we have performed the following experiment. On one hand, 10000 rules have been pushed into the switch S_3 (IUT) using the ONOS controller. On the other hand, a logic circuit has been obtained using Algorithm 2 for the specification containing the same 10000 rules installed in the IUT. Then, a comparison has been made between the time taken by Open vSwitch to process one packet and the time taken to simulate a single pattern of the logic circuit.
The time Open vSwitch takes to process one packet and the time taken to simulate a single pattern of the logic circuit have been measured as follows. To determine the time to process one packet, monitors have been installed on each switch port (interface). The time difference between the packet ingress and the packet egress has been measured to be ~ 0.29ms while executing Open vSwitch under $VM2$ (Table 5.2).
For measuring the time taken to simulate a single pattern of the logic circuit, some concerns should be taken into consideration. Indeed, as the time to simulate a single pattern in ABC (given a synthesized circuit) is considerably low, precision issues may occur. Furthermore, reading the file and writing the response to the standard output take most of the simulation time. For this reason, the relevant values have been extracted from the packet used, and the Open vSwitch has been simulated one thousand times. Finally, the average time taken to simulate a single pattern has been computed to be ~ 0.003408ms.

5.8.2.3 Discussion

As a conclusion of the presented experiments, it can be seen that the logic circuit simulation is more than 85 times faster compared to the switch packet processing. Arguably, the difference in the input / output interfaces for the different environments (packet input in a switch vs. file read in a circuit simulation) can also affect the time estimations.
However, capturing packets at a network interface is done when the packet has been processed by the interface and processed by the operating system, therefore, reading a packet is done from the internal (RAM) memory of the devices. On the other hand, the file used to simulate a pattern in the circuit simulation is performed from a hard drive (HD). For that reason, it is reasonable to assume the time measurements performed over the switch have an inherent advantage (RAM access is much faster than HD access). Therefore, theoretically, the speed-up may be even larger when considering the same input / output interfaces.
It is of special interest to accurately estimate the obtained speed-up due to its potential applications not only for testing reasons but, for optimizing the switch implementations. Performing

such investigation of the speed-up obtained via the logic circuit representation of a set of rules is left for the future work.

5.9 Chapter Conclusions

In this chapter, we have proposed a logic circuit based approach for testing SDN-enabled devices. It allows to take advantage of well established test generation strategies for logic circuits as well as of scalable manipulation over Boolean vectors and functions. The switch has been tested for its forwarding functionality in the data plane and has been modelled as a stateless system. We have also introduced some mutation operators over the switch rules and have discussed how logic circuit and related SAT solving can be utilized for detecting equivalent mutants.
Finally, we have considered run-time verification of switches and have investigated the use of logic circuits in this case.
Preliminary experiments with Open vSwitch have confirmed the effectiveness of the proposed approach. The meaningful results motivate us to apply such approach for testing other SDN components. Thus, in Chapter 7, we adapt the proposed approach of this chapter to test a specific controller application.
In the next chapter, we consider the switch-to-controller communication as the system under test where we take into consideration its behaviour in the southbound interface. In contrast to the present work, the switch in this case is modelled and analyzed as a ‘stateful’ system.

6

Test Generation for OpenFlow Switches: An Extended Finite State Machine based Approach

Contents

In the previous chapter, we have presented a novel formal testing approach to certify the correctness of the forwarding functionality of an SDN-enabled switch, i.e., we have considered the switch as a stateless system, defined by a set of pre-configured rules. Notwithstanding the effectiveness potential of the presented solution in detecting implementation forwarding errors, it does not cover the behaviour of the switch in its interaction with the controller.

In this context, we intend to address this *challenge* by proposing an appropriate model based approach that allows testing the interaction of the switch with the controller. For this purpose, the method has the potential of generating test suites with guaranteed fault coverage.

6.1 Introduction

The literature has mostly overlooked the problem of errors that might occur in the switch link with the controller. Nevertheless, unexpected bugs still happen at this level and must be addressed. Further on, the OF specification is extremely complex and lacks uniform standardization. For example, just the rule installation command ($Flow\_Mod$) is more than two pages long [113]. This might increase misinterpretation or cause conflicts due to multiple or duplicated requirements.

To address this challenge, in this chapter, we propose an EFSM based technique for test generation that aims to verify the OF switch-to-controller interaction with respect to requirements described in the OF specification. Correspondingly, the EFSM model is derived based on the OF requirements. Potential *incorrect* implementations are also modelled as EFSMs that are obtained by injecting specific types of (user-driven) faults into the model, i.e., through mutants' generation. For each mutant, a *distinguishing sequence* (DS) is sought that separates the original specification from the mutated one. We present an effective algorithm that derives a test suite TS formed by the corresponding distinguishing sequences.

Several approaches have been proposed in the literature for deriving conformance tests when the system specification is represented by an EFSM. On one hand, a group of these works have proven to be effective in deriving tests with guaranteed fault coverage. For example, these approaches have been proposed in the works by Bochmann et al. [20], El-Fakih et al. [49], and Petrenko et al. [118]. However, the author is not aware of the corresponding techniques being applied to tackle the correctness of SDN-enabled components. On the other hand, another group of works relying on EFSM for test generation have been proposed for SDN application area [160, 176], however, no related fault model has been introduced and thus, no fault coverage has been proven (see Chapter 3).

This chapter proposes to lessen this gap and advance the state of the art research on model based testing applied to SDN. An effective heuristic approach to derive distinguishing sequences for the specification EFSM and its mutants is presented for testing SDN switch-to-controller communication.

The main contributions of this chapter are the following. Firstly, we formalize a part of the OF requirements and propose an EFSM based test generation approach that is applied to an OF switch in its northbound interface (or controller southbound). Secondly, to demonstrate the effectiveness of the proposed approach, an experimental evaluation is performed. The evaluation aims at the assessment of the derived test suites fault coverage on one hand and at the execution of the derived tests against an OF implementation under test, on the other hand.

The conducted evaluation has shown on the one hand the effectiveness in terms of fault coverage of the derived test suites. Indeed, compared to randomly generated test suites, the ones derived by our approach have shown an average mutation score of 60.07% against 21.75% for the randomly generated TSs. Further on, the average mutation score is even significantly higher than the maximal score with random test suites. On the other hand, experiments have revealed several implementation faults and specification ambiguities when we have tested a switch implementation, namely Open vSwitch $2.5.0$. Examples of detected faults include several misbehaviours when rules with some specific values of the 'action' field of the $Flow\_Mod$ input have been installed, in addition, a misbehaviour in updating the statistics about the installed rules has been observed.

Section 6.2 describes the problem statement, the context of our contribution and its objectives. Section 6.3 describes the formal specification of the switch EFSM model. Section 6.5 presents the test generation approach and details the proposed algorithms. Section 6.6 defines the

evaluation metrics used to assess the performance of our solution, reports the evaluation results, and then presents the experimental results with an emphasis on the revealed errors with a switch implementation. Finally, Section 6.7 concludes the chapter.

6.2 Problem Statement

In an SDN architecture, the switch component not only assures a forwarding functionality to steer traffic flows in the data plane, but also interacts constantly with the SDN controller to acquire forwarding rules that determine this forwarding behaviour. The process of acquiring these rules defines the behaviour of the switch with respect to the controller. The interaction is performed via the controller southbound protocol. The protocol defines the communication between the control plane and the data plane. The agreed protocol (OpenFlow requirements) suggests ensuring the southbound communication and specifies the corresponding interchanged messages.
Nonetheless, the OF specification is extremely complex, lacks uniform standardization and is expressed in informal language. This might increase misinterpretation or cause conflicts due to multiple or duplicated requirements, etc. Therefore, all of these factors would result in a high likelihood of the implementations of the switch exhibiting diverging behaviours from their OF requirements.
We consider an OF switch in its communication with the controller as the system under test as shown in Figure 6.1. The system under test takes as input OF messages from the controller and outputs replies (OF messages) to the controller as specified by the OF requirements.
We propose to investigate how to guarantee the correct behaviour of the switch communications with the controller.

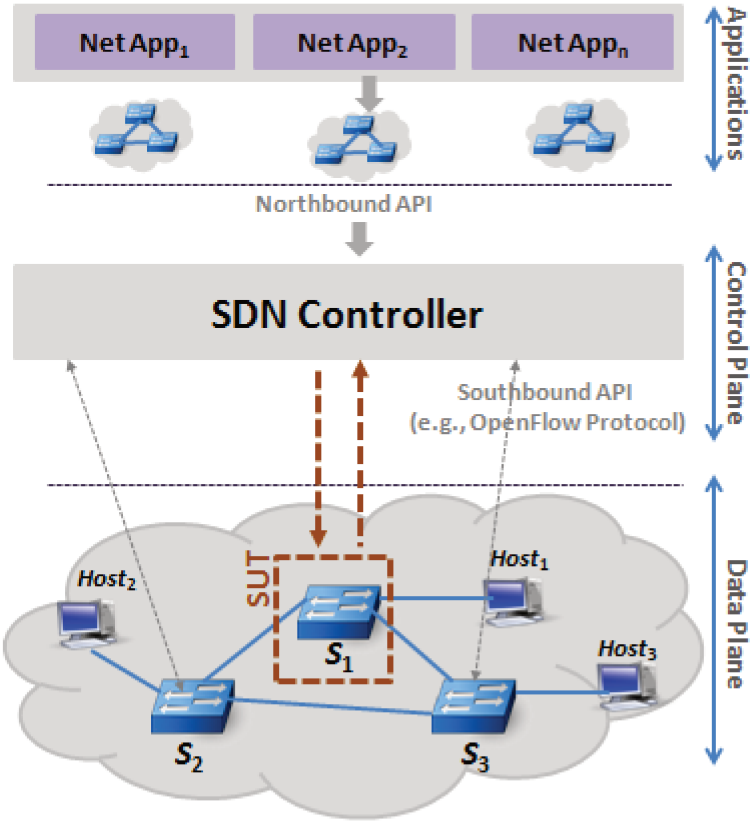


Figure 6.1 – Topology showing a switch-to-controller communication as the SUT

The research question we are addressing in this chapter can then be formulated as follows. *Given the switch specified by its communication with the controller, how to guarantee the correctness of its behaviour, i.e., how to ascertain the implementation of the switch meets the defined set of requirements?*

6.3 Extended Finite State Machine Model for an OF Switch

Formal models may be used as the basis for automating parts of the testing process and can lead to more efficient and effective testing [66]. Moreover, FSMs/EFSMs are widely used and have proven their effectiveness in various application domains, such as modeling and testing communication protocols, and other reactive systems. These formal models can be successfully adopted in specifying the properties of the OF switch and in capturing its functioning, particularly its communication with the controller. It is therefore of paramount importance to have a model which can capture the main interaction part between the controller and the switch, is able to model the main communication messages going from switch to controller (and vice versa), and can be easily extended to include additional parts of the OF specification or even the entire specification. The model proposed in this chapter is an attempt in that direction.

The proposed EFSM model (i.e., the specification $\mathcal{S}$) for the switch is derived from the OF requirements [113] only considering its interaction with the controller because in this chapter our goal is to test a switch at the controller southbound interface (and not at the data plane interface as achieved in Chapter 5).

The model is partially illustrated in Figure 6.2. The characteristics of the EFSM are depicted in Table 6.1. The model is composed of five states, two non parameterized inputs and nine parameterized inputs, eleven non parameterized and seven parameterized outputs. It contains also two context variables, seven output parameter functions, fifteen predicates and three context update functions. Note a level of abstraction in the model, for example not all inputs/outputs of the original protocol are modelled.

As the requirements do not describe precisely what the switch should reply in case of the success of a request received from the controller (e.g., response to a successful FLOW_MOD), in our model, we assume that the reply is a non-parameterized NULL_o output.

The sets of states *S*, inputs *X* and outputs *Y* are depicted in Equation 6.1.

$$
\begin{aligned}
S = &\{\text{CLOSED}, \text{WAIT\_HELLO}, \text{WAIT\_FEATURE}, \text{CONNECTION\_ESTABLISHED}, \text{FAIL\_MODE}\};\\
X = &\{\text{connected}, \text{HELLO}_i, \text{NULL}_i, \text{disconnected}, \text{FEATURE\_REQ}, \text{ADD}, \text{DELETE},\\
&\quad \text{MULTIPART\_REQ}, \text{BARRIER\_REQ}, \text{ECHO\_REQ}, \text{PACKET\_OUT}\};\\
Y = &\{\text{HELLO}_o, Error, \text{ERROR}_1, \text{ERROR}_2, \text{ERROR}_3, \text{ERROR}_4, \text{ERROR}_5, \text{ERROR}_6, \text{ERROR}_7,\\
&\quad \text{ERROR}_8, \text{ERROR}_9, \text{ERROR}_{10}, \text{MULTIPART\_REP}, \text{FEATURE\_REP}, \text{ECHO\_REP},\\
&\quad \text{BARRIER\_REP}, \text{NULL}_o, \text{FLOW\_REMOVED}\}.
\end{aligned}
\tag{6.1}
$$

The EFSM reflects that the switch supports connection version negotiation (parameterized input HELLO_i) and preserves the behaviour of correct exchange of FEATURE; ADD and DELETE (modelling the FLOW_MOD message); BARRIER; ECHO; PACKET_OUT and MULTIPART messages.

The handshake and version negotiation are handled by predicates of t_0, t_1 and t_2. For example, transition t_2 in Figure 6.2 is depicted in Equation 6.2 where WAIT_HELLO and WAIT_FEATURE are the initial and final states of t_2 respectively, HELLO$_i$ is the parameterized input. P_2 is the predicate checking the values of parameters *type* and *version* of the input. NULL$_o$ is the output sent by the switch representing a success of the HELLO$_i$ request.

$$t_2 = (\text{WAIT\_HELLO}, \text{HELLO}_i, P_2, -, \text{NULL}_o, -, \text{WAIT\_FEATURE}) \tag{6.2}$$

Once in the WAIT_FEATURE state and upon receipt of the parameterized input FEATURE_REQUEST (t_5 in Equation 6.3), if the predicate of t_5 is evaluated to True (indicating the non expiry of a timeout), the machine produces the parameterized output FEATURE_REPLY (containing the switch capabilities) and moves to the state CONNECTION_ESTABLISHED. In case FEATURE_REPLY is not sent after a timeout (predicate of t_3 evaluates to True), the machine moves to the initial state CLOSED indicating a disconnection (transition t_3 in Figure 6.2).

$$\begin{aligned} t_5 =&(\text{WAIT\_FEATURE}, \text{FEATURE\_REQUEST}, P_5, op_5, \text{FEATURE\_REPLY}, \\ &up_5, \text{CONNECTION\_ESTABLISHED}) \end{aligned} \tag{6.3}$$

Being in the WAIT_FEATURE state and upon receipt of one of the parameterized inputs CONNECTED or HELLO$_i$, the machine stays in the same state and produces non parameterized $ERROR_1$ and $ERROR_2$ replies respectively (transitions t_{30} and t_{31} in Figure 6.2).
Once the connection is successfully established and the machine being in the CONNECTION_ESTABLISHED state, some of the exchanged messages are modelled as self-looping transitions in state CONNECTION_ESTABLISHED (transitions from t_6 to t_{23}) including for example transition t_6 depicted in Equation 6.4.

$$t_6 = (\text{CONNECTION\_ESTABLISHED}, \text{ADD}, P_6, op_6, \text{NULL}_o, up_6, \text{CONNECTION\_ESTABLISHED}) \tag{6.4}$$

We note that the values of input/output parameters have rather small domains. For example, the input parameter *version* of HELLO$_i$ input takes only the single value $0x04$ and the input parameters for the input ADD are $[type \quad table \quad match \quad action \quad flags]$ and their values have small domains as well. The same for output parameters.
The machine has the set of two context variables; $C_v = \{nb\text{FLOWS}, \text{TLS\_}timeout\}$ denoting respectively the number of rules in the switch and the TLS session timeout. $nb\text{FLOWS}$ can take distinct values in the range $[0..max\_entries]$ where $max\_entries$ is a constant that denotes the maximal number of rules the switch can insert and depends on the switch characteristics (takes the value of 10^6 in our model). In case TLS_$timeout$ expires, the switch loses the connection with the controller and the machine moves to state CLOSED (transition t_4 in Figure 6.2). The OF requirements specify that the connection maintenance is done by the underlying TLS/TCP connection mechanisms and since currently supported protocols have the same default timeout value of 300 seconds, we set TLS_$timeout$ to this value.

No. of states	No. of transitions	No. of predicates	No. of update functions
5	31	15	3

Table 6.1 – The characteristics of the EFSM switch model

Note that the finite state model proposed in this chapter can be extended to model additional parts of the OF requirements, or even the entire specification. For example, the configuration

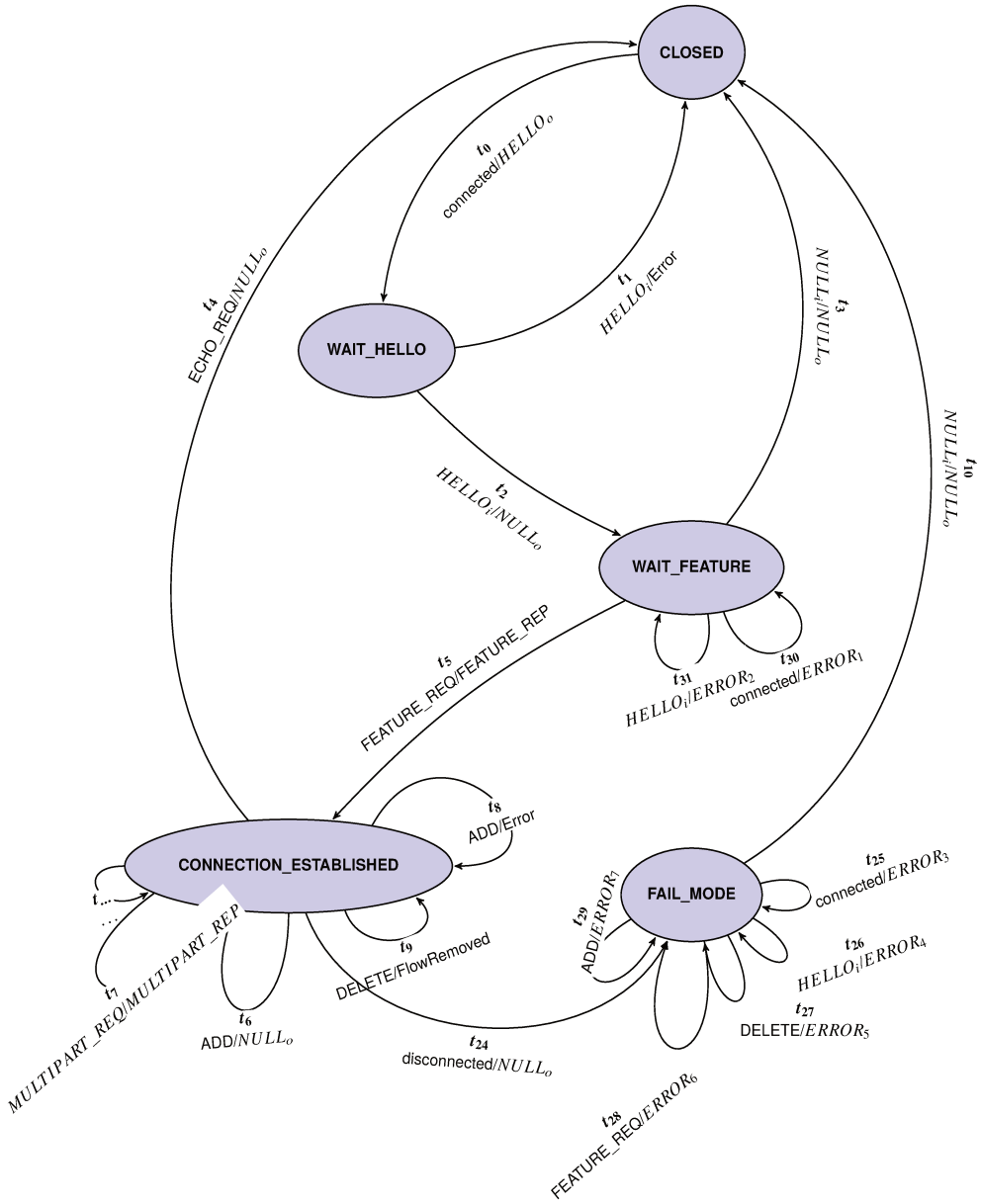


Figure 6.2 – Part of the specification EFSM of the switch

messages handling groups, queues, and meters can be added as self-looping transitions after the connection establishment, or other states can be added as well. The model allows the detection of potential faults and misinterpretations as it will be shown by our experiments. However, its expanding would eventually require more significant upfront time. Naturally, an evaluation of such expanding is needed and hence can form a direction of future work.

6.4 Introducing User-Defined Mutations and a Fault Model

In this work, we assume that the specification machine $\mathcal{S}$ has a set of selected transitions, referred to as *suspicious* that are defined by a 'user' (can be an expert, tester, developer, etc.). We focus on output, transfer, predicate and update function faults at these transitions.
Given a 'suspicious' transition $t = (s_i, x, P, op, y, up, s_j)$ of $\mathcal{S}$, t has:

- An *output fault* if its (parameterized) output is distinct from that specified in $\mathcal{S}$.

- A *transfer fault* if its final state is different from that specified by $\mathcal{S}$.
- An *update function fault* if its update function is omitted.
- A *predicate fault* if its predicate is negated[1].

Note that the introduced faults could be more sophisticated. For example, one can consider altering the operators of an update function. Nonetheless, even with these types of faults, we show that our approach is able to detect faults in an OVS implementation. Also, we note that in this work, only first-order mutants are considered.
As in Chapter 5, we introduce a fault model as the tuple of the specification, conformance relation and fault domain, $\langle \mathcal{S}, \simeq, \mathcal{FD} \rangle$ [120].
In this chapter, the specification machine $\mathcal{S}$ is represented by an EFSM, and the conformance relation $\simeq$ is the *quasi-equivalence*, i.e., an implementation $\mathcal{M}$ conforms to the specification $\mathcal{S}$ if for each input sequence for which $\mathcal{S}$ is defined, $\mathcal{M}$ produces the same output sequence as defined by $\mathcal{S}$. $\mathcal{FD}$ is a set of implementation machines; faulty implementations are simulated by the mutants of interest.
As usual, we are interested in deriving *exhaustive* test suites for $\langle \mathcal{S}, \simeq, \mathcal{FD} \rangle$, i.e., such test suites that detect each non-conforming (faulty) implementation $\mathcal{M} \in \mathcal{FD}$.

6.5 EFSM based Technique for Test Generation

Definition 6.1.

(I) A *path* in $\mathcal{S}$ is the set of (parameterized) inputs of the successive transitions that are enabled from one configuration to another.

(II) Let $\mathcal{S}$ have the initial configuration $(s_0, \mathbf{v}_0)$. A *test sequence* is the sequence of input / output pairs of $\mathcal{S}$ that starts from $(s_0, \mathbf{v}_0)$ to a given configuration $(s, \mathbf{v})$ (i.e., a path from $(s_0, \mathbf{v}_0)$ to $(s, \mathbf{v})$).

Definition 6.2.
Given a (parameterized) input sequence α, if α is defined at the initial states for machines $\mathcal{S}$ and $\mathcal{M}$, configurations $(s, \mathbf{v})$ of $\mathcal{S}$ and $(s\prime, \mathbf{v}\prime)$ of $\mathcal{M}$ are *distinguishable* by α (DS) if the (parameterized) output sequences produced respectively by $\mathcal{S}$ and $\mathcal{M}$ in response to α are different, i.e.,

$$out(\mathcal{S}, \alpha) \neq out(\mathcal{M}, \alpha).$$

We furthermore refer to $\mathcal{S}$ as the specification machine describing the desired behaviour of an OF switch, while $\mathcal{M}$ denotes a potential faulty implementation of it. $\mathcal{M}$ is *distinguishable* from $\mathcal{S}$ (can be detected) if there exists such a DS α. Otherwise, $\mathcal{M}$ is *quasi-equivalent* to $\mathcal{S}$.

Our approach is a depth first search (DFS) based heuristic that progressively constructs a test suite TS. As a first step, a set of *user-defined* mutants for suspicious transitions is derived.

[1]The specification can stay deterministic if other outgoing transitions are mutated accordingly. In our model, no more than two transitions with the same (parameterized) input (and different predicates) are defined at a state, thus when introducing a predicate fault we can simply change the predicates assigned to such two outgoing transitions. We assume this mutant is still of first order because it can correspond to a single fault in an implementation, e.g., unintentional swapping of the 'if-then-else' statements.

Some examples of the hand-seeded faults are as follows. An example considers a variable that exceeds its extreme value, for the very simple reason that software developers often just forget to check such condition. Another example considers a type of fault that consists in swapping the predicates of two outgoing transitions from the same state with same (parameterized) input which corresponds to developers unintentionally swapping the condition to check (in the 'if-then-else' statements for example). Another relevant example is linked to the coding activity such as potential mistakes in the action part in the $Flow\_Mod$ message or small differences in the matching fields which might 'tickle' subtle bugs [26]. Other faults concern misunderstanding or misinterpretation of the requirements. Such faults are deliberately seeded into the specification.

Then as a second step, for each mutant $\mathcal{M}_i$ of the set, for checking the distinguishability between a reached configuration $(s, \mathbf{v})$ where the suspicious transition is defined in $\mathcal{S}$ and the corresponding configuration reached in $\mathcal{M}_i$, the approach tries to append the DSs if they exist up to a certain predefined positive depth l.

The construction of a distinguishing sequence, say σ, of $\mathcal{S}$ and a given $\mathcal{M}_i$ is performed in the following way. It is formed of a preamble α and a postamble that is appended to α to form the $distSeq$. The preamble α is an input sequence that takes $\mathcal{S}$ from the initial configuration $(s_0, \mathbf{v_0})$ to the configuration where the suspicious transition is defined. The postamble $distSeq$ is constructed by a depth first search that will repeatedly expand deeper configuration nodes in the EFSM and explore the successive configurations. In other words, configurations at depth l (which corresponds to length $|\alpha| + l$ starting from the initial configuration) are treated as if they have no outgoing transitions (successors). The algorithm therefore progressively increases the depth until it finds that the outputs of $\mathcal{S}$ and $\mathcal{M}_i$ are different or that the limit l for the depth is reached (i.e., the length $|\alpha| + l$ is reached).

The approach is formalized in the following algorithms.

Algorithm 5 captures the main flow of the proposed approach. It first assigns TS and $\mathcal{FD}\prime$ to empty sets. Then for each mutant $\mathcal{M}_i$ in the set $\mathcal{FD}$, it looks for the corresponding DS. Since our EFSM is initially connected, we are only interested in configurations that are reachable from the initial configuration. A preamble α is generated to first reach the suspicious transition.

The specification $\mathcal{S}$ is then simulated over α and the sequence σ is first empty and is later extended with the (parameterized) sequence distSeq if it exists to satisfy the distinguishability between the two configurations reached in $\mathcal{S}$ and the mutant respectively.

Therefore, for a $length$ going from $|\alpha|$ to $(|\alpha| + l)$, i.e., from the depth of the starting suspicious transition to the defined depth limit, the algorithm repeatedly calls DistinguishingSequenceAppend illustrated in Algorithm 6 which takes care of suitable extensions. Eventually, Algorithm 5 will find a DS if one exists up to depth l (i.e., up to length $(|\alpha| + l)$ from the initial configuration). Finally, if such sequence exists, it is added to TS while the mutant is marked as distinguishable, i.e., added to the set $\mathcal{FD}\prime \subseteq \mathcal{FD}$. When all derived mutants are considered, TS is returned along with $\mathcal{FD}\prime$ containing distinguished mutants.

Proposition 6.3.
If Algorithm 5 returns a test suite TS then this test suite is *exhaustive* with respect to the fault model $\langle \mathcal{S}, \simeq, \mathcal{FD}\prime \rangle$.

Proof.
Indeed, $\mathcal{FD}\prime$ contains only distinguishable mutants. Moreover, the derived test suite TS is formed by all the input sequences that distinguishes the specification $\mathcal{S}$ and each mutant of the set of mutants $\mathcal{FD}\prime$. Therefore, each faulty implementation / mutant is detected by the derived test suite TS. □

Algorithm 5: EFSM based Approach: Algorithm that derives a test suite TS and a set $\mathcal{FD}'$ of distinguishable mutants

Input : EFSM $\mathcal{S}$, initially connected and deterministic
A set $\mathcal{FD}$ of mutants
An upper bound positive integer $l > 0$

Output : A test suite TS and a set of mutants $\mathcal{FD}'$ that can be detected or $NoSolution$

1 $TS \leftarrow \emptyset$
2 $\mathcal{FD}' \leftarrow \emptyset$
3 **foreach** $\mathcal{M}_i \in \mathcal{FD}$ *representing a fault at a 'suspicious' transition* $t_i = (s_i, x_i, P_i, op_i, y_i, up_i, s_i')$ **do**
4 Let α be a (parameterized) input sequence that takes $\mathcal{S}$ from the initial configuration $(s_0, \mathbf{v_0})^{\mathcal{S}}$ to the configuration $(s_i, \mathbf{v_i})^{\mathcal{S}}$ where the predicate P_i of t_i can be evaluated to True and a (parameterized) input x_i is defined
5 Simulate $\mathcal{S}$ applying α and reach configuration $(s_i, \mathbf{v_i})^{\mathcal{S}}$
6 $\sigma \leftarrow empty$
7 **for** $length \leftarrow |\alpha|$ **to** $(|\alpha| + l)$ **do**
8 DistSeq ← DistinguishingSequenceAppend($\mathcal{S}$,$\mathcal{M}_i$, $(s_i, \mathbf{v_i})^{\mathcal{S}}$, $length$) ▷ depth limited search
9 **if** *DistSeq* $\neq NoSolution$ **then**
10 $\sigma \leftarrow \sigma$.DistSeq
11 **if** $|\sigma| > |\alpha|$ **then**
12 $TS \leftarrow TS \cup \{\sigma\}$
13 $\mathcal{FD}' \leftarrow \mathcal{FD}' \cup \{\mathcal{M}_i\}$
14 **if** $|TS|$==*0* **then**
15 **return** $NoSolution$
16 **return** TS *and* $\mathcal{FD}'$

Example

Below, we show an example of a mutant and the corresponding distinguishing sequence DS returned by Algorithm 5.

Consider the transition depicted in Equation 6.6 where up_6 indicates that $nb\text{FLOWS}$ is increased by one. A mutant M_0 considers an issue that sometimes developers forget to verify the update of a variable. The idea here is that M_0 is derived by omitting the updating function up_6, shown in Equation 6.5, of t_6.

$$\begin{aligned} &up_6 : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+ \\ &nb\text{FLOWS} \leftarrow nb\text{FLOWS} + 1 \end{aligned} \tag{6.5}$$

It suggests the switch will not update its statistics after adding a new rule. The corresponding

Algorithm 6: DistinguishingSequenceAppend($\mathcal{S}$,$\mathcal{M}$, $(s_i, \mathbf{v_i})$, $length$)

Input : Initially connected and deterministic EFSMs $\mathcal{S}$ and $\mathcal{M}$
A configuration $(s_i, \mathbf{v_i})$
An upper bound positive integer $length$

Output : A test sequence or $NoSolution$

```
1  NoSolutionHappen ← False
2  seq ← input sequence of the path up to (s_i, v_i)
3  if out(S, seq) ≠ out(M, seq) then
4      return seq
5  if |seq|==length then
6      return NoSolution
7  foreach transition t = (s_i, x_i, P_i, op_i, y_i, up_i, s_i') outgoing from the configuration (s_i, v_i)
     with P_i evaluating to True do
8      successor ← (s_i', v_i') ;        ▷ v_i' is the result of up_i applied to v_i
9      path ← DistinguishingSequenceAppend(S,M, successor, length)
10     if path == NoSolution then
11         NoSolutionHappen ← True
12     else
13         if path ≠ NoSolution then
14             return path
15 if NoSolutionHappen then
16     return NoSolution
```

distinguishing sequence σ returned by Algorithm 5 is shown in Equation 6.7.

$$t_6 = (\text{CONNECTION\_ESTABLISHED}, \text{ADD}, P_6, op_6, \text{NULL}_o, up_6, \text{CONNECTION\_ESTABLISHED}) \quad (6.6)$$

$$\begin{aligned}\sigma = &\text{connected}(\text{OFPT\_HELLO}, 0x04)\\ &\text{HELLO}_i(\text{OFPT\_HELLO}, 0x04)\\ &\text{FEATURE\_REQ}(\text{FEATURE\_REQUEST}, 0x04)\\ &\text{BARRIER\_REQ}(\text{BARRIER\_REQUEST})\\ &\text{ADD}(\text{OFPT\_FLOW\_MOD}, 0, any, 1, 0)\\ &\text{BARRIER\_REQ}(\text{BARRIER\_REQUEST})\\ &\text{MULTIPART\_REQ}(\text{OFPMP\_TABLE})\\ &\text{BARRIER\_REQ}(\text{BARRIER\_REQUEST}).\end{aligned} \quad (6.7)$$

6.6 Experimental Evaluation for Testing an OF Switch

6.6.1 Evaluation of the Approach

In the first part of the evaluation, given the depth l to which Algorithm 5 is allowed to explore the generated transitions tree starting from the suspicious transition of interest, we investigate the impact of this depth on the effectiveness of the generated test suites (fault coverage). For this purpose, we vary the depth l and we generate corresponding test suites, then we measure the mutation score of each generated test suite.

In the second part, we further assess the effectiveness of the test suites derived by the proposed approach by comparing them to test suites generated randomly, i.e., an approach that randomly selects inputs from the input set to form test suites. We refer to this approach as RG (Random Generation). The goal is to compare the test suites derived by our test generation method with size-equivalent test suites that do not follow any systematic test generation strategy. Though this provides only a baseline and a comparison with alternative testing methods is definitely relevant, it is a necessary starting point.

In order to perform an experimental evaluation, a number of software tools have been developed. The experimentation process is composed of four main steps. The first step is to generate mutants of different kinds. The second step is to generate the test suites based on the two approaches, namely the proposed EFSM based approach and the random test generation approach (RG). The third step of the experiment is to produce JUnit files that can run the test suites. The fourth and final step is running the generated test suites against the generated mutants.

To conduct our evaluation, automation of test suite generation is paramount. Alternatively, one can produce them manually though this is very time consuming and likely error-prone. Similarly, mutant generation must be automated. However in our experiments, we generate two sets of mutants. A first set of user-defined mutants are manually generated and added to the second set of automatically (randomly) generated ones.

We have therefore developed a testing framework. The framework is composed of two main modules. The first module allows the test engineer to produce test suites according to two different test generation methods; using our proposed approach and using the random generation approach (RG). The mutants have been generated and the test suites have been automatically constructed. The second module of the testing framework helps executing the randomly generated test suites against the different generated mutants.

Note that the random test suites generation is based on the random function provided by the Java programming language.

We discuss our measurement of performance and effectiveness (at finding faults) using two metrics as follows. On one hand, concerning the *performance*, we focus precisely on the execution CPU time metric. On the other hand, to assess the effectiveness of our testing technique, we use *fault coverage*, specifically mutation analysis is used to compare the capability (of fault detection) of the derived test suites based on our approach against the capability of randomly generated test suites in terms of the number of killed mutants.

One issue to be addressed is the detection of equivalent mutants, i.e., mutants that have the same behaviour as the specification machine and therefore cannot be killed by test sequences. There are studies proposing techniques to automate the detection of equivalent mutants, and a commonly used heuristic is to consider survived mutants not killed by any test suite in the overall test pool (i.e., in test suites being compared) as equivalent mutants [39]. In this work, we have used this heuristic. Since we compare test suites between one another, this assumption should not introduce a significant threat to validity of our results.

In our experiments, we produce output, transfer, predicate and update function mutants.
We have created a total of 288 mutants where 70 have been manually generated. 20 of them (not killed by any of our test suites) have been manually checked to be equivalent. Overall, we have therefore used 268 mutants where 67 are output, 67 are transfer, 67 are predicate, and 67 are update function. 10 test suites have been generated using our approach. This has been performed by increasing the depth l until 10 and for each fixed depth $l = \{1, \ldots, 10\}$, a corresponding test suite has been generated.
To generate size-equivalent random test suites, for each generated test suite using our approach, we have measured the average length of all of its test sequences. For each of these length values, a corresponding test suite of that same length has been randomly generated using the RG approach. Hence, 10 test suites have been randomly generated.
Figure 6.3 illustrates the mutation score as the depth increases. The mean mutation score and the execution CPU time (in minutes) are shown in Figure 6.4.

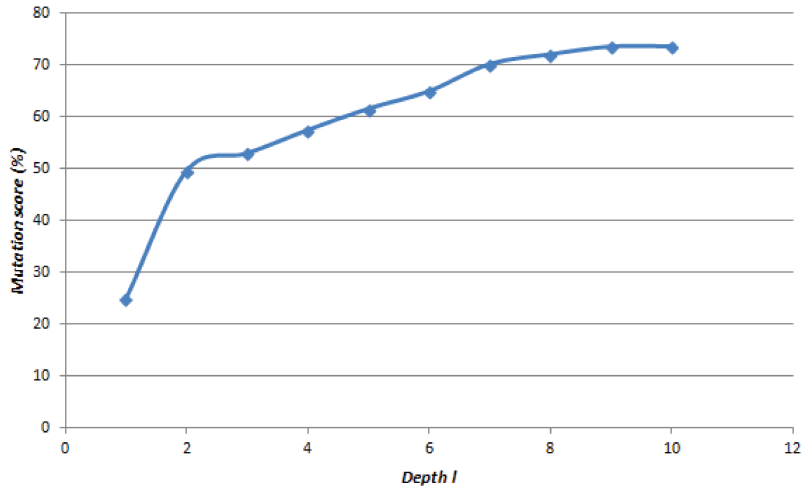


Figure 6.3 – Mutation score as the depth increases

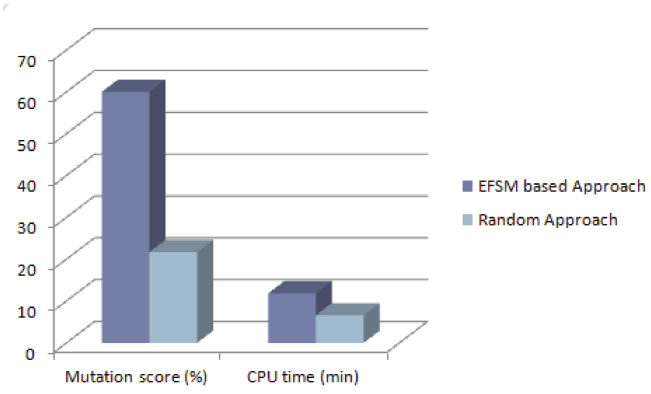


Figure 6.4 – Average mutation score and execution time for TSs derived by the proposed approach Vs TSs randomly generated

The choice of the depth l, to limit the test sequence length, impacts the number of detected mutants and hence detected faults as shown in Figure 6.3. We can observe that the mutation score increases when l increases.
The results from randomly generated test suites show a mean mutation score of 21.75% (Figure 6.4). This is significantly lower than the 60.07% average that has been obtained with the

proposed EFSM based method (with the depth l varying from 1 to 10). The average mutation score is even significantly higher than the maximal score with random test suites. We can then conclude that, factoring out the cost of testing, the EFSM based testing technique is rather relevant. It is able to outperform the effectiveness of randomly generated test suites and achieve the highest mutation score. We conjecture that this is due to not only the ability of our technique to cover more transitions in the EFSM model, but also to its ability to distinguish between configurations which the random approach does not do.
However, as shown in Figure 6.4, the CPU time that our proposed technique requires compared to the RG approach is high. Therefore, there is a trade-off between the performance and the quality of the generated test suites using the two methods.
Our main conclusion is that the proposed EFSM based heuristic is reasonably effective at detecting faults (60.07% average as opposed to 21.75% for size-equivalent random test suites).
The next subsection presents the evaluation of the effectiveness of the EFSM based approach in revealing OF switch implementation faults.

6.6.2 Experiments on Testing an OVS Implementation

To evaluate the effectiveness of the EFSM based approach in revealing OF switch implementation faults, we have developed another module part of our testing framework including two main components. First, there is a translation module that maps the generated test sequences described in Section 6.5 into OF-syntax messages. It uses an input file specifying the derived inputs to fill the different input parameter values and construct a TS ready to be executed against the SUT. A second component takes care of the comparison of observed and expected outputs specified in the input file as well. The framework allows to send TSs to the OVS, collect the observed outputs and compare them against the specified expected ones to finally print a conclusion displaying 'OK' or 'Fail' message to the user.
Experiments have been performed on OVS version $2.5.0$. As in the previous chapters, Mininet tool has been utilized. In the testing set up, Mininet has been executed under Mint 18.1 with 8GB of RAM and 1 core of a 2.4 GHz Intel Core i7. The Floodlight 1.1 controller has been used. Communication with the SUT happens at the level of the southbound interface. The SUT is connected to a testing engine containing an emulated controller that is capable of sending and capturing OF messages in a controlled manner.
Figure 6.5 shows the SUT (VM_1) and the testing engine (VM_2). The source code of OVS $2.5.0$ has been directly cloned from its Git repository on the virtual machine VM_1. The SUT may receive any of the inputs from the input alphabet X emitted from the controller set on the virtual machine $VM2$.
The test suites TSs of interest have been executed against the OVS implementation. In the following, we present the bugs detected in OVS $2.5.0$ implementation along with the related DSs and their lengths.
The existence of a fault in OVS 2.5 implementation has been revealed by a test sequence (DS) of one of the generated TSs. It concerns the installation of a new rule with parameter $action$ set to 'OFPActionPushVlan' which pushes a new VLAN tag onto the packet matching the entry. In this case, the SUT has replied with an $error$ message with parameters $type$ = OFPET_BAD_ACTION and $code$ = OFPBAC_BAD_ARGUMENT. The observed reply reveals a fault in the SUT in association with installing a new rule having an action that pushes a new VLAN tag to the matching packet. The switch has behaved as if an action in the $Flow\_Mod$ message ADD had a value that is invalid, however the 'OFPActionPushVlan' action is specified to be supported. The length of the DS capable of detecting such fault is 7.
Another fault has been detected by another test sequence and concerns the installation of a

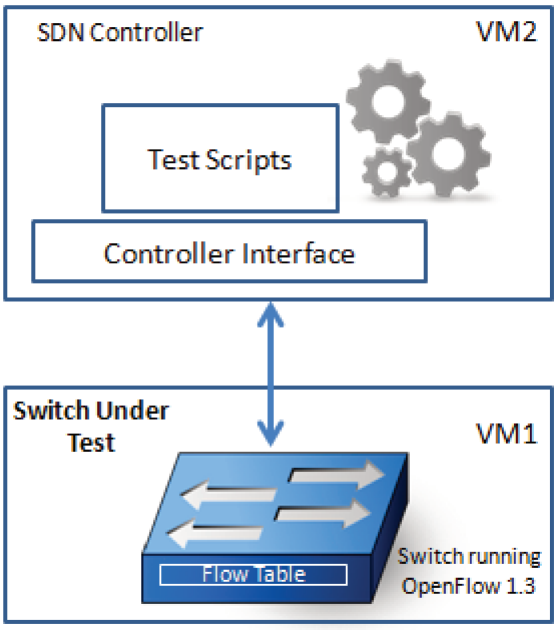


Figure 6.5 – Testbed framework for an OF switch analysis

new rule with parameter $action$ set to OFPP_ALL which is supposed to forward the matching packet to all ports of the switch under test. In this case, the SUT has replied with an $error$ message as well instead of a NULL_o reply. The parameters of the observed $error$ reply are $type$ = OFPET_BAD_ACTION and $code$ = OFPBAC_BAD_ARGUMENT. In this case, the length of the DS capable of detecting this fault is 7 as well.

A similar fault has been detected which concerns the installation of a new rule with parameter $action$ set to OFPP_IN_PORT. When the SUT receives a $Flow\_Mod$ message with the indicated parameter, it is supposed to reply with a NULL_o reply and install the rule which forwards the matching packet to all ports except the input one. However, the observed reply has been an $error$ message similar to the previously mentioned one. The length of the DS capable of detecting such fault is also 7.

The next detected fault concerns the MULTIPART_REQ for reporting statistics about installed rules. The input sequence $\alpha \in TS$ is intended to add a rule to the SUT (when $nbFlows$ is less than $max\_entries$) and request the SUT about statistics. The expected reply should show increment of the total number of rules $nbFlows$ by one. However, the SUT reply contains the exact same number of rules than before applying the DS. This means that the switch when adding the rule to its table, does not update statistics. The DS capable of detecting such fault is derived based on an update function mutant and it contains a $Flow\_Mod$ input and a MULTIPART_REQ input for adding a new rule and then getting the statistics. The $type$ parameter of the MULTIPART_REQ is set to OFPMP_TABLE. The reply to this DS contains the active number of installed rules. The DS in this case has length 8.

Another type of fault has been detected as well. It concerns a table overflow. The DS of interest includes the parameterized input ADD 10^6 times followed by another input ADD. In this case, the SUT has replied with 10^6 NULL messages followed by a NULL and has added the 'extra' rule. Note that in section 6.4 of the OpenFlow requirements [113], it is stated that "If a switch cannot find any space in the requested table in which to add the incoming flow entry, the switch must send an ofp_error_msg with OFPET_FLOW_MOD_FAILED type and OFPFMFC_TABLE_FULL code". This indeed reports a fault in the OVS 2.5 under test and confirms our intuition that in the implementation, the variable $nbFlows$ is not checked for reaching

its extreme value $max\_entries$. The DS has length $10^6 + 6$. Thus it has not been derived by our implementation but rather derived manually using our model. Once the configuration of interest is reached in the model, the transition with the parameterized input ADD and having the predicate $[nbFlows < max\_entries]$ has been simulated to be traversed 10^6 times.

Discussion

There are several threats that could potentially affect the validity of our study.
One of the threats susceptible to affect our study is the one referring to generalizability of our findings. In this preliminary study, it is clear that the results we have obtained are a priori limited as they are based on one case study (as a baseline) involving the comparison of our approach to the RG approach. Therefore, more investigations and comparisons are necessary in order to be able to generalize.
Another point worth mentioning is related to checking the scalability of the proposed approach with respect to the size and complexity of the built model. In other words, if the proposed model is extended to incorporate additional parts of the OF requirements, it would be interesting to investigate how the complexity of the proposed test derivation algorithm will grow accordingly. This question also forms a direction of planned future work.
In contrast, as we have used mutation score being a surrogate metric of evaluating the effectiveness at detecting faults and as a test suite mutation score has proven to be correlated with its real fault detection rate [3], we believe there is little threat to the validity in general. Besides, we have experimented on an OF switch implementation which has allowed us to detect some faults.

6.7 Chapter Conclusions

In this chapter, an EFSM based testing technique for an OF switch-to-controller communication has been presented. Given an EFSM model of the switch-to-controller interaction derived from OF requirements, and a set of mutants representing faults defined by a user at suspicious transitions, the method derives a test suite formed by *distinguishing sequences* aiming at detecting the mutants of interest.
We have evaluated the proposed test derivation technique by comparing it to a random generation approach. The results have shown that the designed algorithm is efficient compared to the random algorithm. Further, preliminary experiments with Open vSwitch have confirmed the effectiveness of the proposed approach in revealing some implementation faults. However, the findings presented in this work should be interpreted in the context of limitations related to the model design decisions and the number of *user-driven* mutants. In the future, we plan to improve the operation efficiency of the approach and apply it to larger SDN models on one hand and compare it to other (EFSM based) test generation techniques on the other hand.
After presenting novel model based testing techniques for testing the switch in its forwarding functionality and in its interaction with the controller, we turn to another crucial SDN component, namely the SDN controller. The next chapter proposes an adaption of the model based technique presented in Chapter 5 to a module of the controller. In fact, we consider a specific module of the controller responsible for translating requested virtual links into pairs of ports through a given switch. We identify that the module of interest exhibits a 'stateless' behaviour. Therefore, appropriate logic circuit is designed to model its behaviour in order to derive high quality test suites.

7

Test Derivation for a Controller Application: An Adaptation of the Logic Circuit based Approach

Contents

Having successfully addressed the challenges of testing an SDN switch in its data plane interface functionality and in its OF interface interaction with an SDN controller in the previous chapters, we now tackle another critical SDN component, particularly the controller. Indeed, to ascertain the correct implementation of the controller, it is important to guarantee the correct behaviour of its modules / applications.

The testing method presented in the current chapter is related to the one of Chapter 5 since the logic circuit notions introduced there, are used here. In particular, this chapter presents a proposal for the adaptation of the model based test generation technique elaborated in Chapter 5 to test one module of an SDN controller that is responsible for 'translating' requests from the application layer into virtual links in the data plane level. The testing technique relies on *Logic Circuits*.

7.1 Introduction

The controller in an SDN architecture is a core SDN component responsible for making decisions on managing forwarding devices in the underlying data plane. An SDN controller monitors the state of the network by gathering statistics from forwarding devices, makes the global routing decisions, and reacts on events such as link congestion. In order to fulfill the long list of tasks, controllers have grown to be rather complex pieces of software, consisting of more than a million lines of code [17], [104]. The proposals put forth for different controllers in the literature do not modify the basic controller architecture and modules, rather they differ in terms of features and capabilities.

The core sub-modules of the controller are mainly related to topology and traffic flow. The 'link discovery' sub-module for example regularly transmits inquiries on external ports utilizing $Packet\_Out$ messages. These inquiry messages return in the from of $Packet\_In$ messages, which allows the controller to build the topology of the network [113]. The topology itself is maintained by the 'topology manager' sub-module. This provides the 'decision making' sub-module to find optimal paths between nodes of the data plane. The paths are built such that the different requests / policies can be implemented during the path installation. In addition, the controller may also have 'statistics manager' and 'queue manager' for collecting performance information and management of different incoming and outgoing packets, respectively. 'Flow manager' is another sub-module which directly interacts with the forwarding devices and installs rules in their flow tables. It utilizes southbound interface for this purpose [113]. Note that the controller also has other sub-modules and modules which are not mentioned here.

The controller as a large and complex software inevitably contains bugs that may disrupt its functioning when triggered.

In previous research, the author et al. have proposed and discussed the use of finite state models for optimizing and testing SDN controller sub-modules / components [19]. Namely, in the mentioned work, the 'queue manager' or 'scheduler' sub-module has been taken as a running example and a discussion on the properties of appropriate non-classical state models has been introduced. The author et al. have presented how optimization of the non-functional parameters and testing of the functional ones can be performed when a state model describes the controller 'queue manager' sub-module. In contrast, in this chapter, the author discusses the use of another appropriate model and a corresponding test generation approach for another controller module, further, only functional aspects of the module of interest are considered in the current work.

In the current chapter, we are rather interested in the specific behaviour of the controller module responsible for the translation of a *requested virtual link* with respect to a given switch in the data plane into a rule[1]. A requested virtual link with respect to a switch is specified by a pair of nodes of the topology. More precisely, we are interested in testing the behaviour of the controller module that given a requested virtual link with respect to a given switch, implements it through this switch. In other words, a virtual link is created, i.e., a rule is pushed in the corresponding switch such that the traffic is forwarded accordingly. We refer to this specific module of the controller as $Link\_Translator$.

Note that the functionality of the module of interest, i.e., the $Link\_Translator$, is implemented in many actual controllers. It involves, among others, the aforementioned 'device manager', 'topology manager' and 'flow installation manager' sub-modules. For example, in the Floodlight controller [1], the $Link\_Translator$ is the core module behind the controller application

[1]Note that the implementation of such virtual link in the underlying $RNCT$ is reflected by the installation of one rule in the switch of interest and hence the creation of a 'virtual' edge from the previous node to the next node (specified in the request) through the switch.

referred to as ‘circuit pusher’. In fact, the ‘circuit pusher’ takes as input a pair of hosts, it computes a path between them and pushes a corresponding rule in each switch of that path such that a virtual link is implemented with respect to each switch of the path.
We discuss the adaptation of the model based testing technique proposed in Chapter 5 to test the described controller module, i.e., the $Link\_Translator$. We first formalize the behaviour of the module and present an algorithm for logic circuit synthesis that builds the corresponding specification. This model can later serve to generate test cases.
The author is aware that an experimental evaluation is needed to study the effectiveness of the proposal and the effectiveness of the derived test suites. This part is left open for further investigations in future work.
Section 7.2 states the problematic the chapter is addressing. Section 7.3 formalizes the controller module under test. Section 7.4 presents the circuit logic synthesis solution for the controller $Link\_Translator$ module of interest. Section 7.5 presents a proposal on the test generation approach. Section 7.6 discusses the test suites execution. Finally, Section 7.7 concludes the chapter.

7.2 Problem Statement

In Chapter 4, we have represented a request emitted from end-users (applications) as a path (e.g., from $Host_A$ to $Host_B$ as shown in Equation 7.1). In this work, the requested virtual links shown in Equation 7.2 and corresponding to that path need to be created / implemented. These requested virtual links are of the form of triples (u, S, v) including always a node of type switch S that is linked with two other nodes u and v that can be of type switch or host.

$$(Host_A, S_0)\,(S_0, S_1)\,\ldots(S_i, S_{i+1})\,\ldots\,(S_{k-1}, S_k)\,(S_k, Host_B) \tag{7.1}$$

$$(Host_A, S_0, S_1)\,(S_0, S_1, S_2)\,\ldots(S_{i-1}, S_i, S_{i+1})\,\ldots\,(S_{k-2}, S_{k-1}, S_k)\,(S_{k-1}, S_k, Host_B) \tag{7.2}$$

For example, in $Path_1$, shown in Equation 7.3 (one of the paths of the data plane in Figure 7.1), the requested links can be represented as $(Host_1, S_1, S_2)$, (S_1, S_2, S_3) and $(S_2, S_3, Host_3)$.

$$Path_1 = (Host_1, S_1)\,(S_1, S_2)\,(S_2, S_3)\,(S_3, Host_3) \tag{7.3}$$

The $Link\_Translator$ module receives a request from a given application as a requested virtual link for any pair of nodes. A requested virtual link in this case extends the definition of a virtual link defined in Chapter 4 by considering any given nodes in the topology. More formally, the $Link\_Translator$ receives a requested virtual link of the form (u, S, v) for any given nodes u and v.
The request is received from a given application (in the application layer). Then, $Link\_Translator$ translates that requested link with respect to a switch S. This is performed by assigning a pair of ports of S linking it to its neighbours in such a way that there exists a path between the initial pair of nodes (u, v) (specified by the request) that passes through S via its neighbours (note that this comes down to the translation of a virtual link in its definition specified in Chapter 4). The $Link\_Translator$ outputs a rule represented here by a pair of ports specifying (i) the input port of the rule as the one of the switch S linking it to the previous node (u) corresponding to the requested link; (ii) the output port specified by the rule action as the port linking the switch S to the next node (v) corresponding to the requested link. Therefore, the implementation of a requested virtual link with respect to a switch S means the installation of a forwarding rule in S assigning correctly the ports of S.
The topology showing the $Link\_Translator$ module is depicted in Figure 7.1.

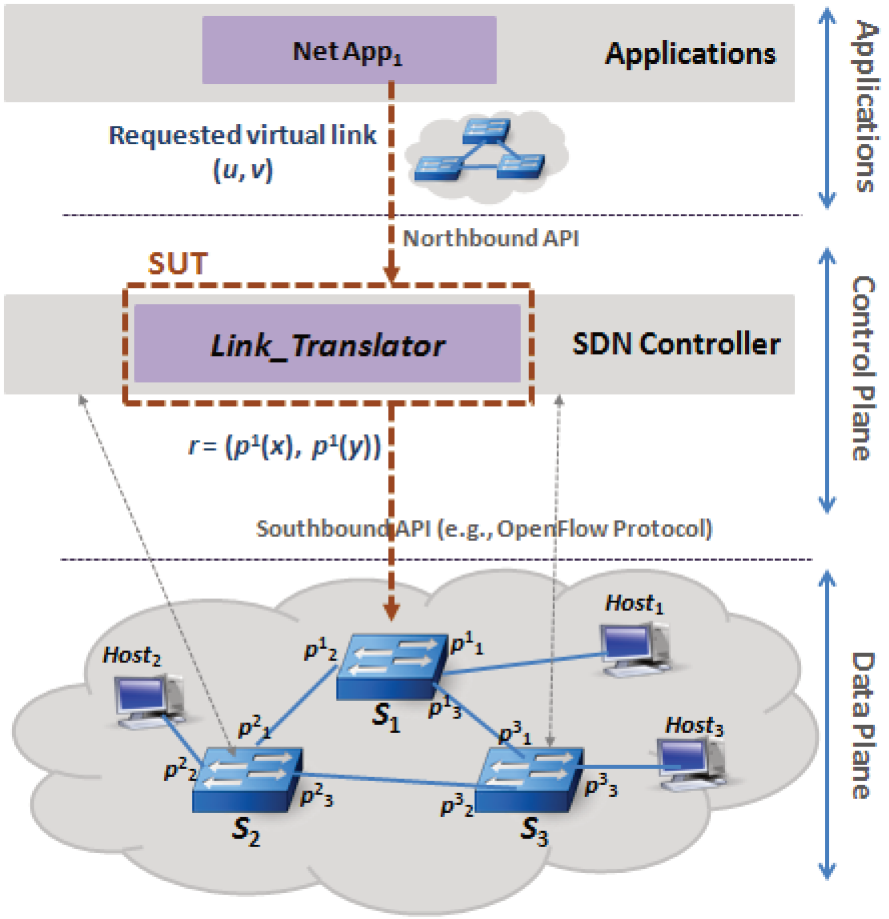


Figure 7.1 – Topology showing a controller application as the SUT

The input to the SUT is a requested virtual link, i.e., a pair (u, v).
The output of the SUT is a rule to switch S_1.
u and v are nodes of the topology.
The rule r is a pair of ports of S_1.

The system under test has a one-direction communication with a given application from which it receives a requested virtual link for any pair of nodes. It has also a one-direction communication with a given switch in the data plane. It sends a rule to the switch of interest specified in the received requested link.
The point of control in the topology is the northbound interface where a request can be sent to the $Link\_Translator$ module, the observation point is the southbound interface where we can observe the input port and the output port of the $Flow\_Mod$ generated by the SUT for the switch of interest S.
The research question we are addressing in this chapter can be formulated as follows.
Given the $Link\_Translator$ controller module accepting as input requested virtual links for any pair of nodes from an application; given also that a requested virtual link is specifying any two devices between which a virtual link (through one switch) should be created / implemented; how to guarantee that the $Link\_Translator$ assigns correctly the ports of the network devices as specified by the request?

7.3 Formal Representation of a Controller Application and Notations

Similar to the representation adopted in Chapter 4, the data plane is represented as an $RNCT$, i.e., an undirected graph $G = (V, E)$ where $E \subseteq \{\{u, v\} \mid u \in V \ \& \ v \in V\}$ with no multiple edges and no self loops.
The implementation of a requested virtual link for any pair of nodes with respect to a switch S then means the 'translation', i.e., the creation of a virtual data link from and to other node (-s) adjacent to S. We refer to the set of neighbours of S as $Adj(S)$. We assume that the different nodes $\forall v \in V$ are identified by their MAC addresses. We refer to the set of ports of S as $Ports(S)$.
As we are interested in checking a requested virtual link for any pair of nodes with respect to a given switch S, we further refer to a requested link as simply a pair of network devices (u, v). Given a requested virtual link $(u, v), u \in V$, $v \in V$, with respect to a given switch S, the problem of translation of the virtual link (u, v) with respect to the switch S is to check that the controller module of interest translates (u, v) into the correct pair of ports of S.
For checking whether the translation for any pair of nodes of a requested virtual link is correct, i.e., $\forall u \in V$, $v \in V$, we need to check whether this pair of nodes is correctly assigned through S. If u and v are connected via S then we need to check whether this pair of nodes is correctly assigned through S, and if u and v are not connected via S, then we need to check indeed no ports are assigned.
This is reasonable because we are only interested in testing whether the $Link\_Translator$ assigns correctly the ports of S. As it will be discussed in Section 7.6, for the test execution, before applying a test case, one of the possibility to translate it to an executable test case is to add / delete some of the physical links in the resource topology such that only a single path is kept which connects any given u and v through u, S and v respectively and where $u \in V, v \in V$.
When a switch S is requested to connect the nodes u and v, we speak about an ordered pair $(u, v) \in V \times V$. Traffic coming to switch S from u, should be forwarded by S to v, more precisely, traffic coming to S from input port linking S to u should be forwarded to the port of S linking S to v.
As studied in Chapter 5, a forwarding rule in a switch S can be written as an implication, identifying the intervals for the parameters (including the input port) and related subset of output ports (recall Definition 5.1 in Chapter 5). In this Chapter, differently from Chapter 5,

we plan to check both, an input port and an output port. We therefore assume that an SDN application configures a switch table in such a way that each rule determines an input port and an output port.
We assume that the $Link\_Translator$ behaviour is represented by a mapping of the set of pairs $\{(u,v) \in V \times V \mid u \neq v\}$ to the set of pairs of ports of S, i.e., $\{(p_{in}(u), p_{out}(v)) \in Ports(S) \times Ports(S) \mid p_{in}(u) \neq p_{out}(v)\}$ as shown in Equation 7.4.

$$\begin{aligned} &Link\_Translator: V \times V \longrightarrow Ports(S) \times Ports(S) \\ &\qquad (u,v) \longmapsto (p_{in}(u), p_{out}(v)) \\ &\text{such that } u \neq v \text{ and } p_{in}(u) \neq p_{out}(v) \end{aligned} \tag{7.4}$$

Example

For the switch S_1 in Figure 7.1, given the requested virtual link $(Host_1, S_3)$, the $Link\_Translator$ translates it to the pair (p_1^1, p_3^1).

Example of mutants

Given the pair of ports $(p_{in}(u), p_{out}(v))$, *output mutants* are of two kinds. Either the rule has an input port different from the one connecting the previous node to S; or the rule has in its actions an output port different from the expected port.

Definition 7.1.

(I) An *output mutant* of the first kind is defined in Equation 7.5. This type of mutant suggests that a rule is emitted for the switch S with $p'_{in}(u)$ instead of $p_{in}(u)$ as the input port, where $p'_{in}(u) \neq p_{in}(u)$.

$$\begin{aligned} &(u,v) \mapsto (p'_{in}(u), p_{out}(v)) \\ &\qquad \text{and } p'_{in}(u) \neq p_{in}(u) \end{aligned} \tag{7.5}$$

(II) An *output mutant* of the second kind is defined in Equation 7.6. This type of mutant suggests that a rule is emitted for the switch S with $p'_{out}(v)$ instead of $p_{out}(v)$ as the output port, where $p'_{out}(v) \neq p_{out}(v)$.

$$\begin{aligned} &(u,v) \mapsto (p_{in}(u), p'_{out}(v)) \\ &\qquad \text{and } p'_{out}(v) \neq p_{out}(v) \end{aligned} \tag{7.6}$$

Definition 7.2.

(I) A requested link mutant with respect to u is defined in Equation 7.7. This type of mutant suggests that the node u of the input pair gets replaced by another node u', where $u \neq u'$.

$$\begin{aligned} &(u',v) \mapsto (p_{in}(u), p_{out}(v)) \\ &\qquad \text{and } u' \neq u \end{aligned} \tag{7.7}$$

(II) A requested link mutant with respect to v is defined in Equation 7.8. This type of mutant suggests that the node v of the input pair gets replaced by another node v', where $v \neq v'$.

$$(u, v') \mapsto (p_{in}(u), p_{out}(v)) \quad \text{and} \quad v' \neq v \tag{7.8}$$

Running Example

Let us consider the $Link\_Translator$ module operating on the simple example of switch S_1 of the topology illustrated in Figure 7.1. In this case, the set of nodes adjacent to S_1 is $Adj(S_1) = \{Host_1, S_2, S_3\}$.
The behaviour of the $Link\_Translator$ is specified in Equation 7.9.

$$\begin{aligned}
&(Host_1, S_2) \mapsto (p_1^1,\ p_2^1)\\
&(Host_1, S_3) \mapsto (p_1^1,\ p_3^1)\\
&(S_2, S_3) \mapsto (p_2^1,\ p_3^1)\\
&(S_3, S_2) \mapsto (p_3^1,\ p_2^1)\\
&(S_2, Host_1) \mapsto (p_2^1,\ p_1^1)\\
&(S_3, Host_1) \mapsto (p_3^1,\ p_1^1)
\end{aligned} \tag{7.9}$$

In the following, we propose to model the $Link\_Translator$ behaviour as a corresponding logic circuit.

7.4 Deriving a Logic Circuit for a Controller Application Specification

Similar to the approach presented in Chapter 5, we propose to build a logic circuit $\mathcal{LC}$ that preserves the behaviour of $\mathcal{S}$ represented by Equation 7.4. This logic circuit $\mathcal{LC}$ allows taking advantage of several scalable manipulations over the Boolean vectors (logic circuits).
We focus on the use of logic synthesis solutions from an LUT for a system of (partially specified) Boolean functions to derive such logic circuit $\mathcal{LC}$. The corresponding procedure is described in Algorithm 7.
Given a mapping forming the specification $\mathcal{S}$, Algorithm 7 derives a logic circuit simulating the behaviour of $\mathcal{S}$ as follows.
First, the numbers of primary inputs and outputs of the logic circuit (lines 1 and 2) are computed. Next, an empty LUT is derived (line 3). The variables of the LUT correspond to the computed number of primary inputs, i.e., $2 \times \lceil log_2(1 + max(V)) \rceil$, where $max(V)$ denotes the maximal value of the identifiers (the MAC addresses) of all nodes in V. The partially specified Boolean functions of the LUT correspond to the computed number of primary outputs, i.e., $2 \times \lceil log_2(1 + |Ports(S)|) \rceil$. Afterwards, for each *mapping* in $\mathcal{S}$, a corresponding Boolean vector encoding the nodes' pair and the ports' pair is derived and added to the LUT (lines 6, 7, 8, 9, 10, and 11). Finally, a logic synthesis is run and $\mathcal{LC}$ is returned (lines 12 and 13).

Algorithm 7: Logic Circuit Derivation from a Controller Application (Link_Translator)

Input : A specification $\mathcal{S}$ represented by a set of *mappings* w.r.t. a switch S

Output : A logic circuit $\mathcal{LC}$ simulating $\mathcal{S}$

1 Determine the number of the primary inputs (PI) for the logic circuit as $2 \times \lceil log_2(1 + max(V)) \rceil$ such that $max(V)$ is the maximal value of all nodes in the set V where all values of nodes are non-negative, and $\lceil a \rceil$ denotes the ceiling function applied to a.

2 Determine the number of the primary outputs (PO) for the logic circuit as $2 \times \lceil log_2(1 + |Ports(S)|) \rceil$, where $|Ports(S)|$ denotes the cardinality of the set of ports of the switch S.

3 Derive an empty LUT L for a system of $2 \times \lceil log_2(1 + |Ports(S)|) \rceil$ Boolean functions of $2 \times \lceil log_2(1 + max(V)) \rceil$ variables

4 **foreach** *mapping* $\in \mathcal{S}$ **do**

5 **foreach** *pair* (u, v) **do**

6 Encode each node by a Boolean vector B_i, $i = \{1, 2\}$ of length $\lceil log_2(1 + max(V)) \rceil$

7 Set B_output to $(00 \ldots 00)$ such that $|B\_output| = 2 \times \lceil log_2(1 + |Ports(S)|) \rceil$

8 Encode $p_{out}(v)$ of the pair $(p_{in}(u), p_{out}(v))$ by a Boolean vector B_output_1 such that $|B\_output_1| = \lceil log_2(1 + |Ports(S)|) \rceil$

9 Encode $p_{in}(u)$ of the pair $(p_{in}(u), p_{out}(v))$ by a Boolean vector B_output_2 such that $|B\_output_2| = \lceil log_2(|Ports(S)|) \rceil$

10 Set B_output to B_output_1 B_output_2

11 Add a new line to the LUT, i.e., set L to $L \cup \{B_1, B_2 \mid B\_output\}$

12 Run a logic synthesis solution for deriving a logic circuit $\mathcal{LC}$ from the LUT L

13 **return** $\mathcal{LC}$

Example

For the running example of the set of mappings listed in Equation 7.9, we consider the switches S_1, S_2 and S_3 and the hosts $Host_1, Host_2$ and $Host_3$ have the MAC addresses' values as illustrated in Table 7.1. Let the ports of S_1 be as follows $p_1^1 = 1$, $p_2^1 = 2$ and $p_3^1 = 3$ linking S_1 to $Host_1$, S_2 and S_3 respectively (as illustrated in Figure 7.1).

The LUT derived by Algorithm 7 has six lines portrayed in Table 7.2.

Node	MAC Address
S_1	9A:D8:73:D8:90:6A
S_2	9A:D8:73:D8:90:6B
S_3	9A:D8:73:D8:90:6C
$Host_1$	00:1B:63:84:45:E1
$Host_2$	00:1B:63:84:45:E2
$Host_3$	00:1B:63:84:45:E3

Table 7.1 – MAC addresses of the nodes of the topology illustrated in Figure 7.1

$i_1 i_2 \dots i_{96}$	$f_1 f_2 \dots f_8$
000000000001101101100011100001000100010111100001100110101101100001110011110110001001000001101011	00010010
000000000001101101100011100001000100010111100001100110101101100001110011110110001001000001101100	00010011
100110101101100001110011110110001001000001101011100110101101100001110011110110001001000001101100	00100011
100110101101100001110011110110001001000001101100100110101101100001110011110110001001000001101011	00110010
100110101101100001110011110110001001000001101011000000000001101101100011100001000100010111100001	00100001
100110101101100001110011110110001001000001101100000000000001101101100011100001000100010111100001	00110001

Table 7.2 – Look-up table for the controller application ($Link\_Translator$) running example

7.5 Test Suite Generation

Now that a logic circuit $\mathcal{LC}$ that simulates the behaviour of the $Link\_Translator$ is derived, the test derivation techniques presented in Chapter 5 can be used. We propose the use of classical logic circuit testing strategies. Thereupon, the derived circuit $\mathcal{LC}$ can be saved into the BLIF format, and the BLIF mutant generator developed by Kushik et al. [85] can then be executed to generate mutants of different types such as for example SSF, SBF, and HDF.
Afterwards, a distinguishing pattern can be found for each non-equivalent mutant and a test suite can be obtained for each of the fault models. The goal of deriving such test suite is to distinguish the output of a correct implementation from an assumed incorrect implementation, i.e., mutant. The derived test to kill each mutant of a particular type can be then applied to the implementation of the $Link\_Translator$ module of the controller.
Further on, in order to check if the fault coverage increases with different circuit representations, as accomplished in Chapter 5, the original BLIF file representing $\mathcal{LC}$ can be re-synthesized as an AND-INVERTER graph (AIG), and the same procedure of test generation applied to $\mathcal{LC}$ can be performed to the newly generated logic circuit to obtain the corresponding test suites.
Finally, to assess the fault coverage of traditional digital circuit fault models, a set of mutants of $\mathcal{S}$ as specified in Definitions 7.1 and 7.2 can be generated.

7.6 Test Suite Execution

In this section, we describe the test execution procedure for one of the modules integrated in the Floodlight controller [1]. This module is a $Link\_Translator$ mapping requested virtual links received from an application into pairs of ports with respect to a switch. The module is referred to as 'circuit pusher'.
This module takes as input a pair of source and destination addresses of any given two hosts, it retrieves these addresses and uses them as attachment points (APs) using 'DeviceManager'

sub-module. Then, it computes a path between these two addresses using 'RoutingManager' sub-module. Finally, it sends one rule per pair of APs in the path using another sub-module [1].

One of the challenges that could be encountered in the experimental evaluation is that the generated test cases can be too abstract and not compatible with an actual implementation (SUT). Indeed, as we consider the behaviour of the $Link\_Translator$ module with respect to a single switch, we might be confronted to the problem of converting / adapting the derived abstract test cases into ones that can be accepted by the SUT (test translation problem).

To be able to execute the derived test suites against the implementation of the controller module under test, we need a way for each test case in the test suite to be converted into a pair of hosts as accepted by the $Link\_Translator$, in this case, the 'circuit pusher'. Thus, the derived abstract test cases (of the form of $(u, v)\ \ \forall u \in V, v \in V$) should be concretized to fulfill the format of test cases accepted by the module of interest and make the execution feasible.

Consider the following situation: we aim to validate that the $Link\_Translator$ behaves correctly with respect to the switch S_1 of the topology of Figure 7.1. To accomplish this, suppose that we want to execute the test suite TS_1 of Equation 7.10. We need a way for each test case in the test suite TS_1 to be converted into a pair of hosts as accepted by the $Link\_Translator$ in the Floodlight controller. The procedure allowing such conversion is described in Algorithm 8.

$$TS_1 = \{(Host_1, S_2), (Host_1, S_3), (S_2, S_3), (S_3, S_2), (S_2, Host_1), (S_3, Host_1)\} \tag{7.10}$$

For each test case $(u, v)\ \forall\ u, v \in V$, Algorithm 8 converts it with respect to a switch S, it sets a path between two hosts $(Host_u, Host_v)$ such that the path contains the requested virtual link $(u, S, v)\ \ \forall u, v \in V$. In this way the 'circuit pusher' module takes as input the pair $(Host_u, Host_v)$. The execution of such test case allows to check whether the translation of the virtual link (u, v) for S is correct, i.e., whether the ports of S are correctly assigned as expected.

Lines 3, 4, and 5 check whether the first component of the pair (u, v) is of type host, if not and there exists a host in the adjacency of u then u is replaced by that host. If there are no hosts in the adjacency of u, then a host is attached to u (lines 7 and 8). The same procedure is performed for the second component v (lines 9, 10, 11 and lines 13 and 14).

The new $RNCT'$ containing only the links of interest in the resource topology is defined (line 15). Finally, line 16 returns the executable test case along with the new $RNCT'$.

For the execution of the test suites, the REST API of the Floodlight controller can be used to send the test inputs.

Example

Consider an example of a derived test case $tc = (S_2, S_3)$ for our running example for the switch S_1 of the SDN topology of Figure 7.1. Algorithm 8 returns a corresponding executable test case $tc_{exe} = (Host_2, Host_3)$ that can be fed as input to the 'circuit pusher' controller module along with a new $RNCT'$ where all the links are down except those of interest, i.e., $(Host_2$, $S_2)$ (S_2, S_1) (S_1, S_3) $(S_3, Host_3)$.

Figure 7.2 illustrates the execution of the obtained executable test case tc_{exe} by Algorithm 8. For each test case of a given test suite, the procedure is repeated. The test suite can then be executed against the implementation and observations can be performed in the southbound interface.

Algorithm 8: Test Translation for the Controller Application Under Test

Input : A test case tc of the form of a pair (u, v) w.r.t. a switch S, $\forall u, v \in V$
An $RNCT$

Output : An executable test case tc_{exe} of the form $(Host_u, Host_v)$ where $Host_u, Host_v \in V$ and $Host_u$ and $Host_v$ are of type hosts
An $RNCT'$ for tc_{exe} execution

1 Let $Ht \subseteq V$ and $Sw \subseteq V$ be the sets of hosts and switches respectively
2 $RNCT' \leftarrow RNCT$
3 **if** *In* $RNCT'$, $u \notin Ht$ **then**
4 **if** *In* $RNCT'$, $\exists$ *a node* $Host_u \in Adj(u)$ *such that* $Host_u \in Ht$ **then**
5 $tc_{exe} \leftarrow (Host_u, v)$
 ▷ In $RNCT'$, replace u by a node of type host adjacent to u
6 **else**
7 Attach a node $Host_u$ such that $Host_u \in Ht$ to node u in $RNCT'$
 ▷ In $RNCT'$, attach to u a node of type host
8 $tc_{exe} \leftarrow (Host_u, v)$
9 **if** *In* $RNCT'$, $v \notin Ht$ **then**
10 **if** *In* $RNCT'$, $\exists$ *a node* $Host_v \in Adj(v)$ *such that* $Host_v \in Ht$ **then**
11 $tc_{exe} \leftarrow (Host_u, Host_v)$
 ▷ In $RNCT'$, replace v by a node of type host adjacent to v
12 **else**
13 Attach a node $Host_v$ such that $Host_v \in Ht$ to node v in $RNCT'$
 ▷ In $RNCT'$, attach to v a node of type host
14 $tc_{exe} \leftarrow (Host_u, Host_v)$
15 $RNCT' \leftarrow$ put down all links in the $RNCT'$ except $(Host_u, u)\ (u, S)\ (S, v)\ (v, Host_v)$
16 **return** tc_{exe} *and* $RNCT'$

7.7 Chapter Conclusions

Motivated by the results showing the effectiveness of the logic circuit based testing technique presented in Chapter 5, in this chapter, we have made a proposal for applying such technique to a module of the controller. The module is responsible for translating, for a given switch, a requested virtual link received from a given application (in the application layer) into a rule pushed to the switch of interest specifying the input and output ports linking the switch to the corresponding neighbours.

We have formalized the behaviour of the module in Section 7.3 and have presented an algorithm for logic circuit synthesis from the set of mappings defining the specification in Section 7.4. Then, Section 7.5 has provided a glance insight on the test generation approach

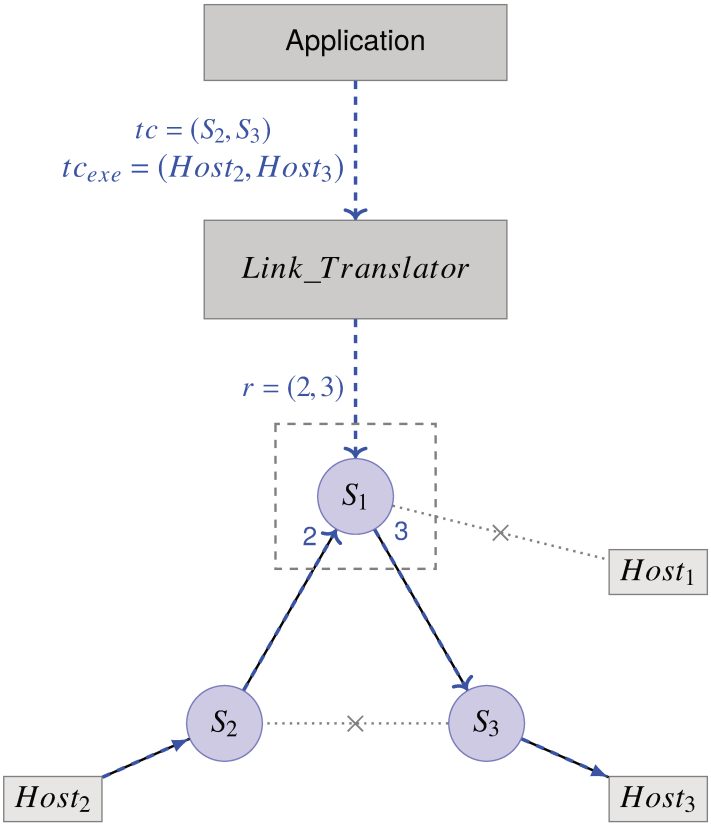


Figure 7.2 – Illustration of the execution of a test case of the running example

with reference to Chapter 5. To illustrate the test cases' execution, a corresponding algorithm has been presented in Section 7.6. The algorithm serves for converting derived test cases into executable ones with respect to an integrated controller application part of the Floodlight controller [1].

By this, the fourth *challenge* stated in Chapter 1 (***RQ4***) has been addressed. Nonetheless, the work presented in the current chapter represents only a first step towards testing the controller, a colossal and compounded SDN component. Therefore, an experimental evaluation is still needed to validate the effectiveness of the approach applied to the controller module and to draw meaningful conclusions. This is planned for future work.

The discussion will be continued and summarized in the next chapter in order to illustrate the remaining research potential of the proposed model based testing approach for the controller component. In particular, the new directions in relation to the preliminary achievements gained throughout this work will be pointed out.

8

Conclusion and Future Work

"The important thing is not to stop questioning."

– Albert Einstein

Contents

This is the last chapter of this thesis. It is divided into two sections, the first of which contains a general summary, including a discussion on how the problems and challenges introduced in Chapter 1 have been addressed throughout this work. Then, the second section outlines the perspectives and some future directions of the contributions presented here with a glance insight at the advantages and limitations of the testing approaches proposed herewith.

8.1 Contributions: Summary

The first part of this thesis has dealt with general introductory and motivating analysis on its topic. Hence, Chapter 1 has introduced the motivation, research questions and contributions of this work. Also, the structure has been provided there and a roadmap has been discussed. Then, in Chapter 2, the background on SDN architectures and their components have been described, herewith the fundamentals on verification and testing. Besides, model based testing concepts have been put in the center of attention. Also, basic knowledge on mutation analysis and black and white box testing approaches has been given.
In Chapter 3, analysis of the related work on verification and testing of SDN architectures and their components has been provided. A taxonomy has been elaborated for verification techniques on one hand and testing techniques on the other hand. A comprehensive classification of the verification techniques into off-line and run-time has been presented while for testing techniques, the underlying test generation methods have been in focus. Finally, based on the

analysis of challenges and limitations of the existing work, characteristics of the approaches proposed in this thesis have been briefly elaborated.

Next, the intention of the second part of this thesis has been to introduce and detail the contributions. In Chapter 4, a new model based approach for testing entire SDN architectures relying on *graph/path enumeration* has been presented. A corresponding *fault model* where the *fault domain* contains potential implementations of *virtual paths* requested by an en-user has been defined. To guarantee the *fault coverage*, the conditions have been proven, when under *black box* and *white box* testing assumptions a *complete* test suite with respect to such fault model can be derived. Furthermore, an experimental evaluation has been conducted and the results have been discussed demonstrating the effectiveness of this first contribution.

By this first contribution, the challenge of guaranteeing the correct behaviour of an entire SDN architecture has been successfully addressed. Nonetheless, this first achievement does not resolve the entire problematic as the functional correct behaviour of SDN components forming this architecture is brought to the picture and should not be taken for granted. On the one hand, it might happen that the architecture is functioning correctly as a whole while hiding faulty SDN components such as the switch and the controller. On the other hand, should a bug be discovered by testing the entire architecture, it certainly becomes of interest to localize its triggering faulty SDN component.

The following three Chapters 5, 6 and 7 of the thesis have achieved such task. In a first step, the *switch* as a critical SDN component has been considered as the system under test, in its data plane interface and in its interaction with the controller in Chapters 5 and 6 respectively. In a second step, a *controller application* responsible for the translation of user requests into links in the data plane has been considered as the system under test in Chapter 7.

Chapter 5 has introduced a new model based testing technique where the specification has been represented by a *Logic Circuit* for assuring the correctness of the *forwarding functionality* of an SDN-enabled switch. The system under test acts as a forwarding device receiving and sending network packets in accordance with a set of configured rules. Hence, it has been modelled and analyzed as a 'stateless' system. A testing approach has been proposed where both, active and passive testing can take advantage of such logic circuit representation and a *fault model* has been proposed. Furthermore, the usefulness of logic circuit based fault models such as for example Single Stuck-at Faults (SSFs) for detecting bugs and misconfigurations in the SDN-enabled switch implementations has been estimated. Correspondingly, potential mutations over the switch rules have been introduced in order to discover which of these mutations can be effectively detected using the proposed logic circuit based approach. Further on, the use of Boolean Satisfiability (SAT) solvers for detecting equivalent mutants has been discussed. Finally, a scalable solution for the switch monitoring on the basis of logic circuits and related operations has been elaborated. Results have shown the effectiveness of the proposed approach.

Chapter 6 has investigated the behaviour of a switch in its interaction with an SDN controller as the SUT. For this purpose, an *Extended Finite State Machine* based technique for test generation has been proposed. The approach has succeeded to verify the OF switch-to-controller interaction with respect to requirements described in the OF specification. An EFSM model has been derived based on the OF requirements, then potential *incorrect* implementations have also been modelled as EFSMs that have been obtained by injecting specific types of (user-driven) faults into the model, i.e., through mutants' generation. For each mutant, a *distinguishing sequence* has been derived that separates the original specification from the mutated one. An algorithm that derives test suites formed by the corresponding distinguishing sequences has been presented. The approach has been evaluated by defining a number of test quality metrics and by comparing it to a random generation approach as a baseline.

The results have shown the effectiveness of such approach in detecting switch-to-controller interaction errors/bugs.
Succinctly, the experimental evaluations conducted in Chapters 4, 5 and 6 have revealed the strengths of the proposed approaches so far by providing high test coverage. This has motivated the adaptation of one of the approaches to another critical SDN component. Specifically, the module of an SDN controller responsible for the translation of end-users' requests, coming from the application layer and represented as *requested virtual links*, into pairs of ports with respect to a given switch in the data plane.
Chapter 7 has been based on Chapter 5. Here, the testing methodology presented in Chapter 5 has been adapted to test an important module of an SDN controller.
To summarize, the proposed model based testing approaches in this work present several advantages. They advance the state of the art of model based testing techniques for SDN. Indeed, these approaches have been shown to be most beneficial to use in detecting errors in the implementations. Moreover, the entire architecture as well as different SDN components have been covered by this work.
Further on, the proposed approaches for testing SDN architectures and their components help reduce considerably the effort spent by the test engineers. This is in the context of not only test generation but also test execution. Farther, the proposed *faults models* allow to provide guarantees about the quality of the derived test suites, in particular, conditions on the *completeness* and *exhaustiveness* of the test suites derived by the proposed approaches are proven. By that, the cost of the process of SDN testing before deployment is definitely cut.

8.2 Perspectives and Future Directions

Despite the advantages of the contributions aforementioned, there is still plenty of work concerning the presented research work and approaches' future achievements.
Firstly, in Chapter 3, our review of the state of the art of techniques for verification and testing SDN architectures / components has not considered works on security and fault tolerance. Including such body of works could be an interesting step towards a more thorough analysis. This could bring to the picture the testing challenges raised by these fields.
Secondly, the testing approach presented in Chapter 4 has considered the system under test as the SDN architecture. The latter can be a part of a more complex orchestration framework, for example, in the context of virtual networking deployments such as the 5G mobile network. In this case, the SDN architecture would operate in the presence of other technologies such as cloud-computing. Such deployments would account for not only traditional end-user based requirements / policies but also for emerging, more sophisticated services. Taking into consideration these factors, it could be appealing to inspect the validity of the proposed graph enumeration based approach for testing the SDN architecture operating in such environments. For instance, more equivalence classes and more conformance relations could be defined. In this context, it is worth mentioning that some of the thesis results have been already extended / improved. For example, the approach proposed in the aforementioned chapter has been augmented with other equivalence classes defined over the SDN architecture inputs, i.e., paths. This has been developed by Yevtushenko et al. [163].
Moreover, the testing approach considered for an SDN-enabled switch in its data plane interface in Chapter 5 has proven that the logic circuit simulation is much faster compared to the switch packet processing. This could be an interesting research direction for optimizing the switch implementations. Performing more investigations of the speed-up obtained via the logic circuit representation of a set of rules could serve as a basis for building a switch directly

from its logic description and hence its optimization. Further, equivalent mutants for switches and the corresponding proposed SAT solving solution can be further explored.
The testing approach presented in Chapter 6 could also be improved. For example, the EFSM model has been synthesized manually based on our domain knowledge. A natural research direction is to automatically extract such model given the OF requirements especially when the latter is augmented to model the entire OF protocol for example. Model learning and approaches for semi-automatic model derivation for instance might be an interesting track to explore in this direction. Besides, the EFSM model is partial which has an impact on the proposed test generation algorithm based on deriving distinguishing sequences between the specification and the mutated models. The correlation between the degree of partiality and the effectiveness of the algorithm might be considered for future investigations.
The compliance of the SDN controller with its specification and the correctness of all of its modules is a challenge that has only been partially addressed in this work. Even though the controller is a big and complex software system, assuring the functional correctness of its modules is of paramount importance. The testing approach presented in Chapter 7 is only a first attempt in this direction. The effectiveness and efficiency of such method should be evaluated in order to prove its success. This is another interesting direction of future research work.
There is also still plenty of research potential resulting from the thesis in general. For example, a thorough evaluation of the complexity of the algorithms proposed in the thesis should be performed in the future. Further on, the issue of test suite minimization is an open area for further inspections as well. Namely, how to choose the minimal number of test cases of interest so that the test suite fault coverage is preserved could be subject of deeper investigations for all the approaches. Further on, the proposed fault models describe just sets of faults responsible for potential inconsistencies/bugs of the SDN architecture and its related SDN components. However, how to ensure that all the faults present in real implementations are captured by such fault models is an open question. Hence, extending the proposed fault models, investigating others, and checking different abstraction levels could be an interesting future objective. Another point worth mentioning is that since the main focus of the thesis has been on model based test derivation methods, more elaborated experiments notably with architectures/components operating in real ‘production’ environments could be performed in the future. Also, the tools used for test generation and execution in this work could be extended and integrated in a testing framework which would make the work of a test engineer/network administrator easier to achieve.

Bibliography

[1] *A Big Switch Networks sponsored community project*. URL: http://www.projectfloodlight.org/floodlight/.

[2] Kanak Agarwal et al. “SDN traceroute: Tracing SDN forwarding without changing network behavior.” In: *Proceedings of the third workshop on Hot topics in software defined networking*. ACM. 2014, Pages 145–150.

[3] Bernhard K Aichernig et al. “Model-based mutation testing of an industrial measurement device.” In: *Proceedings of the International Conference on Tests and Proofs*. Springer. 2014, Pages 1–19.

[4] Elvira Albert et al. “SDN-Actors: Modeling and Verification of SDN Programs.” In: *Proceedings of the International Symposium on Formal Methods*. Springer. 2018, Pages 550–567.

[5] Syed Taha Ali et al. “A Survey of Securing Networks Using Software Defined Networking.” *IEEE Trans. Reliability* 64.3 (2015), Pages 1086–1097.

[6] Amer Aljaedi and C Edward Chow. “Pathseer: A centralized tracer of packet trajectories in software-defined datacenter networks.” In: *Principles, Systems and Applications of IP Telecommunications (IPTComm)*. IEEE. 2016, Pages 1–9.

[7] Izzat Alsmadi, Milson Munakami, and Dianxiang Xu. “Model-based testing of SDN firewalls: a case study.” In: *Proceedings of the Second International Conference on Trustworthy Systems and Their Applications (TSA)*. IEEE. 2015, Pages 81–88.

[8] Rajeev Alur and David Dill. “Automata for modeling real-time systems.” In: *Proceedings of the International Colloquium on Automata, Languages, and Programming*. Springer. 1990, Pages 322–335.

[9] Rajeev Alur and David L Dill. “A theory of timed automata.” *Theoretical computer science* 126.2 (1994), Pages 183–235.

[10] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[11] Carolyn Jane Anderson et al. “NetKAT: Semantic foundations for networks.” In: *ACM SIGPLAN Notices*. Volume 49. 1. ACM. 2014, Pages 113–126.

[12] Christian Attiogbé. “Building Correct SDN-Based Components from a Global Formal Mode.” *arXiv preprint arXiv:1806.09476* (2018).

[13] Tigran Avanesov et al. “Network troubleshooting with sdn-radar.” In: *Proceedings of the International Symposium on Integrated Network Management (IM)*. IFIP/IEEE. 2015, Pages 1137–1138.

[14] Thomas Ball et al. “Vericon: Towards verifying controller programs in software-defined networks.” In: *Proceedings of the ACM SIGPLAN Notices*. Volume 49. 6. ACM. 2014, Pages 282–293.

[15] Bruno Barras et al. “The Coq proof assistant reference manual: Version 6.1.” Doctoral dissertation. Inria, 1997.

[16] Ryan Beckett et al. “An assertion language for debugging SDN applications.” In: *Proceedings of the third workshop on Hot topics in software defined networking*. ACM. 2014, Pages 91–96.

[17] Pankaj Berde et al. “ONOS: towards an open, distributed SDN OS.” In: *Proceedings of the third workshop on Hot topics in software defined networking*. ACM. 2014, Pages 1–6.

[18] UC Berkeley. “Berkeley logic interchange format (BLIF).” *Oct Tools Distribution* 2 (1992), Pages 197–247.

[19] Asma Berriri, Natalia Kushik, and Djamal Zeghlache. “On using finite state models for optimizing and testing SDN controller components.” *Russian Physics Journal* 59.8/2 (2016), Pages 5–7.

[20] Gregor V Bochmann and Alexandre Petrenko. “Protocol testing: review of methods and relevance for software testing.” In: *Proceedings of the ACM SIGSOFT international symposium on Software testing and analysis*. ACM. 1994, Pages 109–124.

[21] Robert S Boyer, Bernard Elspas, and Karl N Levitt. “SELECTa formal system for testing and debugging programs by symbolic execution.” *ACM SigPlan Notices* 10.6 (1975), Pages 234–245.

[22] Robert Brayton and Alan Mishchenko. “ABC: An academic industrial-strength verification tool.” In: *Proceedings of the International Conference on Computer Aided Verification*. Springer. 2010, Pages 24–40.

[23] Kai Bu et al. “Is every flow on the right track?: Inspect SDN forwarding with RuleScope.” In: *Proceedings of the 35th International Conference on Computer Communications (INFOCOM)*. IEEE. 2016, Pages 1–9.

[24] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. “Why and where: A characterization of data provenance.” In: *Proceedings of the International conference on database theory*. Springer. 2001, Pages 316–330.

[25] Marco Canini et al. “A NICE Way to Test OpenFlow Applications.” In: *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. San Jose, CA: USENIX, 2012, Pages 127–140. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/canini.

[26] Marco Canini et al. “Automating the testing of OpenFlow applications.” In: *Proceedings of the 1st International Workshop on Rigorous Protocol Engineering (WRiPE)*. 2011.

[27] Chen Chen et al. “Proof-based Verification of Software Defined Networks.” In: *Proceedings of the ONS*. 2014.

[28] Alessandro Cimatti et al. “Nusmv 2: An opensource tool for symbolic model checking.” In: *Proceedings of the International Conference on Computer Aided Verification*. Springer. 2002, Pages 359–364.

[29] Benoit Claise. *Cisco systems netflow services export version 9*. 2004.

[30] Duncan Clarke, Insup Lee, and Hong-liang Xie. “VERSA: A Tool for the Specification and Analysis of Resource-Bound Real-Time Systems.” *Journal of Computer and Software Engineering* 3 (1995).

[31] Edmund M Clarke, E Allen Emerson, and Joseph Sifakis. “Model checking: algorithmic verification and debugging.” *Communications of the ACM* 52.11 (2009), Pages 74–84.

[32] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. “Automatic verification of finite-state concurrent systems using temporal logic specifications.” *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8.2 (1986), Pages 244–263.

[33] G Adam Covington et al. “A packet generator on the NetFPGA platform.” In: *Proceedings of the 17th Symposium on Field Programmable Custom Computing Machines*. IEEE. 2009, Pages 235–238.

[34] Jason Croft et al. “Systematically Exploring the Behavior of Control Programs.” In: *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. Santa Clara, CA: USENIX Association, 2015, Pages 165–176. ISBN: 978-1-931971-225. URL: https://www.usenix.org/conference/atc15/technical-session/presentation/croft.

[35] Levente Csikor and Dimitrios P Pezaros. “End-host Driven Troubleshooting Architecture for Software-Defined Networking.” In: *Proceedings of the Global Communications Conference (GLOBECOM)*. 2017, Pages 1–7.

[36] Martin Davis, George Logemann, and Donald Loveland. “A machine program for theorem-proving.” *Communications of the ACM* 5.7 (1962), Pages 394–397.

[37] Leonardo De Moura and Nikolaj Bjørner. “Z3: An efficient SMT solver.” In: *Proceedings of the International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, Pages 337–340.

[38] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters.” *Communications of the ACM* 51.1 (2008), Pages 107–113.

[39] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. “Hints on test data selection: Help for the practicing programmer.” *Computer* 11.4 (1978), Pages 34–41.

[40] Lin Deng, Jeff Offutt, and Nan Li. “Empirical evaluation of the statement deletion mutation operator.” In: *Proceedings of the Sixth International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2013, Pages 84–93.

[41] David 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, Pages 390–393. URL: http://dl.acm.org/citation.cfm?id=647765.735832.

[42] Mihai Dobrescu and Katerina Argyraki. “Software dataplane verification.” *Communications of the ACM* 58.11 (2015), Pages 113–121.

[43] Mihai Dobrescu and Katerina Argyraki. “Toward a verifiable software dataplane.” In: *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM. 2013, Page 18.

[44] Avri Doria et al. *Forwarding and control element separation (ForCES) protocol specification*. Technical report. 2010.

[45] Ramakrishnan Durairajan, Joel Sommers, and Paul Barford. “Controller-agnostic SDN debugging.” In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM. 2014, Pages 227–234.

[46] E Allen Emerson. “The beginning of model checking: A personal perspective.” In: *25 Years of Model Checking*. Springer, 2008, Pages 27–45.

[47] *Event-B and Rodin Documentation Wiki*. URL: http://wiki.event-b.org/index.php/Main_Page.

[48] Khaled El-Fakih, Nina Yevtushenko, and Natalia Kushik. “Adaptive distinguishing test cases of nondeterministic finite state machines: test case derivation and length estimation.” *Formal Aspects of Computing* 30.2 (2018), Pages 319–332.

[49] Khaled El-Fakih et al. “Fault diagnosis in extended finite state machines.” In: *Proceedings of the IFIP International Conference on Testing of Software and Communicating Systems*. Springer. 2003, Pages 197–210.

[50] Khaled El-Fakih et al. “Fsm based interoperability testing methods for multi stimuli model.” In: *Proceedings of the IFIP International Conference on Testing of Communicating Systems*. Springer. 2004, Pages 60–75.

[51] Seyed K Fayaz and Vyas Sekar. “Testing stateful and dynamic data planes with FlowTest.” In: *Proceedings of the third workshop on Hot topics in software defined networking*. ACM. 2014, Pages 79–84.

[52] Seyed Kaveh Fayaz et al. “BUZZ: Testing Context-Dependent Policies in Stateful Networks.” In: *Proceedings of the NSDI*. 2016, Pages 275–289.

[53] Seyed K Fayaz et al. “Scalable Testing of Context-Dependent Policies over Stateful Data Planes with Armstrong.” *arXiv preprint arXiv:1505.03356* (2015).

[54] Nick Feamster, Jennifer Rexford, and Ellen Zegura. “The road to SDN.” *Queue* 11.12 (2013), Page 20.

[55] Paulo Fonseca and Edjard Mota. “A survey on fault management in software-defined networks.” *IEEE Communications Surveys & Tutorials* (2017).

[56] L Girish and Sridhar KN Rao. “Mathematical tools and methods for analysis of SDN: A comprehensive survey.” In: *Proceedings of the 2nd International Conference on Contemporary Computing and Informatics (IC3I)*. IEEE. 2016, Pages 718–724.

[57] Ramesh Govindan et al. “Evolve or die: High-availability design principles drawn from googles network infrastructure.” In: *Proceedings of the ACM SIGCOMM Conference*. ACM. 2016, Pages 58–72.

[58] Bernhard JM Grün, David Schuler, and Andreas Zeller. “The impact of equivalent mutants.” In: *Proceedings of the International Conference on Software Testing, Verification, and Validation Workshops*. IEEE. 2009, Pages 192–199.

[59] Natasha Gude et al. “NOX: towards an operating system for networks.” *ACM SIGCOMM Computer Communication Review* 38.3 (2008), Pages 105–110.

[60] Arjun Guha, Mark Reitblatt, and Nate Foster. “Machine-verified network controllers.” In: *Proceedings of the ACM SIGPLAN Notices*. Volume 48. 6. ACM. 2013, Pages 483–494.

[61] Stephen Gutz et al. “Splendid isolation: A slice abstraction for software-defined networks.” In: *Proceedings of the first workshop on Hot topics in software defined networks*. ACM. 2012, Pages 79–84.

[62] Nikhil Handigol et al. “I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks.” In: *Proceedings of the NSDI*. Volume 14. 2014, Pages 71–85.

[63] Nikhil 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*. ACM. 2012, Pages 55–60.

[64] Ahmed El-Hassany et al. “SDNRacer: concurrency analysis for software-defined networks.” In: *Proceedings of the ACM SIGPLAN Notices*. Volume 51. 6. ACM. 2016, Pages 402–415.

[65] Brandon Heller et al. “Leveraging SDN layering to systematically troubleshoot networks.” In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM. 2013, Pages 37–42.

[66] Robert M Hierons et al. “Using formal specifications to support testing.” *ACM Computing Surveys (CSUR)* 41.2 (2009), Page 9.

[67] Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, 2003.

[68] Alex Horn, Ali Kheradmand, and Mukul Prasad. “Delta-net: Real-time Network Verification Using Atoms.” In: *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, MA: USENIX Association, 2017, Pages 735–749.

[69] Fei Hu, Qi Hao, and Ke Bao. “A survey on software-defined network and openflow: From concept to implementation.” *IEEE Communications Surveys & Tutorials* 16.4 (2014), Pages 2181–2206.

[70] Te-Yuan Huang et al. “Teaching computer networking with mininet.” In: *Proceedings of the ACM SIGCOMM*. 2014.

[71] Ali Hussein et al. “SDN verification plane for consistency establishment.” In: *Proceedings of the Symposium on Computers and Communication (ISCC)*. IEEE. 2016, Pages 519–524.

[72] Yosr Jarraya, Taous Madi, and Mourad Debbabi. “A survey and a layered taxonomy of software-defined networking.” *IEEE communications surveys & tutorials* 16.4 (2014), Pages 1955–1980.

[73] Einar Broch Johnsen et al. “ABS: A core language for abstract behavioral specification.” In: *Proceedings of the International Symposium on Formal Methods for Components and Objects*. Springer. 2010, Pages 142–164.

[74] René Just et al. “Are mutants a valid substitute for real faults in software testing?” In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, Pages 654–665.

[75] Miyoung Kang et al. “Formal modeling and verification of SDN-OpenFlow.” In: *Proceedings of the Sixth International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2013, Pages 481–482.

[76] Miyoung Kang et al. “Process algebraic specification of software defined networks.” In: *Proceedings of the Fourth International Conference on Computational Intelligence, Communication Systems and Networks (CICSyN)*. IEEE. 2012, Pages 359–363.

[77] Peyman Kazemian, George Varghese, and Nick McKeown. “Header Space Analysis: Static Checking for Networks.” In: *Proceedings of the NSDI*. Volume 12. 2012, Pages 113–126.

[78] Peyman Kazemian et al. “Real Time Network Policy Checking Using Header Space Analysis.” In: *Proceedings of the NSDI*. 2013, Pages 99–111.

[79] Zeus Kerravala. “As the value of enterprise networks escalates, so does the need for configuration management.” *The Yankee Group* 4 (2004).

[80] Ahmed Khurshid et al. “Veriflow: Verifying network-wide invariants in real time.” *ACM SIGCOMM Computer Communication Review* 42.4 (2012), Pages 467–472.

[81] Ahmed Khurshid et al. “Veriflow: Verifying network-wide invariants in real time.” In: *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2013, Pages 15–27.

[82] Hyojoon Kim et al. “Kinetic: Verifiable dynamic network control.” In: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2015, Pages 59–72.

[83] James C King. “Symbolic execution and program testing.” *Communications of the ACM* 19.7 (1976), Pages 385–394.

[84] Diego Kreutz et al. “Software-defined networking: A comprehensive survey.” *Proceedings of the IEEE* 103.1 (2015), Pages 14–76.

[85] NG Kushik, JE López, and NV Yevtushenko. “Investigation of correlation of test sequences for reliability testing of digital physical system components.” *Russian Physics Journal* 59.8 (2016), Pages 1274–1280.

[86] Maciej Kuzniar, Marco Canini, and Dejan Kostic. “OFTEN testing OpenFlow networks.” In: *Proceedings of the European Workshop on Software Defined Networking (EWSDN)*. IEEE. 2012, Pages 54–60.

[87] Maciej Kuzniar, Peter Peresini, and Dejan Kostic. *Proboscope: Data plane probe packet generation*. Technical report. 2014.

[88] Maciej Kuniar, Peter Pereni, and Dejan Kosti. “What you need to know about SDN flow tables.” In: *Proceedings of the Passive and Active Measurement*. Springer. 2015, Pages 347–359.

[89] Maciej Kuzniar et al. “A SOFT way for openflow switch interoperability testing.” In: *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM. 2012, Pages 265–276.

[90] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[91] David Lebrun, Stefano Vissicchio, and Olivier Bonaventure. “Towards test-driven software defined networking.” In: *Proceedings of the Network Operations and Management Symposium (NOMS)*. IEEE. 2014, Pages 1–9.

[92] Yahui Li et al. “A Survey on Network Verification and Testing with Formal Methods: Approaches and Challenges.” *IEEE Communications Surveys & Tutorials* (2018).

[93] Ying-Dar Lin et al. “OFBench: Performance Test Suite on OpenFlow Switches.” *IEEE Systems Journal* (2017).

[94] Hongqiang Harry Liu et al. “Crystalnet: Faithfully emulating large production networks.” In: *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM. 2017, Pages 599–613.

[95] Nuno P Lopes et al. “Checking Beliefs in Dynamic Networks.” In: *Proceedings of the NSDI*. 2015, Pages 499–512.

[96] Kshiteej Mahajan et al. “Jury: Validating controller actions in software-defined networks.” In: *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2016, Pages 109–120.

[97] Haohui Mai et al. “Debugging the Data Plane with Anteater.” In: *Proceedings of the ACM SIGCOMM Conference*. SIGCOMM ’11. New York, NY, USA, 2011, Pages 290–301.

[98] Rupak Majumdar, Sai Deep Tetali, and Zilong Wang. “Kuai: A model checker for software-defined networks.” In: *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD)*. IEEE. 2014, Pages 163–170.

[99] A Matrosova, Eugeniy Mitrofanov, and Toral Shah. “Multiple stuck-at fault testability of a combinational circuit derived by covering ROBDD nodes by Invert-And-Or sub-circuits.” In: *Proceedings of the East-West Design & Test Symposium (EWDTS)*. IEEE. 2015, Pages 1–4.

[100] Rick McGeer. “A safe, efficient update protocol for OpenFlow networks.” In: *Proceedings of the 1st workshop on Hot topics in software defined networks*. ACM. 2012, Pages 61–66.

[101] Rick McGeer. “Verification of switching network properties using satisfiability.” In: *Proceedings of the International Conference on Communications (ICC)*. IEEE. 2012, Pages 6638–6644.

[102] Nick McKeown et al. “OpenFlow: enabling innovation in campus networks.” *ACM SIGCOMM Computer Communication Review* 38.2 (2008), Pages 69–74.

[103] Nick McKeown et al. “OpenFlow: enabling innovation in campus networks.” *ACM SIGCOMM Computer Communication Review* 38.2 (2008), Pages 69–74.

[104] Jan Medved et al. “Opendaylight: Towards a model-driven sdn controller architecture.” In: *Proceedings of the 15th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. 2014, Pages 1–6.

[105] *Mininet: An Instant Virtual Network on Your Laptop (or Other PC)-Mininet*. 2018.

[106] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. “DAG-aware AIG rewriting a fresh look at combinational logic synthesis.” In: *Proceedings of the 43rd Annual Design Automation Conference*. ACM. 2006, Pages 532–535.

[107] Gilbert N Nde and Rahamatullah Khondoker. “SDN testing and debugging tools: A survey.” In: *Proceedings of the International Conference on Informatics, Electronics and Vision (ICIEV)*. IEEE. 2016, Pages 631–635.

[108] Tim Nelson et al. “A balance of power: Expressive, analyzable controller programming.” In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM. 2013, Pages 79–84.

[109] Tim Nelson et al. “Tierless Programming and Reasoning for Software-Defined Networks.” In: *Proceedings of the NSDI*. Volume 14. 2014, Pages 519–531.

[110] Bruno Astuto A Nunes et al. “A survey of software-defined networking: Past, present, and future of programmable networks.” *IEEE Communications Surveys & Tutorials* 16.3 (2014), Pages 1617–1634.

[111] Jeff Offutt. “A mutation carol: Past, present and future.” *Information and Software Technology* 53.10 (2011), Pages 1098–1107.

[112] Robert Olsson. “Pktgen the linux packet generator.” In: *Proceedings of the Linux Symposium, Ottawa, Canada*. Volume 2. 2005, Pages 11–24.

[113] *Openflow Switch Specification Version 1.3.4*. Open Networking Foundation. 2014. URL: https://www.opennetworking.org.

[114] Aurojit Panda et al. “Verifying isolation properties in the presence of middleboxes.” *arXiv preprint arXiv:1409.7687* (2014).

[115] Janak H Patel. “Stuck-at fault: a fault model for the next millennium.” In: *Proceedings of the International Test Conference*. IEEE. 1998, Page 1166.

[116] Peter Peresini and Marco Canini. “Is Your OpenFlow Application Correct?” In: *Proceedings of the ACM CoNEXT Student Workshop*. EPFL-POSTER-170417. 2011.

[117] Peter Pereni, Maciej Kuniar, and Dejan Kosti. “Monocle: Dynamic, fine-grained data plane monitoring.” In: *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. ACM. 2015, Pages 1–13.

[118] Alexandre Petrenko, Sergiy Boroday, and Roland Groz. “Confirming configurations in EFSM testing.” *IEEE Transactions on Software Engineering* 30.1 (2004), Pages 29–42.

[119] Alexandre Petrenko, Adenilso Simao, and José Carlos Maldonado. *Model-based testing of software and systems: recent advances and challenges*. 2012.

[120] Alexandre Petrenko, Omer Nguena Timo, and S Ramesh. “Test generation by constraint solving and FSM mutant killing.” In: *Proceedings of the IFIP International Conference on Testing Software and Systems*. Springer. 2016, Pages 36–51.

[121] Ben Pfaff et al. “The design and implementation of open vswitch.” In: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2015, Pages 117–130.

[122] P. Phaal, S. Panchen, and N. McKee. *InMon Corporation’s sFlow: A Method for Monitoring Traffic in Switched and Routed Networks*. RFC Editor. United States, 2001.

[123] Vladislav Podymov and Uliana Popesko. “UPPAAL-based software-defined network verification.” In: *Proceedings of Tools & Methods of Program Analysis (TMPA)*. IEEE. 2013, Pages 9–14.

[124] Junaid Qadir and Osman Hasan. “Applying formal methods to networking: theory, techniques, and applications.” *IEEE Communications Surveys & Tutorials* 17.1 (2015), Pages 256–291.

[125] Mark Reitblatt et al. “Abstractions for network update.” In: *Proceedings of the ACM SIGCOMM conference on Applications, technologies, architectures, and protocols for computer communication*. ACM. 2012, Pages 323–334.

[126] Mark Reitblatt et al. “Consistent updates for software-defined networks: Change you can believe in!” In: *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. ACM. 2011, Page 7.

[127] Elisa Rojas et al. “Are we ready to drive software-defined networks? A comprehensive survey on management tools and techniques.” *ACM Computing Surveys (CSUR)* 51.2 (2018), Page 27.

[128] Charalampos Rotsos et al. “OFLOPS: An open framework for OpenFlow switch evaluation.” In: *Proceedings of the International Conference on Passive and Active Network Measurement*. Springer. 2012, Pages 85–95.

[129] Natali Ruchansky and Davide Proserpio. “A (not) nice way to verify the OpenFlow switch specification: formal modelling of the OpenFlow switch using alloy.” In: *ACM SIGCOMM Computer Communication Review*. Volume 43. 4. ACM. 2013, Pages 527–528.

[130] Colin Scott et al. “Troubleshooting blackbox SDN control software with minimal causal sequences.” In: *Proceedings of the ACM SIGCOMM Conference, Chicago, Illinois, USA*. 2014.

[131] Robert Colin Scott et al. “What, where, and when: Software fault localization for sdn.” *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-178* (2012).

[132] Sandra Scott-Hayward, Christopher Kane, and Sakir Sezer. “Operationcheckpoint: Sdn application control.” In: *Proceedings of the 22nd International Conference on Network Protocols (ICNP)*. IEEE. 2014, Pages 618–623.

[133] Divjyot Sethi, Srinivas Narayana, and Sharad Malik. “Abstractions for model checking SDN controllers.” In: *Proceedings of the Formal Methods in Computer-Aided Design (FMCAD)*. IEEE. 2013, Pages 145–148.

[134] Sakir Sezer et al. “Are we ready for SDN? Implementation challenges for software-defined networks.” *IEEE Communications Magazine* 51.7 (2013), Pages 36–43.

[135] Ehab Al-Shaer and Saeed Al-Haj. “FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures.” In: *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*. ACM. 2010, Pages 37–44.

[136] Ehab Al-Shaer et al. “Network configuration in a box: Towards end-to-end verification of network reachability and security.” In: *Proceedings of the 17th International Conference on Network Protocols (ICNP)*. IEEE. 2009, Pages 123–132.

[137] Alexander Shalimov et al. “Advanced study of SDN/OpenFlow controllers.” In: *Proceedings of the 9th central & eastern european software engineering conference in russia*. ACM. 2013.

[138] Nick Shelly et al. “Destroying networks for fun (and profit).” In: *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*. ACM. 2015, Page 6.

[139] Vadym Shkarupylo and Olga Polska. “The approach to SDN Network topology verification on a basis of Temporal Logic of Actions.” In: *Proceedings of the 14th International Conference on Advanced Trends in Radioelecrtronics, Telecommunications and Computer Engineering (TCSET)*. IEEE. 2018, Pages 183–186.

[140] Richard William Skowyra et al. “Verifiably-safe software-defined networks for CPS.” In: *Proceedings of the 2nd ACM international conference on High confidence networked systems*. ACM. 2013, Pages 101–110.

[141] Richard Skowyra et al. “A verification platform for sdn-enabled applications.” In: *Proceedings of the International Conference on Cloud Engineering (IC2E)*. IEEE. 2014, Pages 337–342.

[142] *Software-Defined Networking: The New Norm for Networks*. URL: https://www.opennetworking.org.

[143] Haoyu Song. “Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane.” In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM. 2013, Pages 127–132.

[144] Gordon Stewart. “Computational verification of network programs in coq.” In: *Proceedings of the International Conference on Certified Programs and Proofs*. Springer. 2013, Pages 33–49.

[145] Radu Stoenescu et al. “Symnet: Scalable symbolic execution for modern networks.” In: *Proceedings of the ACM SIGCOMM Conference*. ACM. 2016, Pages 314–327.

[146] Radu Stoenescu et al. “Symnet: Static checking for stateful networks.” In: *Proceedings of the workshop on Hot topics in middleboxes and network function virtualization*. ACM. 2013, Pages 31–36.

[147] Fan-Hsun Tseng et al. “sPing: a user-centred debugging mechanism for software defined networks.” *IET Networks* 6.2 (2017), Pages 39–46.

[148] Tiziano Villa et al. “Explicit and implicit algorithms for binate covering problems.” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16.7 (1997), Pages 677–691.

[149] Willem Visser et al. “Model checking programs.” *Automated Software Engineering* 10.2 (2003), Pages 203–232.

[150] Juan Wang et al. “Towards a security-enhanced firewall application for openflow networks.” In: *Cyberspace Safety and Security*. Springer, 2013, Pages 92–103.

[151] Yangyang Wang, Jun Bi, and Keyao Zhang. “A tool for tracing network data plane via SDN/OpenFlow.” *China Information Sciences* 60.2 (2017), Page 022304.

[152] *What are sdn southbound apis?* URL: https://www.sdxcentral.com/networking/sdn/definitions/southbound-interface-api/.

[153] Wenfei Wu, Ying Zhang, and Sujata Banerjee. “Automatic synthesis of NF models by program analysis.” In: *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM. 2016, Pages 29–35.

[154] Yang Wu et al. “Automated Bug Removal for Software-Defined Networks.” In: *Proceedings of the NSDI*. 2017, Pages 719–733.

[155] Andreas Wundsam et al. “OFRewind: enabling record and replay troubleshooting for networks.” In: *Proceedings of the USENIX Annual Technical Conference*. USENIX Association. 2011, Pages 327–340.

[156] Wenfeng Xia et al. “A survey on software-defined networking.” *IEEE Communications Surveys & Tutorials* 17.1 (2015), Pages 27–51.

[157] Dianxiang Xu. “A tool for automated test code generation from high-level Petri nets.” In: *Proceedings of the International Conference on Application and Theory of Petri Nets and Concurrency*. Springer. 2011, Pages 308–317.

[158] Yutaka Yakuwa, Nobuyuki Tomizawa, and Toshio Tonouchi. “Efficient model checking of OpenFlow networks using SDPOR-DS.” In: *Proceedings of the 16th Network Operations and Management Symposium (APNOMS), Asia-Pacific*. IEEE. 2014, Pages 1–6.

[159] Hongkun Yang and Simon S Lam. “Real-time verification of network properties using atomic predicates.” *IEEE/ACM Transactions on Networking* 24.2 (2016), Pages 887–900.

[160] Jiangyuan Yao et al. “Formal modeling and systematic black-box testing of SDN data plane.” In: *Proceedings of the 22nd International Conference on Network Protocols (ICNP)*. IEEE. 2014, Pages 179–190.

[161] Jiangyuan Yao et al. “Test oriented formal model of SDN applications.” In: *Proceedings of the International Performance Computing and Communications Conference (IPCCC)*. IEEE. 2014, Pages 1–2.

[162] Jiangyuan Yao et al. “Testing Black-Box SDN Applications with Formal Behavior Models.” In: *Proceedings of the 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2017, Pages 110–120.

[163] Nina Yevtushenko et al. “Test Derivation for the Software Defined Networking Platforms: Novel Fault Models and Test Completeness.” In: *Proceedings of the East-West Design & Test Symposium (EWDTS)*. IEEE. 2018, Pages 1–6.

[164] Xu Yuzhuang et al. “An algebraic approach for verifying compositions of SDN components.” In: *Proceedings of the International Conference on Communications in China (ICCC Workshops)*. IEEE. 2016, Pages 1–5.

[165] Vladimir Anatol’evich Zakharov, RL Smelyansky, and Evgenii Viktorovich Chemeritsky. “A formal model and verification problems for software defined networks.” *Modelirovanie i Analiz Informacionnyh Sistem* 20.6 (2013), Pages 36–51.

[166] Andreas Zeller. “Yesterday, my program worked. Today, it does not. Why?” In: *Proceedings of the ACM SIGSOFT Software engineering notes*. Volume 24. 6. Springer-Verlag. 1999, Pages 253–267.

[167] Hongyi Zeng et al. “A survey on network troubleshooting.” *Technical Report Stanford/TR12-HPNG-061012, Stanford University, Tech. Rep.* (2012).

[168] Hongyi Zeng et al. “Automatic Test Packet Generation.” *IEEE/ACM Trans. Netw.* 22.2 (2014), Pages 554–566. ISSN: 1063-6692.

[169] Hongyi Zeng et al. “Automatic test packet generation.” In: *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM. 2012, Pages 241–252.

[170] Hongyi Zeng et al. “Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks.” In: *Proceedings of the NSDI*. Volume 14. 2014, Pages 87–99.

[171] Hui Zhang et al. “Enabling layer 2 pathlet tracing through context encoding in software-defined networking.” In: *Proceedings of the third workshop on Hot topics in software defined networking*. ACM. 2014, Pages 169–174.

[172] Peng Zhang et al. “Mind the gap: Monitoring the control-data plane consistency in software defined networks.” In: *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. ACM. 2016, Pages 19–33.

[173] Shuyuan Zhang and Sharad Malik. “SAT based verification of network data planes.” In: *Proceedings of the International Symposium on Automated Technology for Verification and Analysis*. Springer. 2013, Pages 496–505.

[174] Tianzhu Zhang et al. “Dealing with misbehaving controllers in SDN networks.” In: *Proceedings of the Global Communications Conference (GLOBECOM)*. IEEE. 2017, Pages 1–6.

[175] Ying Zhang et al. “SLA-verifier: Stateful and quantitative verification for service chaining.” In: *Proceedings of the Conference on Computer Communications (INFOCOM)*. IEEE. 2017, Pages 1–9.

[176] ZhiHao Zhang, DongMing Yuan, and HeFei Hu. “Multi-Layer Modeling of OpenFlow based on EFSM.” In: *Proceedings of the 4th International Conference on Machinery, Materials and Information Technology Applications*. 2016, Pages 524–529.

[177] Yusu Zhao et al. “CeGen: A Cost-Effective Probe Generation Scheme for Rule Verification in SDN.” *IEEE Communications Letters* (2017).

[178] Yusu Zhao et al. “SDN-enabled Rule Verification on Data Plane.” *IEEE Communications Letters* (2017), Pages 1–1.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Méthodes de Test Basées sur les Modèles pour la Validation des Réseaux Logiciel (SDN)

Mots clés : Méthodes de Test Basées sur les Modèles, Modèles Formels, Testing, Réseaux Logiciels (SDN).

Résumé : Les réseaux logiciels (connus sous l'appellation : Software Defined Networking, SDN), qui s'appuient sur le paradigme de séparation du plan de contrôle et du plan d'acheminement, ont fortement progressé ces dernières années pour permettre la programmabilité des réseaux et faciliter leur gestion. Reconnu aujourd'hui comme des architectures logicielles pilotées par des applications, offrant plus de programmabilité, de flexibilité et de simplification des infrastructures, les réseaux logiciels sont de plus en plus largement adoptés et graduellement déployés par l'ensemble des fournisseurs. Néanmoins, l'émergence de ce type d'architectures pose un ensemble de questions fondamentales sur la manière de garantir leur correct fonctionnement. L'architecture logicielle SDN est elle-même un système complexe à plusieurs composants vulnérables aux erreurs. Il est essentiel d'en assurer le bon fonctionnement avant déploiement et intégration dans les infrastructures.
Dans la littérature, la manière de réaliser cette tâche n'a été étudiée de manière approfondie qu'à l'aide de vérification formelle. Les méthodes de tests s'appuyant sur des modèles n'ont guère retenu l'attention de la communauté scientifique bien que leur pertinence et l'efficacité des tests associés ont été largement démontrés dans le domaine du développement logiciel. La création d'approches de test efficaces et réutilisables basées sur des modèles nous semble une approche appropriée avant tout déploiement de réseaux virtuels et de leurs composants. Le problème abordé dans cette thèse concerne l'utilisation de modèles formels pour garantir un comportement fonctionnel correct des architectures SDN ainsi que de leurs composants. Des approches formelles, structurées et efficaces de génération de tests sont les principales contributions de la thèse. En outre, l'automatisation du processus de test est mise en relief car elle peut en réduire considérablement les efforts et le coût.
La première contribution consiste en une méthode reposant sur l'énumération de graphes et qui vise le test fonctionnel des architectures SDN. En second lieu, une méthode basée sur un circuit logique est développée pour tester la fonctionnalité de transmission d'un commutateur SDN. Plus loin, cette dernière méthode est étendue pour tester une application d'un contrôleur SDN. De plus, une technique basée sur une machine à états finis étendus est introduite pour tester la communication commutateur-contrôleur.
Comme la qualité d'une suite de tests est généralement mesurée par sa couverture de fautes, les méthodes de test proposées introduisent différents modèles de fautes et génèrent des suites de tests avec une couverture de fautes guarantie.

Title: Model Based Testing Techniques for Software Defined Networks

Keywords: Model Based Testing, Formal Models, Testing, Software Defined Networking (SDN).

Abstract: Having gained momentum from its concept of decoupling the traffic control from the underlying traffic transmission, Software Defined Networking (SDN) is a new networking paradigm that is progressing rapidly addressing some of the long-standing challenges in computer networks. Since they are valuable and crucial for networking, SDN architectures are subject to be widely deployed and are expected to have the greatest impact in the near future. The emergence of SDN architectures raises a set of fundamental questions about how to guarantee their correctness. Although their goal is to simplify the management of networks, the challenge is that the SDN software architecture itself is a complex and multi-component system which is failure-prone. Therefore, assuring the correct functional behaviour of such architectures and related SDN components is a task of paramount importance, yet, decidedly challenging.
How to achieve this task, however, has only been intensively investigated using formal verification, with little attention paid to model based testing methods. Furthermore, the relevance of models and the efficiency of model based testing have been demonstrated for software engineering and particularly for network protocols. Thus, the creation of efficient and reusable model based testing approaches becomes an important stage before the deployment of virtual networks and related components. The problem addressed in this thesis relates to the use of *formal models* for guaranteeing the correct functional behaviour of SDN architectures and their corresponding components. Formal, and effective test generation approaches are in the primary focus of the thesis. In addition, automation of the test process is targeted as it can considerably cut the efforts and cost of testing.

The main contributions of the thesis relate to model based techniques for deriving high quality *test suites*. Firstly, a method relying on *graph enumeration* is proposed for the functional testing of SDN architectures. Secondly, a method based on *logic circuit* is developed for testing the forwarding functionality of an SDN switch. Further on, the latter method is extended to test an application of an SDN controller. Additionally, a technique based on an *extended finite state machine* is introduced for testing the switch-to-controller communication. As the quality of a test suite is usually measured by its *fault coverage*, the proposed testing methods introduce different *fault models* and seek for test suites with *guaranteed fault coverage* that can be stated as sufficient conditions for a test suite *completeness / exhaustiveness*.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France