Formalization and Verification of the OpenFlow Bundle Mechanism Using CSP

Huiwen Wang Huibiao Zhu* Yuan Fei Lili Xiao
Shanghai Key Laboratory of Trustworthy Computing,
School of Computer Science and Software Engineering,
East China Normal University, Shanghai, China

***Abstract*—Software Defined Network (SDN) is an emerging architecture of computer networking. The most important feature of SDN is that it separates the control plane from the data plane. OpenFlow is considered as the first and currently most popular standard southbound interface of SDN. It is a communication protocol which enables the SDN controller to directly interact with the forwarding plane. The widespread use makes the reliability of OpenFlow important. The OpenFlow bundle mechanism is a new mechanism proposed by OpenFlow protocol to guarantee the completeness and consistency of the messages transmitted between SDN switches and controllers during the communication process. Due to the requirement of reliability and security of OpenFlow, we think that it is of great significance to formally analyze and verify the safety-relevant properties of the mechanism. In this paper, we apply Communication Sequential Processes (CSP) and the model checker Process Analysis ToolKit (PAT) to model and verify the OpenFlow bundle mechanism. Our formalization and verification show that the mechanism can satisfy four properties: deadlock freeness, parallelism, atomicity and order property, from which we can conclude that the mechanism offers a better way to guarantee the completeness and consistency.**

***Keywords*—OpenFlow; Bundle Mechanism; Formalization; Verification**

I. Introduction

Software Defined Network (SDN) is an emerging architecture of computer networking which is totally different from the current network infrastructure. SDN separates the control plane and the data plane to realize the flexibility and programmability of the network. OpenFlow protocol was originally proposed for campus network [1]. It is the first and currently the most popular southbound interface of SDN. It first defines the communication protocol which enables the SDN controller to directly interact with the forwarding plane consisting of network devices such as switches [2]. Therefore, formalizing and verifying the mechanism of OpenFlow protocol is necessary.

OpenFlow protocol supports three kinds of messages: controller-to-switch messages, asynchronous messages and symmetric messages [3]. The bundle message is a kind of controller-to-switch messages put forward to guarantee the consistency and completeness of communication. In recent years, several methods have been given to solve problems like packet losses and duplications [4]. The bundle mechanism is one of the solutions proposed by OpenFlow. A bundle is a sequence of OpenFlow messages sent from the controller to the switch and all the modifications in these messages are applied as a single operation. The bundle mechanism groups related changes and applies them together so that the completeness can be guaranteed [5]. And it also supports the order property. It means that the modifications inside a bundle should be applied in sequence. Our work in this paper is to model and verify the mechanism.

The remainder of this paper is organized as follows. In section 2, we present an overview of the OpenFlow bundle mechanism and a brief introduction to CSP. Section 3 shows the formalization of the OpenFlow bundle mechanism. The implementation and verification of the model are presented in section 4. Finally, we conclude the paper and discuss some possible future work in section 5.

*Corresponding Author. E-mail address: hbzhu@sei.ecnu.edu.cn (H. Zhu).

II. Background

In this section, we give an overview of the OpenFlow bundle mechanism and a brief introduction to the CSP is presented as well.

A. The OpenFlow Bundle Mechanism

OpenFlow specification has defined a configuration that the switch uses bundle messages as a transmission manner. A sequence of OpenFlow messages sent from the controller to the switch is stored in a bundle. Each bundle is specified with a unique bundle ID in a specific controller connection.

The bundle messages are transmitted between the controller and the switch. And there are two kinds of bundle messages: the bundle control message and the bundle add message. The bundle control message is used to perform operations on a bundle, e.g. open operation, close operation and commit operation. The bundle add message is formatted as a regular OpenFlow message which includes modifications in the message to be applied later. The bundle message has two properties: atomicity and order property, which are shown by the parameter flag. Atomicity means that the modifications should be done all or nothing. And order property means that the modifications should be executed in the order they are added in the bundle.

When a bundle is open, messages can be stored in the bundle without being applied. When the switch commits a bundle, the bundle should be closed and the switch should pre-validate the modifications inside the bundle and commit the

DOI reference number: 10.18293/SEKE2018-127

bundle afterwards. The whole process includes the following five oprations: opening a bundle, adding a message to a bundle, closing a bundle, committing a bundle and discarding a bundle. The detailed descriptions are presented as below:

1) Opening a bundle: The switch opens a new bundle according to the connection ID and bundle ID pairs. The switch should guarantee the validity of the connection ID, by checking whether the bundle already exists and verifying the correctness of other properties.
2) Adding a message to a bundle: The switch adds messages into a bundle and then the switch fetches the bundle by the ID pairs. Once fetched successfully, the switch validates properties of the added OpenFlow messages. For example, the switch should guarantee the consistency of the OpenFlow messages.
3) Closing a bundle: The switch closes the fetched bundle. A bundle should be closed before it is committed while a nonexistent bundle can not be closed.
4) Committing a bundle: The switch commits the fetched bundle. If a bundle is not closed, the switch closes the bundle and then commits it. The process of committing should satisfies the features of atomicity and order property. The modifications should be applied in all or nothing and in order. Whether the committing is successful or not, the switch discards the bundle afterwards.
5) Discarding a bundle: A bundle can be discarded during the whole process from the creation of the bundle. But if the bundle does not exist, the operation fails.

In addition, the switch must support the exchange of echo messages during the transmission of bundle messages to guarantee the liveness of the connection.

B. CSP

A brief overview of CSP is given in this subsection. Process algebra is a representative of the formal methods, which can illustrate concurrent systems easily using intuitive expressions and strict mathematical theories. CSP is proposed by C.A.R Hoare in 1978, which enriches process algebra [6]. Due to the powerful expression, CSP has been successfully applied in many fields [7], [8]. CSP processes are composed of primitive processes and actions. The syntax of a subset of the CSP is given as below. P and Q are two processes. a and b are two actions. And c is a channel.

$$\begin{aligned} P, Q ::= & Skip | Stop | a \rightarrow P | c?x \rightarrow P | c!v \rightarrow Q | P \lhd b \rhd Q \\ & | P; Q | P \Box Q | P || Q | P ||| Q | P[|X|]Q \end{aligned}$$

- $Skip$ denotes that a process does nothing but terminates successfully.
- $Stop$ denotes that the process is in the state of deadlock and does nothing.
- $a \rightarrow P$ represents that the process first engages in action a, then the subsequent behaviour is like P.
- $c?x \rightarrow P$ receives a message through the channel c and assigns it to a variable x, then behaves like P.
- $c!v \rightarrow Q$ sends a message v using the channel c, then the behaviour is like P.
- $P \lhd b \rhd Q$ denotes if the condition b is true, the behaviour is like P, otherwise, like Q.
- $P; Q$ performs P and Q sequentially.
- $P \Box Q$ behaves like either P or Q and the choice is made by the environment.
- $P||Q$ denotes that P runs in parallel with Q.
- $P|||Q$ indicates that P interleaves Q which means P and Q run concurrently and randomly.
- $P[|X|]Q$ indicates that P and Q perform the concurrent events on the set X of channels.

III. Formalization of the OpenFlow Bundle Mechanism

In this section, we model the OpenFlow bundle mechanism using CSP. The formalization is based on the description of the mechanism presented in section 2.

A. Sets, Messages and Channels

For convenience, we define seven sets in our model. The set $ConnectionID$ represents identities of connections and the set $BundleID$ represents identities of bundles. The set $Type$ denotes types of bundle messages, e.g $echo$, $open$, $close$, $commit$ and $discard$. The set $Flag$ shows the two properties of a bundle, e.g. atomicity and order property. The set $Operator$ represents operations of switches. The set $State$ indicates the states of the bundle including $open$ and $close$. And the set $Content$ represents other message contents. Based on the definitions above, we model the messages used in the mechanism as below:

$$MSG =_{df} MSG_{echo} \cup MSG_{con\_swt} \cup MSG_{swt\_bun},$$

$$MSG_{con\_swt} =_{df} MSG_{con} \cup MSG_{add} \cup MSG_{err},$$

$$MSG_{swt\_bun} =_{df} MSG_{req} \cup MSG_{rep} \cup MSG_{err},$$

$$\begin{aligned} MSG_{echo} =_{df} \{ & connection\_id.content | content \in Content, \\ & connection\_id \in ConnectionID\}. \end{aligned}$$

$$\begin{aligned} MSG_{con} =_{df} \{ & connection\_id.bundle\_id.type.flag | \\ & connection\_id \in ConnectionID, \\ & bundle\_id \in BundleID, \\ & type \in Type, flag \in Flag\}, \end{aligned}$$

$$\begin{aligned} MSG_{add} =_{df} \{ & connection\_id.bundle\_id.flag.content | \\ & connection\_id \in ConnectionID, \\ & bundle\_id \in BundleID, \\ & flag \in Flag, content \in Content\}, \end{aligned}$$

$$\begin{aligned} MSG_{err} =_{df} \{ & connection\_id.bundle\_id.type | \\ & connection\_id \in ConnectionID, \\ & bundle\_id \in BundleID, type \in Type\}, \end{aligned}$$

$$\begin{aligned} MSG_{req} =_{df} \{ & connection\_id.bundle\_id.operator.content | \\ & connection\_id \in ConnectionID, \\ & bundle\_id \in BundleID, \\ & operator \in Operator, content \in Content\}, \end{aligned}$$

$$\begin{aligned}MSG_{rep} =_{df} \{&connection\_id.bundle\_id.operator.state|\\ &connection\_id \in ConnectionID,\\ &bundle\_id \in BundleID,\\ &operator \in Operator, state \in State\}.\end{aligned}$$

We define three kinds of channels to model the communication among components:

- $ComCS$ is a channel for controllers and switches to transmit bundle messages.
- $ComSB$ is a conceptual channel used to represent the communication between the switch and its bundle process field.
- $ComEcho$ is an optional channel for controllers and switches to transmit echo messages.

B. Components

In this subsection, we use CSP to model OpenFlow controllers and switches which adopt bundle messages in the transmission manner. There are two levels in the model system. The first level happens between the controller and the switch. In the second level, the switch will search the bundle field it stores for the specific connection. To simplify, we abstract a component $Bundle$ communicating with the switch to simulate the process that the switch searches the bundle field. Our whole model is composed of three processes: *Controller*, *Switch* and *Bundle*.

Controller. The controller sends a sequence of OpenFlow modification requests in a bundle to switches. After that the controller will receive the response messages from switches. Each connection between a controller and a switch has an identity. At the same time, the controller supports exchanging echo messages to check the liveness of the connection. We model the behaviors as followed:

$$\begin{aligned}&Controller(connection\_id, bundle\_id, type) =_{df}\\ &\left(\begin{array}{l}(ComEcho!Msg_{echo} \rightarrow ComEcho?Msg_{echo})\\ \lhd type = echo \rhd\\ (ComCS!Msg_{con\_swt} \rightarrow ConCS?Msg_{con\_swt})\end{array}\right)\\ &\rightarrow Controller(connection\_id, bundle\_id, type);\end{aligned}$$

Switch. After receiving the message sent from the controller, the switch will search its bundle field to perform validations corresponding to the message received. We abstract the process as the communication between the switch and its bundle field. The switch sends a request message to its bundle field and receives a response after the bundle field checks the request message. Then the switch sends a corresponding response message to the controller. We model the behaviors as below:

$$\begin{aligned}&Switch(connection\_id, bundle\_id, type) =_{df}\\ &\ ComEcho?Msg_{con\_swt} \rightarrow\\ &\left(\begin{array}{l}(ComEcho!Msg_{echo})\\ \lhd type = echo \rhd\\ (ComSB!Msg_{swt\_bun} \rightarrow ComSB?Msg_{swt\_bun}\\ \rightarrow ComCS?Msg_{con\_swt})\end{array}\right)\\ &\rightarrow Switch(connection\_id, bundle\_id, type);\end{aligned}$$

Bundle. It is a process abstracted out from the performance of switches after receiving bundle messages. The bundle fields store bundles created by a switch. And the pre-validations of messages happen in this process. The detailed description can be found in section 2.The behaviors are modelled as below:

$$\begin{aligned}&Bundle(connection\_id, bundle\_id, operator) =_{df}\\ &Com?MSG_{swt\_bun} \rightarrow\\ &Assort(connection\_id, bundle\_id, operator);\\ &\left(\begin{array}{l}Com!MSG_{swt\_bun} \rightarrow\\ Bundle(connection\_id, bundle\_id, operator)\end{array}\right);\end{aligned}$$

$$\begin{aligned}&Assort(connection\_id, bundle\_id, operator) =_{df}\\ &if(operator == opn)\\ &then(openBundle(connection\_id, bundle\_id, operator))\\ &elseif(operator == add)\\ &then(addBundle(connection\_id, bundle\_id, operator))\\ &elseif(operator == cls)\\ &then(clsBundle(connection\_id, bundle\_id, operator))\\ &elseif(operator == cmt)\\ &then(cmtBundle(connection\_id, bundle\_id, operator))\\ &elseif(operator == dis)\\ &then(delete());\end{aligned}$$

The five operations are described respectively as below:

1) $openBundle(connection\_id, bundle\_id, operator)$ creates a new bundle in the switch's bundle field. The switch must perform the following validations. The connection should be reliable and the $bundle\_id$ should refer to a nonexistent bundle. And if the two conditions are satisfied, the value of $bundle\_id$ should be set to $false$ and the state of the bundle is set to *open* and then return $true$.

$$\begin{aligned}&openBundle(connection\_id, bundle\_id) =_{df}\\ &\left(\begin{array}{l}setContent(true, setBundleID(false),\\ setState(open))\\ \lhd \left(\begin{array}{c}(connection\_id \in ConnectionID)\\ \wedge (bundle\_id \notin BundleID)\end{array}\right) \rhd\\ setContent(false)\end{array}\right)\\ &; openBundle(connection\_id, bundle\_id);\end{aligned}$$

2) $addBundle(connection\_id, bundle\_id, operator)$ adds messages to a bundle. When the switch adds messages, it should first fetch the bundle using the $bundle\_id$ and $connection\_id$ pair. We define the parameter $vars$ as the number of messages to be added. If the fetched bundle is open and other properties are legal, the switch can add the message successfully and return $true$.

$$\begin{aligned}&addBundle(connection\_id, bundle\_id) =_{df} |||_{i \in vars}\\ &\left(\begin{array}{l}setContent(true, setMsgNum(num + 1))\\ \lhd \left(\begin{array}{c}(connection\_id \in ConnectionID)\\ \wedge (bundle\_id \in BundleID)\\ \wedge (state_{bundle\_id} = open)\end{array}\right) \rhd\\ setContent(false)\end{array}\right)\\ &; addBundle(connection\_id, bundle\_id);\end{aligned}$$

3) $clsBundle(connection\_id, bundle\_id, operator)$ closes a bundle after finishing adding messages. And the

bundle to be closed should be fetched successfully with $connection\_id$ and $bundle\_id$ pairs and its state should be open. Once all these validations are successful, the bundle state is set to $close$ and return $true$.

$$
\begin{array}{l}
clsBundle(connection\_id, bundle\_id) =_{df} \\
\left(\begin{array}{l} setContent(true, setState(close)) \\ \lhd \left(\begin{array}{l} (connection\_id \in ConnectionID) \\ \quad \wedge (bundle\_id \notin BundleID) \\ \quad \wedge (state_{bundle\_id} = open) \end{array} \right) \rhd \\ setContent(false) \end{array} \right) \\
; clsBundle(connection\_id, bundle\_id);
\end{array}
$$

4) $cmtBundle(connection\_id, bundle\_id, operator)$ commits the bundle. After fetching the bundle with legal $connection\_id$ and $bundle\_id$ pairs, the switch performs the following actions. Firstly, if the bundle state is $open$, the switch closes the bundle and then continues to commit the bundle. Secondly, it verifies the flag associated with the bundle and performs the corresponding actions.

$$
\begin{array}{l}
cmtBundle(connection\_id, bundle\_id) =_{df} \\
appMsg(connection\_id, bundle\_id); \\
\left(\begin{array}{l} setContent(true, setCmtNum(MsgNum)); \\ disBundle(connection\_id, bundle\_id) \end{array} \right) \\
\lhd \left(\begin{array}{l} (connection\_id \in ConnectionID) \\ \quad \wedge (bundle\_id \in BundleID) \\ \quad \wedge (state_{bundle\_id} = close) \end{array} \right) \rhd \\
\left(\begin{array}{l} \left(\begin{array}{l} clsBundle(connection\_id, bundle\_id); \\ cmtBundle(connection\_id, bundle\_id) \end{array} \right) \\ \lhd \left(\begin{array}{l} (connection\_id \in ConnectionID) \\ \quad \wedge (bundle\_id \in BundleID) \\ \quad \wedge (state_{bundle\_id} = open) \end{array} \right) \rhd \\ \left(\begin{array}{l} setContent(false, setCmtNum(0)); \\ disBundle(connection\_id, bundle\_id) \end{array} \right) \end{array} \right) \\
; cmtBundle(connection\_id, bundle\_id);
\end{array}
$$

5) $disBundle(connection\_id, bundle\_id, operator)$ discards the bundle fetched with the $connection\_id$ and $bundle\_id$ pair. Then it sets the $bundle\_id$ to be 1 and deletes all the messages in the bundle regardless of the bundle state.

$$
\begin{array}{l}
disBundle(connection\_id, bundle\_id) =_{df} \\
\left(\begin{array}{l} setContent(true, setBundleID(true), \\ setMsgNum(null)) \\ \lhd \left(\begin{array}{l} (connection\_id \in ConnectionID) \\ \quad \wedge (bundle\_id \in BundleID) \end{array} \right) \rhd \\ setContent(false) \end{array} \right) \\
; disBundle(connection\_id, bundle\_id);
\end{array}
$$

And the mechanism should support the exchange of echo messages and creations of multiple bundles. We add the two processes into our model as below:

Echo. Exchanging echo request and echo reply messages is supported during the whole process of the bundle. Echo messages are not included in a bundle and only transmitted between the controller and the switch with the channel $ComEcho$. And multiple connections may be included. Multiple bundles can be created in parallel. We use *N* to represent the number of interleaving connections.

$$
\begin{array}{l}
Echo() =_{df} |||_{i \in N} \\
\quad \left(\begin{array}{l} Controller(i, bundle\_id, echo) \\ |[ComEcho]| \\ Switch(i, bundle\_id, echo) \end{array} \right)
\end{array}
$$

BundleSys. The process of the creation of the bundle can be modeled as the parallel composition of the the controller, switch and bundle process and multiple connections may be included. *N* and *M* represents the number of connections and the number of bundles respectively.

$$
\begin{array}{l}
BundleSys() =_{df} |||_{i \in N, j \in M, type \in Type} \\
\quad \left(\begin{array}{l} Controller(i, j, type) |[ComCS]| \\ Switch(i, j, type) |[ComSB]| \\ Bundle() \end{array} \right)
\end{array}
$$

C. System

After modeling the processes *Echo()* and *Bundle()*, the OpenFlow Bundle architecture can be modeled as the concurrence of the two processes.

$$System() =_{df} Echo() ||| BundleSys()$$

Fig.1 shows the whole system we have modeled. We mark the $Echo$ part of the system with color blue and the $BundleSys$ part with color black.

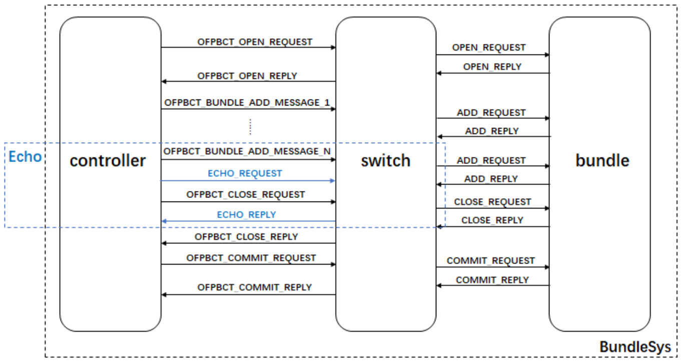


Fig. 1. Modeled System of OpenFlow Bundle Mechanism

IV. Implementation and Verification

In this section, we encode the CSP description above in PAT code and perform a series of validations. The verification is about four key properties of the OpenFlow Bundle Messages transmission. They are *Deadlock Freeness*, *Parallelism*, *Atomicity* and *Order Property*. In the following part, we give the detailed description.

A. Implementation in PAT

PAT is a model checker tool for automatic system analysis based on CSP [9]. Many systems can be verified in PAT such as concurrent real-time system, probabilistic systems and other domains [10], [11]. In this subsection, we encode our model in PAT and verify it.

We define some significant channels and variables. N represents the number of connections between multiple controllers and a switch. M denotes the number of bundles stored in bundle field. P indicates the number of messages added into a bundle. We use channel $ComCS[N]$, channel $ComSB[N]$ and channel $ComEcho[N]$ to describe different transmissions.

In the trial, we set N to be 3, M to be 2 and M to be 4 randomly respectively. And the buffer size is set to be 0 to ensure the communication among components is synchronous.

```
#define N 3;
#define M 2;
#define P 4;
channel ComCS[N] 0;
channel ComSB[N] 0;
channel ComEcho[N] 0;
```

We define some arrays to determine whether the process is successfully executed. We set the variables to be 1 to represent true and 0 to represent false. We also define some other arrays to record the state of the bundle, the messages added inside the bundle and the message committed as the executed results. Some of the declarations are given as below:

```
var EchoReturn = true;
var msgNum[N*M];
var cmtNum[N*M];
```

We define five functions with the same passing parameters and call them with different values. Then we implement the processes from the creation to the discard of a bundle. Because there may be multiple controllers connected to a switch and more than one bundle, we use *subBundleSys* to represent a single connection and use *i*,*j* as *connection_id* and *bundle_id* to distinguish it.

```
subBundleSys(i,j)=
(OpenBundle(i,j);AddBundle(i,j);
 CloseBundle(i,j);CommitBundle(i,j)
|||Echo(i);
```

The OpenFlow Bundle message transmission system can be implemented by taking advantages of non-determinism and interleaving, with i identifying the number of each connection. We give the definition of the complete system as follows:

```
BundleSystem()=
Init();(|||i:{0..N-1}@subBundleSys(i,0));
```

B. Verification

In this subsection, we use model checker PAT to simulate the execution of transmission of OpenFlow bundle messages and verify the properties. PAT searches the state space of the system until it locates a counterexample or exhausts the state space.

Property 1: Deadlock Freeness

If a component waits to receive information and no other components feel like sending messages to it, the system gets stuck in a deadlock state. A security protocol should be free of deadlock. PAT tool provides a primitive assertion to verify the property as below:

$$\#assert\ BundleSystem()\ deadlockfree;$$

We can conclude from Fig 2 that the system is free of deadlock.

Property 2: Parallelism

The system must support exchanging echo messages. Parallelism means echo messages can be transmitted without waiting for the end of the bundle and support multiple controller channels as well. It allows multiple bundles to be created in parallel. Randomly, we define three bundles created by different controllers and we add four messages into each bundle. Our goal is that in the end, the number of messages that each bundle commits successfully is four.

$$\begin{aligned}\#define\ Parallelism(cmtNum[0] == 4\&\&\\ cmtNum[1] == 4\&\&\\ cmtNum[2] == 4);\\ \#assert\ BundleSystem()\ reachesParallelism;\end{aligned}$$

Fig 2 shows that the verification result of parallelism is valid which means that the mechanism can support exchange echo messages with multiple bundle messages transmitted in parallel.

Property 3: Atomicity

Atomicity means that the switch should commit the messages inside the bundle in an all-or-nothing way. If one or more messages stored in the bundle can not be committed properly, then no messages will be committed. All the messages should be pre-validated. We set the values to true or false to determine whether it can be committed successfully. Then we check whether the number of committed messages of each bundle is zero or all.

$$\begin{aligned}&\#define\ Atm0(cmtNum[0] == 0\ xor\ cmtNum[0] == 4);\\ &\#define\ Atm1(cmtNum[1] == 0\ xor\ cmtNum[1] == 4);\\ &\#define\ Atm2(cmtNum[2] == 0\ xor\ cmtNum[2] == 4);\\ &\#define\ Atomicity(Ato0\ \&\&\ Ato1\ \&\&\ Ato2);\end{aligned}$$

If the verification is valid, the OpenFlow bundle mechanism can guarantee atomicity.

$$\#assert\ BundleSystem()\ \models\ Atomicity;$$

As shown in Fig 2, the bundle commits either all the messages inside it or none of the messages which satisfies the atomicity property.

Property 4: Order Property

If the switch supports the order property, it should strictly commit the messages according to the order they added. We use array *msgApplied* to represent whether the message is committed. There are three valid conditions listed as follows. None of the messages is committed. The former messages are committed and the latter ones are not. And all of the messages are committed.

$$\begin{aligned}&\#define\ NoApl(msgApplied[0] == 0\&\&\\&\qquad msgApplied[1] == 0);\\&\#define\ ForApl(msgApplied[0] == 1\&\&\\&\qquad msgApplied[1] == 0);\\&\#define\ AllApl(msgApplied[0] == 1\&\&\\&\qquad msgApplied[1] == 1);\\&\#define\ Order(NoApl\ xor\ ForApl\ xor\ AllApl);\end{aligned}$$

The system should satisfy any one of the three conditions.

$$\#assert\ BundleSystem()\ \models\ Order;$$

The messages are committed in sequence which satisfies the order property as shown in Fig 2.

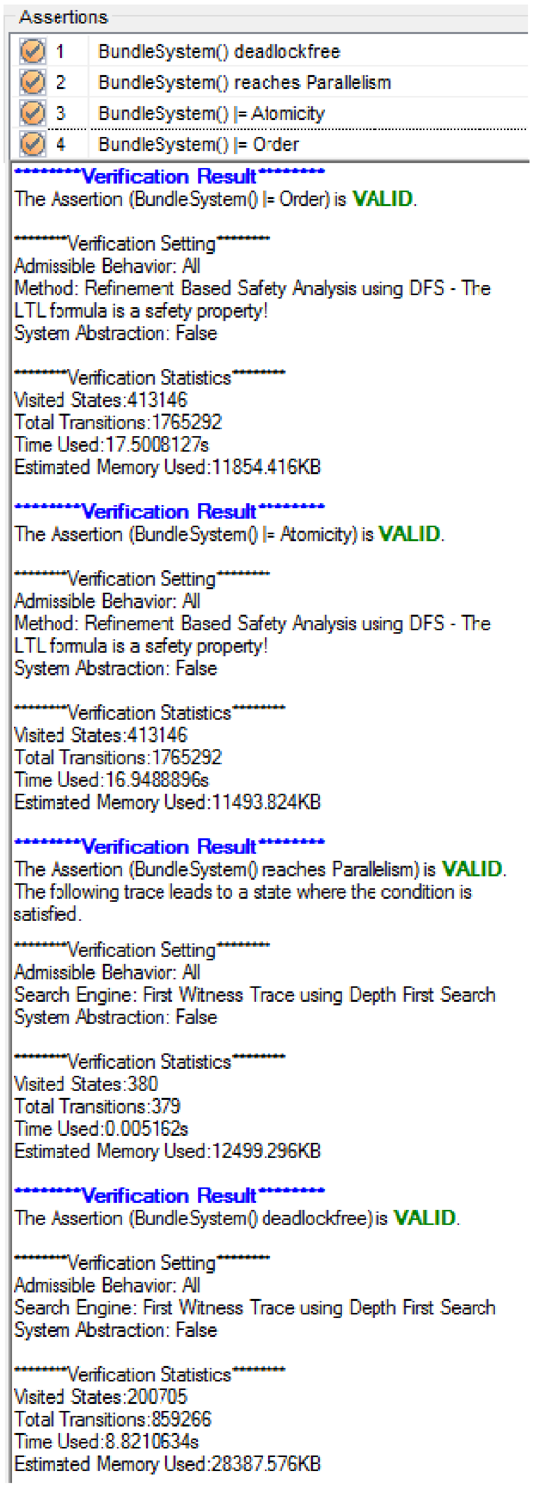
Fig.2 shows the verification results of all the properties and they are all valid. We can conclude that the OpenFlow bundle mechanism can guarantee the four properties to keep consistency and completeness of the communication.

V. Conclusion

The OpenFlow bundle mechanism is proposed to provide a method for guaranteeing consistency and completeness of network updates. In this paper, we construct a formal model for the OpenFlow bundle mechanism based on CSP. In addition, we encode the CSP description in the model checker tool PAT and perform the validation of four properties including deadlock freeness, parallelism, atomicity and order property. Corresponding to the verification results, we conclude that the mechanism can guarantee these transmission properties.

In the future, we will continue our work on the formalization and verification of OpenFlow. As the communication process we model in this paper is synchronous, We plan to explore a more general method which can fully guarantee consistency and completeness of the communication process in asynchronous situations based on time.

Acknowledgement. This work was partly supported by Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things (No. ZF1213).



```
Assertions
1  BundleSystem() deadlockfree
2  BundleSystem() reaches Parallelism
3  BundleSystem() |= Atomicity
4  BundleSystem() |= Order

********Verification Result********
The Assertion (BundleSystem() |= Order) is VALID.

********Verification Setting********
Admissible Behavior: All
Method: Refinement Based Safety Analysis using DFS - The
LTL formula is a safety property!
System Abstraction: False

********Verification Statistics********
Visited States:413146
Total Transitions:1765292
Time Used:17.5008127s
Estimated Memory Used:11854.416KB

********Verification Result********
The Assertion (BundleSystem() |= Atomicity) is VALID.

********Verification Setting********
Admissible Behavior: All
Method: Refinement Based Safety Analysis using DFS - The
LTL formula is a safety property!
System Abstraction: False

********Verification Statistics********
Visited States:413146
Total Transitions:1765292
Time Used:16.9488896s
Estimated Memory Used:11493.824KB

********Verification Result********
The Assertion (BundleSystem() reaches Parallelism) is VALID.
The following trace leads to a state where the condition is
satisfied.

********Verification Setting********
Admissible Behavior: All
Search Engine: First Witness Trace using Depth First Search
System Abstraction: False

********Verification Statistics********
Visited States:380
Total Transitions:379
Time Used:0.005162s
Estimated Memory Used:12499.296KB

********Verification Result********
The Assertion (BundleSystem() deadlockfree) is VALID.

********Verification Setting********
Admissible Behavior: All
Search Engine: First Witness Trace using Depth First Search
System Abstraction: False

********Verification Statistics********
Visited States:200705
Total Transitions:859266
Time Used:8.8210634s
Estimated Memory Used:28387.576KB
```

Fig. 2. Verification Results of Four Properties

References

[1] Mckeown, Nick, et al. "OpenFlow:enabling innovation in campus networks." Acm Sigcomm Computer Communication Review 38.2(2008):69-74.
[2] Kreutz D, Ramos F M, Verissimo P, et al. Software-Defined Networking: A Comprehensive Survey. Proceedings of the IEEE, 2015, 103(1): 14-76.
[3] Lara A, Kolasani A, Ramamurthy B. Network Innovation using OpenFlow: A Survey. IEEE Communications Surveys & Tutorials, 2014, 16(1):493-512.
[4] Kohler, Thomas, F. Drr, and K. Rothermel. "Update consistency in software-defined networking based multicast networks." Network Function Virtualization and Software Defined Network IEEE, 2016:177-183.
[5] Azodolmolky, Siamak. Software Defined Networking with OpenFlow. Packt Publishing, 2013.
[6] C. A. R. Hoare. Communicating Sequential Processes. Prentice/Hall International, 1985.
[7] Lowe G, Roscoe B. Using CSP to detect errors in the TMN protocol. IEEE Transactions on Software Engineering, 1997, 23(10): 659-669.
[8] Roscoe A W, Huang J. Checking noninterference in Timed CSP. Formal Aspects of Computing, 2013, 25(1): 3-35.
[9] PAT, PAT: Process analysis toolkit. [Online]. Available:http://pat.comp.nus.edu.sg/
[10] J. Sun, Y. Liu, and J. S. Dong, Model checking CSP revisited: Intoducing a process analysis toolkit, in Leveraging Applicaions of Formal Methods, Verification and Validation. Springer, pp. 307-322, 2009.
[11] Yuanjie, S. I., et al. "Model checking with fairness assumptions using PAT." Frontiers of Computer Science 8.1(2014):1-16.