Model Checking Software-Defined Networks with Flow Entries that Time Out

Vasileios Klimis, George Parisis and Bernhard Reus
University of Sussex, UK
{v.klimis, g.parisis, bernhard}@sussex.ac.uk

Abstract—Software-defined networking (SDN) [1] revolutionised network operation and management along with future protocol design; a virtually centralised and programmable controller ‘programs’ network switches through (virtually) centralised, programmable controllers, which deploy network functionality by installing rules in the flow tables of network switches. Although this is a powerful abstraction, buggy controller functionality could lead to severe service disruption and security loopholes, motivating the need for (semi-)automated tools to find, or even verify absence of, bugs. Model checking SDNs has been proposed in the literature, but none of the existing approaches can support dynamic network deployments, where flow entries expire due to timeouts. This is necessary for automatically refreshing (and eliminating stale) state in the network (termed as soft-state in the network protocol design nomenclature), which is important for scaling up applications or recovering from failures. In this paper, we extend our model (MoCS) to deal with timeouts of flow table entries, thus supporting soft state in the network. Optimisations are proposed that are tailored to this extension. We evaluate the performance of the proposed model in UPPAAL using a load balancer and firewall in network topologies of varying size.

I. INTRODUCTION

Software-defined networking (SDN) [1] revolutionised network operation and management along with future protocol design; a virtually centralised and programmable controller (closely related to ‘programs’ network switches through interactions (standardised in OpenFlow [2])) that alternate switches’ flow tables. In turn, switches push packets to the controller when they do not store state relevant to forwarding these packets. Such a paradigm departure from traditional networks enables the rapid development of advanced and diverse network functionality; e.g., in designing next-generation inter-data centre traffic engineering [3], load balancing [4], firewalls [5] and Internet exchange points (IXPs) [6]. Although this is a powerful abstraction, buggy controller functionality could lead to severe service disruption and security loopholes. This has led to a significant amount of research on SDN verification and/or bug finding, including static network analysis [7], [8], [9], dynamic real-time bug finding [10], [11], [12], [13], and formal verification approaches, including symbolic execution [14], [15], [16] and model checking [17], [10], [16], [18] methods. A comprehensive review of existing approaches along with their shortcomings can be found in [19].

Model checking is a renowned automated technique for hardware and software verification and existing model checking approaches for SDNs have shown promising results with respect to scalability and model expressivity, in terms of supporting realistic network deployments and the OpenFlow standard. However, a key limitation of all existing approaches is that they cannot model forwarding state (added in network switches’ flow tables by the controller) that expires and gets deleted. Without this, one cannot model nor verify the correctness of SDNs with soft-state which is prominent in the design of protocols and systems that are resilient to failures and scalable; e.g., as in [21], where flow scheduling is on a per-flow basis, and numerous network protocols where in-network state is not explicitly removed but expires, so that overhead is minimised [22].

In this paper, we extend our model (MoCS) [17] to support soft-state, complying with the OpenFlow specification, by allowing flow entries to time out and be deleted. We propose relevant optimisations (as in [17]) in order to improve verification performance and scalability. We evaluate the performance of the proposed model extensions in UPPAAL using a load balancer and firewall in network topologies of varying size.

II. MoCS SDN MODEL

The MoCS model [17] is formally defined by means of an action-deterministic transition system. We parameterise the model by the underlying network topology, λ, and the controller program, CP, in use. The model is a 6-tuple $M(\lambda,CP) = (S, s_0, A, \rightarrow, AP, L)$, where $S$ is the set of all states the SDN may enter, $s_0$ the initial state, $A$ the set of actions which encode the events the network may engage in, $\rightarrow \subseteq S \times A \times S$ the transition relation describing which execution steps the system undergoes as it performs actions, $AP$ a set of atomic propositions describing relevant state properties, and $L : S \rightarrow 2^{AP}$ is a labelling function, which relates to any state $s \in S$ a set $L(s) \in 2^{AP}$ of those atomic propositions that are true for $s$. Such an SDN model is composed of several smaller systems, which model network components (hosts, switches and the controller) that communicate via queues and, combined, give rise to the definition of $\rightarrow$. A detailed description of MoCS’ components and transitions can be found in [17]. Due to lack of space, in this paper, we only discuss aspects of the model that are required to understand and verify the soundness of the proposed model extensions, and examples used in the evaluation section. Figure 1 illustrates a high-level view of OpenFlow interactions, modelled actions and queues, including the proposed extensions discussed in Section III.

States and queues: A state is a triple $(\pi, \delta, \gamma)$, where $\pi$ is a family of hosts, each consisting of a receive queue (rcvq); $\delta$ is a family of switches, consisting of a switch packet queue

...
Fig. 1: A high-level view of OpenFlow interactions (left half) and modelled actions (right half). A red solid-line arrow depicts an action which, when fired, (1) dequeues an item from the queue the arrow begins at, and (2) adds an item in the queue the arrowhead points to (or multiple items if the arrow is double-headed). Deleting an item from the target queue is denoted by a reverse arrowhead. A forked arrow denotes multiple targeted queues.

\[(pq, \text{switch forward queue (fq), switch control queue (cq), switch flow table (ft); } \gamma \text{ consists of the local controller state (cs) and a family of controller queues: request queue (rq), barrier-reply queue (brq) and flow-removed queue (frq). So } \pi \text{ and } \delta \text{ describe the data-plane and } \gamma \text{ the control plane. The network components communicate via the shared queues. Each transition models a certain network event that will involve some of the queues, and maybe some other network state. Concurrency is modelled through interleavings of these events.} \]

**Transitions:** Each transition is labelled with an action \(\alpha \in A\) that indicates the nature of the network event. We write \(s \leadsto s'\) and \((s, \alpha, s') \in \leftrightarrow\) interchangeably to denote that the network moved from state \(s\) to \(s'\) by executing transition \(\alpha\). The parts of the network involved in each individual \(\alpha\), i.e. packets, rules, barriers, switches, hosts, ports and controller states, are included in the transition label as parameters; e.g., \(\text{match}(sw, pkt, r) \in A\) denotes the action that switch \(sw\) matches packet \(pkt\) by rule \(r\) and, as a result, forwards it accordingly, leading to a new state after transition.

**Atomic propositions:** The propositions in \(AP\) are statements on (1) controller states \((cs \in CS)\), which allows one to reason about the controller’s internal data structures, and (2) packet header fields. Those packets may be in any switch buffer \(pq\) or client buffer \(rcvq\) (but no other buffers). For instance, \(\exists pkt \in sw.pq. P(pkt)\) is a legitimate atomic proposition that states that there is a packet in \(sw\)’s packet queue that satisfies packet \(pkt\) property \(P\).

**Topology:** \(\lambda\) describes the network topology as a bijective map which associates one network interface (a pair of networking device and physical port) to another.

**Specification Logic:** The properties of the SDNs to be checked in this paper are safety properties, expressed in linear-time temporal logic without ‘next-step’ operator, \(\text{LTL}\backslash \{\Box\}\). We have enriched the logic by modal operators of dynamic logic [23], allowing formula construct of the form \([\alpha(x)]P\) stating that whenever an event \(\alpha(x)\) happened, \(P\) must hold. Note that \(P\) may contain variables from \(x\). This extension is syntax sugar in the sense that the formulæ may be expressed by additional state; e.g., \([\text{match}(sw, pkt, r)](r, fwdPort = \text{ctrl})\) states that if \(\text{match}\) happened, it was via a rule that dropped the packet. This permits specification formulæ to be interpreted not only over states, but also over actions that have happened. The model checking problem then, for an SDN model \(M_{(\lambda, CP)}\) with a given topology \(\lambda\), a control program \(CP\) and a formula \(\varphi\) of the specification logic as described above, boils down to checking whether all runs of \(M_{(\lambda, CP)}\) satisfy \(\varphi\), short \(M_{(\lambda, CP)} \models \varphi\).

**SDN Operation:** End-hosts send and receive packets (\(send\) and \(recv\) actions in Figure 1) and switches process incoming packets by matching them (or failing to) with a flow table entry (rule). In the former case (\(match\) action), the packet is forwarded as prescribed by the rule. In the opposite case (\(nomatch\) action), the packet is sent to the controller (\PacketIn message on the left side of Figure 1). The controller’s packet handler is executed in response to incoming \PacketIn messages; as a result of its execution, its local state may change, a number of packets (\PacketOut message) and rule updates (\FlowMod message), interleaved with barriers (\BarrierReq message), may be sent to network switches. Network switches react to incoming controller messages; they forward packets sent by the controller as specified in the respective \PacketOut message (\fwd action), update their own forwarding tables (\add\del actions), respecting set barriers and notifying the controller (\BarrierRes message) when said barriers are executed (\brepl action). Finally, upon receiving a \BarrierRes message, the controller executes the respective handler (\bsync action), which can result in the same effects as the \PacketIn message handler.

**Abstractions:** To obtain finitely representable states, all queues in the model must be finitely representable. For packet queues we use multisets subject to \((0, \infty)\) abstraction; a packet either does not appear in the queue or appears an unbounded number of times. The other queues are simply modelled as finite sets. Modelling queues as sets means that entries are not processed in the order of arrival. This is intentional for packet queues but for controller queues this may limit behaviour unless the controller program is order-insensitive. We focus on those controller programs in this paper.

### III. Modelling Flow Entry Timeouts

In order to model soft-state in the network, we enrich our model with two new actions that model flow entry timeouts and subsequent handling of these timeouts by the controller program. Note that in our model, timeouts are not triggered by any kind of clock; instead, they are modelled through the interleaving of actions in the underlying transition system that ensure that flow removal (and subsequent handling by the
controller program) will appear as it would for any possible value of a timeout in a real system.

The new actions are defined as follows: \texttt{frmvd(sw, r)} models the timeout event, as an action in the transition system that removes the flow entry (rule) \( r \) from switch \( sw \) and notifies the controller by placing a \texttt{FlowRemoved} message (see Figure 1) in the respective queue \((\text{frq})\). The \texttt{fsync(sw, r, cs)} action models the call to the \texttt{FlowRemoved} message handler. As a result of the handler execution, the controller’s local state \((cs)\) may change, a number of packets \((\text{PacketOut} \text{message})\) and rule updates \((\text{FlowMod} \text{message})\), interleaved with barriers \((\text{BarrierReq} \text{message})\), may be sent to network switches. In order to model timeouts, rules are augmented with a \texttt{timeout} bit which, when true, signals that the installed rule can be removed at any time, i.e., the \texttt{frmvd}-action can be interleaved, in any order, with any other action that is enabled at any state later than the installation of this rule.

To support our examples, we add to the set of \texttt{FlowMod} messages \texttt{modify flow entry} instruction. In [17] we only used \texttt{add(sw, r)} and \texttt{del(sw, r)} messages, for installing and deleting rule \( r \) at switch \( sw \), respectively. We now add \texttt{mod(sw, f, a)} to these messages. This instructs switch \( sw \) that if a rule is found in \( sw.\text{ft} \) that matches field \( f \), its forwarding actions are modified by \( a \). If no such rule exists, \texttt{mod(\cdot)} does not do anything.

\textbf{Optimisation:} To tackle the state-space explosion, we exploit the fact that some traces are observationally \(w.r.t.\) the property to be proved equivalent, so that only one of those needs to be checked. This technique, referred to as \textit{partial-order reduction} (POR) [24], reduces the number of interleavings (traces) one has to check. To prove equivalence of traces, one needs actions to be permutable and invisible to the property at hand. This is the motivation for the following definition:

\textbf{Definition 1 (SAFE ACTIONS)} Given a context \( \text{CTX} = (\text{CP}, \lambda, \varphi) \), and SDN model \( \mathcal{M}_{\lambda,CP} = (S, A, \rightarrow, s_0, AP, L) \), an action \( \alpha(\cdot) \in A(s) \) is called \textit{safe} if it is (1) independent of any other action \( \beta \) in \( A \), i.e. executing \( \alpha \) after \( \beta \) leads to the same state as running \( \beta \) after \( \alpha \), and (2) \textit{unobservable} for \( \varphi \) (also called \textit{\varphi-invariant}), i.e., \( s \models \varphi \) iff \( \alpha(s) \models \varphi \) for all \( s \in S \) with \( \alpha \in A(s) \).

The following property of controller programs is needed to show safety:

\textbf{Definition 2 (ORDER-SENSITIVE CONTROLLER PROGRAM)} A controller program \( \text{CP} \) is order-sensitive if there exists a state \( s \in S \) and two actions \( \alpha, \beta \) in \( \{\text{ctrl}(\cdot), \text{bsync}(\cdot), \text{fsync}(\cdot)\} \) such that \( \alpha, \beta \in A(s) \) and \( s \xrightarrow{\alpha} s_1 \xrightarrow{\beta} s_2 \) and \( s \xrightarrow{\beta} s_3 \xrightarrow{\alpha} s_4 \) with \( s_2 \neq s_4 \).

In [17] we already showed that certain actions are safe and can be used for PORs. We now show that the new \texttt{fsync(\cdot)} action is safe on certain conditions.

\textbf{Lemma 1 (SAFENESS)} For transition system \( \mathcal{M}_{\lambda,CP} = (S, A, \rightarrow, s_0, AP, L) \) and a formula \( \varphi \in \text{LTL}_{\lambda,\varphi} \), \( \alpha = \text{fsync}(sw, r, cs) \) is safe iff the following two conditions are satisfied:

\textbf{Independence} \( \text{CP} \) is not order-sensitive

\textbf{Invisibility} if \( Q(q) \) occurs in \( \varphi \), where \( q \in CS \), then \( \alpha \) is \( \varphi \)-invariant

\textbf{Proof.} See Appendix B.

Given a context \( \text{CTX} = (\text{CP}, \lambda, \varphi) \) and an SDN network model \( \mathcal{M}_{\lambda,CP} = (S, A, \rightarrow, s_0, AP, L) \), for each state \( s \in S \) define \textit{ample}(\( s \)) as follows: if \( \{\alpha \in A(s) \mid \alpha \text{ safe } \} \neq \emptyset \), then \textit{ample}(\( s \)) = \{\alpha \in A(s) \mid \alpha \text{ safe } \}. \) Otherwise, \textit{ample}(\( s \)) = \( A(s) \). Next, we define \( \mathcal{M}_{\lambda,CP}^{\text{fr}} = (S^{\text{fr}}, A, \rightarrow, s_0, AP, L^{\text{fr}}) \), where \( S^{\text{fr}} \subseteq S \) the set of states reachable from the initial state \( s_0 \) under \( \rightarrow_{\text{fr}} \), \( L^{\text{fr}}(s) = L(s) \) for all \( s \in S^{\text{fr}} \) and \( \rightarrow_{\text{fr}} \subseteq \rightarrow_{\text{fr}} \times A \times S^{\text{fr}} \) is defined inductively by the rule:

\[ s \xrightarrow{\alpha} s' \quad \text{if} \quad \alpha \in \text{ample}(s) \]

Now we can proceed to extend the POR Theorem of [17]:

\textbf{Theorem 1 (FLOW-REMOVED EQUIVALENCE)} Given a property \( \varphi \in \text{LTL}_{\lambda,\varphi} \), it holds that \( \mathcal{M}_{\lambda,CP}^{\text{fr}} \) satisfies \( \varphi \) iff \( \mathcal{M}_{\lambda,CP} \) satisfies \( \varphi \).

The proof is a consequence of Lemma 1 applied to the proof of Theorem 2 in [17]. See Appendix B for a detailed proof.

\textbf{IV. EXPERIMENTAL EVALUATION}

In this section we experimentally evaluate the proposed extensions in terms of verification performance and scalability. We use a realistic controller program that enables a network switch to act both as a load balancer and stateful firewall (see §A-CP1). The load balancer keeps track of the active sessions between clients and servers in the cluster (see Figure 2), while, at the same time, only allowing specific clients to access the cluster. Soft state is employed here so that flow entries for completed sessions (that were previously admitted by the firewall) time out and are deleted by the switch without having to explicitly monitor the sessions and introduce unnecessary signalling (and overhead). In the underlying SDN model, the \texttt{frmvd} action is fired, which, in turn, deletes the flow entry from the switch’s table and notifies the controller of that. This enables the \texttt{fsync} action that calls the flow removal handler.

![Fig. 2: Four clients and two servers connecting to an OF-switch. ■ is not white-listed.](image-url)
and resulting load, is uniformly distributed to all available servers, and (2) that traffic from non-whitelisted clients is blocked. More concretely, “a packet from a ‘dodgy’ address should never reach the servers, and the difference between the number of assigned sessions at each server should never be greater than 1”, formally,

\[
\forall s_i, s_j \in \text{Servers} \forall \text{pkt} \in s_i, \text{rcvq}. \\
\neg \text{pkt.src} = \text{dodgy} \land |s\text{Load}[s_i] - s\text{Load}[s_j]| < 2 \quad (\varphi)
\]

where \(s\text{Load}\) stores the active session count for each server.

In the first (buggy) version of the controller’s packet handler (shaded grey in §A-CP1) and flow removal handler §A-CP2, the controller program assigns new sessions to servers in a round-robin fashion and keeps track of the active sessions (array \text{deplSessions} in the provided pseudocode). When a session expires, the respective flow table entry is expected to expire and be deleted by the switch without any signalling between the controller, clients or servers. As stated above, this controller program does not satisfy safety property \(\varphi\) because the controller does nothing to rebalance the load when a session expires. Our model implementation discovered the bug in the topology shown in Figure 2 with 3 sessions in 11ms exploring 202 states.

In the second (still buggy) version of the controller, session scheduling is more sophisticated (shaded blue in §A-CP1); a session is assigned to the server with the least number of active sessions. Although the updated load balancing algorithm does keep track of the active sessions per server, this controller is still buggy because no rebalancing takes place when sessions expire. In a topology of 4 clients and 2 servers, we were able to discover the bug in 52ms after exploring 714 states.

We fix the bug by allowing the controller program to rebalance the active sessions, when (1) a session expires and (2) the load is about to get out of balance, by moving one session from the most-loaded to the least-loaded server (§A-CP3). In the same topology as above, we verified the property in 625ms after exploring 15068 states.

Next, we evaluate the performance of the proposed model and extensions for verifying the correctness of the property in a given SDN. We do that by verifying \(\varphi\) with the correct controller program, discussed above, and scaling up the topology in terms of clients, servers and active sessions. Results are listed in Table I and state exploration is illustrated in Figure 3.

Table I lists performance of the model checker for verifying the correct controller program with PORs disabled on the left and with PORs enabled on the right, respectively. For each chosen topology we list the number of states explored, CPU time used, and memory used. The topology is shaped as in Figure 2, and parametrised by the number of clients (ranging from 3 to 5) and servers (ranging from 2 to 5), as indicated in the table. The number of required packets and rules, respectively, is shown in grey. These numbers are always uniquely determined by the choice of topology. Where there are no entries in the table (indicated by a dash) the verification did not terminate within 24 hours.

The results clearly show that the verification scales well with the number of servers but not with the number of clients. They also demonstrate that the POR optimisation massively reduces the state space and thus the verification time. This is not surprising as the number of possible interleavings is massively increased by the non-deterministic rule timeout events.

![Fig. 3: Explored States (logarithmic scale). Wide bars represent the optimised model and narrow ones (inside) the unoptimised model. Unicoloured bars represent non-termination.](image)

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
<th>without POR</th>
<th>with POR</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td>CPU time</td>
<td>Resident memory [KIB]</td>
<td>States</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>13</td>
<td>15,068</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>16</td>
<td>15,068</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>25</td>
<td>15,068</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>31</td>
<td>15,068</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>47</td>
<td>–</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>25</td>
<td>–</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>33</td>
<td>–</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>41</td>
<td>–</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>21</td>
<td>–</td>
</tr>
</tbody>
</table>

V. CONCLUSION AND FUTURE WORK

We have proposed model checking of SDN networks with flow entries (rules) that time out. Timeouts pose problems due to the great number of resulting interleavings to be explored. Our approach is the first one to deal with timeouts, exploiting partial-order reductions, and performing reasonably well for small networks. We demonstrated that bug finding works well for SDN networks in the presence of flow entry timeouts. Future work includes exploring flow removals with timeouts that are constrained by integer to enforce certain orderings of timeout messages as well as improvements in performance, for instance, by using bounded model checking tools for concurrent programs.

1 It is worth stressing that modelling such functionality is not supported by existing model checking approaches, such as [17] and [18], where flow table entries can only be explicitly deleted by the controller.

2 UPPAAL [25] is the back-end verification engine for MOCS and all experiments were run on an 18-Core iMac pro, 2.3GHz Intel Xeon W with 128GB DDR4 memory.

3 Note that the sync-optimisation was not enabled in the examples above.
REFERENCES


APPENDIX

A. Controller Programs

CP1 implements the PacketIn message handler that processes packets sent by switches when the nomatch action is fired. The two different versions of functionality discussed in the paper are defined by the leastConnectionsScheduling constant. When leastConnectionsScheduling is false, server selection is done in a round-robin fashion, whereas, in the opposite case, the controller assigns the new session to the server with the least number of active sessions.

Controller Program CP1: PacketIn Message Handler

```java
1: handler pktIn(pkt, sw)
2:   if pkt.srcIP != dodgy_client then
3:     if ~deplSessions[pkt.srcIP] then
4:       if leastConnectionsScheduling then
5:         // Round-Robin rotation
6:         server ← server mod 2 + 1
7:       else
8:         // Least-Connections scheduling
9:         server ← min(sLoad[])
10:   end if
11:  rule1.srcIP ← pkt.srcIP
12:  rule1.in_port ← pkt.in_port
13:  rule1.fwdPort ← server
14:  if leastConnectionsScheduling then
15:    rule2.srcIP ← server
16:    rule2.dstIP ← pkt.srcIP
17:    rule2.fwdPort ← pkt.in_port
18:    rule2.flowRemoved ← true
19:    // Initialisation of drop rule rule_d
20:    rule_d.srcIP ← dodgy_client
21:    rule_d.fwdPort ← drop
22:    // Deployment of rules
23:    send_message(FlowMod(add(rule1)), sw)
24:    send_message(FlowMod(add(rule2)), sw)
25:    send_message(FlowMod(add(rule_d)), sw)
26:  end if
```

CP2 implements the naive (and buggy) FlowRemoved message handler. When soft state expires in the network, the handler merely updates its local state to reflect the update in the load.

Controller Program CP2: Naive FlowRemoved message handler

```java
1: handler flowRmv(rule, sw)
2:   sLoad[rule.srcIP] ← sLoad[rule.srcIP] + +
3:   deplSessions[rule.destIP] ← false
4: end handler
```

CP3 implements a more sophisticated (and correct) FlowRemoved message handler. When soft state expires in the network, the handler updates its local state to reflect the update
in the load and re-assigns active sessions from the most to the least loaded server, by updating the flow table of the switch accordingly.

**Controller Program CP.3:** Correct \textit{FlowRemoved} message handler

```
1: handler flowRmv(rule, sw)
2:  sLoad[rule.srcIP] = 
3: if max(sLoad[]) − min(sLoad[]) > 1 then
4:  r ← rule in sw.flt | fwdPort = max(sLoad[])
5:  rs ← symmetric rule of r
6:  cm ← mod(r, fwdPort ← min(sLoad[]))
7:  cmₐ ← mod(rs, srcIP ← min(sLoad[]))
8:  send_message (FlowMod(cm, sw))
9:  send_message (FlowMod(cmₐ, sw))
10: sLoad[max(sLoad[])] = −
11: sLoad[min(sLoad[])] +=
12: end
13: end handler
```

**B. Proofs**

**Lemma 1 (Safeness)** For transition system \(\mathcal{M}_{(\lambda, \text{cp})} = (S, A, \rightarrow, s₀, AP, L)\) and a formula \(\varphi \in \text{LTL}_{\{\lambda\}}\), \(\alpha = \text{fsync}(sw, r, cs)\) is safe iff the following two conditions are satisfied:

- **Independence** CP is not order-sensitive
- **Invisibility** \(\notin \mathcal{Q}(q)\) occurs in \(\varphi\), where \(q \in CS\), then \(\alpha\) is \(\varphi\)-invariant

**Proof.** To show safety we need to show two properties: \textit{independence} (action is independent of any other action) and \textit{invisibility} w.r.t. the context, in particular controller program, topology function and formula \(\varphi\).

**Independence:** Recall that two actions \(\alpha\) and \(\beta \neq \alpha\) are independent iff for any state \(s\) such that \(\alpha \in A(s)\) and \(\beta \in A(s)\):

1. \(\alpha \in A(\beta(s))\) and \(\beta \in A(\alpha(s))\)
2. \(\alpha(\beta(s)) = \beta(\alpha(s))\)

(1) It can be easily checked that no instance of safe actions \(\text{fsync}(-)\) disables any other action, nor is any safe \(\text{fsync}(-)\) disabled by any other action, so the first condition of independence holds.

(2) For any safe \(\alpha = \text{fsync}(-)\) and any other action \(\beta\) we can assume already that they meet Condition (1). To show that any interleaving with any action \(\beta \neq \alpha\) leads to the same state, we observe that

- if \(\beta\) is not an \(\text{fsync, ctrl or bsync}\) action, then the mutations of queues by these actions do not interfere with each other.
- The interesting cases occur when \(\beta\) is in \{\text{fsync}(-), \text{ctrl}(-), \text{bsync}(-)\}. From the first condition we know that CP is not order-sensitive, which implies that \(\alpha\) and \(\beta\) are independent. Order-insensitivity is a relatively strong condition but it ensures correctness of the lemma and thus partial order reduction. Thus any interleaving of \(\alpha\) and \(\beta\) leads to the same state.

**Invisibility:** \(\alpha = \text{fsync}(sw, r, cs)\) may only affect \(\text{frq, sw.fq, sw'.cq}\) (for some switches \(sw'\)), and the control state \(cs\). We know by definition of our Specification Language that it cannot refer to \(\text{frq}\) or any \(sw'.cq\). In case the control state changes, \(\alpha\) is invisible to \(\varphi\) because of the second condition (Invisibility) of Lemma 1.

\[\square\]

**Theorem 1 (Flow-Removed Equivalence)** Given a property \(\varphi\), \(\mathcal{M}_{(\lambda, \text{cp})}^{fr}\) satisfies \(\varphi\) iff \(\mathcal{M}_{(\lambda, \text{cp})}^{\text{fr}}\) satisfies \(\varphi\).

**Proof.** If \(\text{ample}(s)\) satisfies the following conditions:

- C1 (Non-)emptiness condition: \(\emptyset \neq \text{ample}(s) \subseteq A(s)\).
- C2 Dependency condition: Let \(s \xrightarrow{\alpha₁} s₁ \ldots \xrightarrow{\alphaₙ} sₙ \xrightarrow{\lambda₁} t\) be a run in \(\mathcal{M}\). If \(\beta \in A(\text{ample}(s))\) depends on \(\text{ample}(s)\), then \(\alphaᵢ \in \text{ample}(s)\) for some \(0 < i \leq n\), which means that in every path fragment of \(\mathcal{M}\), \(\beta\) cannot appear before some transition from \(\text{ample}(s)\) is executed.
- C3 Invisibility condition: If \(\text{ample}(s) \neq A(s)\) (i.e., state \(s\) is not fully expanded), then every \(\alpha \in \text{ample}(s)\) is invisible.
- C4 Every cycle in \(\mathcal{M}\) contains a fully expanded state \(s\) (i.e., \(\text{ample}(s) = A(s)\)).

then for each path in \(\mathcal{M}\) there exists a stutter-trace equivalent path in \(\mathcal{M}_{\text{fr}}\), and vice-versa, denoted \(\mathcal{M} \equiv \mathcal{M}_{\text{fr}}\) – as we now show.

C1 The (non-)emptiness condition is trivial since by definition of \(\text{ample}(s)\) it follows that \(\text{ample}(s) = \emptyset\) iff \(A(s) = \emptyset\).

C2 By assumption \(\beta \in A(\text{ample}(s))\) depends on \(\text{ample}(s)\). But with our definition of \(\text{ample}(s)\) this is impossible as all actions in \(\text{ample}(s)\) are safe and by definition independent of all other actions.

C3 The validity of the invisibility condition is by definition of \(\text{ample}\) and safe actions.

C4 We now show that every cycle in \(\mathcal{M}_{(\lambda, \text{cp})}^{\text{fr}}\) contains a fully expanded state \(s\), i.e. a state \(s\) such that \(\text{ample}(s) = A(s)\). By definition of \(\text{ample}(s)\) it is equivalent to show that there is no cycle in \(\mathcal{M}_{(\lambda, \text{cp})}^{\text{fr}}\) consisting of safe actions only. We show this by contradiction, assuming such a cycle of only safe actions exists.

Distinguish two cases.

**Case 1** A sequence of safe actions of same type.

Let \(\rho\) an execution of \(\mathcal{M}_{(\lambda, \text{cp})}^{\text{fr}}\) which consists of only one type of \(\text{fsync}\)-actions: \(\rho = s₁ \xrightarrow{\text{fsync}(sw₁, r₁, cs₁)} s₂ \xrightarrow{\text{fsync}(sw₂, r₂, cs₂)} \ldots s_{i-1} \xrightarrow{\text{fsync}(sw_{i-1}, r_{i-1}, cs_{i-1})} sᵢ\). Suppose \(\rho\)

\(^4\)Generalisations by a more clever analysis of the controller program are a future research topic.
is a cycle. According to the $ctrl$ semantics, for each transition $s \xrightarrow{fsync(sw,rx)} s'$, where $s = (\delta, \gamma)$, $s' = (\delta', \gamma')$, it holds that $\gamma', frq = \gamma, frq \setminus \{r\}$ as we use sets to represent $frq$ buffers. Hence, for the execution $\rho$ it holds $\gamma_1, frq = \gamma_1, frq \setminus \{r_1, r_2, \ldots r_{i-1}\}$ which implies that $s_1 \neq s_i$. Contradiction.

Case 2 A sequence of different safe actions. Suppose there exists a cycle with mixed safe actions starting in $s_1$ and ending in $s_i$. Distinguish the following cases.

i) There exists at least a $fsync$ action in the cycle. According to the effects of safe transitions, the $fsync$ action will switch to a state with smaller $frq$. It is important here that no action of other type than $fsync$ accesses $frq$. This implies that $s_1 \neq s_i$. Contradiction.

ii) No $fsync$ action in the cycle. This is already established in [17].