Fault Tolerance Using an SDN Pattern Framework

Nikolaos E. Petroulakis[1,2], George Spanoudakis[2] and Ioannis G. Askoxylakis[1]
[1] Institute of Computer Science, Foundation for Research and Technology - Hellas, Heraklion, Greece
[2] Department of Computer Science, City University of London, London
Email: npetro@ics.forth.gr, g.e.spanoudakis@city.ac.uk, asko@ics.forth.gr

***Abstract*—Software Defined Networking (SDN) and Network Function Virtualization (NFV) are a promising combination for programmable connectivity, rapid service provisioning and service chaining, as they offer the necessary end-to-end optimizations. However, with the actual exponential growth of connected devices, future networks such as SDN/NFV require an open-solutions architecture, facilitated by standards and a strong ecosystem. Such networks need to support communication services that offer guarantees about fault tolerance, redundancy, resilience and security. The construction of complex networks preserving Security and Dependability (S&D) properties is necessary to avoid system vulnerabilities, which may occur in the various layers of SDN architectures. In this work, we propose a pattern framework built in an SDN controller able to import design patterns in a rule-based language and provide fault tolerance in SDN networks. To evaluate the importance and the functionality of the SDN framework, abstract pattern instances are proposed to provide network fault tolerance, detection and restoration in SDN network infrastructures.**

***Index Terms*—Software Defined Networking; SDN Pattern Framework; Fault Tolerance; Design Patterns; Security; Drools.**

I. Introduction

With the fast growth of Software Defined Networking (SDN) together with Network Function Virtualization (NFV) and their integration with 5G network architectures, the design of networks enters in a new era. In addition, SDN and NFV have been used as key networking technologies to tackle the issue of connectivity and availability under heterogeneous communication technologies. Since SDN allows network programmability and control to be decoupled from the forwarding plane, a dynamical reconfiguration of their resources, as well as keeping network connectivity, can be directly reprogrammed by the control plane. One of the challenges of SDN is to drive the reconfiguration through specifications embedded in critical infrastructures. Furthermore, applications running on top of SDN architecture need to be resource and network-aware, in order to take full advantage of underlying network programmability and to become more agile. Moreover, the usage of SDN may be also introduced to provide security in the Internet of Things (IoT) as proposed in [1]. However, the design of such networks effectively encounters difficulties that stem from the widely distribution and heterogeneous nature of SDN and the extent of intelligence, dependability and security that they need to demonstrate during their operation.

SDN requires a careful investigation of new security and dependability capabilities and risks, which have not been necessary legacy systems. The network programmability introduced by SDN is expected to allow a software-based operator control of the network through a so-called network operating system deployed on an external controller. The design and implementation of SDN infrastructures can be based on an architectural framework where the network elements are integrated through design patterns which are re-usable solutions to common problems and building blocks to architectures. In order to design and operate SDN network architectures, design patterns can be used to encode proven dependencies between individual network components and corresponding properties of orchestrations (composition) involving them. They can define generic ways of composing (i.e., establishing the connectivity between) and configuring the heterogeneous network elements and software components that may exist at all layers of the SDN implementation stack, including: switches and hosts, services, cloud, IoT enabling platforms, as well as software components at the SDN application layer. In order to achieve this, patterns specify abstract and generic interaction and orchestration protocols, enhanced (if necessary) by transformations to ensure the semantic compatibility of data.

Security and Dependability (S&D) patterns can be used for the definition of optimal paths which are able to guarantee S&D properties in deployed networks. Since SDN controllers can support management, monitoring and self-healing mechanisms of networks, the design of suitable patterns requires services to reconfigure the network to provide security, dependability, traffic engineering, load balancing, and fault tolerance mechanisms such as replication, diversity and redundancy. A first definition of our pattern-based approach for designing reliable cyber-physical systems was given in [2]. Furthermore, in [3], we extended the original approach by proposing a pattern framework as an SDN application outside the SDN controller. In this work, we present the implementation of a prototype *SDN Pattern Framework* built in an SDN controller able to provide monitoring and management capabilities of SDN network architectures with respect also to S&D based on executable patterns.

The remainder of this paper is organized as follows. In Section II, an overview of related work is presented. In Section III, we present the proposed *SDN Pattern Framework*. The framework consists of the definition of the pattern schema and the development of a pattern engine module and a pattern graphical user interface (GUI) built in an SDN controller that enable the interaction between core functionalities of the controller and the SDN network architectures. It can be

978-1-5090-5019-2/17/$31.00 ©2017 IEEE

used by developers or administrators of SDN systems, not necessarily experts in SDN controllers to (a) create designs of their systems in ways guaranteed to satisfy S&D properties such as network connectivity, fault detection, restoration and tolerance, (b) verify if existing designs systems satisfy required properties and (c) monitor and manage SDN network at runtime. In Section IV, we evaluate the pattern framework proposing abstract specification instances of SDN patterns able to handle network component as retrieved by the inventory of the controller and provide connectivity and fault tolerance on SDN. The patterns are able to install flows and decrease the failover time in case of faults and network failures providing fault tolerance in SDN network architecture. Finally, Section V provides conclusions and future work.

II. Related Work

The design of a system has been simplified and facilitated by the modelling of design patterns. With the softwarization of networks in SDN, design patterns can be applied in all the different layers of SDN architectures. Fault tolerance patterns have been proposed in works such as [4] and [5]. Although authors in [6] and [7] present fault tolerant patterns for software, the proposed architectural, detection, mitigation and recovery patterns can be also applied on SDN networks. Flow policy patterns as expressed by Frenetic languages such as Pyretic, can generate flow rules able to be installed in programmable switches of SDN networks [8]. In our approach, we also provide paths as flow rules based on the security and dependability requirements. Moreover, design patterns have been also proposed for the graph algorithms in works such as [9] and [10].

Network fault tolerance appears to be a critical topic for research [11]. Suitable fault tolerance mechanisms have been developed in SDN that intend to guarantee fault tolerance. FatTire [12] is a language for writing fault-tolerant SDN programs in terms of paths through the network and explicit fault-tolerance requirements. The main features of FatTire include fast-failover OpenFlow mechanisms, and correct behaviour during periods of failure recovery. It focuses on data plane including the fault tolerance between the path between the source and destination and intermediate functions such as IDS and firewall. Compared to our fault tolerance framework, FatTire is based on Netcore compiler and it is not compatible with controllers such as OpenDaylight(ODL)[1]. Coronet [13] on the other hand proposes a fault tolerant system for NOX SDN controller able to protect data plane link/switch failures and is based on Dijkstra's shortest path algorithm. Ravana [14] is a fault tolerant SDN controller platform evaluated in the Ryu controller that supports fault tolerance for both controller and switches. SMaRtLight [15] is a practical fault-tolerant SDN controller supporting primary and backup controllers (master slave) are used to replicate SDN controllers. LegoSDN [16] is able to tolerate SDN application failures, while AFRO [17] proposes an automatic failure recovery for POX SDN controller based on behalf of simpler, failure-agnostic controller modules.

[1]http://www.opendaylight.org

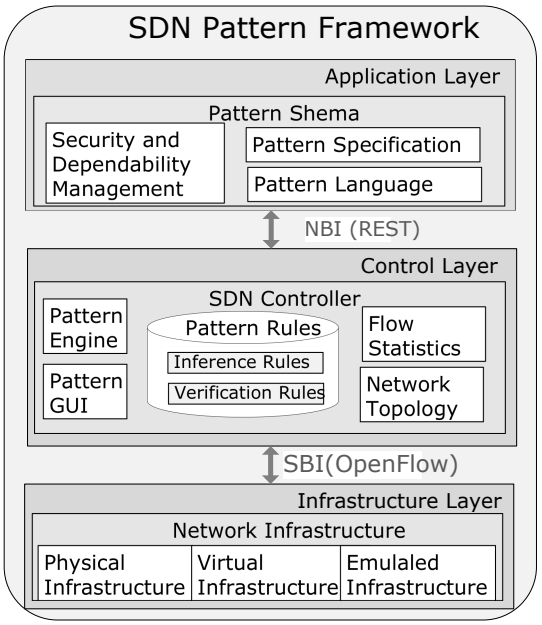


Fig. 1: Architecture of the *SDN Pattern Framework*

III. SDN Pattern Framework Architecture

One the core part of this work is the presentation of the developed *SDN Pattern Framework*. The building blocks which constitute the framework are (i) the *Pattern Schema* describing the structure of the pattern, (ii) the *Pattern Engine* module and the *Pattern Graphical User Interface (GUI)* module which are implemented in the SDN Controller, and (iii) the network elements in the *Network Infrastructure* which the controller is able to interact and apply the imported SDN patterns. The architecture of the framework and the interaction between blocks are presented in the Figure 1. The analysis of the blocks on the implemented framework and specifically the technology which are used during the design, the implementation integration and deployment of pattern framework are described in the following subsections.

A. Pattern Schema

The pattern schema includes the specification and language of SDN patterns and can be used as an instrument for designing, verifying and modifying the topology of SDN networks, at design time or runtime. At design time, the procedure includes the definition of a design problem and the required S&D property that must be guaranteed by the SDN. In verification, an existing SDN network design (topology) and the required S&D properties are provided, and patterns are applied to analyze the former and ensure that the latter is satisfied. The analysis is based on checking if the topology of the pattern matches totally or partly the network design and on the fact that the individual components that constitute the network

with the particular topology have certain properties that can guarantee end-to-end network level S&D properties. Finally, at runtime patterns are applied to alter the topology and the forwarding rules of an operational network in order to ensure the satisfaction of S&D properties.

1) Pattern Specification: The pattern specification is defined as follows:

i) **Name**: should include a clear statement of the role and the definition of the pattern.
ii) **Problem**: should analyze the problem that the pattern should solve.
iii) **Existing solutions**: should describe the current state of the art solutions for the problem.
iv) **Our solution**: should describe the pattern, emphasizing the key technical points of solution.
v) **Evaluation**: should provide an evaluation of the pattern either experimentally or formally.
vi) **Contributions**: should define the contributions and advances of the proposed pattern over state of the art existing solutions.

2) Pattern Language: SDN patterns can be expressed as production rules to enable reasoning. In implementing our approach, we have selected Drools rule engine to express SDN patterns as rules, as this supports backward and forward chaining inference and verification by implementing and extending the Rete algorithm [18]. Drools rules can encode the topology of a pattern and the process of finding suitable component compositions in order to guarantee the required property. Drools production rules are being stored in the production memory and are used to process data inserted in the working memory (Knowledge Base) as facts by pattern matching. Each rule consists of two parts: the *when* condition and the *then* actions. When a network that matches the topology of an SDN pattern does not satisfy the required property, the pattern may be used to substitute, add or remove components in order to satisfy the property. A Drools rule that encodes an SDN pattern includes the inputs of the pattern's components, the type of composition and the required property in Left Hand Side (LHS). When the conditions in the LHS are satisfied, then the rule is fired to execute the actions as described in its Right Hand Side (RHS). In the RHS, the new requirements of the compositions or atomic components can be inserted, updated or deleted.

In order to specify and express SDN patterns, the semantics of the pattern language should be defined. In Table I the most useful semantics are presented. In the LHS, the network components which constitute the topology of the pattern are defined. Different network topology facts such as *Nodes*, *Links* and *Flows* are included in the list. Moreover, the *Requirement* represents the constraints of the topology and the required property. In the RHS, the pattern provides the solution by inserting, modifying, updating or retracting facts from the knowledge base which will also update the inventory list in the controller. Each component is converted through the respective Java class to an understandable format to the SDN controller. Finally, the semantics of Drools language give the potentiality to represent more complex patterns by adding more variables and pattern properties.

TABLE I: Pattern Language Semantics

Type	Syntax	Description
rule	rule "name"	name of the rule
Left Hand Side (LSH)		
when	**Network Pattern Elements (Facts)**	
	Node (address, ports, txPackets, rxPackets)	match network nodes such as switches and hosts
	Link (srcId, srcPort, destId, destPort)	match links between source and destination nodes
	Path (srcId, destId)	match paths between source node intermediate links and destination node
	Flow (switchId, inPort, outPort, priority)	match flow rules between nodes
	Requirement (src, dest, pro, satisfied)	match requirements of pattern such as source, destination and property and satisfied
	Conditional Elements	
	==	match conditions
	contains	contains object (logical)
	not	not match (logical)
	!=	not match (arithmetic)
Right Hand Side (RSH)		
then	**Actions**	
	modify ($fact){pro=pro'}	modify knowledge base fact
	retract ($fact)	retract knowledge base fact
	insert (new Fact ())	insert knowledge base fact
	update ($fact)	update knowledge base fact
	Java commands	other Java language syntax

B. SDN Controller

The SDN Pattern Framework is based on the ODL SDN controller as it provides a wide range of abstractions and functionalities that facilitate controller application development and a numerous built-in modules that can be either reused or even extended. ODL is selected as an open source controller which has attracted a lot of attention in the networking area with more active contributors compared to other open-source solutions. The controller is based on Apache Karaf and uses Apache Maven[2] for building projects. To implement functionalities in ODL, certain purpose-built modules as well as enhancements to existing SDN controller modules need to be installed and set up properly. Finally, for the evaluation and testing of the SDN Pattern Framework different software tools such as Eclipse[3], Git[4], and Postman[5] are necessary for the

[2]https://karaf.apache.org, https://maven.apache.org
[3]http://www.eclipse.org
[4]https://git-scm.com/
[5]https://www.getpostman.com

development of the *Pattern Engine* and *Pattern GUI* modules ODL modules as is described below.

1) Pattern Engine Module: This module has been created to enable the capability to insert, modify, execute and retract patterns at design or at runtime in the SDN controller. Since Drools rule engine is based on Maven, one core part of this framework is the integration of all the dependencies in the ODL controller, as well as the integration of the entities that interact with the controller to run Drools at design and at runtime. As ODL does not support Drools Maven libraries by default, some modifications must be done in order to import the required packages (knowledge-api, drools-core, drools-compiler, drools-templates, drools-decisiontables). Furthermore, to support insertion, modification and deletion of facts and rules by administrators or users, suitable northbound interfaces (YANG APIs and the respective REST APIs) are also implemented. Finally, appropriate YANG interfaces are also implemented to retrieve network components such as switches, service functions and end-hosts, active links and statistics from the controller as required by the Drools patterns.

2) Pattern GUI Module: This module has been developed as an additional module on the ODL controller to monitor, manage and assess the proof of concept implementation of the SDN Pattern Framework. Patterns can be inserted as Drools rule as a plain text or as an external file in Json format. Suitable javascripts and html files have been implemented in order to support the insertion of patterns either at design or at runtime. More precisely, angularJS[6] is used as a javascript-based library for front-end web application. The use of predefined pattern/rule templates is also considered to be enhanced in a following version as a more convenient way to manage patterns. The imported patterns are also previewed automatically in an interactive table as presented in the Figure 2. Appropriate APIs have been implemented in order an administrator to be able to insert, modify, delete and enable/disable imported rules using the implemented controls.

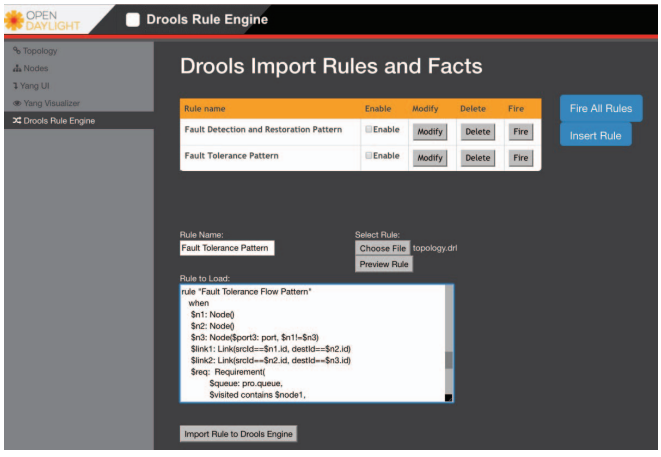


Fig. 2: OpenDaylight SDN Pattern Framework GUI

[6]https://angularjs.org

C. Network Infrastructure

The last part of the SDN Pattern Framework includes the infrastructure of the network. The framework is able to interact with SDN architectures to provide security and dependability in SDN networks. Apart from the SDN controller, which is the core part of SDN infrastructure, end hosts such as clients, servers, service functions together with programmable switches are also required. Since ODL supports a number of south-bound protocols such as OpenFlow, this capability enables the interaction with infrastructures as provided by physical networks including hardware switches, hosts, virtual infrastructures such as OpenStack[7] or network emulators such as Mininet[8] platform suitable for testing and emulation.

IV. Fault Tolerance SDN Patterns

In this section, SDN fault tolerance patterns are presented based on the pattern schema and evaluated in the proposed SDN pattern framework.

A. Name

The family of the fault tolerance SDN patterns includes (i) the Fault Tolerance Flow Pattern and (ii) the Fault Detection and Restoration Pattern.

B. Problem

The use of SDN networks imposes the necessity to provide higher fault and intrusion tolerance compared to legacy networks as new threats are being introduced. More specifically, the ability to control networks by means of software and centralization of network control makes SDN vulnerable to failures or attacks. Fault tolerance in network architectures requires the design of a network able to guarantee avoidance of single or multiple link failures, faulty end-hosts and switches, or attacks.

C. Existing solutions in SDN

The most common solutions to guarantee fault tolerance and avoid single point of failure in SDN, include the replication of paths forwarding traffic in parallel, the use of redundant paths and the ability to switch in case of failure (failover) and traffic diversity. As described in Section II, data plane fault tolerance solutions exist in the bibliography such as FatTire and Coronet. However, existing data plane fault tolerance solutions do not provide the required open and flexible design as our proposed approach.

D. Our solution

The key technical solutions of the problem include the creation of fault tolerance SDN patterns based on the defined pattern schema. The abstract forms of the fault tolerance patterns, as expressed by Drools rules, can establish a fault tolerance mechanism for the detection and restoration of network faults or attacks and can be applied first locally and then globally as presented below.

[7]https://www.openstack.org
[8]https://www.mininet.org

1) Fault Tolerance Flow Pattern: This pattern is able to provide proactively connectivity and path protection solutions for fast failure recovery against faults by identifying the shortest path and installing the respective flows to preplan and reserve paths, which is extremely valuable for large-scale SDN systems.

Pattern Topology: The discovery of suitable interconnected nodes such as hosts and switches on existing network infrastructures can be done by pattern matching. This includes the identification of all the available paths and choose the most appropriate ones in order to define the degree of the redundancy of the network topology.

Pattern Requirement: The pattern constraints are defined as requirements which represent the property that the pattern guarantee such as fault tolerance.. This is based on the Dijkstra algorith which adapts breadth-first algorithm to find single source shortest path.

Pattern Action: When the pattern identifies the shorted path between source and destination, the actions of the pattern includes the installation of suitable flow rules in the openflow-enabled switches. The describe pattern as Drools rule is presented in the Rule 1.

Rule 1: Fault Tolerance Pattern Rule

```
rule "Fault Tolerance Pattern"
  when
    $n1: Node()
    $n2: Node()
    $n3: Node($port3: port, $n1!=$n3)
    $link1: Link(srcId==$n1.id, destId==$n2.id)
    $link2: Link(srcId==$n2.id, destId==$n3.id)
    $req:  Requirement(
              pro.name =="Fault Tolerance",
              pro.src!=$n1, pro.dest!=$n3,
              $visited: pro.visited,
              $queue: pro.queue,
              $visited contains $n2,
              $queue not contains $n2,
              $visited not contains $n3)
then
   $queue.remove($n2);
   $queue.add($n3);
   $visited.add($n3);
   update($queue);
   update($visited);
    Flow flow = new Flow(
        $n2.id, $link1.destPort, $link2.srcPort);
   insert(flow);
end
```

2) Fault Detection and Restoration Pattern: Apart from the proactive definition of the respective flows, this pattern provides a reactive mechanism to dynamically allocate path flows for fast fault detection and restoration, which is required to detect link failures and packet losses in order to restore network availability.

Pattern Topology: The pattern topology includes the source node, the destination nodes and the active path for data transmission.

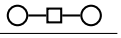
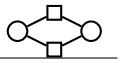
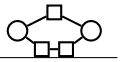
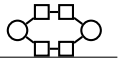
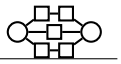
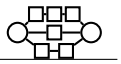
Pattern Requirement: When there are dropped packets between the two nodes or the link is down, the pattern detects it as failure.

Pattern Action: After the detection of such failure, the flow rules is removed from the inventory list and a new requirement is inserted in the knowledge base which activates the fault tolerance flow pattern which is able to restore the link by finding alternative paths. In Rule 2, an abstract overview of the pattern rule can be found.

Rule 2: Fault Detection and Restoration Pattern Rule

```
rule "Fault Detection and Restoration Pattern"
  when
    $n1: Node($tx: txPacket)
    $n2: Node($rx: rxPacket)
    $path: Path(src==$n1.id,destination==$n2.id)
    $req: Requirement(
            pro.name =="Fault Detection", $tx>$rx)
then
  retract($path);
   insert(new Requirement($n1, $n2,
                          "Fault Tolerance"));
end
```

TABLE II: Flow Configuration Experiments

Topo	Hosts	Switches	Links	Config Time	Degree	Topology
1	2	1	2	0.1sec	1	
2	2	2	4	0.3sec	2	
3	2	3	5	0.4sec	2	
4	2	3	6	0.6sec	2	
5	2	3	8	1.1sec	3	
6	2	6	9	1.3sec	3	

E. Evaluation

To evaluate the fault tolerance patterns in the SDN Pattern framework, we conducted different types of preliminary experiments in the Mininet emulator. The first set of experiments evaluates the fault tolerance pattern as for the needed time needed for the installation of the required flows for different custom topologies including different number of hosts, switches and links as presented in Table II. The second set includes the evaluation on the time needed to detect and restore path, if possible, in case of link failures. The purpose of this experiments is to send 10 packets from source to destination for the different network topologies as were defined in the previous experiments. Since the fault tolerance pattern defines the shortest path between source and destination, the purpose of the experiment is to measure the detection and restoration time in case of a failure. The results of the experiments can be found in Figure 3.

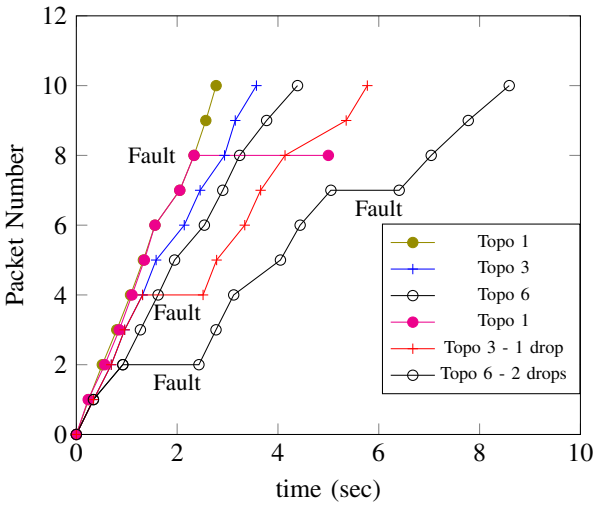


Fig. 3: Fault Detection and Restoration Experiments

F. Contributions

The contributions of our proposed proposed patterns in the SDN pattern framework can be summarized as follows:

- Fault tolerance patterns can create a proactive and reactive mechanism able to handle failures and faults.
- Other proposed fault tolerance mechanisms (FatTire, Pyretic) are not implemented for the ODL controller but for other controllers such as POX and NOX.
- The proposed pattern language can be used by experts and non experts programmers as it offers a interface for fast interaction with the controller.
- The open architecture of our proposed patterns can be strengthened by the addition of more S&D patterns.
- The proven capability of the pattern framework through the patterns at runtime gives great potentiality for further development.

V. Conclusions and Future Work

This work presented the development of an SDN Pattern Framework able to handle faults and failures in SDN network infrastructures. The flexibility of the framework to insert patterns as Drools rules in the controller, shows the capability of the controller to guarantee properties and handle incidents. Moreover, the proposed pattern language, is easily understandable and open to modifications based on the required need of administrators and users. There is also no need to modify internal controller modules since the interaction is applied through pattern rules. Thus, the framework can be easily extended to interact with multiple controller capabilities and functionalities. Finally, as future work we intend to use and extend the automated reactiveness of the framework, ant to also evaluate for Byzantine Fault Tolerance and Service Function Chaining (SFC) as proposed in our work [19] via the development of the associated S&D patterns.

Acknowledgement

This work has received funding from the European Unions Horizon2020 research and innovation programme under the Marie Skodowska-Curie grant agreement No734815 (CyberSure).

References

[1] Ricard Vilalta, Raluca Ciungu, Arturo Mayoral, Ramon Casellas, Ricardo Martinez, David Pubill, Jordi Serra, Raul Munoz, and Christos Verikoukis. Improving security in internet of things with software defined networking. In *Global Communications Conference (GLOBECOM), 2016 IEEE*, pages 1–6. IEEE, 2016.

[2] Nikolaos E Petroulakis, George Spanoudakis, Ioannis G Askoxylakis, Andreas Miaoudakis, and Apostolos Traganitis. A pattern-based approach for designing reliable cyber-physical systems. In *Global Communications Conference (GLOBECOM), 2015 IEEE*, pages 1–6. IEEE, 2015.

[3] Nikolaos E Petroulakis, George Spanoudakis, and Ioannis G Askoxylakis. Patterns for the design of secure and dependable software defined networks. *Computer Networks*, 109:39–49, 2016.

[4] Matthias Tichy, Daniela Schilling, and Holger Giese. Design of self-managing dependable systems with uml and fault tolerance patterns. In *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 105–109. ACM, 2004.

[5] Titos Saridakis. A system of patterns for fault tolerance. In *EuroPLoP*, pages 535–582, 2002.

[6] Robert Hanmer. *Patterns for fault tolerant software*. John Wiley & Sons, 2013.

[7] Ferran Adelantado and Christos Verikoukis. Detection of malicious users in cognitive radio ad hoc networks: A non-parametric statistical approach. *Ad Hoc Networks*, 11(8):2367–2380, 2013.

[8] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. Modular sdn programming with pyretic. *Technical Report of USENIX*, 2013.

[9] Dietmar Kühl. Design patterns for the implementation of graph algorithms. In *Master's thesis, Techinische Universität*. Citeseer, 1996.

[10] Wenfei Fan, Xin Wang, Yinghui Wu, and Jingbo Xu. Association rules with graph patterns. *Proceedings of the VLDB Endowment*, 8(12):1502–1513, 2015.

[11] Jue Chen, Jinbang Chen, Fei Xu, Min Yin, and Wei Zhang. *When Software Defined Networks Meet Fault Tolerance: A Survey*, pages 351–368. Springer International Publishing, Cham, 2015.

[12] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. Fattire: Declarative fault tolerance for software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 109–114. ACM, 2013.

[13] Hyojoon Kim, Mike Schlansker, Jose Renato Santos, Jean Tourrilhes, Yoshio Turner, and Nick Feamster. Coronet: Fault tolerance for software defined networks. In *2012 20th IEEE International Conference on Network Protocols (ICNP)*, pages 1–2. IEEE, 2012.

[14] Naga Katta, Haoyu Zhang, Michael Freedman, and Jennifer Rexford. Ravana: Controller fault-tolerance in software-defined networking. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, page 4. ACM, 2015.

[15] Fábio Botelho, Alysson Bessani, Fernando MV Ramos, and Paulo Ferreira. On the design of practical fault-tolerant sdn controllers. pages 73–78, 2014.

[16] Balakrishnan Chandrasekaran and Theophilus Benson. Tolerating sdn application failures with legosdn. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, page 22. ACM, 2014.

[17] Maciej Kuźniar, Peter Perešíni, Nedeljko Vasić, Marco Canini, and Dejan Kostić. Automatic failure recovery for software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 159–160. ACM, 2013.

[18] Drools. Business rules gement system solution. www.drools.org.

[19] Konstantinos Fysarakis, Nikolaos E. Petroulakis, Andreas Roos, Khawar Abbasi, Petra Vizaretta, George Petropoulos, Ermin Sakic, George Spanoudakis, and Ioannis Askoxylakis. A Reactive Security Framework for Operational Wind Parks Using Service Function Chaining. In *22nd IEEE Symposium on Computers and Communications*, 2017.