Correct Software by Design for Software-Defined Networking: A Preliminary Study

Liang Hao, Xin Sun, Lan Lin, Zedong Peng
Department of Computer Science, Ball State University, Muncie, IN 47306, USA
{lhao, xsun6, llin4, zzpeng}@bsu.edu

Abstract

We report our experience of applying rigorous software specification and design methodologies to the development of applications for the emerging software-defined networking (SDN) paradigm. While much of the prior work in the SDN space focused on creating novel algorithms and protocols, in this paper we take the position that the implementation of those algorithms and protocols on the SDN platform is a hard problem on its own that deserves a systematic treatment from the software engineering perspective. Through a concrete case study of implementing an essential switching algorithm as an SDN app, we expose the challenges stemmed from the unique three-tier architecture of SDN, and propose a rigorous approach that flows from functional requirements through stepwise refinement to design and implementation. Our case study shows promises of the proposed approach in supporting correctness arguments for the software developed for the SDN platform.

1 Introduction

Software-defined networking (SDN) is an emerging technology that has completely transformed modern networking, with widening adoptions in industries such as IT, telecommunications, retails, and healthcare, to name a few. At the core of the technology is the ability to apply software solutions to hard, long-standing networking problems while cutting the operation costs via automation.

Until very recently the focus of the SDN community was on creating new and SDN-specific algorithms and protocols that can take advantage of the unique capabilities of SDN to solve sophisticated networking problems, such as highly-dynamic, fine-grained traffic engineering and adaptive intrusion detection. However, little attention was given to the software engineering aspects of app development, as it was assumed that the implementation of those algorithms and protocols was straightforward. More recently there has been some work, including the authors' own, on the testing and orchestration of the SDN apps that leveraged techniques from the software engineering discipline. But to the best of our knowledge, there has not been a systematic investigation on the implementation of the SDN apps guided by software engineering principles and methodologies.

In this paper we take the position that implementing SDN apps is a hard problem on its own that deserves a systematic treatment from the software engineering perspective. We present a case study of implementing a basic yet essential switching algorithm on the SDN platform. Our solution takes two iterations of rigorous software specification and design. It flows from functional requirements through stepwise refinement to design and implementation. We report our experience that shows promises of the proposed approach, which also supports correctness arguments for the software developed for the SDN platform.

2 The MAC Learning Algorithm

We introduce in this section a preliminary case study, i.e., the MAC learning algorithm. We first describe the algorithm and its implementation on a traditional switch. We then describe how the SDN architecture differs fundamentally from the traditional network architecture, and expose the challenges of migrating the same algorithm to SDN that stem from the architectural difference.

2.1 On a Traditional Switch

The MAC learning algorithm is implemented on every traditional switch, typically in the firmware. It is the core switching algorithm that enables a switch to forward packets toward their destinations. The algorithm builds the switch table and at the same time utilizes the table to determine the switch port to which a packet should be directed. An entry in the switch table contains (1) the hardware identification number, termed *Media Access Control (MAC) address*, of some host or router in the network; this address is

DOI reference number: 10.18293/SEKE2020-081

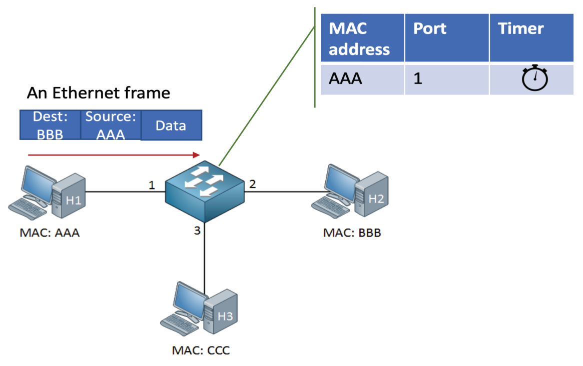


Figure 1. The learning algorithm working on a traditional network

used as the key for indexing the table, (2) the switch port leading toward that MAC address, and (3) a timer to delete the entry in the future in case it becomes stale. The table is stored in the memory and is initially empty. More specifically, the algorithm has the following components (also illustrated in Figure 1):

• For each incoming packet received on a port, the switch creates an entry in the switch table if such an entry does not exist. The entry contains (1) the MAC address in the packet's *source* address field, (2) the port from which the packet arrived, and (3) a timer set to expire after some period of time. If such an entry already exists, the timer will be refreshed. If there exists an entry with the same key (i.e., the MAC address) but a different port, the port will be updated based on the new information, and the timer be reset.

• For each incoming packet, the switch uses the MAC address in the packet's *destination* address field to look up the table. If the MAC address is listed, the switch will send the packet out the associated port; otherwise, the switch will *flood* the packet out all active ports except the incoming port (the port the packet was received on).

• The switch deletes an entry in the table when the associated timer expires. This is to handle potential topology change, e.g., hosts being removed or relocated.

2.2 The SDN Architecture

The unique three-tier architecture of SDN (illustrated in Figure 2) has important implications on the development of software for SDN, so we briefly describe it here. At the bottom tier are the hardware boxes, commonly called *SDN switches*. Compared to hardware boxes (such as switches and routers) in a traditional network, SDN switches are much dumber (and also cheaper). A traditional switch or router has the intelligence, provided by the device firmware, to decide for itself how to handle incoming packets, as the firmware implements various networking algorithms and protocols. In contrast an SDN switch does not have such

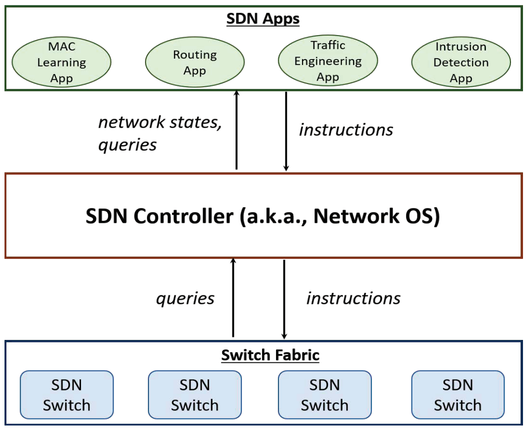


Figure 2. The three-tier SDN architecture

intelligence; it relies on the controller (the middle tier in the SDN architecture) to provide "instructions" (technically called *FlowMod messages* or simply *FlowMods*) on how to handle packets and then act accordingly. The controller is a software platform that runs on any commodity PC server, and can be viewed as the operating system for the network. On the one hand the controller interacts with the hardware boxes and provides instructions (i.e., FlowMods) to them upon request. On the other hand it provides a set of APIs that supports individual SDN apps running on top of it. The SDN apps (the top tier) collectively implement the intelligence of the network. Each app typically manages/optimizes one aspect of the network, such as switching, routing, traffic engineering, intrusion detection, etc. They obtain an abstract representation of the network state (e.g., topology, traffic load) from the controller and output to the controller instructions on how the network state should be modified and how incoming packets should be handled. The controller then compiles the instructions received from all apps to generate FlowMod messages and sends FlowMod messages to the SDN switches.

2.3 The Challenges of Migrating the Algorithm to SDN

As explained above, in the SDN paradigm the MAC learning algorithm is to be implemented as an app running on top of the controller. This app will be responsible for building the tables, one for each switch. The switches are only capable of querying the controller (which in turn consults the app) to obtain necessary instructions in the form of FlowMod messages to forward incoming packets. Such a query from a switch contains the source and destination MAC addresses of the packet and the switch port on which the packet was received. This allows the app to create or update the entry corresponding to the source MAC address in the querying switch's table. A FlowMod message from the controller back to the switch is in the format of "send any packet with source MAC address s and destination MAC address d to the port p" or "flood any packet with source MAC address s and destination MAC address d".

It is important to note that, the SDN architecture requires the querying switch to *cache* any FlowMod message received from the controller in a local table structure called *FlowMod table*. The switch will then use the cached FlowMod messages to process subsequent packets with the *same source and destination MAC addresses*, without querying the controller again. A switch may cache as many instructions as its memory space permits. As a critical measure to save memory space, any cached instruction will be deleted after it has not been utilized for a period of time. This caching technique is critical to optimizing the packet processing speed on the switches, as querying the controller introduces significant delays. As an example, imagine that a large file is being transmitted over a local-area network from the host s to the host d. Tens of thousands of packets will be transmitted, all with the same source and destination MAC addresses. A switch w queries the controller when it receives the *first* packet and caches the FlowMod message from the controller. The switch will then use the cached FlowMod to forward all subsequent packets of the same file without querying the controller again. Hence the controller will only see the first packet of that file.

The fact that the SDN controller, and consequently all apps running on top it, only see a *small fraction* of all the packets in the network has an important implication on the implementation of the MAC learning algorithm, or more specifically, on refreshing the timers associated with table entries. Recall that an entry will be refreshed every time a packet from the same source MAC address is received. Because the controller does not see most of the packets, it will not be able to effectively refresh the timers. As a result, many of the timers will unnecessarily expire, causing the app to instruct the switches to flood much more frequently than necessary. Continuing from the file transmission example and imagining the switch w receives a new packet with s as the *destination* MAC address shortly after the file transmission. It queries the controller which in turn consults the app. But the app has timed out the table entry corresponding to the MAC address s because it did not see any of the subsequent packets of that file. Thus the app, through the controller, will instruct the switch to flood the packet. Clearly the flooding is unnecessary in this case because the information contained in the expired table entry is still valid. As flooding causes significant bandwidth overhead that degrades the network performance, it is highly undesirable. (On the other hand, the switch does see all the packets of the file. We will explore in Section 5 how this knowledge of the switch may be leveraged to prevent the MAC learning app from unnecessarily timing out table entries.)

3 Leveraging Rigorous Software Specification and Design Methodologies

Developing a reliable SDN app, just as developing a reliable piece of *any* software, relies on rigorous methods for code development and testing, and a development process that is based on more than heuristics. In our opinion, each SDN app should first be treated as a black box, and flow naturally through a sequence of requirements specification, design, implementation, and testing steps. In migrating the MAC learning algorithm from traditional network to SDN, we applied two rigorous methods for software specification and design, i.e., Prowell and Poore's *sequence-based specification and stepwise refinement* [10, 12, 13] and Exman's *linear software models and the modularity matrix* [8, 7].

Sequence-based specification was developed in the 90's by the University of Tennessee Software Quality Research Laboratory. It converts ordinary, functional requirements to a precise specification that defines software's response to any possible input sequence, through a systematic *sequence enumeration* process. In this process, sequences of system inputs are enumerated in length-lexicographic order and mapped to software's outputs, and grouped in equivalence classes based on behavior described in (software) requirements. The completed enumeration encodes a formal model in the form of a finite state machine (a Mealy machine) that is refined into design and implementation [10, 12, 13].

Linear software models and the modularity matrix were recently developed by Iaakov Exman in the study of real software system composition, as a formal theory of modularity. He proposed that the composition of a software system can be represented by a modularity matrix, whose rows and columns represent *functionals* (a generalization of methods) and *structors* (a generalization of classes), respectively, and 1/0-valued matrix elements indicate asserted links (associations) (or lack of) between rows (functionals) and columns (structors). He proved that a standard modularity matrix, in which one has only linearly independent structors and functionals, must be both square and block-diagonal, with disjoint diagonal blocks representing independent system modules. He showed that canonical systems strictly obey linear software models, and larger systems tend to agree with bordered linear software models with a few outliers near the diagonal block borders. The outliers point to areas of coupling that need to be resolved in system design [8, 7].

We first applied sequence-based specification to derive a rigorous specification from functional requirements for our chosen case study, and refined it into a state-based specification and design. Then we applied the modularity matrix to validate the modular design refined from the formal specification. We derived the specification in two iterations, with new findings at the end of the first iteration, which we incorporated into the second iteration's work product.

4 Our Solution: The First Iteration

We started with a natural language description of the behavior of the MAC learning algorithm, i.e., the software requirements, as shown in Table 1. In developing the requirements we also identified a system boundary that cuts the interfaces between the software and its external entities in the software's environment, i.e., the switches (communication with the switches is through the SDN controller), the memory (that stores the lookup tables for the switches), and the timers. Figure 3 depicts our identified system boundary for the first increment.

Table 1. MAC learning algorithm requirements: The first increment

Tag	Requirement
1	On receiving a packet with source MAC address sa, destination MAC address da from switch s and in-port p, if the lookup table for s does not contain entries for either sa or da, the learning switch should add a new entry (sa, p, t) to the same lookup table, start timer t, and flood the same packet to all the ports of s except the in-port p.
2	The output of the learning switch is solely determined by the incoming packet information, the current lookup table status, and the timer events, as encapsulated in the most recent input.
3	On receiving a packet with source MAC address sa, destination MAC address da from switch s and in-port p, if the lookup table for s does not contain an entry for sa but contains an entry for da, the learning switch should add a new entry (sa, p, t) to the same lookup table, start timer t, and forward the same packet to the port of da as specified in the lookup table (for s).
4	On receiving a packet with source MAC address sa, destination MAC address da from switch s and in-port p, if the lookup table for s contains an entry for sa but does not contain an entry for da, the learning switch should overwrite the entry (sa, p, t) to the same lookup table, restart timer t, and flood the same packet to all the ports of s except the in-port p.
5	On receiving a packet with source MAC address sa, destination MAC address da from switch s and in-port p, if the lookup table for s contains entries for both sa and da, the learning switch should overwrite the entry (sa, p, t) to the same lookup table, restart timer t, and forward the same packet to the port of da as specified in the lookup table (for s).
6	On receiving a timer going off event for the MAC address sa in the lookup table for switch s, the learning switch should delete the entry (sa, p, t) from the same lookup table, and stop timer t.

From the identified system boundary we collected software's inputs (stimuli) and outputs (responses), as shown in Table 2 and Table 3.

The sequence enumeration proceeds as follows. One explicitly enumerates all possible stimulus sequences first according to the length, and within the same length lexicographically. For each enumerated sequence, one maps it to a software's response based on the requirements, and declares it equivalent to a prior sequence if both sequences take the software to the same situation (i.e., internal state). If a stimulus sequence is operationally not realizable (for instance, when a button-pressing event happens before the power-on event), the sequence is mapped to a special *illegal* response, otherwise, it is *legal*. If a sequence is declared equivalent to a prior sequence, it is *reduced*, otherwise, it is *unreduced*. One proceeds from Length n to Length $n + 1$ only extending both legal and unreduced sequences (by every stimulus), until there are no more sequences to extend. At that point the enumeration is complete.

Table 2. MAC learning algorithm stimuli: The first increment

Stimulus (Parameterized)	Shorthand
Packet(source MAC address, destination MAC address, switch, in-port)	$pa(sa, da, s, p)$
Look up table(switch)	$lt(s)$
Timer going off(switch, source MAC address)	$t(s, sa)$

Table 3. MAC learning algorithm responses: The first increment

Response	Shorthand
Forward	forward
Flood	flood
Add a new entry to the lookup table	add
Overwrite an existing entry in the lookup table	overwrite
Delete an existing (expired) entry from the lookup table	delete
Start timer	start
Restart timer	restart
Stop timer	stop

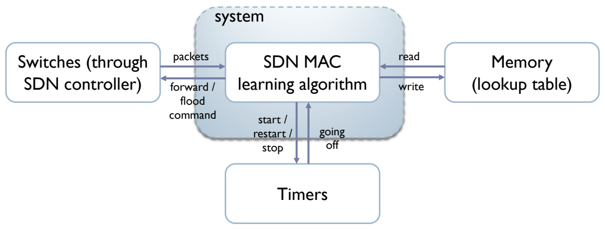


Figure 3. The system boundary for the first increment

An enumeration of the learning switch algorithm is shown in Table 4. The columns are stimulus sequences, their mapped responses, and requirements traces, respectively. We omit the column that shows reductions to prior sequences (as explained later, all the enumerated sequences are reduced to the first sequence in the table). We started with the empty sequence λ. All the others are Length 1 sequences with either an incoming packet ($pa(sa, da, s, p)$) as the current input, or a timer going off event ($t(s, sa)$). When a packet comes in, we used predicates (in square brackets) to refine the condition based on the lookup table status, i.e., whether the source address sa and the destination address da are in the lookup table for switch s, respectively, in order to define the software's unique response deterministically. It turned out that all the Length 1 sequences are reduced to λ, suggesting a stateless software control that maps current input (with predicate refinement) to current output. We held the following assumptions in sequence enumeration: (1) The $pa(sa, da, s, p)$ and $t(s, sa)$ events are queued by the SDN controller for processing, hence cannot happen simultaneously; and (2) All the timers are set to go off after the same time interval once started/restarted.

Table 4. A MAC learning algorithm enumeration: The first increment

Sequence	Response	Trace
λ	0	Method
$pa(sa, da, s, p)[sa \notin lt(s), da \notin lt(s)]$	add (sa, p, t) to $lt(s)$, start t, flood $pa(sa, da, s, p)$ to all the ports of s except p	$1, 2$
$pa(sa, da, s, p)[sa \notin lt(s), da \in lt(s)]$	add (sa, p, t) to $lt(s)$, start t, forward $pa(sa, da, s, p)$ to the port of da in $lt(s)$	$2, 3$
$pa(sa, da, s, p)[sa \in lt(s), da \notin lt(s)]$	overwrite (sa, p, t) in $lt(s)$, restart t, flood $pa(sa, da, s, p)$ to all the ports of s except p	$2, 4$
$pa(sa, da, s, p)[sa \in lt(s), da \in lt(s)]$	overwrite (sa, p, t) in $lt(s)$, restart t, forward $pa(sa, da, s, p)$ to the port of da in $lt(s)$	$2, 5$
$t(s, sa)$	delete (sa, p, t) in $lt(s)$, stop t	$2, 6$

As one could observe, our first iteration of the specification was a direct, intuitive migration of the traditional MAC learning algorithm, replacing distributed intelligence with centralized intelligence, solely based on the algorithm's behavior. What were overlooked are the constraints enforced by the unique SDN architecture, i.e., the different roles taken by the controller and the switches, as well as the caching of instructions (in the form of FlowMod messages) on the switches. Without considering these requirements the controller would see and handle every packet that goes through every switch, making itself excessively "fat" and causing unnecessary, significant delays to degrade network performance. This observation led to our second iteration of the specification to be discussed next.

5 Our Solution: The Second Iteration

In the second iteration we took into consideration important architectural differences introduced by SDN, i.e., the two-level architecture in which the controller and the switches take on different roles. We notice the following for the SDN environment:

• Lookup tables are on the controller, rather than on the switches, and are outside of the system boundary (of the specified learning algorithm).

• The controller only sees the packets that the switches do not know how to handle (forward).

• The controller sends the switches FlowMod messages that are maintained by the switches. The FlowMod messages have different information packed than the information packed in a lookup table entry.

Table 5. MAC learning algorithm requirements: The second increment

Tag	Requirement
1	The SDN controller maintains a lookup table for each switch mapping MAC addresses to ports on that switch. Each switch maintains a FlowMod table that maps (source MAC address, destination MAC address, in-port) to (out-port, timer value). Recall from Section 2.3 that the FlowMod table contains the cached FlowMod messages which are instructions from the controller to the switch.
2	On receiving a packet with source MAC address sa, destination MAC address da, in-port p of switch s, if the FlowMod table of switch s has an entry for (sa, da, p), denoted by (op, t), the switch will forward the packet to port op, and restart timer with value t; otherwise, it will send the packet on to the controller for processing.
3	When the controller receives a packet with source MAC address sa, destination MAC address da from switch s and in-port p, if the lookup table for s does not contain an entry for da, the learning switch should add/overwrite an entry (sa, p) to the lookup table, and flood the same packet to all the ports of s except the in-port p.
4	The output of the learning switch is solely determined by the incoming packet information, the current lookup table status, or the FlowRemoved message, as encapsulated in the most recent input.
5	When the controller receives a packet with source MAC address sa, destination MAC address da from switch s and in-port p, if the lookup table for s contains an entry for da (with out-port op), the learning switch should add/overwrite an entry (sa, p) to the lookup table, forward the same packet on to port op, and write a FlowMod message $(s, sa, da, p, op, t, add)$ and its reversed FlowMod message $(s, da, sa, op, p, t, add)$, where t is the timer value, to switch s.
6	When a FlowMod table entry expires, the switch automatically removes it from the FlowMod table, and sends a FlowRemoved message to the controller. When the controller receives a FlowRemoved message from switch s, with source MAC address sa, destination MAC address da, and in-port p, it deletes the lookup table entry (sa, p) for s, and sends a FlowMod message $(s, da, sa, p, delete)$ to remove the reversed FlowMod table entry maintained by s.
7	Any FlowMod table entry maintained by the switch that goes stale (it only goes stale when the out-port becomes incorrect and cannot reach the destination) must eventually time out (expire) to prevent packet loss.

• No timer is associated with lookup table entries. Timer effect is simulated on the switches.

The new knowledge we learned contributed to our derived requirements for the second increment shown in Table 5. A new system boundary was identified as depicted in Figure 4, from which we defined stimuli and responses in Table 6 and Table 7.

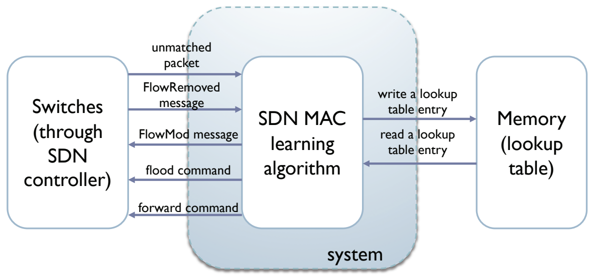

unmatched packet
FlowRemoved message
FlowMod message
flood command
forward command
Switches (through SDN controller)
SDN MAC learning algorithm
write a lookup table entry
read a lookup table entry
Memory (lookup table)
system

Figure 4. The system boundary for the second increment

Table 6. MAC learning algorithm stimuli: The second increment

Stimulus (Parameterized)	Shorthand
Unmatched packet(switch, source MAC address, destination MAC address, in-port)	$pa(s, sa, da, p)$
Look up table(switch)	$lt(s)$
FlowRemoved message(switch, source MAC address, destination MAC address, in-port)	$frm(s, sa, da, p)$

Table 7. MAC learning algorithm responses: The second increment

Response	Shorthand
Forward	forward
Send a FlowMod message to the switch with the following information: switch, source MAC address, destination MAC address, in-port, out-port, timer value, type (add or delete)	$flowmod(s, sa, da, ip, op, t, add)$ or $flowmod(s, sa, da, ip, delete)$
Flood	flood
Add/Overwrite a new entry to the lookup table with the following information: switch, source MAC address, in-port	$add\text{-}lt(s, sa, p)$
Delete an existing (expired) entry from the lookup table with the following information: switch, source MAC address, in-port	$delete\text{-}lt(s, sa, p)$

We completed a sequence enumeration for the second increment in Table 8. Similarly all Length 1 sequences are reduced to the empty sequence indicating a stateless software control for this simple SDN app. This is because we used predicates to refine the lookup table status, at a certain level of abstraction, at the receipt of an unmatched packet or a FlowMod message from the switch, to deterministically identify software's behavior while keeping the enumeration productive.

Table 8. A MAC learning algorithm enumeration: The second increment

Sequence	Response	Trace
λ	0	Method
$pa(s, sa, da, p)$ $[da \notin lt(s)]$	add-lt(s, sa, p), flood $pa(s, sa, da, p)$ to all the ports of s except p	$1, 2, 3, 4$
$pa(s, sa, da, p)$ $[da \in lt(s)]$	add-lt(s, sa, p), forward $pa(s, sa, da, p)$ to the port of da in $lt(s)$ (denoted by op), $flowmod(s, sa, da, p, op, t, add)$, $flowmod(s, da, sa, op, p, t, add)$	$1, 2, 4, 5, 7$
$frm(s, sa, da, p)$	delete-lt(s, sa, p), $flowmod(s, da, sa, p, delete)$	$1, 2, 4, 6, 7$

Refinement of a sequence-based specification into design and implementation proceeds with selecting a software architecture and capturing how to gather each stimulus, generate each response, and maintain each system state.

Towards the implementation of an SDN MAC learning algorithm, we selected Mininet [1] as the network emulator, and Floodlight [4] as the open SDN controller. There is a learning switch already provided by Floodlight, which we disabled and replaced with a simple learning switch implemented from scratch based on our formal specification. This consists of two Java files that define a class and an interface: `SimpleLearningSwitch.java` and `ISimpleLearningSwitchService.java`.

In Table 9 and Table 10 we show how one could write Java code to gather each stimulus and generate each response. No state data is needed for this simple stateless control as identified by the specification.

We defined functionals from the requirements in Table 11, and structors from our design (classes and methods) in Table 12. Figure 5 shows a standard modularity matrix for the MAC learning algorithm based on these definitions that obeys linear software models. Due to the simplicity of this app (a single class implementing the algorithm), structors correspond to a group of class methods rather than a group of classes.

Testing of the SDN learning switch algorithm turned out to be a trivial testing problem given the stateless (rather than a stateful) software control. Using predicate refinement, we were able to enforce a trajectory onto a specific lookup table state, enabling a direct mapping from the current input (e.g., an incoming packet) to the current output, and exhaustive testing of all scenarios of uses.

Table 9. MAC learning algorithm stimuli gathering

Stimulus	Design / Implementation
$pa(s, sa, da, p)$	SimpleLearningSwitch.receive(IOFSwitch sw, OFMessage msg, FloodlightContext cntx), msg.getType() == PACKET_IN, sw $\Rightarrow$ s, cntx $\Rightarrow$ (sa, da), msg $\Rightarrow$ p
$lt(s)$	Map⟨IOFSwitch, Map⟨MacAddress, OFPort⟩⟩, SimpleLearningSwitch.macToSwitchPortMap, macToSwitchPortMap.get(s)
$frm(s, sa, da, p)$	SimpleLearningSwitch.receive(IOFSwitch sw, OFMessage msg, FloodlightContext cntx), msg.getType() == FLOW_REMOVED, sw $\Rightarrow$ s, msg $\Rightarrow$ (sa, da, p)

Table 10. MAC learning algorithm response generation

Response	Design / Implementation
forward	SimpleLearningSwitch.pushPacket(IOFSwitch sw, Match m, OFPacketIn msg, OFPort outport)
$flowmod(s, sa, da, ip, op, t, add)$ or $flowmod(s, sa, da, ip, delete)$	SimpleLearningSwitch.writeFlowMod(IOFSwitch sw, OFFlowModCommand command, OFBufferId buffered, Match m, OFPort output), sw $\Rightarrow$ s, command $\Rightarrow$ OFFlowModCommand.ADD or OFFlowModCommand.DELETE, m $\Rightarrow$ (sa, da, ip), outport $\Rightarrow$ op, SimpleLearningSwitch.FLOWMOD_DEFAULT_IDLE_TIMEOUT $\Rightarrow$ t
flood	SimpleLearningSwitch.writePacketOutForPacketIn (IOFSwitch sw, OFPacketIn msg, OFPort porttype) Pass in OFPort.FLOOD as porttype
$add\text{-}lt(s, sa, p)$	SimpleLearningSwitch.addToPortMap(IOFSwitch sw, MacAddress sourceMac, OFPort inport)
$delete\text{-}lt(s, sa, p)$	SimpleLearningSwitch.removeFromPortMap(IOFSwitch sw, MacAddress sourceMac)

Table 11. MAC learning algorithm functionals

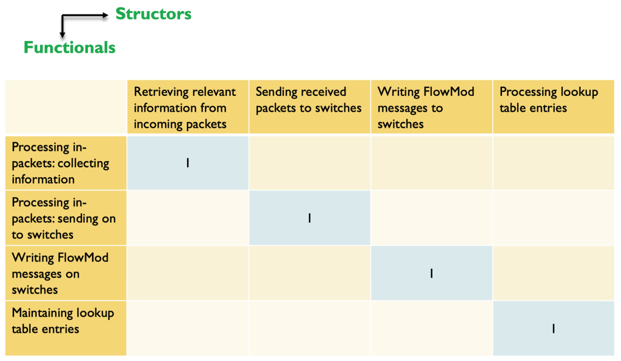
Functional	Requirement
Processing in-packets: collecting information	3, 4, 5, 6
Processing in-packets: sending on to switches	3, 4, 5
Writing FlowMod messages on switches	2, 4, 5, 6, 7
Maintaining lookup table entries	1, 3, 4, 5, 6

6 Related Work

Software-defined networking is a new paradigm in computer networking that is gaining significant momentum. The key concept of softwarization was first introduced in the seminal work [11]. For the next several years, the research primarily focused on the technologies that enabled the core platform. This includes the development of the controller software(e.g., Floodlight [4] and OpenDaylight [2]), the communication protocol (called OpenFlow) between the hardware boxes and the software controller [3],

Table 12. MAC learning algorithm structors

Structor	Class Method
Retrieving relevant information from incoming packets	SimpleLearningSwitch.receive, createMatchFromPacket
Sending received packets to switches	SimpleLearningSwitch.pushPacket, SimpleLearningSwitch.writePacketOutForPacketIn
Writing FlowMod messages to switches	SimpleLearningSwitch.writeFlowMod
Processing lookup table entries	SimpleLearningSwitch.addToPortMap, SimpleLearningSwitch.removeFromPortMap, SimpleLearningSwitch.inLookupTable


Structors
Functionals

	Retrieving relevant information from incoming packets	Sending received packets to switches	Writing FlowMod messages to switches	Processing lookup table entries
Processing in-packets: collecting information	I			
Processing in-packets: sending on to switches		I		
Writing FlowMod messages on switches			I	
Maintaining lookup table entries				I

Figure 5. A standard modularity matrix for the SDN MAC learning algorithm

the pipeline packet processing on the hardware boxes [5], and so on.

More recently and as the core SDN platform has been established, the networking community has begun to shift its attention to the development of SDN apps. The development has largely focused on the creation of advanced networking algorithms and techniques that utilize the unique capabilities provided by the SDN platform, such as direct programmability and centralized control, to solve hard and long-standing problems in networking, such as dynamic traffic engineering [9], efficient and intelligent anomaly detection [15], optimized energy efficiency [14], to name a few. Unfortunately the problem of implementing those algorithms and techniques into software applications is largely overlooked, as it has been assumed that the implementation is straightforward. This paper takes the position that the implementation is in fact a non-trivial problem, and our major contributions are to shed light on the implementation complexity, and to propose and sketch an initial solution. As such this work complements the prior research well. To the best of our knowledge, no prior work has systematically addressed the software engineering problems in SDN application development.

Finally, we wish to note that there has been effort (including the authors' own work) in applying software engineering techniques in the SDN context, but mainly to the testing [6, 16] and orchestration [17] of SDN apps. In contrast, this paper focuses on the implementation of those applications.

7 Conclusion

The emerging software-defined networking paradigm revolutionizes computer networking and presents new challenges to the software engineering community. The architecture enabled by SDN enforces re-examination of the many assumptions (that have been taken for granted in developing traditional networking software), when one develops SDN control software. We propose a systematic and methodical approach to SDN app development through rigorous software specification and design methodologies, that can be applied to developing new SDN apps, or migrating existing algorithms and protocols to the SDN environment. We illustrate a preliminary case study of the MAC learning algorithm that shows promises of achieving correct software by design for software-defined networking, through a sequence of refinement steps. Our future work includes further validation of our approach via more case studies of more sophisticated SDN apps to address scalability, and utilizing/augmenting existing tool support.

Acknowledgments

This work was generously funded by Air Force Research Laboratory(AFRL) through the NSF Security and Software Engineering Research Center (S^{2}ERC), and by the National Science Foundation (NSF) under Grants CNS-1660569 and 1835602. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of AFRL, S^{2}ERC, or NSF.

References

[1] Mininet: An Instant Virtual Network on your Laptop (or other PC). `http://mininet.org`.

[2] The OpenDaylight Project. `http://opendaylight.org`.

[3] OpenFlow Specifications. `https://www.opennetworking.org/software-defined-standards/specifications/`.

[4] Project Floodlight: Open Source Software for Building Software-Defined Networks. `http://www.projectfloodlight.org/floodlight/`.

[5] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *SIGCOMM Computing Communications Review*, 43(4):99–110, 2013.

[6] M. Canini, D. Venzano, P. Perešini, D. Kostič, and J. Rexford. A NICE way to test Openflow applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, pages 127–140, San Jose, CA, 2012.

[7] I. Exman. Linear software models: Standard modularity highlights residual coupling. *International Journal of Software Engineering and Knowledge Engineering*, 24(2):183–210, 2014.

[8] I. Exman. Conceptual integrity of software systems: Architecture, abstraction and algebra. In *Proceedings of the 29th International Conference on Software Engineering and Knowledge Engineering*, pages 416–421, Pittsburgh, PA, 2017.

[9] C.-Y. Hong, S. Mandal, M. Al-Fares, M. Zhu, R. Alimi, K. N. B., C. Bhagat, S. Jain, J. Kaimal, S. Liang, and et al. Before and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in Google's software-defined WAN. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 74–87, Budapest, Hungary, 2018.

[10] L. Lin, S. J. Prowell, and J. H. Poore. An axiom system for sequence-based specification. *Theoretical Computer Science*, 411(2):360–376, 2010.

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *Computer Communication Review*, 38:69–74, 2008.

[12] S. J. Prowell and J. H. Poore. Foundations of sequence-based software specification. *IEEE Transactions on Software Engineering*, 29(5):417–429, 2003.

[13] S. J. Prowell, C. J. Trammell, R. C. Linger, and J. H. Poore. *Cleanroom Software Engineering: Technology and Process*. Addison-Wesley, Reading, MA, 1999.

[14] M. Rahnamay-Naeini, S. S. Baidya, E. Siavashi, and N. Ghani. A traffic and resource-aware energy-saving mechanism in software defined networks. In *International Conference on Computing, Networking and Communications*, pages 1–5, Kauai, HI, 2016.

[15] A. Santos da Silva, J. A. Wickboldt, L. Z. Granville, and A. Schaeffer-Filho. ATLANTIC: A framework for anomaly traffic detection, classification, and mitigation in SDN. In *IEEE/IFIP Network Operations and Management Symposium*, pages 27–35, Istanbul, Turkey, 2016.

[16] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, and et al. Troubleshooting blackbox SDN control software with minimal causal sequences. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 395–406, Chicago, IL, 2014.

[17] X. Sun and L. Lin. Leveraging rigorous software specification towards systematic detection of SDN control conflicts. In *Proceedings of the 31st International Conference on Software Engineering and Knowledge Engineering*, pages 193–258, Lisbon, Portugal, 2019.