Vol.2021-DPS-186 No.11
Vol.2021-CSEC-92 No.11
2021/3/15

Coqを用いたソフトウェアスイッチの設計と実装

斎藤 文弥[1,a)]　高野 祐輝[1,b)]　宮地 充子[1,2,c)]

概要：多くの端末がネットワークに繋がるようになるにつれ，スイッチなどを含むネットワークファンクションの重要性が増している．スイッチはネットワークとユーザーの端末を繋ぐ機能があり，管理が容易なソフトウェアスイッチにおいても安全なソフトウェアスイッチの設計は重要である．しかし，ソフトウェアスイッチではしばしばバグやエラーなどにより通信障害が発生するため，動作の検証が必要である．Coq は定理証明支援系言語であり，定義された関数や式の正しさをプログラム上で証明可能である．本論文では，パケットハンドリング機構の一つである Raw Socket を用いて各端末の MAC アドレスを管理するソフトウェアスイッチの設計と実装を行った．特に，MAC アドレスの管理については Coq で動きを検証した．この結果，形式的に検証されたソフトウェアスイッチの基礎実装を実現した．

キーワード：ネットワークファンクション，Coq，プログラム検証

Design and Implementation of Software Switch using Coq

Abstract: As more and more devices are connected to the network, the importance of Network functions, including switches, is increasing. Switches have the function of connecting networks and users' devices, and it is important to design safe software switches even for software switches which are easy to manage. However, since network disturbance often occurs in software switches due to bugs and errors, we have to verify the operation of them. An interactive theorem prover language, called Coq, can prove some defined functions and formulas on the program. In this paper, a software switch controlling the MAC addresses of each device is designed and implemented using Raw Socket, which is one of the packet handling mechanisms. Especially, we verified the operation of the MAC addresses management in Coq. As a result, formally verified fundamental implementation of the software switch was realized.

Keywords: network function, Coq, program verification

1. はじめに

1.1 研究背景

スマートフォンなどの IoT 機器が広く普及しており，それらの機器同士を繋ぐネットワークファンクションの開発は日々進められている．ネットワークファンクションは身近なファイヤーウォールや Wi-Fi などのスイッチングハブから，IT 企業が管理する大規模サーバーまで様々な種類がある．つまり，現在多くの人々がネットワークファンクションを通してインターネットを利用しているのである．しかし，ネットワークファンクションの動作に起因する通信障害は至る所で発生している．実際に Ramesh Govindan らは大規模なコンテンツプロバイダーにおいてネットワーク内で管理操作が行われている際に多くの障害が発生していることを指摘している [1]．様々なデータがインターネット上で送受信されており通信障害発生時の損失は計り知れない．したがって，現代社会において重要な機能を持つネットワークファンクションの動作の検証は喫緊の課題である．

1.2 本研究の目的

現在までに様々な手法ネットワークファンクション検証が提案されている．しかし，必要な仕様の一部しか実現していない手法や検証実行時に複雑な知識が必要な手法が多

[1] 大阪大学
Osaka University
[2] 北陸先端科学技術大学院大学
Japan Advanced Institute of Science and Technology
[a)] fsaito@cy2sec.comm.eng.osaka-u.ac.jp
[b)] ytakano@cy2sec.comm.eng.osaka-u.ac.jp
[c)] miyaji@comm.eng.osaka-u.ac.jp

ⓒ 2021 Information Processing Society of Japan

い．既存研究である VigNAT [2] はツールチェーンを利用することにより，検証をステップごとに進めてプロセスを明確にした．一方で，ツールチェーンで使用される各ツールについての理解を必要とするため，要求される知識が多く容易には使用できない．一方，別の既存研究 Aragog [3] は検出したい違反を記述するための言語と状態遷移モデルを組み合わせた実行時検証システムである．実際に実行しながら検証を行うことが可能だが，実行時検証であるためにネットワークファンクションの動作に対するバグの早期発見が難しい．本研究では Coq を利用することによって，単純な手法を用いて実行前に検証されたネットワークファンクションの実装を目的としている．

1.3 本論文の構成

2 章では本研究に関する既存研究である A Fromally Verified NAT と Aragog について記載する． 3 章ではネットワークファンクションの設計原理および，定理証明支援系言語の一つである Coq を利用した設計を記載する． 4 章では Coq で検証されたプログラムを C 言語の関数として抽出し実装する． 5 章では既存研究との比較評価と開発したソフトウェアの機能評価と考察を記載する． 6 章では本研究のまとめを記載する．

2. 関連研究

2.1 Vigor

Arseniy らが提案した Vigor [2] は NAT を検証するために開発されたツールチェーンで Vigor により検証された NAT は VigNAT と呼ばれる．NAT を含むネットワークファンクションは常に稼働する環境が変わりバージョンも開発され続けていくため，動作の安全性を検証することは困難である．この研究で扱っている NAT に必要とされる条件は，低レベルなプロパティ (RFC3022 [4] に則って意味的に正しいことと，クラッシュせずメモリーを大量に消費しない) を満たすことである．NAT の検証においてこれまでの提案方法ではこれらの条件を同時に満たすものは無かったが，このツールチェーンの活用と既に検証されたデータ構造をもつライブラリ (libVig) によって Vigor は可能にしている．

次に Vigor で使用されるツールをリストアップしつつ，それらの役割を説明する．

Clang LLVM compiler [5]
LLVM は様々なモジュールと再利用可能なコンパイラ，ツールチェーン技術の集まり．幅広い機能を持つがこの研究では定義していない動作を検出するために使用される．

VeriFast proof checker [6]
分離論理に基づき，事前条件と事後条件が注釈されている C 言語のプログラムで動作する証明認証器．

KLEE symbolic execution engine [7]
シンボリック実行を行うエンジン．シンボリック実行とは入力をシンボル化 (変数化) することを言い，レジスタやメモリなどの実行環境も含めた全ての入力がシンボル化される．

Validator
全てのシンボリックトレースを条件を満たしているトレースであることを示す証明に変換する．そして，そのトレースを証明認証器に送る役割を果たす．

続いて VigNAT による検証の手順について説明する．VigNAT が正しく動作することを検証するために，VigNAT の構成要素が正しい状態にあることを一つずつ証明して，必要とされる条件を満たす NAT であることを示す．その手順を以下に列挙する．

(1) NAT をステートレスな部分とステートフルな部分に分割する．libVig は後者に振り分けられる．
(2) 形式的手法を用いて libVig に格納されているデータ構造が正しいことを検証する．
(3) KLEE により全探索シンボリック実行を行って，ステートレスパートに存在する全てのパスに対して探索し，低レベルなプロパティを検証する．このステップは libVig が正しく用いられていることと libVig のモデルが正しいことを満たしていることを前提に進められる．これらの証明は Validator と VeriFast を組み合わせて帰納的に証明する．
(4) 同じように Validator と VeriFast を利用して VigNAT の意味的正しさを証明する．

2.2 Aragog

次に Nofel Yaseen らによって提案されたネットワークファンクションの実行時検証システム Aragog [3] を紹介する．Aragog は，より広範に様々なネットワークファンクションに対して使用でき，さらに実行中に検証を行えることから実用性にも優れたシステムである．

ⓒ 2021 Information Processing Society of Japan

Aragog は違反を検知するための仕様を記述する言語と，検証に必要なリソースを可能な限り減らすシステムを組み合わせたものである．これらを利用して以下の手順で実行中のネットワークファンクションを検証する．

(1) Aragog に用意された仕様記述言語を使い，検証する違反を書く．
(2) 検証に必要とされる処理だけを選別し，検証器に転送する．
(3) 全てのノードの状態と状態遷移を監視して，違反が起こった場合にはアラートを発する．

2.2.1 仕様記述言語

このシステムでは既存のプログラミング言語ではなく，独自の仕様記述言語を利用する．一般のプログラミング言語などでは違反仕様の記述量が多くなるような場合も，Aragog では短く簡潔に書くことができる．このプロセスにおいて行われることは大きく分けて次の三項目である．

フィルタリング
: 全ての処理は，あるノードにおいて情報を格納または破棄する，というように記述される．その格納や破棄といったイベントが違反検証において考慮されるべきか否かを選別する．

カテゴライズ
: どのイベントを一緒に考慮する必要があり，どのイベントを別々に考慮することができるかを示す．これにより検証を細分化し，プロセスを容易にする．

違反の記述
: イベントの種類やノードを指定し，違反の内容を具体的に記述する．

また，違反の仕様記述に必要なフォーマットのみが用意されている言語であり，ソースコードの可読性や冗長性においても優れている．

2.2.2 状態管理マシン

この研究では大規模なネットワークファンクションの検証にも対応することを目的に，状態遷移の概念を利用する．大規模なネットワークファンクションは多くの入力や状態，制御フローを抱えており，これらの挙動を全て検証することは現在のコンピュータの処理能力では難しい．そこで全ての状態遷移に対して動作の検証を行うのではなく，状態に影響を及ぼさない遷移については検証のプロセスを省略することによって処理の数を減らすことができる．この手法を実現するためにローカル状態マシンとグローバル状態マシンを生成する．

まず，ローカル状態マシンにおいて全ての状態とその各々の遷移を設定し，違反へと至る経路のモデルを構築する．そのモデルにおいて状態を変化させない遷移を抑制可能な遷移としてラベル付けして，実際に処理を始める．処理の際に，ある遷移が抑制可能な遷移でなかった場合，グローバル状態マシンに送られて検証を必要としているソフトウェア上でデータの格納や破棄が行われる．そこで，ユーザーの定義する違反が発生していると違反の検出として警告される．

2.3 既存研究のまとめと本研究の目的

紹介した研究はいずれもネットワークファンクションの動作の検証を行うシステムであるが，それぞれの手法にもメリットとデメリットが存在する．また，評価指標も様々でありソフトウェアによってより速度を重視した検証手法を採用したり，速度を犠牲にして誤動作のない安全性を追求したモデルを利用したりと使用する状況によってシステムの優位性が変わる．そういった指標のもとで本研究において提案する手法は，誤動作を抑制し安全性を追求した手法である．この目標を実現するために定理証明支援系言語である Coq を用いて，データを管理するプログラムの動作の仕様に対する忠実性および安全性を証明し，ソフトウェア検証の一つの手法として提案する．

3. 設計

本章では，ネットワークファンクションの設計原理と設計を説明する．

3.1 設計原理

3.1.1 満たされるべき要件

現在，様々なネットワークファンクションが使用されているが，それらはバグや意図しない動作を起こしてしまう不完全さを内包している．そこで，ネットワークファンクションに必要とされる要件を以下に挙げる．

- ソフトウェアがクラッシュしない
- ソフトウェアが仕様に従う
- 全パケットが処理される

一つ目のクラッシュしないことは全ての電子機器やネットワーク技術に共通する要件であり，実用性の側面から最も重要視されなければならない要件である．クラッシュは定義されていない入力や予期しない状態に陥ることによって発生するが，最悪の場合，データの破損や通信の断絶によって計り知れない損害を生み出してしまう恐れがある．従って，クラッシュは最も忌避すべき事態であり，起こらないよう十分に検証すべきである．

二つ目に仕様に従うことも重要である．仮にソフトウェアが仕様に従っていなかった場合，開発者ですらデータの処理や通信プロセスがどのような手順で行われたか把握できず，そのソフトウェアの動作に関して保証できない．すなわち，意図せずに第三者に重要なデータを渡してしまう等の事故が発生する可能性があり，仕様に従うことが保証されていないソフトウェアも致命的な欠陥があるといえる．

最後に，ネットワークファンクションは全パケットを認識しつつ処理しなければならない．パケットが破棄される

ことも処理の一つなので許容されるが，未処理のパケットがどのプロセスにも該当せず存在すれば，そのデータはネットワークファンクション上で完全に無視される．全てのパケットが仕様に基づいて処理されるような仕様にすべきである．

3.1.2 動作検証

設計したネットワークファンクションが 3.1.1 節にあるような要件を満たしているかを検証する手法は多く提案されているが，一部の要件のみしか満たされていなかったり，複雑な手法を用いた検証であったりするため，ユーザーにとって扱いにくい側面がある．そこで本研究では，よりシンプルな検証手法としてプログラム検証言語を利用する．この手法では実際に IP パケットなどのデータを使って正常に動作することを検証するのではなく，ソースプログラムにおいて作成したデータ構造や関数の動作を事前に検証して確認する．これによって，ユーザーに必要とされることはネットワークファンクションにおけるプログラムの仕様とプログラム検証言語の文法や証明の手順を把握しておくことである．

3.2 設計

本節では上の設計原理に基づいた設計手法を説明する．設計するソフトウェアスイッチの仕様を以下に列挙する．

- Raw Socket を利用して受信したパケットからデータを取得する．
- リンク層において動作し，イーサネットと MAC アドレスを管理してネットワーク転送を行う．
- Coq によって作成，検証されたネットワーク転送テーブルとして機能する関数を用いる．

3.2.1 Raw Socket

インターネット通信を行う際には TCP/IP の階層モデルのプロトコルが使用されるが，それらを利用するためには記述するプログラミング言語と TCP/IP の階層モデルを結びつける必要がある．この役割を担うのがソケットであり，TCP/IP 通信をソケット通信と呼ぶこともある．通常，ソケットは各階層のプロトコルでパケットのヘッダ情報をもとに処理され，その都度パケットから不必要になった情報は削除されていく．しかし，Raw Socket では各階層で処理される前のパケットをそのまま入手してプロトコルを実行することができる．これを利用して宛先，送信元 IP アドレスと MAC アドレスの情報を得ることができるため，上位層のプロトコルを経由せずにスイッチングの機能を実装することが可能となる．

3.2.2 ネットワーク転送テーブル

本研究で実装するソフトウェアスイッチはリンク層で動作するスイッチであり，イーサネットと MAC アドレスを管理する．転送テーブルの構築のプロセスの例を図 1 とともに示す．ただし，ホスト h2 がソフトウェアスイッチの

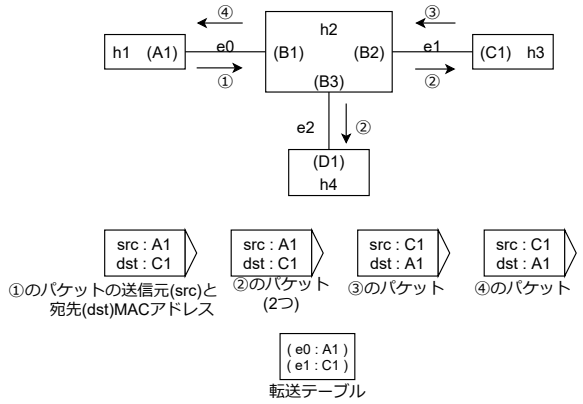


図 1 ネットワーク転送の仕組み

機能を持つ．また図 1 では本来 6byte ある MAC アドレスを括弧の中に簡略化して表す．

(1) ホスト h1 からホスト h3 への通信要求をイーサネット e0 を通して伝達する．また h1 の MAC アドレスと e0 のペアを転送テーブルに追加する．
(2) ホスト h2 から h3 への通信要求を e0 を除く全てのイーサネットに対して伝達する．
(3) 該当する h3 から h1 への応答をイーサネット e1 を通して伝達する．また h3 の MAC アドレスと e1 のペアを転送テーブルに追加する．
(4) h2 から h1 へ応答を伝達する．

転送テーブルはパケット 1 を h2 が受け取った際に，e0 と送信元 MAC アドレスである A1 をペアとして記録する．同様に，パケット 3 にて e1 と C1 をペアとして記録する．このようにしてネットワーク転送を行う．

3.2.3 Coq で検証された関数の利用

本論文で提案するソフトウェアスイッチの特徴として検証された関数の利用がある．この関数は図 1 の転送テーブルで動作するもので，誤ったネットワークにパケットを転送してしまわないように動作の保証をするべきである．そこで，ソースコードで定義した関数の動作を検証できる Coq を用いる．実際に，ネットワーク転送テーブルを構築する際に必要な型や関数，動作の検証を順に説明する．

まず今回使用する型とその型の変数を用意する初期化関数及びその検証についてソースコード 1 とともに説明する．

1 行目にネットワーク転送テーブルを管理する map という特殊な型を定義する．map は他のプログラミング言語で言うところの辞書型であり，key と value を一つのセットとして格納する．また map は関数に似た働きも併せ持っており，引数として key を渡すと map に該当する key が存在した場合は相当する value を返し，該当する key が無かった時には v を返す．次に 3 行目の empty という関数を見る．これは新しく map を生成するときに使われる関数で該当する key が無かった時に必ず引数の v を返す．4 行目の fun は無名関数を表しており，初期状態では empty は

ソースコード 1 Coq での型の定義と初期化関数

```
1  Definition map (A : Type) := string →A.
2
3  Definition empty {A : Type} (v : A) : map A
       :=
4          (fun _ => v).
5
6  Theorem apply_empty : forall (A : Type) (x :
       string) (v : A),
7          (empty v) x = v.
8  Proof.
9          intros.
10         unfold empty.
11         reflexivity.
12 Qed.
```

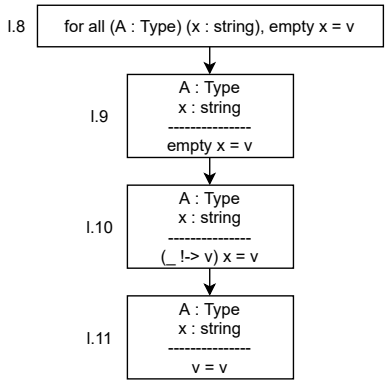


図 2 apply empty の証明の流れ

いかなる入力に対しても v を返す関数であることを示す．6 行目から 11 行目は上で定義した empty に対する動作の正しさを証明する定理である．すなわち，empty によってつくられた map 型の変数に引数としていかなる値を与えようと返される値は v であることを立証する定理である．8 行目から 11 行目の証明の部分は図 2 に沿って説明する．

8 行目で証明したい式と用いられる変数が用意される．次に変数の型を導入して固定し，証明する式と変数を分割する．10 行目で empty という関数を 3 行目と 4 行目で定義したように展開する．その 10 行目の左辺の括弧はいかなる入力に対しても v を返す関数であることを示しており，そのプロセスが処理されると 11 行目のような式になる．reflexivity は左辺と右辺の対称性をチェックするコマンドであり，正しいことが確認されると証明が終わる．このように Coq では各プロセスを段階ごとに分けて証明が進められる．

続いて map を更新するための関数 update とその動作の検証を行う定理について図 3 とソースコード 2 に基づいて説明する．

map に新たにデータを格納する際には key となる値が既に存在するかしないかによって動作が異なる．既に存在する場合には value の値のみを更新する必要があり，それぞれの分岐に対して動作の検証を行う．

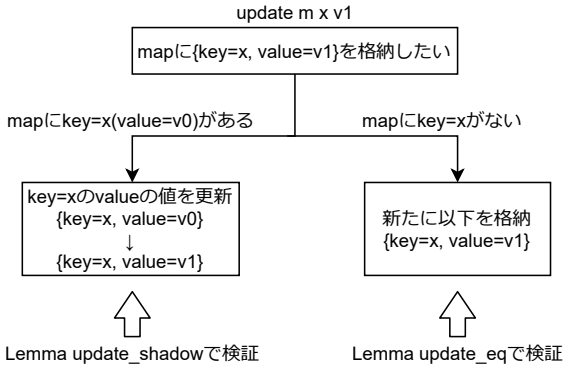


図 3 map の更新の流れ

ソースコード 2 map の更新関数 update とその動作の検証

```
1  Definition update {A : Type} (m : map A) (x
       : string) (v : A) :=
2          fun x' ⇒if eqb_string x x' then v
               else m x'.
3
4  Theorem eqb_string_refl : forall s : string,
5          true = eqb_string s s.
6
7  Theorem update_eq : forall (A : Type) (m :
       map A) x v,
8          (update m x v) x = v.
9  Proof.
10         intros. unfold update.
11         rewrite<-eqb_string_refl. reflexivity
               .
12 Qed.
13
14 Axiom functional_extensionality : forall {X Y
       :Type} {f g : X ->Y},
15         (forall (x : X), f x = g x) -> f =
               g.
16
17 Theorem update_shadow : forall (A : Type) (m
       : map A) x v0 v1,
18         (update (update m x v0) x v1) = (
               update m x v1).
19 Proof.
20         intros. unfold update.
21         apply functional_extensionality.
22         unfold eqb_string. intros x0.
23         destruct (string_dec x x0) as [Hs_eq
               | Hs_not_eq].
24         - reflexivity.
25         - reflexivity.
26 Qed.
```

関数 update は引数に map，key，value を取り map を

ソースコード 3 Coq における関数と OCaml への抽出

```
1  Definition empty {A : Type} : map A :=
2      (fun _ => None).
3
4  Definition update {A : Type} (m : map A) (x
       : string) (v : A) :=
5      fun x' ⇒
           if eqb_string x x' then v else m x'.
6
7  Definition get {A:Type} (m: map A) (x:string
       ) :=
8      m x.
9
10 Extraction "mapping.ml" empty update get.
```

ソースコード 4 OCaml におけるプログラム

```
1  let empty v _ =
2      v
3
4  let update m x v x' =
5      match eqb_string x x' with
6      | True -> v
7      | False -> m x'
8
9  let get m =
10     m
11
12 let _ = Callback.register "empty" empty
13 let _ = Callback.register "update" update
14 let _ = Callback.register "get" get
```

返す．4 行目と 5 行目は文字列を比較し等しければ true を返す関数 eqb_string に対する動作の定理であり，ここではその証明は省略する．7 行目から 12 行目における定理は map に格納されている key と代入される引数 key の変数が同じ x であることから eqb_string_refl を利用して証明が行われる．14 行目から 15 行目では右辺と左辺の map が等価であることを示すために，関数の等価性を表す公理を導入する．定理 update_shadow では証明する式の両辺が map であることから，実際に引数となる x0 を用意してそれらの返り値の比較をすることによって証明を進める．最後に x と x0 が等しい時と等しくない時に場合分けし，式の反射性を用いて証明を完了する．

4. 実装

本研究で設計するソフトウェアスイッチは C 言語でパケットキャプチャとネットワーク転送を行う．従って，Coq で定義し検証した関数を C 言語に抽出する必要がある．この時，OCaml[*1] という言語を介して C 言語に関数を抽出する．本章では Coq で検証した関数を C 言語の関数から呼び出すまでのプロセスをプログラムコードとともに説明する．

Coq で関数を定義して Extraction することで.ml(OCaml の拡張子) に関数を抽出することができる．次に OCaml のプログラムを C 言語から呼び出すためのセットアップを行う．ここで新しく定義した関数 get は引数に map と key を渡して value を返す関数である．

OCaml における関数を C 言語のプログラムから共有ライブラリとして利用できるようにするために Callback.register によって各々の関数をライブラリに登録する．

[*1] Coq は OCaml をベースに開発された言語である．Coq で定義された型や関数を OCaml のコードに抽出することができ，さらに既存のライブラリを用いることにより C 言語から OCaml の関数を呼び出すことが可能となる．

表 1 既存研究と本研究の比較

	VigNAT	Aragog	本研究
誤動作修正の容易さ	✓		✓
実行時検証		✓	
コンパイル時検証	✓		✓
型システムによる保護			✓

4.1 内部構成

4.1.1 Coq から C 言語

脚注にあるように Coq は OCaml をベースに構築された言語であり，Coq で書かれたコードを OCaml に変換することが可能である．その際，処理に必要な関数を抽出するとともにその引数として用いられる型なども同時に読み込むため，OCaml についての深い理解が無くとも C 言語から OCaml の関数を呼び出すことができる．次に C 言語のプログラムにおいてライブラリに格納された関数として使用可能にするために関数を呼び出すように書き換えておく．これらの手順を取ることによって Coq で検証された関数を，実際にパケット転送を実行する C 言語のプログラムに活用することが可能となる．

4.1.2 C 言語を利用したパケットキャプチャ

初めに C 言語でソケットの種類を指定し Raw Socket を確保しなければならない．なぜならソケットを用いた通信は他にもあり，イーサネットヘッダや TCP/UDP ヘッダが処理されたソケットが存在するためである．各イーサネットに対して Raw Socket を用意することで送られてきたパケットのデータを読み込み，MAC アドレスなどの管理が可能になる．そして読み込んだデータを Coq から呼び出した関数を用いて処理することによって，次に行うプロセスを確定する．

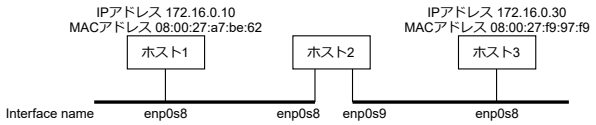


図 4 実装環境

ソースコード 5 ホスト 1 における ping コマンド

```
1 $ping 172.16.0.30
2 PING 172.16.0.30 (172.16.0.30) 56(84)
      bytes of data.
3 64 bytes from 172.16.0.30: icmp_seq=1 ttl
      =64 time=0.946 ms
4 64 bytes from 172.16.0.30: icmp_seq=2 ttl
      =64 time=1.14 ms
5 64 bytes from 172.16.0.30: icmp_seq=3 ttl
      =64 time=0.905 ms
6
7 --- 172.16.0.30 ping statistics ---
8 3 packets transmitted, 3 received, 0% packet
       loss, time 2026ms
9 rtt min/avg/max/mdev =
      0.905/0.995/1.136/0.100 ms
```

ソースコード 6 ホスト 2 における送受信

```
1 recv from enp0s8 (98 octets)
2 Dst MAC addr : 08:00:27:f9:97:f9
3 Src MAC addr : 08:00:27:a7:be:62
4 send to enp0s9 (98 octets)
5
6 recv from enp0s9 (98 octets)
7 Dst MAC addr : 08:00:27:a7:be:62
8 Src MAC addr : 08:00:27:f9:97:f9
9 send to enp0s8 (98 octets)
```

ソースコード 7 ホスト 3 におけるパケットの送受信

```
1 tcpdump -i enp0s8
2 tcpdump: verbose output suppressed, use -v
      or -vv for full protocol decode
3 listening on enp0s8, link-type EN10MB (
      Ethernet), capture size 262144 bytes
4 07:57:07.553726 IP 172.16.0.10 > h3: ICMP
      echo request, id 19, seq 1, length 64
5 07:57:07.553751 IP h3 > 172.16.0.10: ICMP
      echo reply, id 19, seq 1, length 64
```

5. 評価

5.1 定性的評価

上の表 1 のように，誤動作について VigNAT は各ツールで証明を行うため各々のステップで誤動作が発見できる．Coq はプログラムで定理を証明する形式で行い，一つずつ動作の検証を進めるため誤動作の発見は容易である．実行時検証は Aragog のみが対応しており簡単に検証可能だが，ソフトウェアが不完全であれば動作時にクラッシュを起こす可能性がある．それに対し VigNAT と本研究は，コンパイル時に検証を行うため動作時のクラッシュを事前に検知することが可能である．最後に本研究で用いる Coq にはコンピュータプログラム用の形式手法の一つである型システムに基づいた型検査を行う機能がある．これによりポインタやメモリに関する違反などのチェックが必要なくなる．

5.2 実行結果

本節では実装した環境と結果を記載する．本研究では図 4 のようにホストとネットワークを用意した．ただし，ホスト 1 とホスト 3 は繋がっておらずホスト 2 を介することでデータ通信が可能になる．このホスト 2 で Raw Socket を利用したパケットキャプチャと Coq で作成したアドレスを管理する関数を用いて転送処理を行う．

ソースコード 5 では 1 行目でホスト 3 の IP アドレスへ ping を送信している．2 行目以降は 3 個のパケットが送信され，応答が返ってきたこととその応答速度が表示されている．次にホスト 2 の動作とホスト 3 における ping 応答を記載する．

ソースコード 6 では前半 6 回のパケット転送は host1 側から送信された ping 要求を host3 側へ転送している処理と，それに対する ping 応答が行われている．その後，host3 側から host1 への ARP 要求と ARP 応答，host1 側から host3 への ARP 要求と ARP 応答が行われている．これにより，実際にソフトウェアスイッチによってパケットが転送されたことを確認した．

5.3 考察

ネットワークファンクションの検証手法は複数のパターンがあり，Aragog の実行時検証や本研究と Vignat の静的検証などがある．実行時検証は実際にネットワークファンクションが稼働している時に，誤った動作やバグの検出を行う．静的検証はネットワークファンクションを稼働させる前に，考慮すべき入力データの全てに対する動作を予め検証する．表 2 のように，静的検証はバグの早期発見と原理的探索が可能であり誤動作が発生した原因の追及とその改善が比較的容易である．しかし考慮されていないデータが入力されてしまった場合，その動作は保証された安全なものではない．そのためより強固な安全性を求めるのであれば，入力されるデータに対し全探索のように検証を行わなければならない．従って多数の端末に繋がる大規模なネットワークファンクションのように入力パターン数が膨大な場合には静的検証の手法を利用するべきではない．そ

表 2 実行時検証と静的検証の比較

	長所	短所
実行時検証	実際のデータに対しての検証が可能	バグが起きた原因探究に時間と労力を要する
静的検証	システムの動作に対する原理的探索が可能 バグの早期発見が可能	入力データの全パターンについて検証が必要

の際は実行時検証の手法を適用すべきである．反対に，限られた少しの端末間で作用する強固なネットワークファンクションを構築したい場合には静的検証を使用するべきである．

6. おわりに

現在，様々なネットワークファンクションが使用されているがそれらには何らかのバグやエラーが存在していることが多い．そのバグの発生を減少させることはコンピュータネットワークにおいて，常に重要視される課題である．また，不特定多数の端末に作用する大規模なネットワークファンクションや金融などの機密性が必須とされるデータを扱うネットワークファンクション等があり種類は多岐にわたる．各々が用いられる場面で最も必要とされる機能は変わり，その動作における検証の手法も変わる．

本研究では，ソフトウェアスイッチの実装についてパケットの転送テーブルを管理する関数の動作検証を定理証明支援系言語の一つである Coq を利用して行った．スイッチは幅広く使用されているネットワークファンクションの一つで企業においても利用されている．そういった場面において送受信されるパケットデータが意図しない端末に送られてしまうことは金銭的にも社会信用度の観点からも被害が出てしまい，経済的損失を生んでしまうおそれがある．こういった実用面に対して，パケットの転送テーブル管理の動作に重点を置いたソフトウェアスイッチを設計した．今後は様々な検証の手法を調べて用途に基づいた手法を用い，安全性と高速化を向上させたネットワークファンクションの実装を目指す．

また今回は，パケット転送を管理する host2 に対してネットワークが二つ接続された環境を構築した．しかし，実際に利用されているスイッチは 3 個以上のネットワークや端末に対して動作する．したがって本研究はさらに 3 個以上の端末に対して動作する，Coq により検証されたソフトウェアスイッチの実装へと発展させたい．

謝辞 本研究の一部は文部科学省「Society5.0 に対応した高度技術人材育成事業成長分野を支える情報技術人材の育成拠点の形成 (enPiT)」さらに文部科学省の平成 30 年度「Society 5.0 実現化研究拠点支援事業」と「ソフトバンク株式会社」の助成を受けています．

参考文献

[1] Govindan, R., Minei, I., Kallahalla, M., Koley, B. and Vahdat, A.: Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016* (Barcellos, M. P., Crowcroft, J., Vahdat, A. and Katti, S., eds.), ACM, pp. 58–72 (online), DOI: 10.1145/2934872.2934891 (2016).

[2] Zaostrovnykh, A., Pirelli, S., Pedrosa, L., Argyraki, K. J. and Candea, G.: A Formally Verified NAT, *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, ACM, pp. 141–154 (online), DOI: 10.1145/3098822.3098833 (2017).

[3] Yaseen, N., Arzani, B., Beckett, R., Ciraci, S. and Liu, V.: Aragog: Scalable Runtime Verification of Shardable Networked Systems, *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, pp. 701–718 (2020).

[4] Srisuresh, P. and Egevang, K.: Traditional IP Network Address Translator (Traditional NAT) (2001). RFC3022.

[5] : Clang 12 documentation (2020).

[6] Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W. and Piessens, F.: VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings* (Bobaru, M. G., Havelund, K., Holzmann, G. J. and Joshi, R., eds.), Lecture Notes in Computer Science, Vol. 6617, Springer, pp. 41–55 (online), DOI: 10.1007/978-3-642-20398-5_4 (2011).

[7] Cadar, C., Dunbar, D. and Engler, D. R.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings* (Draves, R. and van Renesse, R., eds.), USENIX Association, pp. 209–224 (2008).

ⓒ 2021 Information Processing Society of Japan