Building Correct SDN Components from a Global Event-B Formal Model

J. Christian Attiogbé(✉)

LS2N - UMR CNRS 6004 - University of Nantes, Nantes, France
Christian.Attiogbe@univ-nantes.fr

Abstract. Software defined networking (SDN) brings flexibility in the construction and management of distributed applications by reducing the constraints imposed by physical networks and by moving the control of networks closer to the applications. However mastering SDN still poses numerous challenges among which the design of correct SDN components (more specifically controller and switches). In this work we use a formal stepwise approach to model and reason on SDN. Although formal approaches have already been used in this area, this contribution is the first state-based approach; it is based on the Event-B formal method, and it enables a correct-by-construction of SDN components. We provide the steps to build, using several refinements, a global formal model of a SDN system; correct SDN components are then systematically built from the global formal model satisfying the properties captured from the SDN description. Event-B is used to experiment with the approach.

Keywords: SDN · Correct design · Event-B · Refinement Decomposition

1 Introduction

An essential constituent of distributed applications is the physical network behind them. Distributed applications very often build on existing middlewares which embody services provided at the network level. Thus the reliability of distributed applications depends not only on their own development but also on the reliability of the network. Due to the involvement of many physical devices, the network level has for many years been a source of severe complexities and constraints leading very often to the adoption of rigid solutions in the deployment of applications. AQ1

Fighting the lack of flexibility of physical networks resulted in the software-defined networking (SDN) initiative [1,15,17]. Software-defined networking provides the opportunity to go deeper in modelling and reasoning on networks, since it enables to define and manage more easily the networks at the software level. Indeed a software-defined network is made of a controller and switches which are abstractly defined before being implemented at software level. In this context an user application does not consider a physical network or a specific middleware but it is rather built on top of a virtual or open network.

K. Bae and P. C. Ölveczky (Eds.): FACS 2018, LNCS 11222, pp. 1–19, 2018.
https://doi.org/10.1007/978-3-030-02146-7_2

Even if software-defined networking makes it possible to control an entire network with software, through programs that tailor network behavior to suit specific applications and environments, programmers still have many difficulties to write correct SDN programs. This is due to the unpredictability of the SDN as a distributed asynchronous system, and the lack of correctly developed SDN frameworks or formally verified SDN frameworks. Much work have been undertaken around SDN; they address different aspects: building simulators and analysers for SDN, building SDN controllers, verifying the controller component of an software-defined network, etc.

However SDN deployment is still at its beginning and programmers or administrators still need trustworthy devices and frameworks. Such devices may be implemented from rigorously founded models and related reasoning and engineering tools. Besides, considering the keen interest and the demands for the deployment of SDN as a flexible infrastructure for specific applications, clouds applications, IoT, etc, which all require security, it is of tremendous importance to have at the disposal of developers trustworthy development, analysis and simulation frameworks. Formal models taking into account several of these aspects are then necessary. That are the challenges that motivate of our work.

The main contributions of this paper are: *(i)* capturing the SDN as a discrete-event system to foster its modelling with an event-based approach; *(ii)* a state-based core model for rigorous analysis, development and simulation frameworks dedicated to SDN applications; it is a global Event-B [2] model, designed as the basis of the stepwise construction of the various components of a SDN; *(iii)* the systematic derivation of correct components (SDN controller and switches) from the global model which is previously proved to have some required properties.

The article is organised as follows. Section 2 gives an overview of software-defined networking, related work and main issues. In Sect. 3 we introduce the main concepts for modelling SDN, an overview of Event-B method and then our approach to build the global abstract model by stepwise refinements. Section 4 shows how one can derive the construction of a correct SDN controller from the global formal model. Section 5 gives the first experimental results related to simulation and verification of global safety/liveness properties. We conclude in Sect. 6 and stress some challenges for future work.

2 Overview of SDN: Concepts and Architecture

The SDN architecture consists of three layers: user application, control, and data forwarding. Control and data are the most relevant ones when studying the SDN. Figure 1 depicts how the SDN is viewed from the user side as a single global switch which denotes an abstraction of an entire network. User applications can directly exploit an abstraction of the network. Network services are solicited directly from host machines linked to a physical device: a *switch* assumed to be under the SDN control.

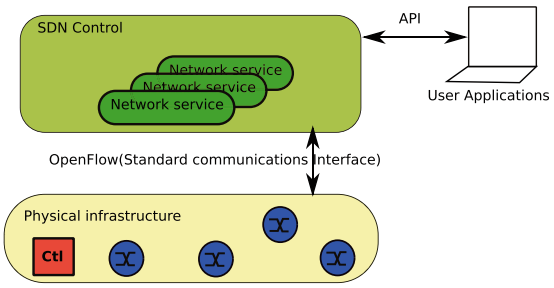

Fig. 1. The layered architecture of a SDN

In software-defined networking there is a physical separation between the *control-plane* (the management of the network and definition of rules to forward data packets) and the *data-plane* (how packets are forwarded according to control-plane) [15,17]. Indeed the network control (the high level or control-plane) is now separated from the packet forwarding (the low level or data-plane) activity and, physical devices inside the low level may be designed more easily; the network control is independent from device providers, the control is brought closer to software controllers and administrators. Traditional network services such as routing, access control, traffic engineering, security, etc can be implemented via various APIs provided by the SDN, instead of being vendor-dependent. The control and data levels are linked by an open communication interface. OpenFlow [1,9] is representative of such communication interfaces. OpenFlow is a standard communication interface, that moves the control of the network from the physical *switches* to logically centralized control software.

SDN has been used in variety of implementations, for example [16] is dedicated to the implementation of wireless networks, while in [5] the authors describe a tool, FlowChecker, which identifies any intra-switch misconfiguration within a Flow Table of a switch. RouteFlow [19] is a controller which implements virtualized IP routing over OpenFlow infrastructure.

2.1 Concepts and Components

We distinguish in Fig. 2 the main components of an SDN. Switches and controller are network devices that interact using packets and messages on data channel and message channel also called secure channel.

Switch. A switch is a device responsible of forwarding packets, to perform a hop-by-hop transfer of information through the network. A switch is configurable by the controller with which it interacts via a secure message channel. A switch interacts with other switches via a data channel.

Controller. A controller is a device responsible of controlling a whole network (a local or medium area network). It is used by the network administrator to dynamically configure, in an evolutive way, the switches with adequate forwarding information; it maintains the connectivity of the switches, etc. The controller

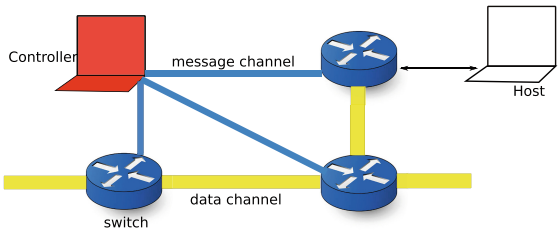

Fig. 2. A detailed architecture

initiates the switches behaviour, maintains them and instructs the switches with respect to specific actions. Packets not processed by the switches are sent to the controller via messages emitted on the message channel. The controller does not use the data channel.

Packet. A packet contains information related to various protocols (Ethernet, IP, etc). A packet has a header related to data and network layers and a body. Inside the header we have for instance: the destination and source addresses for each layer, the type of protocol, ...

Message. A message contains a control or management information addressed by the controller to a switch. The control information is for instance: which packet to drop, the indication of a port on which the switch may forward a data packet. A switch can also emit a message to a controller. In this case either the message contains a response to a control order (for instance the controller asked for the status of a switch) or a packet for which the switch does not have an entry in its table for forwarding the packet to its destination.

Flow Table. A flow table is a part of a switch. It describes the switch elementary behaviour. A flow table is made of several entries sent by the controller. Each entry has a header information and a body. The header may contain a message priority set by the controller. The body of the message can be a data packet, or a rule to process the incoming packets.

Interaction between switches and controller. A properly configured switch has routes to forward received packets coming from other network services. If the switch lacks of forwarding information, it sends the received packets to the controller. The controller is linked to the available switches and manages them directly with orders sent via messages on a *secure reliable channel*. These messages are used to configure and maintain the switches, defining for each one the rules to forward packets it receives. At this stage we have a simple interaction between application level, and the provided high level network services. But this interaction is more complex if we look at it in details. Consider for this purpose a detailed view depicted in Fig. 2, of the interaction with the SDN network.

2.2 OpenFlow: a Standard Interface

OpenFlow is a standard communications interface, supported by the Open Network Foundation [1,9,17]. OpenFlow has been precisely defined but not formally.

As such, OpenFlow provides a means for specifying data level or control-plane logics and also protocols. However there is no mandatory formal specification or formal requirements; accordingly the network systems resulting from OpenFlow may be incorrect or not satisfying safety conditions.

The OpenFlow semantics being informal, tool builders may assume particular behaviour and functioning for the network devices, leading to inconsistencies and incorrect behaviours; that is the case for the order in which packets are processed inside a switch.

2.3 Issues and Related Work

There is a keen interest for SDN, justifying several papers from both industry and academia. Important efforts are devoted to the implementation of SDN [16,18, 19]. SDN provides flexible network systems and distributed systems development but there is no guarantee that these systems are safe or correct. SDN as an asynchronous system undergoes the impact of time passing and non-determinism or concurrency of events. Packets may be received and distributed in any order causing for instance inconsistent interpretation in the switches when a forwarding route arrives after the related packet is sent. One of the main issues in SDN is the inconsistent packet forwarding during a network update which results in an update inconsistency [8]. Update consistency requires that packets are either forwarded by an old version of the forwarding table or by the new version of the table (after an update), but not by an interleaving of the old and the new version.

These issues require efforts to build robust tools and protocols on the basis of thoroughly studied SDN models. Much work have been devoted to various aspects of SDN among which the modelling and reasoning on the SDN controller[10], the analysis of the SDN traffic [8,14].

According to the state of the art [4,6,12,15] most investigations address the implementation issue as an important challenge; some of the aspects taken into account in these papers are: scalability, performance, security, simulation. The correctness of the implementations has received less attention.

In [13] the authors address the challenge of building robust firewalls for protecting OpenFlow-based networks where network states and traffic are frequently changed. They propose the FlowGuard framework for the detection and the resolution of firewall policy violation. They use an algorithmic approach.

VeriFlow is a verification tool proposed in [14] for checking in real-time network invariants and policy correctness in OpenFlow networks. This work is based on direct implementation that monitors the update events occurring on the network; it uses an algorithmic approach for the forwarding rules.

SDNRacer [8] is a network analyzer which can ensure that a network is free of harmful errors such as data races or per-packet inconsistencies. The authors

provide a formal semantics enriched with a happens-before model, to capture when events can happen concurrently.

The work in [10] is devoted to the verification of an SDN controller; the authors provide an operational model for OpenFlow and formalize it in the Coq proof assistant. This model is then used to develop a verified compiler and a run-time system for a high-level network programming language.

To sum up, the previous references are the preliminary steps towards making SDN networks more reliable; but much works remain to be done:

- making easier for developers the construction and verification of controllers from various existing well-researched models,
- enhancing machine-assisted configuration of controllers and OpenFlow-based switches,
- promoting the reuse of correct SDN components in the deployment of new SDN (that is interoperability).

The goal of our work is to serve these purposes by contributing with a global, extensible, refinable formal model. It is the first event-based one, making it easy to derive simulators and also to prove safety and liveness properties. It is provided as a reusable formal basis for any one interested, avoiding hence to repeat the efforts through the chains of work.

Unlike in the case dedicated to implementations, we follow an approach similar to those addressing modelling and reasoning on controllers, by defining for the SDN a global formal model from which the models of the components can be derived and then correctly implemented.

3 Stepwise Refinement-Based Modelling of SDN

We use Event-B [2] and adopt a correct-by-construction approach.

3.1 An Overview of Event-B

Event-B [2,11] is a modelling and development method where components are modelled as abstract machines which are composed and refined into concrete machines. An *abstract machine* describes a mathematical model of a system behaviour[1]. In an Event-B modelling process, abstract machines constitute the dynamic part whereas *contexts* are used to describe the static part. A *context* is seen by machines. It is made of carrier sets and constants. It may contain properties (defined on the sets and constants), axioms and theorems. A machine is described, using named clauses, by a state space made of typed variables and invariants, together with several *event* descriptions.

State Space of a Machine. The variables constrained by the invariant (typing predicate, properties) describe the state space of a machine. The change from one state to the other is due to the effect of the events of the machine. Specific

[1] A system behaviour is a discrete transition system.

properties required by the model may be included in the invariant. The predicate $I(x)$ denotes the invariant of machine, with x the state variables.

Events of an Abstract Machine. Within Event-B, an event is the description of a system transition. Events are spontaneous and show the way a system evolves. An event e is modelled as a *guarded substitution*: $e \mathrel{\widehat{=}} eG \Longrightarrow eB$ where eG is the event *guard* and eB is the event *body* or *action*. An event may occur only when its guard holds. The action of an event describes, with simultaneous generalised substitutions, how the system state evolves when this event occurs: disjoint state variables are updated simultaneously.

The effect of events are modelled with generalised logical substitution (S) using the global variables and constants. For instance a basic substitution `x := e` is logically equivalent to the predicate *x' such that* $x' = e$. This is symbolically written $x' : (x' = e)$ where x' corresponds to the state variable x after the substitution and e is an expression. In the rest of the paper, the variable x is generalised to the list of state variables.

Several events may have their guards held simultaneously; in this case, only one of them occurs. The system makes internally a nondeterministic choice. If no guard is true the abstract system is blocking (deadlock).

In Event-B *proof obligations* are defined to establish model consistency via invariant preservation. Specific properties (included in the invariant) of a system are also proved in the same way.

Refinement. An important feature of the Event-B method is the availability of refinement technique to design a concrete system from its abstract model by stepwise enrichment of the abstract model. During the refinement process new variables (y) are introduced; the invariant is strengthened without breaking the abstract invariant, and finally the events guards are strengthened. In the invariant $J(x, y)$ of the refinement, abstract variables (x) and concrete variables (y) are linked. The refinement is accompanied with proof obligations in order to prove their correctness with respect to the abstract model.

Rodin Tool. `Rodin`[2] is an open tool dedicated to building and reasoning on B models, using mainly provers and the ProB model-checker. `Rodin` is made of several modules (plug-ins) to work with B models and interact with related tools.

3.2 Abstractions for SDN Modelling

An SDN is made of a controller connected to several switches. The controller is linked to the switches via a secure message channel which conveys message flows between the controller and the switches. The switches are interconnected via a data channel which conveys data packets. Consequently, the elementary abstractions using the Event-B notations, are the basic sets that represent: the switches (SW_ID), the packets (PACKET), the messages (MESG), the

[2] http://wiki.event-b.org/index.php/Main_Page.

packet headers (HEADER), the states of a switch (SW_STATE). The messages have types and may contain packets[3]: $mesgType \in MESG \nrightarrow MESGTYPE$, $mesgPk \in MESG \nrightarrow PACKET$.

A packet has several headers (MAC source address, MAC destination address, MAC type, IP source address, IP destination address, IP protocol, transport source port, transport destination port), for simplification we consider only one of such header: $pHeader_i \in PACKET \nrightarrow HEADER$. In the model these headers are specified like the function $pHeader_i$. The previous sets and constants are gathered in a CONTEXT EnvCtx0 (see Fig. 3), seen by an abstract MACHINE which contains the variables and the typing predicate and properties in the VARIABLES and INVARIANT clauses. The context is then successively extended as the machines are refined.

```
CONTEXT EnvCtx0
SETS
    PACKET set of packets (exchanged between switches, controllers, hosts)
    MESG set of messages (exchanged between switches and controller)
    MESGTYPE message types
    ENTRY set of entries of the flow table
    HEADER header+Actions: set of actions applied by switch to match packets
    SW_ID switch ID
    SW_STATE Openflow switch state
CONSTANTS
    PKIn PKOut BarrierQ BarrierR FlowMd askStatus Status AddE DelE ModE
AXIOMS
    MESGTYPE = {PKIn, PKOut, BarrierQ, BarrierR, FlowMd, askStatus, Status,
            AddE, DelE, ModE}
END
```

Fig. 3. Event-B specification of the primary context

The SDN is a set of components that work concurrently in an asynchronous manner; we build a first global abstract model that simulates the functioning of this asynchronous system. The global abstract model will be the basis for the development of the components.

To structure this abstract model, we consider the data model and the discretisation of the behaviour (a set of observed events) of each of its components as a family of events. This is important for mastering the interaction between components and the forthcoming decomposition of the model.

Switches. Each switch has a flow table which contains the elementary behaviour of the switch according to the packets entering the switch. The behaviour of a switch is as follows: when it receives a message from the controller, it analyses the information inside the message and accordingly performs the instructions

[3] The symbol $\nrightarrow$ denotes a partial function; $\leftrightarrow$ denotes a relation.

of the controller, for example updating its table, delivering a packet to a given port indicated in the message, dropping a packet or buffering a packet contained in the message. When a switch receives a packet from another switch, either it forwards the packet to another switch according to the rules in its current table, or it forwards the packet to the controller if there is no rule matching the packet headers. Accordingly, we have a set of switches: $switches \subseteq SW\_ID$. Each switch has:

- a flow table which may be empty or made of several entries:
  $flowTable \in ENTRY \nrightarrow switches$.
  Each entry has several headers (similar as for packets); each one is specified as follows[4]:
  $eHeader_i \in ENTRY \nrightarrow HEADER$
  $\text{dom}(eHeader_i) = \text{dom}(flowTable)$
- a status: $swStatus \in SW\_ID \nrightarrow SW\_STATE \wedge \text{dom}(swStatus) = switches$
- a buffer $swIncomingMsg$ containing all messages received by the switches:
  $swIncomingMsg \subseteq MESG \times switches$
- a buffer $swIPk$ for all packets it receives, before treatment:
  $swIPk \in PACKET \leftrightarrow switches$; $swIncomingPk$ is the set of packets such that $swIncomingPk \subseteq PACKET$ and $swIncomingPk = \text{dom}(swIPk)$.
  Each packet has a header: $pHeader_i \in PACKET \nrightarrow HEADER$
- a buffer $swOMsg$ that contains messages to be sent to the controller:
  $swOMsg \in MESG \leftrightarrow switches$; $swOutgoingMsg$ is a set of messages such that $swOutgoingMsg \subseteq MESG \wedge swOutgoingMsg = \text{dom}(swOMsg)$
- a buffer $swOPk$ containing packets to be sent to other switches or to the controller: $swOPk \in PACKET \leftrightarrow switches$ and $swOutgoingPk$ the set of packets such that $swOutgoingPk \subseteq PACKET \wedge swOutgoingPk = \text{dom}(swOPk)$.

Behaviour of the switch. We capture the behaviour of the switch by considering how it is involved in the interaction with its environment. Each impact of the environment is considered as an event. The (re)actions of the switch are modelled as events that in turn impact or not the environment. We have then a set of events characterizing the switches; they are as follows.

sw_rcv_matchingPkt: the condition for the occurrence of this event is that there is in the incoming packets of a switch sw, a packet pkt, received from another switch via the data channel $((pkt \mapsto sw) \in dataChan)$, which header $(ahd = pHeader1(pkt))$ is pattern-matched with one entry of the flow table of sw: $(\exists ee.(((ee \in ENTRY) \wedge (ee \in \text{dom}(flowTable))) \wedge (eHeader1(ee) = ahd)))$; the effect of the event is that the packet should be forwarded to another switch: $swIPk := swIPk \cup \{pkt\}$
The Event-B specification of the event is given in the Fig. 4.

sw_rcv_unmatchingPkt: its occurs when a switch receives a packet (from another switch) which header does not match any entry of the flow table.

[4] dom denotes the domain of a relation; ran denotes the range.

sw_sndPk2ctrl: its occurs when a switch emits to the controller, a message containing an unmatched packet;

sw_sendPckt2sw: a switch sends a packet to another switch via the data channel;

sw_newFTentry: the occurrence of this event expresses that a new entry is added to the flow table. ···

```
event sw_rcv_matchingPkt // a switch receives a packet matching a flow table entry
ANY sw pkt ahd
WHERE          /* the guard */
    sw ∈ switches  ∧  pkt ∈ PACKET  ∧  (pkt ↦ sw) ∈ dataChan
    ahd ∈ HEADER  ∧  pkt ∈ dom(pHeader1)
    ahd = pHeader1(pkt)
    ∃ee · (((ee ∈ ENTRY) ∧ (ee ∈ dom(flowTable))) ∧ (eHeader1(ee) = ahd))
    sw ∈ dom(swIPk)  ∧  sw ↦ pkt ∉ swIPk
THEN          /* the substitution */
    swIncomingPk := swIncomingPk ∪ {pkt} // input buffer updated;
    dataChan := dataChan \ {pkt ↦ sw}
    swIPk := swIPk ∪ {sw ↦ pkt} // packet will be forwarded
END
```

Fig. 4. Event-B specification of the event sw_rcv_matchingPkt

Controller. A controller administrates the switches with control messages. It has buffers which contain messages or packets to be sent/received to/from switches: a buffer for incoming packets ($ctlIncomingPk \subseteq PACKET$); a buffer for outgoing packets ($ctlOutgoingPk \subseteq PACKET$).

The controller emits/receives messages on/from the message channel. These messages contain either data packets or instructions to control the switches. Among the control messages we have: the Add order to add an new entry into the table flow of a switch; Modf to modify an entry into the flow table of a switch; Del to delete an entry into the flow table of a switch.

Behaviour of the controller. As for the switch, the behaviour of the controller is captured and modelled as a set of events denoting how the controller interacts with its environment. Each impact of the environment is considered as an event; the (re)actions of the controller are modelled as events that in turn impact or not the environment.

As illustration, among the events of the controller we have the following:

ctl_emitPkt: this event occurs when the controller emits to a switch sw, through a message, one of its pending packets; the condition for this occurrence is that there is some pending packets in the dedicated buffer ($pkt \in ctlOutgoingPk$). The effect of the event is that a message containing the packet is added to the secure channel: $secureChan := secureChan \cup \{msg \mapsto sw\}$ and the buffer

is updated: $ctlOutgoingPk := ctlOutgoingPk \setminus \{pkt\}$. Figure 5 gives the Event-B specification of the event; all the remaining events are specified in a similar way.

ctl_rcvPacketIn: this event occurs when the controller receives a packet from a switch which previously received it but does not have an entry matching it.

ctl_askBarrier: the occurrence of this event specifies when the controller asks for a barrier; that means the controller orders the switch to perform some control with urgency and to send a barrier acknowledgement.

```
event ctl_emitPkt // the controller emits a mesg conveying a packet
ANY sw pkt msg
WHERE          /* the guard */
    sw ∈ switches // in destination to one of the switches
    pkt ∈ PACKET
    pkt ∈ ctlOutgoingPk // one of the packet to be sent on the sw
    msg ∈ MESG // a given message to convey the packet
    (msg ↦ PKOut) ∈ mesgType // a packet of type OUT
    (msg ↦ pkt) ∈ mesgPk // the message contains the packet
THEN          /* the substitution */
    secureChan := secureChan ∪ {msg ↦ sw} //emission on the channel
    ctlOutgoingPk := ctlOutgoingPk \ {pkt}
END
```

Fig. 5. Event-B specification of the event ctl_emitPkt

The global abstract model comprises in an EVENT clause, all the events characterizing the switches and the controller; the occurrence of each event is due to some conditions of the SDN and this occurrence has effect on the SDN. In Event-B a *guard* captures each condition; an Event-B *substitution* describes the effect of the event.

Interaction between Controller and Switches. The interaction is based on communications via channels; we distinguish a data packet channel and a control message channel. The channels are modelled with sets. A switch or a controller writes/reads messages on/from the channels according to their behaviour.

$$secureChan \subseteq MESG \times switches$$
$$dataChan \subseteq PACKET \times switches$$

A first abstract Event-B model is obtained by gathering all these abstractions on data and behaviour.

3.3 Correctness Conditions of the Model

The correctness of the global model depends on the properties formulated in the invariant, enhanced and proved during the model construction. Such properties

are carefully derived from the understanding and the structuring of the SDN (see Sect. 3.2).

For instance the model ensures that: *outgoing packets are sent by one of the switches or by the controller.*
This property (SP) is progressively built with the following parts of the invariant. The packets delivered by a switch to other switches according to routing information are called outgoing packets (see Sect. 3.2).

Each switch is equipped with a buffer of data packets it received ($swIPk$) and a buffer of data packets it forwarded ($swOPk$):
$swIPk \in switches \leftrightarrow PACKET \quad \wedge \quad swOPk \in switches \leftrightarrow PACKET$.

The packets forwarded by a switch to other switches should come from its proper buffer: $\forall sw.swOPk(sw) \subseteq swIPk(sw)$

The union of the packets to be delivered by the switches is denoted by $swOutgoingPk$: $swOutgoingPk \subseteq \text{ran}(swOPk)$

The packets in transit through the switches are gathered in the $swSentPkts$ variable: $swSentPkts \subseteq PACKET$

In the same way all the packets sent by the controller are gathered in the $ctlSentPkts$ variable: $ctlSentPkts \subseteq PACKET$

Finally the property SP is expressed by:

$$swOutgoingPk \subseteq swSentPkts \cup ctlSentPkts$$

In a similar way, may properties among which the two following ones, are progressively formulated as parts of the invariant.

The packets in the data channel should be sent by the controller or by the switches.

$$\text{dom}(dataChan) \subseteq swSentPkts \cup ctlSentPkts$$

The contents of the switches buffers ($swIncomingPk$) should come from the controller or other switches.

$$swIncomingPk \subseteq ctlSentPkts \cup swSentPkts$$

These properties are first stated in the abstract model and then refined along the development process according to the refinement of the state variables.

3.4 Model Construction Strategy: The Refinements

Despite the general development strategy in Event-B which consists in building an abstract global model of a system and then to use several refinements to make it precise, it is still challenging to determine the refinement steps according to the problem at hand. In this work we have considered as one of our targets, the main components of the SDN. That is, we tried to deal with details related to the targeted components (switches, controller). The questions are: what are the main features of the switches and how they impact their environments? what are the main features of the controller behaviour and how they impact

its environment? By answering these questions we finished by introducing, for instance, that switches use various ports and they receive/emit messages on ports. Consequently the first abstract model of channels is impacted and then refined.

We focus on the architecture of the SDN, and then tried to list the details that will support actually the achievement of the network services (routing, access control, traffic engineering; see Sect. 2). We have listed, the detailed structure of packets, the structure of messages, the fine-grained processing of packets inside the switches. Then we order these details and tried to handle them one by one. It follows that we have to detail in the refinements: the structure of packets with various headers and body parts; the structure of messages, and accordingly the refinement of the abstract channels; the behaviour of the events that specify the behaviour of both switches and controller. This guided us to master the gradual modelling. From the methodological point of view this is a recipe for (Event-B) specifiers.

We also follow the basic recommendations of Event-B to consider small steps of refinement at time. Table 1 gives an overview of the refinement chains.

Table 1. The refinement steps

`GblModel0`	The first abstract model; all the events are specified at a high level; for instance we do not have yet information on ports, etc
`GblModel0_1`	Refinement. Ports and headers are introduced in the state space thus refined; the related events are refined
`GblModel0_2`	Refinement. Priorities are introduced in the state space; messages are sent from the controller with a priority in their header
`GblModel0_3`	Refinement. The events guard are refined according to priority rules

3.5 Data Refinement

The set of ports (PORTID) is introduced as data refinement details in the GblModel0_1 refined abstract model. Packets are sent on ports according to the actions defined in the entries of the flow table. One port (also called action) may be the destination of a set of packets.

$$actionsQueues \in PORTID \nrightarrow \mathbb{P}(PACKET) \ // \text{ packets targeting a port}$$
$$actions \in ENTRY \nrightarrow \mathbb{P}(ACTION) \ // \text{ ports concerned by an entry}$$
$$\text{dom}(actions) = \text{dom}(flowTable) \ // \text{ all entries have target ports}$$

An entry may specify several actions or ports. The various fields in SDN packets are also introduced as data refinement with the functions: *macSrc*, *macDst*, *IpSrc*, *IpDst*, *IpProto*, *TpSrc*, *TpDst*, *TpSrcPt*, *TpDstPt*.

3.6 Behavioural Refinement

Explicit priority. The controller (via a human administrator) can introduce *priorities* as an information contained in the messages. Priorities are comparable, they are numbers. Consequently, we introduced this refinement level where the messages are refined by adding to them a field which represents their priority. In Event-B, this is a function giving the priority of each message:
$msgPriority \in MESG \nrightarrow MSG\_PRIORITY$ where $MSG\_PRIORITY$ is the set of priorities (a subset of naturals).
Accordingly, the event ctl_emitPkt for instance, is now refined in the model (`GblModel0_2`); its substitution sets the priority of the message which is sent.

Implicit priority. We introduced implicit priorities via a partial order on messages to be sent; in the sequel the symbol $\prec$ denotes this partial order.

To avoid inconsistencies in the behaviour of switches, the messages they sent should be reordered. In practice, when for instance the flow table is modified by an instruction coming from the controller, the outgoing packet in a switch may be forwarded in a wrong destination due to the modification. Besides, the controller can use the barrier to impose a quick modification.

Accordingly the modification messages coming from the switch should have lower priority compared with the forwarding messages. A priority rule which reorder the events, is that: the add control messages are processed after the forwarding of all data packets. The involved events in the model are: sw_newFTentry, sw_sendPckt2sw, sw_sendPk2Ctrl. Therefore we have the following ordering:

$$\begin{array}{c} \mathsf{sw\_newFTentry} \prec \mathsf{sw\_sendPckt2sw} \\ \mathsf{sw\_sendPk2ctrl} \prec \mathsf{sw\_sendPckt2sw} \\ \mathsf{sw\_sendPk2ctrl} \prec \mathsf{sw\_newFTentry} \end{array}$$

As far as the Del order is concerned, as lost packets in the network can be claimed, we use this hypothesis to consider that the Del order has priority on the forward packet. For the Add order, this does not present an inconsistency risk for outgoing packets. For this reason the Add order can be processed in any order. Barrier messages coming from the controller are the most priority ones. Unmatched packets to be returned to the controller are lower priority than the packet to be forwarded to other switches: a rule is that packets to the controller are sent if there is no packet to be forwarded to other switches.

These priorities have been implemented (in `GblModel0_3`) as a refinement of our model. The guards of the involved events have been strengthen with these rules.

4 Deriving Correct Controller and Switch Components

The purpose is to derive SDN components from the global model resulting from the chain of refinements; such derivation is enabled with Event-B via the use of *model decomposition* techniques: the Abrial-style decomposition (called the A-style decomposition) [3] based on shared variables, and the Butler-style decomposition (called the B-style decomposition) [7,20] based on shared events. Indeed

when decomposing into sub-models a model where events use state variables, either the decomposition is based on the partition of the state variables (and some events may need variables in different sub-models; these are shared events) or the decomposition is based on the partition of events in which case some variables may be needed by events in different sub-models, these are shared variables). In the A-style decomposition, which we have used, events are first partitioned between Event-B sub-components. Then, according to shared variables only modified by one of the sub-components, the events which modify the variables in one component, are introduced as *external events* in the sub-components which do not modify the variables. These external events simulate the behaviour of the events which modify the variables, in the components where the variables are not modified. To avoid inconsistency, external events should not be refined. In the B-style decomposition, variables are first partitioned between the sub-components and then shared events (which use the variables of both sub-components) are split between the sub-components according to the used variables. We used the A-style decomposition because it is more relevant when we consider that the events describe the behaviour of each specific component (controller, switch) of the SDN.

Decomposition into a Controller and Switches

According to our modelling approach (see Sect. 3.2) where events are gathered by family, it is straightforward to list the events that describe the behaviour of the controller in order to separate them from the events related to the switches. The controller component is made of all the events, already introduced as such and prefixed with ctrl, which simulate the behaviour of the controller (see Sect. 3.2): ctl_emitPkt, ctl_rcvPacketIn, ctl_askBarrier, etc. Formally, the decomposition is as if a model $\mathcal{M}_\Sigma$ composed of components $\mathcal{S}_{\sigma_1}$ and $\mathcal{C}_{\sigma_2}$, such that[5] $\mathcal{M}_\Sigma \vDash P$, is split into sub-models $\mathcal{S}_{\sigma_1}$ and $\mathcal{C}_{\sigma_2}$ such that $\mathcal{S}_{\sigma_1} \vDash P$ and $\mathcal{C}_{\sigma_2} \vDash P$, with $\Sigma = \sigma_1 \cup \sigma_2$ the alphabet of $\mathcal{M}_\Sigma$ and σ_1 (resp. σ_2) the alphabet of $\mathcal{S}_{\sigma_1}$ (resp. $\mathcal{C}_{\sigma_2}$).

We experimented with the decomposition plugin of the Rodin toolkit using the A-Style decomposition approach.

A challenging issue here is the question of partitioning a set of identical behaviours; for instance if we would like to decompose the behaviours of the switches as a partition. This question is out of the scope of the existing decomposition techniques because of the non-determinism of data and event modelling.

5 Experimentations and Assessment

The global abstract model has been incrementally worked out by combining invariant verification, refinements and simulation. This is done with the Rodin platform[6]. In Table 2 we give the statistics of the performed proofs

[5] The symbol $\vDash$ denotes the logical satisfaction.

[6] http://wiki.event-b.org/index.php.

on the abstract model and its refinements; the Rodin platform generates consistency proof obligations and refinement proof obligations for the submitted models (see Sect. 3.1). These proof obligations were mostly automatically discharged by the Rodin prover; the remaining are interactively proved. The complete model is available at http://pagesperso.ls2n.fr/~attiogbe-c/mespages/nabla/sdn/SDN-WP2.pdf

The model of the last refinement level has been decomposed into a controller component and switches which preserve all the proved properties. One benefit of deriving components from a global formal model is the ability to study required properties involving the components and their environment. We have illustrated this study with a few properties expected from SDN. Both safety and liveness properties have been considered. This can be extended to other specific properties, following a similar approach.

Table 2. Proof statistics

Elt Name	Total	Auto	Manual	Undischarged
`SDN-WP2`	210	202	8	0
`GlModel0`	97	94	3	0
`GlModel0_1`	63	59	4	0
`GlModel0_1_1`	2	2	0	0
`GblModel0_2`	2	2	0	0
`GblModel0_3`	0	0	0	0

We have expressed and proved several global properties on the model before its decomposition into components. For instance: *The data packets received by any switch are sent by the controller or by the other switches.*

Proof. Assume *ctlSentPkts* be the set of packets sent by the controller; we have to prove that $\forall sw.(sw : switches \Rightarrow swIPs[\{sw\}] \subseteq ctlSentPks)$. If *swIncomingPks* is the union of the buffers of the switches, then it suffices to establish that $swIncomingPks \subseteq ctlSentPkts$. □

Several such safety properties (e.g. Table 3 and Sect. 3.3) have been expressed as invariants in the Event-B model and proved using the Rodin prover.

Table 3. A part of the considered safety properties

$\mathbf{SP}_a$	Any packet in the data channel was sent by the controller or the switches
$\mathbf{SP}_b$	Any packet in the switches buffers was sent by the controller or the switches
$\mathbf{SP}_c$	The packets sent via the message channel are contained in $ctl\_sentPkts$

Similarly, liveness properties study is undertaken using stepwise checking of basic properties. For instance, we prove that, the data packets generated by the controller, are finally emitted by this later. The formula $\mathbf{LP}_{deliv}$ (see Table 4) expresses this property. Literally it describes that after the occurrence of the event `ctl_havePacket` we will finally (F) observe the occurrence of `ctl_emitPkt`. The other formula in Table 4 are similar; the X symbol stands for the next operator. Event-B provides, via the ProB tool integrated in the Rodin, the facilities to state and prove liveness properties. ProB supports LTL,

its extension LTL[e] and CTL properties with the standard modal and temporal operators.

Table 4. A part of the liveness properties in LTL/ProB

$\mathbf{LP}_{OKstatus}$	e(ctl_askStatusMsg) ⇒ F(e(ctl_rcvStatus))
$\mathbf{LP}_{deliv}$	e(ctl_havePacket) ⇒ F(e(ctl_emitPkt))
$\mathbf{LP}_{OKMach}$	e(ctl_emitPkt) ⇒ X(e(sw_rcv_machingPkt))

6 Conclusion

We have shown how to build correct controller and switches components from the refinement of a global formal Event-B model of an SDN system. The correctness was established according to the properties captured and formulated as invariants from the SDN system requirements. The global model was first built by a systematic construction using refinements and then decomposed into the target components. The construction of the abstract model itself was achieved so as to be reusable as a recipe for Event-B developers, following the steps we had identified. We overcame the challenging modelling in Event-B, of an SDN system, viewing it as a discrete events system, and thus as an interaction between its main components. We evaluated our model and components for their conformance to the properties required for SDN systems. We experimented with the various aspects related to property proving and simulation, using the `Rodin` tool. As far as we know, among the related work using formal approaches, this study is the first one proposing an event-based approach for studying SDN systems.

We provided a core event-based model to set the foundation of frameworks dedicated to the development, analysis and simulation of SDN-based applications.

As future work, instead of a one-shot derivation of a specific code for the controller, we are investigating a parametric environment to enable the construction of specific controllers targeting various languages. The same idea is relevant for the switches. In light-weight distributed applications requiring the deployment of ad-hoc SDN, it is desirable to build various specific SDN switches from a single abstract model. Consequently, a process of generic refinement into code will be beneficial.

Acknowledgment. We thank the reviewers for helping to improve the paper.

References

1. Software-Defined Networking: The New Norm for Networks. ONF White paper (2012)
2. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
3. Abrial, J.-R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: application to Event-B. Fundam. Inform. **77**(1–2), 1–28 (2007)
4. Akhunzada, A., Gani, A., Anuar, N.B., Abdelaziz, A., Khan, M.K., Hayat, A., Khan, S.U.: Secure and dependable software defined networks. J. Netw. Comput. Appl. **61**, 199–221 (2016)
5. Al-Shaer, E., Al-Haj, S.: FlowChecker: configuration analysis and verification of federated openflow infrastructures. In: Proceedings of 3rd ACM Workshop on Assurable and Usable Security Configuration, SafeConfig '10, USA, pp. 37–44. ACM (2010)
6. Alsmadi, I., Xu, D.: Security of software defined networks: a survey. Comput. Secur. **53**, 79–108 (2015)
7. Butler, M.: Decomposition structures for Event-B. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 20–38. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00255-7_2
8. El-Hassany, A., Miserez, J., Bielik, P., Vanbever, L., Vechev, M.: SDNRacer: concurrency analysis for software-defined networks. In: Proceedings of 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'16, USA, pp. 402–415 (2016)
9. Foundation, O.W.: Openflow switch specification. ONF **TS-006**(Version 1.3.0), 1–106 (2012)
10. Guha, A., Reitblatt, M., Foster, N.: Machine-verified network controllers. In: Proceedings of 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, USA, pp. 483–494. ACM (2013)
11. Hoang, T.S., Kuruma, H., Basin, D.A., Abrial, J.-R.: Developing topology discovery in Event-B. Sci. Comput. Program. **74**(11–12), 879–899 (2009)
12. Horvath, R., Nedbal, D., Stieninger, M.: A literature review on challenges and effects of software defined networking. Procedia Comput. Sci. **64**, 552–561 (2015); In: Conference on ENTERprise Information Systems/International Conference on Project MANagement/Conference on Health and Social Care Information Systems and Technologies, CENTERIS/ProjMAN/HCist (2015). Accessed 7–9 Oct 2015
13. Hu, H., Han, W., Ahn, G.-J., Zhao, Z.: FLOWGUARD: building robust firewalls for software-defined networks. In: Proceedings of 3rd Workshop on Hot Topics in Software Defined Networking, HotSDN '14, USA, pp. 97–102. ACM (2014)
14. Khurshid, A., Zou, X., Zhou, W., Caesar, M., Godfrey, P.B.: VeriFlow: verifying network-wide invariants in real time. In: Proceedings of 10th USENIX Conference on Networked Systems Design and Implementation, NSDI'13, USA, pp. 15–28. USENIX Association (2013)
15. Kreutz, D., Ramos, F.M.V., Verssimo, P.E., Rothenberg, C.E., Azodolmolky, S., Uhlig, S.: Software-defined networking: a comprehensive survey. Proc. IEEE **103**(1), 14–76 (2015)
16. Mahmud, A., Rahmani, R.: Exploitation of openflow in wireless sensor networks. In: Proceedings of 2011 International Conference on Computer Science and Network Technology, vol. 1, pp. 594–600 (2011)
17. OpenNetworkFoundation. SDN Architecture Overview. (V1.0) (2013)

18. Raza, M.H., Sivakumar, S.C., Nafarieh, A., Robertson, B.: A comparison of software defined network (SDN) implementation strategies. Procedia Comput. Sci. **32**, 1050–1055 (2014); In: The 5th International Conference on Ambient Systems, Networks and Technologies (ANT-2014), the 4th International Conference on Sustainable Energy Information Technology (SEIT-2014)
19. Rothenberg, C.E., Chua, R., Bailey, J., Winter, M., Corra, C.N.A., de Lucena, S.C., Salvador, M.R., Nadeau, T.D.: When open source meets network control planes. Computer **47**(11), 46–54 (2014)
20. Silva, R., Butler, M.: Shared event composition/decomposition in Event-B. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 122–141. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_7