Vol.2019-CLE-27 No.8
2019/3/21

Tremaを用いたSDN構築演習における誤り絞り込みのための通信動作の依存関係分析システムの開発

浅野 晶文[1,a)]　立岩 佑一郎[1]　金 鎔煥[1]　片山 喜章[1]　新村 正明[2]

概要：本稿では，大学生向けに展開される Trema を用いた仮想ネットワーク上での SDN の構築演習を対象とする．この演習では，SDN での通信データの取り扱いの理解とコントローラをプログラミングする能力の習得を目的としている．演習問題では，通信例として，送信する通信データと伝達経路が与えられる．出来上がったネットワークが通信例を満たさないとき，その通信の結果やネットワークの状態から学習者は誤りを絞り込む．この際，伝達経路の検出や，通信データの出力時のルール（PacketOut やフローテーブル）の記録，そのルールを設定した実行文（PacketOut または FlowMod の送信実行文）を求めることができないという問題から，誤りの絞り込みが困難な学習者がいる．本稿では，誤りの絞り込みが難しい学習者のために，伝達経路，入力されたパケットと出力時のルールの関係，コントローラの実行文と PacketOut，FlowMod の関係という手がかりを依存関係分析から得るシステムを提案する．

1. はじめに

大学等の教育機関では，通信を動的に制御する技術の Software-Define Network（以下，SDN）によるネットワーク構築演習（以下，演習）が行われるようになってきている [1] [2]．演習では，SDN を実現する OpenFlow[3] を利用したフレームワークのうち，Trema[4] を利用する．OpenFlow は従来のネットワーク機器における通信データの転送機能とシステム制御機能を分離し，システム制御機能を OpenFlow コントローラ (以下，コントローラ) という 1 つの制御用プログラムで実現し，通信データの転送機能を OpenFlow スイッチ (以下，スイッチ) というコントローラに対応可能なネットワークスイッチを複数用いて実現している．コントローラとスイッチの間では，OpenFlow メッセージによってコントローラがスイッチに対して指示を送る．スイッチでは OpenFlow メッセージのうち，FlowMod により通信データの転送ルールを定めたフローテーブルの更新，PacketOut により出力ポートを決定して通信データの送出を行う．これにより，通信データの転送やその伝達経路が動的に制御されている．演習では，ネットワークの構成と通信例，通信テストの手順が与えられる．通信例は送信データと伝達経路である．学習者は，OpenFlow メッセージのうち，FlowMod や PacketOut を送るコントローラをプログラミングし，通信データの転送の制御を行い通信例を満たすことを目指す．コントローラのプログラミングは以下の手順で行われる．

手順 1　コントローラのプログラムの設計・コーディングする．

手順 2　コントローラを実行しコントローラをテストする．

手順 3　2 の結果に基づき，コントローラの誤りを発見，特定する．

手順 4　3 の結果に基づき，修正のために 1 に戻る．

手順 2 のテストでは，ping や traceroute による通信テストを行う．手順 3 の誤りの特定では，通信テストの結果から，学習者はコントローラのどの行に誤りがあるか絞り込み特定を試みる．しかし，次のような問題により，コントローラの誤りを見つけられない学習者がいる．

問題 1　ping，traceroute は，スイッチを含めた通信データの伝達経路を検出できない．そのため，学習者は伝達経路の誤りを引き起こしたスイッチを絞り込めない．

問題 2　スイッチは，任意の通信データの出力ポートの決定に用いたルール（PacketOut やフローテーブル）を記録していない．そのため，学習者は伝達経路の誤りを引き起こしたルールを特定できない．

問題 3　Trema は，ルールをスイッチに設定したコントローラの実行文（PacketOut または FlowMod の送信実行文）を求められない．そのため，学習者は伝達経路の誤りを引き起こした実行文を絞り込めない．

[1] 名古屋工業大学
Nagoya Institute of Technology
[2] 信州大学
Shinshu University
[a)] asano@moss.elcom.nitech.ac.jp

そこで，次の特徴から得た情報を手がかりとして出力する通信動作の依存関係分析システムを提案する．上記の問題のうち，特徴 1 が問題 2，特徴 2 が問題 1，特徴 3 が問題 3 を解決する．

特徴 1　スイッチの OpenFlow メッセージへの処理及び出力ポート決定の動作を模倣し，入力された通信データと出力ポート，OpenFlow メッセージの関係を求める．

特徴 2　通信データを収集し，特徴 1 の関係を利用することで，伝達経路を推定する．

特徴 3　コントローラの実行履歴と送信した OpenFlow メッセージを収集し，両者の関係を求める．

2. 関連研究

従来の誤り絞り込みの手順として，信州大学にて実践されている演習 [1] では，学習者は ping の結果やフローテーブルの確認，コントローラの標準出力を利用した実行された文の確認を行う．この手順では，部分的な伝達経路の推定や最終的なフローテーブルの情報の確認ができるが，推定ができない伝達経路や任意の時刻におけるフローテーブルの情報を得られないため，問題 1 や問題 2 を解決するには不十分である．

Trema における既存の実行ログの自動収集システムとしては Tremashark[5] がある．Tremashark は Wireshark のプラグインとして実装されているコントローラのデバッグの支援を目的としたシステムである．このシステムでは，ネットワークを構成するホスト・スイッチから出力された Syslog，ホスト・スイッチで伝達された通信データ，コントローラとスイッチ間で伝達された OpenFlow メッセージの送信と受信の情報，標準入力からのテキスト情報を収集し，それらの情報を 1 つに集約して時系列に並べたファイルを作成する．このファイルを Wireshark で読み込むと，収集された情報を一連の流れとして確認することができる．パケットとそのほかの情報を自動収集し，伝達経路の手作業での検出や PacketOut，FlowMod の記録を行える点で本研究と類似しているが，コントローラの実行文の情報やスイッチのフローテーブルの情報を収集せず，問題 2 や問題 3 を解決できない．

SDN におけるパケットの到達性を検証するシステムとしては Netplumber[6] がある．Netplumber は，フローテーブルの情報を用いてパケットの到達性やループ経路が存在するかなどを検証する．また，ある時刻におけるフローテーブルの状態を再現できるようにしており，特定の時刻における検証もできる．フローテーブルの情報からパケットのふるまいを調べることや，ある時刻でのフローテーブルを再現する点で本研究と同様であるが，実際に流れたパケットとの関連性やコントローラの動作との関係性を示さないため，問題 3 を解決できない．

3. SDN 構築演習

3.1 演習の概要

本稿で想定している SDN の構築演習は，大学の工学部情報系の演習授業であり，信州大学にて実践されている演習 [1] をモデルとする．この演習では，学習者に数台のネットワーク機器によって構成されるネットワーク構成と達成条件，通信テストの手順と利用するコマンドが与えられている．学習者は与えられているネットワーク構成において，通信例が満たせるようにコントローラをプログラミングする．学習者は作成したプログラムを提出し，指導者は提出されたプログラムを評価・採点する．通信例では，送信データとその伝達経路が示される．ネットワーク構成の情報として以下が与えられている．

(1) 各ホストのホスト名
(2) 各ホストの IP アドレス
(3) 各スイッチのスイッチ名
(4) 各ホストと各スイッチのポートとの配線接続関係

この演習では Linux の Network Namespace[7]，Open vSwitch[8]，OpenFlow のプログラミングフレームワークである Trema を利用する．学習者は Trema を利用するにあたり，コントローラのプログラムを Ruby[9] で記述する．この演習ではネットワークを構成するためのシェルスクリプトファイルが学習者に与えており，学習者はこのシェルスクリプトを実行することで仮想ネットワーク環境を用意する．また，学習者はフローテーブルの情報を確認することが可能である．学習者に与えられている通信テストの手順は以下である．

手順 1　シェルスクリプトを実行して仮想ネットワーク環境を設定する．

手順 2　Trema の run コマンドを実行してコントローラを実行する．

手順 3　ICMP エコーを行うコマンド (ping) を任意の回数だけ実行する．

手順 4　Trema の killall コマンドを実行してコントローラを終了する．

対象とする学習者は，TCP/IP や VLAN，ルーティング技術について学習済みであり，当該授業等において事前にネットワークの構築を学習するものとする．また，SDN によるネットワークの構築については経験がないものとする．演習は以下を目的とする．

目的 1　SDN での通信データの取り扱いについての理解

目的 2　コントローラをプログラミングする能力の習得

3.2 Trema によるプログラミング

Trema におけるコントローラのプログラミングでは，Controller クラスを継承することでコントローラの実装に

表 1　演習で利用する Controller クラスのハンドラメソッド

メソッド名 (引数)	メソッドの説明
start()	Trema コントローラの実行開始時に一度だけ実行される
switch_ready(datapath_id)	スイッチとの接続が確立された時に，スイッチを識別する数値 datapath_id を入力として実行される
packet_in(datapath_id，message)	PacketIn メッセージを取得した時に，スイッチを識別する数値 datapath_id，PacketIn メッセージの情報 message を入力として実行される

表 2　演習で利用する Controller クラスのメソッド

メソッド名	メソッドの説明
send_packet_out	PacketOut メッセージを作成し送信する
send_flow_mod_add	フローを追加する FlowMod メッセージを作成し送信する
send_flow_mod_delete	フローを削除する FlowMod メッセージを作成し送信する

必要な機能を追加する．演習では，Controller クラスで用意されているメソッドのうち，表 1，表 2 で示すものを利用する．学習者は表 1 のハンドラメソッドを拡張する形でプログラミングを行う．また，学習者が Controller クラス内で任意のメソッドを定義することも可能とする．

3.3　演習の流れ

この演習では，学習者は以下の手順でコントローラのプログラミングを行い，提出用のプログラムを作成する．

手順 1　コントローラプログラムを作成する．

手順 2　本章 1 節で示したコントローラを実行する手順を行い，コントローラをテストする．

手順 3　2 の結果に基づき，コントローラの誤りを発見，特定する．

手順 4　3 の結果に基づき，修正のために 1 に戻る．

学習者は，手順 2 のテストの結果によって，作成したコントローラプログラムが本章 1 節で示した通信例を満たしているかを確認する．手順 3 において，誤りは導通に失敗していることから発見される．手順 3 における誤りの特定では，学習者は手順 2 の結果からコントローラのプログラムのうち，どの行の実行文が誤りであるか絞り込み特定を試みる．この際，誤りの絞り込みのために学習者はテストで行った ping の結果やフローテーブルの確認を行う．

4.　提案システム

4.1　準備

本節では，本章で用いる共通の演算子，データ構造について述べる．

- 演算子 {}：引数を要素とする集合を返す．引数は可変長でカンマで区切る．
- 演算子＜＞：指定された順で引数を要素とする系列を返す．引数は可変長でカンマで区切る．例として＜a，b，c＞では，先頭が a，2 番目が b，3 番目が c の系列を返す．
- 演算子 ()：引数を要素とする組を返す．
- 演算子 .：組が持つ値へアクセスする．例として，組 A が要素 b を持つとき，b の値へのアクセスは “$A.b$” と記述される．
- 系列の要素 X_i：系列 X の i 番目の要素を X_i と表記する．
- 時刻：時間 h，分数 m，秒数 s，マイクロ秒数 ms による組（h，m，s，ms）で管理する．

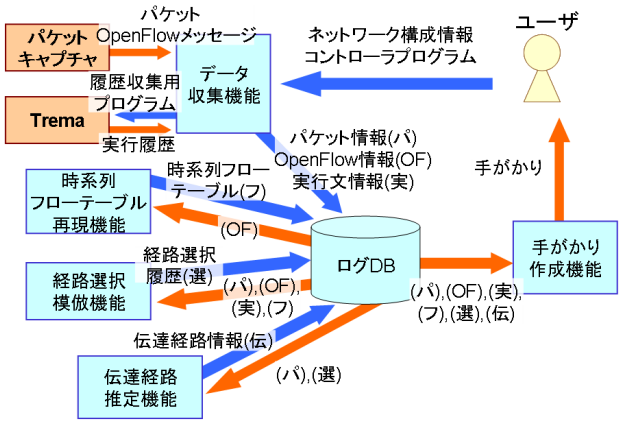

図 1　システム構成図

4.2　システム構成

提案システムの構成図を図 1 に示す．1 章に挙げた特徴のうち，時系列フローテーブル再現機能，経路選択模倣機能が特徴 1，伝達経路推定機能が特徴 2，データ収集機能が特徴 3 に対応する．ユーザである学習者は，提案システムにネットワーク構成情報とコントローラプログラムを入力し，提案システムを実行して通信テストを行う．この際，提案システムのデータ収集機能が通信テストのログデータを収集する．提案システムを終了すると，データ収集機能はログデータの収集を終了する．その後，時系列フローテーブル再現機能，経路選択模倣機能，伝達経路推定機能を順に実行され，それらで作成したデータをログデータベースに保管する．手がかり作成機能はログデータベースのデータから手がかりを作成し学習者に出力する．

4.3　ログデータベース

ログデータベースではパケット情報 (表 3)，OpenFlow 情報 (表 4)，実行文情報 (表 5)，時系列フローテーブル (表 6)，経路選択履歴 (表 7)，伝達経路情報 (表 8) を管理する．表 3 のネットワーク機器名はホスト，スイッチのいずれかが識別可能な名前である．マッチフィールドの情報 match

は OpenFlow のマッチフィールドに対応する情報である．アクションリストの情報 action は OpenFlow のアクションリストのうち，アクション Forward，Drop，Modify-Field とその引数に対応する情報であり，（アクション ac，引数 arv）とする．

表 3 パケット情報のテーブル定義

カラム名	データ型	説明
id	文字列型	識別 ID
time	時刻	パケットの取得時刻
src	文字列型	送信元のネットワーク機器名
dst	文字列型	送信先のネットワーク機器名
type	文字列型	パケットの分類
data	文字列型	パケットのバイナリデータ

表 4 OpenFlow 情報のテーブル定義

カラム名	データ型	説明
id	文字列型	識別 ID
time	時刻	メッセージの取得時刻
sname	文字列型	対象のスイッチ機器名
type	文字列型	メッセージの種類
data	文字列型	メッセージのバイナリデータ

表 5 実行文情報のテーブル定義

カラム名	データ型	説明
id	文字列型	識別 ID
stime	時刻	ハンドラメソッドの開始時刻
etime	時刻	ハンドラメソッドの終了時刻
linedata	文字列型	実行された行番号
ofid	文字列型	関係する OpenFlow 情報の識別 ID

表 6 時系列フローテーブル定義

カラム名	データ型	説明
id	文字列型	識別 ID
sname	文字列型	フローを持つスイッチ機器名
stime	時刻	フローの適用開始時刻
etime	時刻	フローの適用終了時刻
match	文字列型	マッチフィールドの情報
action	文字列型	アクションリストの情報
ofid	文字列型	関係する OpenFlow 情報の識別 ID

表 7 経路選択履歴のテーブル定義

カラム名	データ型	説明
rpid	文字列型	受信側のパケット情報の識別 ID
routeid	文字列型	経路選択に利用した時系列フローテーブル，実行文情報，OpenFlow 情報の識別 ID
spid	文字列型	送信側のパケット情報の識別 ID

表 8 伝達経路情報のテーブル定義

カラム名	データ型	説明
pktid	文字列型	パケット情報の識別 ID の系列

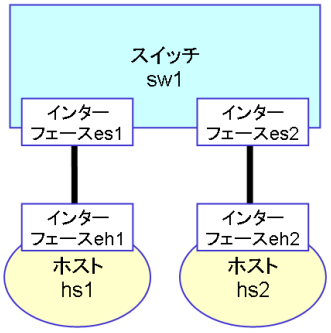

図 2 ネットワーク構成の例

4.4 データ収集機能

この機能ではネットワーク構成情報 NWT，コントローラプログラムを入力とした場合，パケットキャプチャと Trema を実行する．

ネットワーク構成情報 NWT は，ネットワークインターフェース名 ni，ni のネットワークインターフェースを持つホストまたはスイッチの名前 na，na とリンクされているホストまたはスイッチの名前 nb による組（ni，na，nb）を要素とする集合で管理される．例として**図 2** に示すネットワーク構成の場合，NWT ＝ {(es1，sw1，hs1)，(es2，sw1，hs2)，(eh1，hs1，sw1)，(eh2，hs2，sw1)} となる．図 2 では，インターフェース名 eh1，eh2 のインターフェースをそれぞれ持つホストのホスト名がそれぞれ hs1，hs2，インターフェース名 es1，es2 のインターフェースを持つスイッチのスイッチ名が sw1 となっているネットワークが構成されている．

パケットキャプチャの実行では，入力されたネットワーク構成情報 $nwt \in NWT$ から，$nwt.ni$ に対してパケットキャプチャを実行する．この際，パケットキャプチャは対象のインターフェースが受信したパケットのみをキャプチャするように設定する．$nwt.ni$ へのキャプチャにおける結果は，パケットをキャプチャした時刻 tp，キャプチャしたパケットの分類 $ptype$，バイナリデータ $pdata$ による組（nwt，$ptype$，$pdata$，tp）を要素とする集合 CA で管理される．また，OpenFlow メッセージをキャプチャするため，ローカルループバックのインタフェースに対してパケットキャプチャを実行する．このキャプチャにおける結果は，OpenFlow メッセージをキャプチャした時刻 tof，キャプチャしたパケットの OpenFlow メッセージの種類 ofm，バイナリデータ $ofdata$，通信相手のスイッチ名 $ofsw$ による組（nwt，ofm，$ofdata$，$ofsw$，tof）を要素とする集合 CB で管理される．

Trema の実行では，Ruby プログラムの文を要素とし，

それらをプログラムでの配置順で格納した系列 R から実行履歴収集用プログラム $R' = Deb(R)$ を作成する．関数 $Deb(R)$ の定義を図 3 に示す．関数 $Deb(R)$ では，Ruby プログラム R に標準出力を行う記述を追加したプログラムを作成する．その後，作成したプログラム R' を用いて Trema コマンドを実行してコントローラを実行する．この際，Trema が出力する標準出力を監視して取得する．標準出力の結果は，図 3 の s' により出力された文字列 str と，$stat$ の実行や Trema により出力される文字列を要素とし，それらが出力順に先頭から並んだ系列 SW で管理される．

パケットキャプチャの結果 CA, CB からは，まず $ca \in CA$ に対し，表 3 の time = $ca.tp$，src = $ca.nwt.nb$，dst = $ca.nwt.na$，type = $ca.ptype$，data = $ca.pdata$ としてパケット情報のデータをログ DB に追加する．一方 $cb \in CB$ に対し，表 4 の time = $cb.tof$，sname = $cb.ofsw$，type = $cb.ofm$，data = $ca.ofdata$ として OpenFlow 情報のデータをログ DB に追加する．

標準出力の結果 SW からは，まず表 1 の “start” 以外のメソッドごとの実行文の情報の系列 $SW' = Lga(SW)$ を作成する．関数 $Lga(SW)$ の定義を図 4 に示す．関数 $Lga(SW)$ では，6〜20 行目でメソッドの開始・終了を見つけ，21 行目以降ではメソッド内で実行された表 2 の関数を記録する．次に表 4 から組 (id，time，sname，type，data) の集合 OFD として，関係する OpenFlow 情報の ID 集合 $OFID = Ofln(SW', OFD)$ を作成する．関数 $Ofln(SW', OFD)$ の定義を図 5 に示す．関数 $Ofln(SW', OFD)$ では，3〜22 行目で実行文の情報と PacketIn の関係性を求め，23 行目以降で実行文の情報と PacketOut，FlowMod の関係性を求めている．最後に表 5 の stime = $SW'_i.st$，etime = $SW'_i.et$，linedata = $SW'_i.L$，ofid = $OFID_i$ として実行文情報のデータをログ DB に追加する．

4.5 時系列フローテーブル再現機能

この機能では，時間ごとのフローテーブルを表す時系列フローテーブルを作成する．表 4 の OpenFlow 情報を入力とし，時系列フローテーブルを追加・更新する．表 4 から組 (id，time，sname，type，data) の time による時間順ソートの系列 $OFDT$ として時系列フローテーブルの追加・更新を以下の手順で行う．

(1) $OFDT \ni ofdt$ のうち，$ofdt.type$ = FlowMod の $ofdt$ それぞれに対して，$ofdt.data$ からマッチフィールド $ofmatch$，アクションリスト $ofact$，コマンド com を抽出する．

(2) 手順 1 で com が追加を表すものに対して，表 6 の sname = $ofdt.sname$，stime = $ofdt.time$，etime = null，match = $ofmatch$，action = $ofact$，ofid = $ofdt.id$ として時系列フローテーブルのデータをログ DB に追加する．

```
1: function Deb(系列 R)
2:   R'=<>
3:   for i = 1 to |R| do
4:     s'="time=Time.new;puts(“［@dbg］line：+R_i の開始行+“-”+R_i の終了行+“,time：”+time.hour.to_s+“：”+time.min.to_s+“：”+time.sec.to_s+“.”+time.usec.to_s"
5:     if R_i が表 1 の “start” 以外のハンドラメソッド定義のヘッダである then
6:       s' = s'+“+“,[method start：”+ハンドラメソッド名+“(”+ハンドラメソッドの引数+“)]”)”
7:       R' の末尾に R_i を追加
8:       R' の末尾に s' を追加
9:     else if R_i が表 2 のメソッドの実行である then
10:      s' = s'+“+“,[method run：”+メソッドの種類+“)]”)”
11:      R' の末尾に R_i を追加
12:      R' の末尾に s' を追加
13:    else if R_i が “end” である then
14:      s' = s'+“)”
15:      R' の末尾に s' を追加
16:      R' の末尾に R_i を追加
17:    else
18:      s' = s'+“)”
19:      R' の末尾に R_i を追加
20:      R' の末尾に s' を追加
21:    end if
22:  end for
23:  return R'
24: end function
```

図 3 関数 Deb の定義

(3) 手順 1 で com が削除を表すものに対して，表 6 の match = $ofmatch$，action = $ofact$，stime <’$ofdt.time$ を全て満たす時系列フローテーブルのデータを見つける．それらデータの etime = $ofdt.time$ とし，ofid に $ofdt.id$ を追加して更新する．

(4) $OFDT \ni ofdt$ のうち，$ofdt.type$ = FlowRemoved の $ofdt$ を昇順に確認し，$ofdt.data$ からマッチフィールド $ofmatch$，アクションリスト $ofact$ を抽出して，表 6 の match = $ofmatch$，action = $ofact$，stime <’$ofdt.time$ を全て満たし stime が最も早い時系列フローテーブルのデータを 1 つ見つける．そのデータの etime = $ofdt.time$ として更新する．

4.6 経路選択模倣機能

この機能では，スイッチで受信したパケットがどのように処理されたかを模倣し，経路選択履歴を作成する．表 6 の時系列フローテーブル，表 3 のパケット情報，表 4 の OpenFlow 情報，表 5 の実行文情報を入力とする．表 3 から組 (id，time，src，dst) で，dst =スイッチの集合 $SRPKD$ と src =スイッチの集合 $SSPKD$，表 6 から組 (id，sname，stime，etime，match，action) の集合 $TFTD$，表 4 から組 (id，time，sname，type，data) の集合 OFD，表 5 から組 (id，ofid) の集合 LND として作成は以下の手順で行う．

(1) $srpkd \in SRPKD$ に対して，$srpkd.dst$ の値が同じ

```
1: function Lga(系列 SW)
2:   LND=<>, L=<>, M= {}, hnd=null, st=null, et=null
3:   for i = 1 to |SW| do
4:     if SW_i が先頭に "[@dbg]" を含む then
5:       SW_i から sline=R_i の開始行，tline=R_i の終了行，tl=(time.hour.to_s，time.min.to_s，time.sec.to_s，time.usec.to_s) を抽出する
6:       if SW_i が "method start : ハンドラメソッド名" を含む then
7:         if st=null then
8:           st=tl
9:         else
10:          組 (hnd，st，et，L，M) を LND に追加
11:          L=<>，M= {}，et=null
12:          st=tl
13:        end if
14:        SW_i から hnd="ハンドラメソッド名 (ハンドラメソッドの引数)" を抽出する
15:      end if
16:      if sline=tline then
17:        sline を L に追加
18:      else
19:        "sline-tline" を L に追加
20:      end if
21:      if SW_i が "method run : メソッドの種類" を含む then
22:        メソッドの種類を抽出して mname とする
23:        if mname が "send_packet_out" then
24:          "PacketOut" を M に追加
25:        else if mname が "send_flow_mod_add" または "send_flow_mod_delete" then
26:          "FlowMod" を M に追加
27:        end if
28:      end if
29:      et=tl
30:    end if
31:  end for
32:  return LND
33: end function
```

図 4 関数 *Lga* の定義

srpkd を集め時間順ソートした系列を作成し，その集合を $PKRDC$ とする．

(2) $pkrdc \in PKRDC$ を 1 つ取り出し，$sspkd \in SSPKD$ のうち，$sspkd.src = pkrdc_i.dst$ となる $sspkd$ の集合 $PKSD$ と，$tftd \in TFTD$ のうち，$tftd.sname = pkrdc_i.dst$ となる $tftd$ の集合 $STFD$ を求め，経路選択履歴を求めてデータベースを更新する関数 $RSids(pkrdc, PKSD, STFD, OFD, LND)$ を実行する．関数 $RSids(pkrdc, PKSD, STFD, OFD, LND)$ の定義を図 6 に示す．関数 $RSids(pkrdc, PKSD, STFD, OFD, LND)$ では，スイッチでの動作を模倣し，参照した時系列フローテーブル，OpenFlow 情報，実行文情報と出力となるパケット情報を求める．

(3) $PKRDC$ が空集合になるまで手順 2 を行う．

4.7 伝達経路推定機能

この機能では，パケットの経路を推定し，伝達経路情報を作成する．経路選択履歴，パケット情報を入力とする．表 3 から組 (id，time，src，dst) の集合 PKD，表 7 から組 (rpid，spid) の集合 RSD として作成は以下の手順で行う．

(1) 系列 PR =<>，集合 PRA = {}

(2) $PKD \ni pkd$ のうち，$pkd.src$ がホストのものを 1 つ取り出し，$pkd.id$ を PR に追加する．

(3) 取り出した pkd について，$RSD \ni rsd$ のうち，$rsd.rpid = pkd.id$ のものを見つける．無ければ手順 5 を行う．複数ある場合，いずれか 1 つを選び，それ以外の rsd それぞれに対し，組 (PR，rsd) として PRA に追加する．

(4) 手順 3 の rsd について，$rsd.spid$ が null でない場合，$PKD \ni pkd$ のうち，$rsd.spid = pkd.id$ であるものを取り出し，$pkd.id$ を PR に追加する．ここで $pkd.dst$ がホストでない場合，手順 3 を行う．

(5) 表 8 の pktid = PR として伝達経路情報のデータをログ DB に追加する．

(6) PRA が空集合でない場合，要素の 1 つ $pra \in PRA$ を取り出し，手順 3 を $rsd = pra.rsd$ として行う．

(7) PKD が空集合でない場合，手順 1 に戻る．

```
1: function Ofln(系列 SW', 集合 OFD)
2:   ID=<>, OFDS= {}, SWS=<>, pret=null
3:   for i = 1 to |SW'| do
4:     if pret=null then
5:       ofd ∈ OFD のうち，ofd.time < SW'_i.st かつ ofd.type=PacketIn を満たす ofd を全て探し，それらを OFDS に追加する
6:     else
7:       ofd ∈ OFD のうち，pret < ofd.time < SW'_i.st かつ ofd.type=PacketIn を満たす ofd を全て探し，それらを OFDS に追加する
8:     end if
9:     pret=SW'_i.st
10:    SW'_i.hnd のハンドラメソッド名が Packet_in である SW'_i を OFDS の要素数を同じ数見つけるまで i を加算し，その間の SW'_i を SWS に追加する
11:    if SWS が空でない then
12:      for j = 1 to |SWS| do
13:        MID= {}
14:        if SWS_j.hnd のハンドラメソッド名が Packet_in である then
15:          SWS_j.hnd からハンドラメソッドの引数 msn，mdata を抽出する
16:          ofds ∈ OFDS のうち，ofds.sname=msn かつ ofds.data=mdata を満たす ofds.time が最も早い ofds を 1 つ見つけて取り出し，ofds.id を MID に追加する
17:        end if
18:        MID を ID に追加する
19:      end for
20:    else
21:      MID= {} を ID に追加する
22:    end if
23:    OFDS= {}
24:    if SW_{i+1} が存在する then
25:      ofd ∈ OFD のうち，SW'_i.et < ofd.time < SW'_{i+1}.st かつ ofd.type=PacketOut または FlowMod を満たす ofd を全て探し，それらを OFDS に追加する
26:    else
27:      ofd ∈ OFD のうち，SW'_i.et < ofd.time かつ ofd.type=PacketOut または FlowMod を満たす ofd を全て探し，それらを OFDS に追加する
28:    end if
29:    for j = 1 to |SWS| do
30:      MID= {}
31:      SWS_j.hnd からハンドラメソッドの引数 msn を抽出する
32:      for k = 1 to |SWS_j.M| do
33:        ofds ∈ OFDS のうち，ofds.sname=msn かつ ofds.type=SWS_j.M_k を満たす cbs.time が最も早い ofds を 1 つ見つけて取り出し，ofds.id を MID に追加する
34:      end for
35:      ID_x(x = i + j − SWS の要素数) に MID の要素を追加する
36:    end for
37:    SWS=<>, OFDS= {}
38:  end for
39:  return ID
40: end function
```

図 5 関数 *Ofln* の定義

```
1: function RSids(系列 PKRD, 集合 PKSD, 集合 STFD, 集合 OFD, 集合 LND)
2:   for i = 1 to |PKRD| do
3:     RSID = {}, opdata = PKRD_i.data, OPT = {}
4:     stfd ∈ STFD のうち, stfd.stime < PKRD_i.time < stfd.etime を満たす stfd 全ての集合 STFD′ とする
5:     stfd′ ∈ STFD′ のうち, PKRD_i.data が満たせる stfd′.match を持ち, stfd′.stime が最も遅いものを見つけ stfd′′ とする
6:     if stfd′′ がある場合 then
7:       for j = 1 to |stfd′′.action| do
8:         if stfd′′.action_j.ac=Drop then
9:           表 7 の rpid=PKRD_i.id, routeid=stfd′′.id, spid=null として経路選択履歴のテーブルにデータを追加し，この For ループを終了
10:        else if stfd′′.action_j.ac=Forward then
11:          stfd′′.action_j.arv の要素全てを OPT に追加
12:          stfd′′.id を RSID に追加し，この For ループを終了
13:        else if stfd′′.action_j.ac=Modify-Field then
14:          opdata=PKRD_i.data に対して stfd′′.action.arv で指定されたヘッダデータを変更したもの
15:          stfd′′.id を RSID に追加
16:        end if
17:      end for
18:    else
19:      “コントローラ” を OPT に追加
20:    end if
21:    if OPT に “コントローラ” が含まれる then
22:      ofd ∈ OFD のうち，ofd.time > PKRD_i.time, ofd.sname=PKRD_i.dst, ofd.type=PacketIn を全て満たし ofd.time が最も早いものを見つけ ofd′ とする
23:      lnd ∈ LND のうち, lnd.ofid が ofd′.id を含むものを見つけ，lnd.ofid に含まれる ofd′.id 以外の要素の集合を OFID′ とする
24:      ofd′.id と lnd.id を RSID に追加する
25:      ofd ∈ OFD のうち, ofd.id ∈ OFID′ かつ ofd.type=PacketOut を満たす ofd の集合 PO とする
26:      while PO が空集合でない do
27:        PO の要素を 1 つ取り出し po とし，po.data から送信先の機器名の集合 OIN，送信するパケットのバイナリデータ data を抽出する
28:        while OIN が空集合でない do
29:          OIN の要素を 1 つ取り出し oin とする
30:          pksd ∈ PKSD のうち，PKRD_i.time < pksd.time, pksd.data=data, pksd.dst = oin を全て満たし pksd.time が最も早いものを見つけ，PKSD から取り出す
31:          表 7 の rpid=PKRD_i.id, routeid=RSID, spid=pksd.id として経路選択履歴のデータを追加
32:        end while
33:      end while
34:    else
35:      while OPT が空集合でない do
36:        OPT の要素を 1 つ取り出し opt とする
37:        pksd ∈ PKSD のうち，PKRD_i.time < pksd.time, pksd.data=opdata, pksd.dst = opt を全て満たし pksd.time が最も早いものを見つけ，PKSD から取り出す
38:        表 7 の rpid=PKRD_i.id, routeid=RSID, spid=pksd.id として経路選択履歴のデータを追加
39:      end while
40:    end if
41:  end for
42: end function
```

図 6　関数 *RSids* の定義

4.8　手がかり作成機能

この機能では，特定の入力情報に対応した手がかりを作成し出力する．入力情報と対応した手がかり作成の手順と得られる出力を以下に示す．

- ホスト名 *host* を入力とした時，表 3 から src = *host* であるデータを参照し，それらの id の集合 $HPID$ を作成する．その後，表 8 から $pktid_1 \in HPID$ であるデータの集合 HPR を得る．最後に，表 3 から id$\in HPR$ であるデータより，組 (id，src，dst) の time による昇順系列を作成して出力する．これにより，ホスト *host* から送信された全パケットの伝達経路が確認できる．
- スイッチで送受信されたパケットの ID*pid* を入力とした時，表 7 から rpid = *id* または spid = *id* であるデータを参照し，それらの routeid の要素の集合 $RSID$ を作成する．その後，表 3，表 4，表 5，表 6 それぞれを確認し，id$\in RSID$ であるデータの集合を作成して出力する．これにより，スイッチで送受信されたパケットに対して，参照されたフローテーブル，実行された実行文，関連する OpenFlow メッセージが確認できる．
- スイッチのあるフローの時系列フローテーブル ID*tid* を入力とした時，表 6 から id = *tid* であるデータを参照し，その ofid を $OFIDF$ とする．その後，表 4 から id$\in OFIDF$ であるデータの集合と，表 5 から ofid の要素 $\in OFIDF$ であるデータの集合を出力する．これにより，そのフローの設定に関係した OpenFlow メッセージと実行された実行文が確認できる．

5.　プロトタイプシステム

提案システムのうち，データ収集機能について実装した．Ruby 及び C を使用して開発し，パケットキャプチャは tcpdump[10] を利用した．プロトタイプシステムでは，Ruby で記述されたコントローラのプログラムソースコードとネットワーク構成情報を記述したテキストファイルを入力としている．また，Trema の killall コマンドによりコントローラが終了した時，パケットキャプチャを終了し，キャプチャ結果とコントローラの標準出力の結果を出力するようになっている．キャプチャ結果は tcpdump による pcap 形式のキャプチャデータである．コントローラの標準出力の結果はテキストファイルである．プロトタイプシステムでは表 3，表 4，表 5 のカラムをそれぞれ組とした情報で構成した個別のテキストファイルを出力する．プロトタイプを利用した場合，3 章に示した通信テストの手順は以下のようになる．

手順 1　シェルスクリプトを実行して仮想ネットワーク環境を設定する．

手順 2　Ruby コマンドでプロトタイプシステムを実行する．プロトタイプシステムにより，Trema のコントローラと tcpdump が実行される．

手順 3　ICMP エコーを行うコマンド (ping) を任意の回数だけ実行する．

手順 4　Trema の killall コマンドを実行してコントローラを終了する．コントローラの終了を受けてプロトタイプシステムが tcpdump を終了させ，キャプチャ結果と標準出力の結果から，3 章の表 3，表 4，表 5 に対応するデータを作成し出力する．

プロトタイプの結果から得た表 4，表 5 のデータを用いて，特徴 3 の手がかり (実行文と OpenFlow メッセージの関係性) を UML 図で示したものを**図 7** に示す．図 7 からは，PacketIn によって実行された実行文の行番号，実行文によって送信されたメッセージが PacketOut と分かる．

表 9　実行環境

ホスト OS	Windows7
ホスト CPU	Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
仮想化ソフトウェア	VMware Workstation 12 Player 12.5.9 build-7535481
ゲスト OS	Ubuntu 16.04LTS
ゲストメモリ	1GB

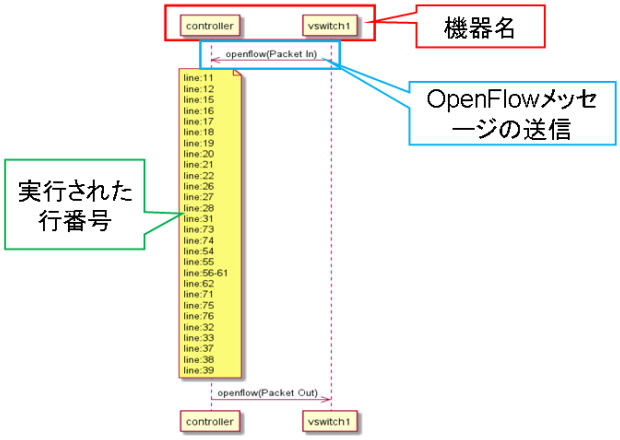

図 7　手がかりの例

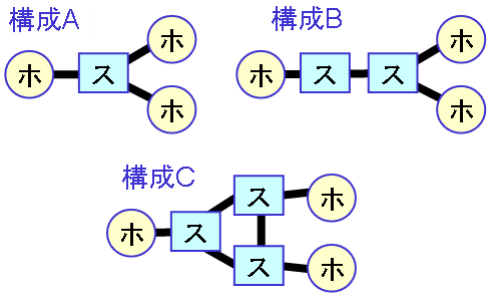

図 8　ネットワーク構成 A〜C

表 10　従来法とプロトタイプの RTT[ms]

	従来法	提案システム
構成 A	4.88	3.95
構成 B	6.71	10.48
構成 C	6.83	19.11

表 11　プロトタイプのデータ量 [KB]

	パケット情報	OpenFlow 情報	実行文情報
構成 A	17	10	2
構成 B	20	18	3
構成 C	21	25	4

6. 評価実験

図 8 に示すネットワーク構成 A〜C において，導通が成功するコントロールプログラムを用いて，実際の演習のように全ホスト間で ping により ICMP パケットを 3 回送信する．その際，従来どおりに Trema を利用した場合とプロトタイプシステムを利用した場合で平均 RTT を比較する．また，パケット情報，OpenFlow 情報，実行文情報のデータ量について確認する．実行環境を表 9 に示す．その結果を表 10，表 11 に示す．表 10 から従来の場合と比較しプロトタイプシステムの RTT のほうが大きいが，演習における通信テストに影響のない程度だった．また，データ量は導通成功時でも表 11 の程度であり，演習に支障のない程度である．

7. おわりに

本稿では，誤り絞り込みが難しい学習者のために，以下の 3 つの手がかりを依存関係分析から得るシステムを提案した．

(1) 伝達経路
(2) 入力されたパケットと出力時のルール (PacketOut，フローテーブル) の関係
(3) コントローラの実行文と PacketOut，FlowMod の関係

また，データ収集機能について実装し，演習で利用できるか評価を行った．今後の課題として，手がかりの表現方法の考察や，絞り込みの効果測定，未実装の機能の開発などが挙げられる．

参考文献

[1] 新村 正明： 情報基礎特論 II，信州大学 (2017 年度).
[2] 長谷川 剛，ほか： 情報ネットワーク学演習 II，大阪大学 (2016 年度).
[3] Open Datapath Standardized Switch Protocol in Software Defined Network (SDN)(online)，入手先 ⟨https://www.opennetworking.org/projects/open-datapath/⟩ (2019.02.26).
[4] Trema Full-Stack OpenFlow Framework in Ruby and C(online)，入手先 ⟨http://trema.github.io/trema/⟩ (2019.02.26).
[5] Tremashark(online)，入手先 ⟨https://www.slideshare.net/chibayasunobu/tremashark⟩ (2019.02.26).
[6] Kazemian, P., Chang, M., Zeng, H., Varghese, G., McKeown, N. and Whyte, S.: *Real Time Network Policy Checking using Header Space Analysis*, Proceeedings of the USENIX Symposium on Networked Systems Design and Implementation(NSDI) (2013).
[7] network_namespaces(7) - Linux manual page (online)，入手先 ⟨http://man7.org/linux/man-pages/man7/network_namespaces.7.html⟩ (2019.02.26).
[8] Open vSwitch(online)，入手先 ⟨http://www.openvswitch.org/⟩ (2019.02.26).
[9] オブジェクト指向スクリプト言語 Ruby(online)，入手先 ⟨https://www.ruby-lang.org/ja/⟩ (2019.02.26).
[10] Manpages of TCPDUMP(online)，入手先 ⟨https://www.tcpdump.org/tcpdump_man.html⟩ (2019.02.26).