A Revised and Verified Proof of the Scalable Commutativity Rule

Lillian Tsai[†], Eddie Kohler[⋆], M. Frans Kaashoek[†], and Nickolai Zeldovich[†]
[†] MIT CSAIL [⋆] Harvard University

1 Introduction

This paper explains a flaw in the published proof of the Scalable Commutativity Rule (SCR) [1], presents a revised and formally verified proof of the SCR in the Coq proof assistant, and discusses the insights and open questions raised from our experience proving the SCR.

2 The Scalable Commutativity Rule

In order to explore the connection between commutativity and scalability in practical systems, Clements et al. [1] defined a new type of commutativity called *SIM commutativity*,[1] a property that can hold of certain interface specifications. This was used to state and prove the Scalable Commutativity Rule (SCR), which claims that every SIM-commutative interface has a conflict-free implementation—that is, on modern machines, a *scalable* implementation. Even if an interface is commutative only in a restricted context, there exists an implementation that scales in that context.

The rest of this section summarizes SIM commutativity and the precise statement of the rule, and describes the published proof of the rule.

2.1 Specifications

Specifications are represented using *actions*, where each action is either an *invocation* (representing an operation call with arguments) or a *response* (representing a return value). Each invocation is made by a specific thread, and the corresponding response is returned to the same thread. The division into invocations and responses models blocking interfaces and concurrent operations [2]. Invocations are written as âĂIJ$f(\text{args})_\alpha$âĂİ and responses are written as âĂIJ$\overline{\text{retval}}_\alpha$,âĂİ where overbars mark responses and Greek letters represent thread IDs.

A particular execution of a system is a *history* or *trace*, which is just a sequence of actions. For example,

$$H = [\mathrm{A}_\alpha, \mathrm{B}_\gamma, \mathrm{C}_\beta, \overline{\mathrm{A}}_\alpha, \overline{\mathrm{C}}_\beta, \overline{\mathrm{B}}_\gamma, \mathrm{D}_\alpha, \overline{\mathrm{D}}_\alpha, \mathrm{E}_\beta, \mathrm{F}_\gamma, \mathrm{G}_\alpha, \overline{\mathrm{E}}_\beta, \overline{\mathrm{G}}_\alpha, \overline{\mathrm{F}}_\gamma]$$

consists of seven invocations and seven corresponding responses across three different threads. In a *well-formed* history, each thread's actions alternate invocations and responses, so each thread has at most one outstanding invocation at any point. H above is well-formed; for instance, in the thread-restricted subhistory $H|\alpha = [\mathrm{A}_\alpha, \overline{\mathrm{A}}_\alpha, \mathrm{D}_\alpha, \overline{\mathrm{D}}_\alpha, \mathrm{G}_\alpha, \overline{\mathrm{G}}_\alpha]$, which selects α's actions from H, invocations and responses alternate as required.

[1] SIM stands for State-dependent, Interface-based, and Monotonic.

A *specification* models an interface's behavior as a prefix-closed set of well-formed histories. A system execution is "correct" according to the specification if its trace is included in the specification. For instance, if $\mathscr{S}$ corresponded to the POSIX specification, then $[\textbf{getpid}_\alpha, \overline{\textbf{92}}_\alpha] \in \mathscr{S}$ (a process may have PID 92) but $[\textbf{getpid}_\alpha, \overline{\textbf{ENOENT}}_\alpha] \notin \mathscr{S}$ (the **getpid()** system call may not return that error). A specification constrains both invocations and responses: $[\textbf{NtAddAtom}_\alpha]$ is not in the POSIX specification because **NtAddAtom** is not a POSIX system call.

An *implementation* is an abstract machine that takes invocations and calculates responses. The original proof of the SCR by Clements et al. [1] (also presented in Section 2.4) uses a class of machines on which conflict-freedom is defined; a good analogy is a Turing-type machine with a random-access tape, where conflict-freedom follows if the machine's operations on behalf of different threads access disjoint portions of the tape.

An implementation M *exhibits* a history H if, when fed H's invocations at the appropriate times, M can produce H's responses (so that its external behavior equals H overall). An implementation M is *correct* for a specification $\mathscr{S}$ if M's responses always obey the specification. This means that every history exhibited by M is either in $\mathscr{S}$ or contains some invalid invocation.

2.2 Commutativity

SIM commutativity aims to capture state dependence at the interface level. State dependence means SIM commutativity must capture when operations commute in some states, even if those same operations do not commute in other states. SIM commutativity captures this contextually, without reference to any particular implementation's state: to reason about *possible* implementations, SIM commutativity captures the scalability inherent in the interface itself. This in turn makes it possible to use the SCR early in software development, during interface design.

Commutativity states that actions may be reordered without affecting eventual results. A history H' is a *reordering* of H when $H|t = H'|t$ for every thread t. This allows actions to be reordered across threads, but not within them. For example, if $H = [\mathrm{A}_\alpha, \mathrm{B}_\beta, \overline{\mathrm{A}}_\alpha \mathrm{C}_\alpha, \overline{\mathrm{B}}_\beta, \overline{\mathrm{C}}_\alpha]$, then $[\mathrm{B}_\beta, \overline{\mathrm{B}}_\beta, \mathrm{A}_\alpha, \overline{\mathrm{A}}_\alpha, \mathrm{C}_\alpha, \overline{\mathrm{C}}_\alpha]$ is a reordering of H, but $[\mathrm{B}_\beta, \mathrm{C}_\alpha, \overline{\mathrm{B}}_\beta, \overline{\mathrm{C}}_\alpha, \mathrm{A}_\alpha, \overline{\mathrm{A}}_\alpha]$ is not, since it doesn't respect the order of actions in $H|\alpha$.

Now, consider a history $H = X \,||\, Y$ (where $||$ concatenates action sequences). Y *SI-commutes* in H when given any re-

ordering Y' of Y, and any action sequence Z,

$$X \,||\, Y \,||\, Z \in \mathscr{S} \quad \text{if and only if} \quad X \,||\, Y' \,||\, Z \in \mathscr{S}.$$

This definition captures state dependence at the interface level. The action sequence X puts the system into a specific state, without specifying a representation of that state (which would depend on an implementation). Switching regions Y and Y' requires that the exact responses in Y remain valid according to the specification even if Y is reordered. The presence of region Z in both histories requires that reorderings of actions in region Y cannot be distinguished by future operations, which is an interface-based way of saying that Y and Y' leave the system in the same state.

Unfortunately, SI commutativity is not sufficient to prove the SCR. To avoid certain degenerate cases,[2] the definition of commutativity must be strengthened to be *monotonic* (the M in SIM). An action sequence Y **SIM-commutes** in a history $H = X \,||\, Y$ when for any *prefix* P of any reordering of Y (including $P = Y$), P SI-commutes in $X \,||\, P$. Equivalently, Y SIM-commutes in H when, given any prefix P of any reordering of Y, any reordering P' of P, and any action sequence Z,

$$X \,||\, P \,||\, Z \in \mathscr{S} \quad \text{if and only if} \quad X \,||\, P' \,||\, Z \in \mathscr{S}.$$

Like SI commutativity, SIM commutativity captures state dependence and interface basis. Unlike SI commutativity, SIM commutativity excludes cases where the commutativity of a region changes depending on future operations. The SCR relies on SIM commutativity.

2.3 Rule

The Scalable Commutativity Rule formally states the following:

> Assume an interface specification $\mathscr{S}$ that has a correct implementation, and a history $H = X \,||\, Y$ exhibited by that implementation. Then whenever Y SIM-commutes in H, there exists a correct implementation of $\mathscr{S}$ whose steps in Y are conflict-free. Since, given reasonable workload assumptions, conflict-free operations empirically scale on modern multicore hardware, this implementation is scalable in Y.

2.4 Proof

The published proof of the SCR proceeds by construction. The pseudocode for the constructed implementation is shown in Figure 1. We briefly describe how the proof proceeds (eliding certain details about interruptability and thread switches).

[2]Consider an interface that supports an invocation **undefinedbehavior** among others, where where if a trace contains **undefinedbehavior**, then all later responses in the trace may return any value whatsoever. This resembles the specification governing C compilers. Every sequence containing **undefinedbehavior** SI-commutes, even if the same sequence without **undefinedbehavior** requires a non-scalable implementation (e.g., by representing a counter that increments by 1 on each call). However, most practical implementations cannot see the future, so they cannot choose a scalable implementation in the hope that **undefinedbehavior** will eventually occur.

```
1  m_rule(s, i) ≡
2    t ← thread(i)
3    If head(s.h[t]) = COMMUTE:     // enter conflict-free mode
4      s.commute[t] ← TRUE; s.h[t] ← tail(s.h[t])
5    If head(s.h[t]) = i:
6      s.h[t].pop()
7    If head(s.h[t]) is a response and thread(head(s.h[t])) = t:
8      r ← head(s.h[t])                          // replay s.h
9    else if s.h[t] ≠ EMULATE:      // H complete/input diverged
10     H' ← a witness consistent with s.h[t]
11     For each invocation x in H':
12       ⟨s.refstate, _, _⟩ ← M(s.refstate, x)
13     s.h[u] ← EMULATE for each thread u
14   If s.h[t] = EMULATE:
15     ⟨s.refstate, r, _⟩ ← M(s.refstate, i)
16   else if s.commute[t]:                     // conflict-free mode
17     s.h[t] ← tail(s.h[t])
18   else:                                     // replay mode
19     s.h[u] ← tail(s.h[u]) for each thread u
20   Return ⟨s, r⟩
```

Figure 1: Constructed scalable implementation m_{rule} for history H and reference implementation M.

Given a specification $\mathscr{S}$, an arbitrary implementation M satisfying $\mathscr{S}$, and a specific history $H = X||Y$ generated by M where Y SIM-commutes in H, the proof constructs an implementation m_{rule} that scales (is conflict-free) within the SIM-commutative region Y.

m_{rule} operates in three modes: *replay*, *conflict-free*, and *emulation* modes. Its state consists of three parts:

1. $s.h[t]$, a per-thread history initialized as $X||\text{COMMUTE}||(Y|t)$
2. $s.commute[t]$, a per-thread flag which is set to true when COMMUTE is reached
3. $s.refstate$, the state of reference implementation M

m_{rule} starts in replay mode. This mode replays the history H as long as each thread invokes operations in the same order as it did in H. While thread t has not yet reached the commutative region, t's invocation matches that of $s.h[t]$ (line 5), and the next action in $s.h[t]$ is a response to t's invocation (line 7), m_{rule} returns the matching response (line 8) and advances the stored history $s.h[u]$ of all threads (line 19).

If the next step in $s.h[t]$ is COMMUTE, then $s.commute[t]$ is set to true (line 4) and m_{rule} enters conflict-free mode. In this mode, m_{rule} starts replaying steps in $Y|t$. Just like before, if the next action in $s.h[t]$ is a response to t's invocation (line 7), m_{rule} returns the matching response (line 8). However, it advances only the stored history of t, namely $s.h[t]$ (line 17). This means that after m_{rule} enters conflict-free mode for a thread t (i.e., $s.commute[t] = true$), it accesses and modifies

only those state components specific to t. Thus, any steps in the conflict-free region Y are conflict-free.

Once $s.h[t]$ has fully replayed or if a thread t *diverges*—that is, t's next invocation does not match the next invocation in $s.h[t]$—then m_{rule} switches to emulate mode for all threads. In this mode, m_{rule} feeds the reference machine M invocations to determine the response to return. Before we can use the reference machine M in this way, however, the state of M must reflect the state of the execution history thus far.

We initialize M's state by finding a *witness* of M that is consistent with the execution history (line 10). A *witness* of the execution history is a sequence of invocations that, when fed to M, generates the execution history. Once a witness is found, we know that feeding M the witness's sequence of invocations (line 12) will bring M to a valid state, where all future responses will be valid according to the spec.

Finding a witness is easy before m_{rule} reaches conflict-free mode: since m_{rule} generated the history $X||Y$, simply take all invocations in X (up to the current point) in order as the witness. However, if m_{rule} has entered the conflict-free mode and executed operations in the SIM-commutative region, the order in which operations were executed in this region may not equal the order in which operations were executed in Y. Here is where SIM commutativity comes in: we can reorder the operations in the commutative region of the execution history to achieve a witness. Because of SIM commutativity, **we can initialize M with a witness comprised of commutative actions in a different order than that in which they were executed, and all future responses will still be valid according to the specification**.

3 Exemplifying the proof's flaw

This last, bold statement is where the proof goes wrong. To help build intuition, we now present a counterexample in which the proof-constructed machine fails: the construction cannot find a witness that will initialize M with a valid state.

3.1 Specification

Imagine a commutative spec $\mathscr{S}$ for opening and closing files with the following two operations:

1. int `open`(): returns an unused file descriptor with value > 0
2. int `close`(int fd): returns `OK` on success, `EBADFD` if *fd* has not been previously opened, or `ECLOSEDFD` if *fd* has already been closed

To better understand $\mathscr{S}$, we look at a couple of examples of valid and invalid histories. Let α and β be thread IDs. The following histories would be valid:

$H_1 = [\texttt{open}()_\alpha, \overline{1}_\alpha, \texttt{open}()_\beta, \overline{2}_\beta, \texttt{close}(1)_\alpha, \overline{\texttt{OK}}_\alpha]$

$H_2 = [\texttt{open}()_\alpha, \overline{1}_\alpha, \texttt{close}(2)_\beta, \overline{\texttt{EBADFD}}_\beta]$

$H_3 = [\texttt{open}()_\alpha, \overline{1}_\alpha, \texttt{close}(1)_\alpha, \overline{\texttt{OK}}_\alpha, \texttt{close}(1)_\alpha, \overline{\texttt{ECLOSEDFD}}_\alpha]$

and the following histories would be invalid:

$H_1' = [\texttt{open}()_\alpha, \overline{1}_\alpha, \texttt{open}()_\beta, \overline{1}_\beta]$
(returns used FD)

$H_2' = [\texttt{open}()_\alpha, \overline{1}_\alpha, \texttt{close}(2)_\alpha, \overline{\texttt{OK}}_\alpha]$
(should return EBADFD)

$H_3' = [\texttt{open}()_\alpha, \overline{1}_\alpha, \texttt{close}(1)_\alpha, \overline{\texttt{OK}}_\alpha, \texttt{close}(1)_\alpha, \overline{\texttt{OK}}_\alpha]$
(should return ECLOSEDFD)

Note that all sequences of `open`() operations are SIM-commutative regions: reordering any number of `open`() operations satisfies $\mathscr{S}$, since the returned FDs are still unique and positive in value.

3.2 Reference Implementation

We now choose a simple reference implementation M that implements $\mathscr{S}$. The implementation has two pieces of global state, namely a counter *gc* initialized as 0 and a *closed* list initially empty. The two operations are implemented as follows:

1. int `open`(): increment *gc* and return the new value
2. int `close`(int fd): if $0 < fd \leq gc$ and $fd \notin closed$, then return `OK` and add *fd* to *closed*. Else if $fd \in closed$, return `ECLOSEDFD`, else return `EBADFD`

M satisfies $\mathscr{S}$: `open`() returns only unused, positive file descriptors since the counter never (disregarding overflows) repeats values. `close`(*fd*) returns `OK` if the file has been opened, since all files below the current value of *gc* must have been opened before, and `ECLOSEDFD` if the file has already been closed. Otherwise, *fd* is invalid and M returns `EBADFD`.

Although M satisfies $\mathscr{S}$, it is *not* scalable for SIM-commutative regions: all `open`() and `close`() operations access and write the shared, global counter and list.

We now have all the pieces to implement (and break) the proof construction from subsection 2.4.

3.3 (Incorrect) Proof Construction

We first choose a SIM-commutative region of $\mathscr{S}$. Since we know `open`() operations are SIM-commutative with each other, we generate the following history using M:

$$H_{commute} = [\texttt{open}()_\alpha, \overline{1}_\alpha, \texttt{open}()_\beta, \overline{2}_\beta]$$

This history is used to set the state of m_{rule}, initializing $s.h[*]$ as

$$s.h[\alpha] = [\text{COMMUTE}, \texttt{open}()_\alpha, \overline{1}_\alpha]$$
$$s.h[\beta] = [\text{COMMUTE}, \texttt{open}()_\beta, \overline{2}_\beta]$$

Now we execute m_{rule} on the following sequence of operations:

$$\texttt{open}()_\beta, \texttt{close}(1)_\alpha$$

Following the proof construction, $\texttt{open}()_\beta$ will first cause m_{rule} to switch to conflict-free mode for β (Figure 1, lines 3-4) because $s.h[\beta][0] = \text{COMMUTE}$. After line 4,

$$s.h[\beta] = [\texttt{open}()_\beta, \overline{2}_\beta]$$

Next, since the invocation $\texttt{open}()$ matches the first invocation by β in $s.h[\beta]$, m_{rule} will return the value 2 (lines 5-8). From line 17, the current state is now

$$s.h[\alpha] = [\text{COMMUTE}, \texttt{open}()_\alpha, \overline{1}_\alpha]$$
$$s.h[\beta] = []$$

The next invocation m_{rule} receives is $\texttt{close}(1)_\alpha$. This switches m_{rule} to conflict-free mode for α (lines 3-4), and sets the state to

$$s.h[\alpha] = [\texttt{open}()_\alpha, \overline{1}_\alpha]$$

m_{rule} cannot, however, replay $s.h[\alpha]$ as it did for β: $\texttt{close}(1)$ is not the next invocation contained in $s.h[\alpha]$, which is $\texttt{open}()$. This means m_{rule} diverges and enters emulate mode (lines 9-13). Our proof construction will now have to get M to a valid state consistent with the current history (line 10) so that we can feed M future invocations during emulation phase (as shown in line 15).

At this point, our recorded history is

$$H_{current} = [\texttt{open}()_\beta, \overline{2}_\beta]$$

There is only one possible reordering for this current history, and thus only one possible witness W, namely the single call $\texttt{open}()$. However, W is not consistent with the execution history $H_{current}$: the history generated by feeding W to M will not equal $H_{current}$, since calling $M.\texttt{open}()$ from a starting state returns 1 instead of 2. Furthermore, feeding W to M will not initialize M with a valid state! Invoking open() only once sets $gc = 1$, and the next future open() call will return 2, an invalid response. Thus, our proof construction fails both to find a witness consistent with the current execution history and to initialize M with a valid state.

One might suggest a potential fix for our proof construction: our construction could feed M *two* open() calls to bring it to a valid state. In other words, the construction does not need to find a witness consistent with our current execution history, but could feed M an arbitrary sequence of invocations (retrying as necessary) until M reaches a valid state, where all future responses will still be valid according to the spec. However, this will also not work. Note that $gc = 2$ and $closed = []$ after M is fed two calls to open(). Although M will never return a previously seen file descriptor for future calls to open(), we also need the next call to $\texttt{close}(1)_\alpha$ to return EBADFD to satisfy the spec (since fd1 has never been opened). But we only have two choices: either to feed $\texttt{close}(1)$ to M or to leave its state with $closed = []$. If we do the former, then $closed = [1]$ and M will return ECLOSEDFD because $1 \in closed$. If we do the latter, then M will return OK for the next $\texttt{close}(1)$ call because $1 \leq gc$! Thus, no matter what choice we make, M returns OK or ECLOSEDFD, and our history will not satisfies the spec:

$$H_{current} = [\texttt{open}()_\beta, \overline{2}_\beta, \texttt{close}(1)_\alpha, \overline{\texttt{OK}}_\alpha] \notin \mathscr{S}$$

and

$$H_{current} = [\texttt{open}()_\beta, \overline{2}_\beta, \texttt{close}(1)_\alpha, \overline{\texttt{ECLOSEDFD}}_\alpha] \notin \mathscr{S}$$

3.4 When might the proof fail?

The proof construction does not *always* fail. Note that M can reach a valid state if either *none* or *all* of the SIM-commutative region in question has been replayed. In other words, if m_{rule} diverges before or after the SIM-commutative region, then we can always get M to a valid state. For the former, when m_{rule} diverges before the SIM-commutative region, we can just feed M the recorded history's invocations sequentially in the order in which they occurred. For the latter, when m_{rule} diverges after the SIM-commutative region, we know there is at least one ordering of *all* operations in the SIM-commutative region that M can generate, namely $H_{commute}$. Thus, feeding M all possible orderings of all the operations in the SIM-commutative region until M reaches a valid state must eventually terminate.

Furthermore, if a prefix of the operations in the SIM-commutative region *with the same order as in* $H_{commute}$ occurs before divergence, then feeding M the operations in this order will also bring M to a valid state. This is because M generated $H_{commute}$ by being fed operations sequentially in this order.

The potential for failure arises only when a *prefix of a reordering* of the SIM-commutative region occurs before divergence, as demonstrated in our example. In this scenario, we do know not know if this reordered sequence of invocations and responses can ever be generated by M.

3.5 Why does the proof fail?

The key problem is that our reference implementation M is incapable of generating some histories required for SIM commutativity. That is, the reference implementation's *induced specification*—the set of histories that it can possibly generate—may be strictly smaller than the defined specification. When this gap arises, and regions that are SIM-commutative in the defined specification do not commute in the induced specification, then M may not be able to achieve a state consistent with the current history of our construction, as shown in our example.

On the other hand, the proof construction likely works whenever the given region SIM-commutes in the *induced* specification. Put another way, our construction likely works as long as "M, the reference implementation, produces the same results for any reordering of the commutative region" (quoted from the published version of the proof in Clements et al. [1]). Any prefix of a reordering of the SIM-commutative region would still SIM-commute in the induced specification because M produces the same result for any

ordering, and thus we can eliminate the failure case of our proof construction.

3.6 How to fix the proof?

We have seen that a bad reference implementation can prevent our proof technique from producing an implementation that scales within a given commutative region. We considered several fixes for this issue.

1. *Induced specification*. As we noted above, the proof construction will likely work if we restrict the rule to regions that are SIM-commutative in the induced specification, rather than those that are SIM-commutative in the defined specification.

2. *Specification oracle*. Alternately, we could remove the reference implementation from the proof entirely, and instead rely on a specification oracle that enumerates valid responses to invocations.

Induced specifications would preserve the somewhat "realistic" feel of the flawed proof, and the useful intuition that a scalable implementation can be obtained from a non-scalable implementation by logging and reconciliation. However, this is stricter than SIM commutativity, which places requirements on the specification, not the implementation. This prevents the SCR from, for example, informing programmers about potential areas to increase the scalability of their implementations.

Specification oracles feel less realistic than reference implementations, but they have the advantage of completely avoiding the issue of whether a given specification can be implemented at all. They also fit nicely into Coq. Because of these reasons, our machine-verified proof uses specification oracles.

4 The Verified Proof

This section describes our machine-verified proof of the SCR based on a specification oracle. The pseudocode for our proof construction m_{oracle} is shown in Figure 2.

4.1 Oracle Proof Construction

An oracle $O_{\mathscr{S}}$ is a function from a history H to $\{true, false\}$ defined as

$$O_{\mathscr{S}}(H) = \begin{cases} true & H \text{ satisfies } \mathscr{S} \\ false & H \text{ does not satisfy } \mathscr{S} \end{cases}$$

Given a specification $\mathscr{S}$, an oracle $O_{\mathscr{S}}$, and a specific history $H = X||Y$ where Y SIM-commutes in H, the proof constructs an implementation m_{oracle} that executes conflict-free within the SIM-commutative region Y.

m_{oracle} operates in three modes: *replay, conflict-free*, and *oracle* modes. Its state consists of three parts (divided into several sub-parts):

1. Copies of H as histories to replay:

```
1  m_oracle(s, i) ≡
2    t ← thread(i)
3    h_copy ← []
4    h_perf ← []
5    If s.mode ≠ ORACLE:
6      if s.X_copy = []:                    // enter conflict-free mode
7        s.mode ← CONFLICT-FREE
8        h_copy ← s.Y_copy[t];
9        h_perf ← s.Y_performed[t]
10     else:                                // still in replay mode
11       h_copy ← s.X_copy
12       h_perf ← s.X_performed
13     if head(h_copy) = i:
14       h_copy.pop()
15     if head(h_copy) is a response & thread(head(h_copy)) = t:
16       r ← head(h_copy)
17       h_perf.append((i, r))
18     else:                                // h.copy empty or input diverged
19       s.mode ← ORACLE
20   If s.mode = ORACLE:
21     for each possible response resp to invocation i:
22       H' ← a history consistent with performed actions
23       if O_S(H'++[(i, resp)]) = TRUE:
24         r ← resp
25         s.oracle_performed.append((i, r))
26         break
27   else if s.mode = CONFLICT-FREE:
28     s.Y_copy[t] ← tail(s.Y_copy[t])
29     s.Y_performed[t] ← h_perf
30   else:                                  // replay mode
31     s.X_copy ← tail(s.X_copy)
32     s.X_performed ← h_perf
33   Return ⟨s, r⟩
```

Figure 2: Verified constructed scalable implementation m_{oracle} for history H and reference implementation M.

- *s.X_copy*, a global list of actions initialized as X
- *s.Y_copy*$[t]$, a per-thread list of actions initialized as $Y|t$

2. Lists of performed actions:
   - *s.X_performed*, a global list of performed actions of X initialized as []
   - *s.Y_performed*$[t]$, a per-thread list of performed actions of $Y|t$ initialized as []
   - *s.oracle_performed*, a global list of performed actions in oracle mode initialized as []
3. *s.mode*, a global flag indicating the current mode of the machine

The replay and conflict-free modes act similar to the corresponding modes of m_{rule} from the prior proof. If m_{oracle} is not already in oracle mode, then m_{oracle} is in replay mode if *s.X_copy* is nonempty, or in conflict-free mode if *s.X_copy* is empty (line 6). If both *s.X_copy* and *s.Y_copy*$[t]$ for all t are empty, m_{oracle} switches to oracle mode (line 19).

In replay mode, if the next requested invocation matches the next invocation in *s.X_copy* and the next action in *s.X_copy* is a response to that invocation, m_{oracle} pops the head off of *s.X_copy*, returns the response, and appends the response to *s.X_performed* (lines 13-17). Otherwise, m_{oracle} has diverged from $X||Y$ and switches to oracle mode.

In conflict-free mode (set up in lines 7-9), if the next requested invocation by t matches the next invocation in *s.Y_copy*$[t]$ and the next action in *s.Y_copy*$[t]$ is a response to that invocation, m_{oracle} returns the response and appends the response to *s.Y_performed*$[t]$ (lines 13-17). Otherwise, the execution has diverged and m_{oracle} switches to oracle mode.

In oracle mode, the next response is determined by querying the oracle function. m_{oracle} iterates through all possible responses r to the invocation i and calls $O_{\mathscr{S}}(H'\text{++}[r])$, where H' is a history consistent with the performed actions in the history (*s.X_performed*, *s.Y_performed*$[t]$ for all t, and *s.oracle_performed*). If the oracle returns *true*, then m_{oracle} stops iterating, returns r, and the chosen response is appended to *s.oracle_performed* (lines 20-25).

More specifically, H' is constructed as

s.X_performed
++ *s.Y_performed*$[t_0]$ ++ ... ++ *s.Y_performed*$[t_{\#threads}]$
++ *s.oracle_performed*

Because of SIM commutativity, any ordering of operations in Y satisfies the spec. Thus, sequentially concatenating the *s.Y_performed*$[*]$ to construct H' generates a valid history.

Note that while in conflict-free mode, m_{oracle} executes in a scalable way: no thread accesses another's state. Thus, m_{oracle} should satisfy the SCR. In the next section, we describe how we verified this claim.

4.2 Coq Formalization

Here we give an overview of how we formalized the proof construction and proved its correctness in Coq. The complete Coq source is available at **https://github.com/tslilyai/coq_scr**.

4.2.1 Definitions

Our Coq model includes definitions for action histories (see Section 2), the machine state and modes as described above, conflict-freedom, SIM commutativity, and machine execution. These definitions are presented in the Appendix (Figure 3). Actions are tuples of $\langle threadID, op, response\rangle$, and histories are (reversed) lists of actions. We create an enum for modes, and use a record to encode state, where the record contains either thread-specific or global histories. Per-thread state is represented as a function from tid to history.

To define conflict-freedom, we need to define per-thread writes. We use two constructions: **diff_histories_tid_set** takes two histories and returns the set of threads whose per-thread histories (*s.Y_performed*$[t]$ or *s.Y_copy*$[t]$) change from the first history to the second. **diff_states_tid_set** takes two states and uses **diff_histories_tid_set** to return the set of threads that have had their per-thread state changed between the two states. Note that per-thread state in our construction changes only if per-thread history changes.

With these constructions, we define a conflict-free step of the machine taken on thread t as follows: let the prior state be s_1 and the consequent state be s_2. s_1 and s_2 must have equivalent values for any global machine state. Furthermore, their per-thread state must either be equivalent or **diff_states_tid_set**(s_1, s_2) must contain at most the single calling thread t.

Note that this definition of conflict-freedom is quite different than the one presented in by Clements et al. in the original SCR paper, which reasoned about conflicts in terms of memory access sets. Instead, our Coq definition is specialized for the mechanics of our proof construction, and allows us to reason on the much higher level of our construction's abstract per-thread state (e.g., *s.Y_copy*$[t]$) rather than individual memory accesses.

4.2.2 Theorem Statements

The final theorem and important lemmas are shown in the Appendix (Figure 4). Key helpers to prove these lemmas include determining the current mode and state of the machine, definitions for switching between modes when appropriate, and lemmas proving correctness of the machine when the machine is at each mode.

Our proof strategy was to first prove two lemmas, namely that the machine m_{oracle} is correct (generates only histories satisfying the spec), and that it is conflict-free during the SIM-commutative region Y. We used these lemmas to prove our final statement of the SCR:

1. Correctness: All histories achievable by m_{oracle} satisfy the spec and m_{oracle} never returns an invalid response

2. Conflict-freedom: Any step m_{oracle} takes in the SIM-commutative region Y is conflict-free

4.3 Proof Assumptions and Evaluation

Our Coq proof makes several assumptions, encoded as parameters or as part of the definition:

- The oracle can enumerate all possible responses. For our proof, we assume something stronger, namely that the number of responses is finite; finiteness was necessary to convince Coq that the oracle function terminates.
- For every invocation that is valid to call, there exists a valid response to return.
- The oracle is correct: $\forall x,\ O_{\mathscr{S}}(x) \iff x \in \mathscr{S}$

We made a number of decisions to ease the proof process. As we described earlier, we reason about conflict-freedom at a high level (without, for example, modeling memory arrays or low-level memory accesses as we did in our first attempt). This abstraction greatly simplified the proof process. We also switched to local reasoning, i.e., reasoning about steps of the machine, rather than proving facts about the machine's entire history, and we found that non-inductive definitions (for example, for reordered histories) made the proofs easier to handle. Finally, we proved all lemmas using the reverse of histories: because histories are defined as a list, the generated inductive cases were more intuitive (new actions are added to the head, not the tail, of the list).

The entire Coq development is 2056 LOC, of which 562 lines are state definitions, lemmas, and theorems. The effort took approximately 3 person-months, including several weeks stuck on verifying the original (incorrect) proof. We believe that without attempting to verify the published proof in Coq, the flaw in the proof may have been difficult to find.

5 Discussion and Conclusion

We have presented an initial proof for the SCR, an example and brief discussion about how it is incorrect, and a new, verified proof for the SCR. However, there remains much that is unsatisfying about the new proof. Most notably, the verified proof relies on the existence of a specification oracle, which causes it to stray even further from an imaginable construction than the original proof. We also assume enumerable responses, which may be practically true but fails to capture the intended semantics of any spec that semantically returns responses in, for example, $\mathbb{R}$. Furthermore, any construction that requires an oracle to iterate through an enumerable (but potentially infinite) number of responses is absurd in practice.

Problems in applying the SCR in practice are not restricted to the proof: the SCR statement itself may be unsuitable for conveying concrete information to implementers about how to design scalable systems in practice. For one, the SCR is a rule that applies only for a particular commutative region, rather than all commutative regions of a spec. If a spec has 100 commutative regions, then we know each commutative region has a implementation scalable *for that region*; however, we do not know if all 100 implementations are different, or if one implementation exists that will scale for all (or even multiple) commutative regions.

Furthermore, as illustrated by our oracle-based proof construction, scalability may not always be optimal: an implementation may scale but not necessarily be more performant.

In its current form, the SCR serves best as a hint that certain implementations can be made more scalable and a suggestion for areas for potential implementation optimization. For example, this use of the SCR served Clements et al. well in constructing sv6 [1]. It remains an open question about whether there is a way to extend the SCR, or modify the proof construction to aid implementers in designing practical, scalable systems. Perhaps there is a way to formulate the SCR to apply not only to one particular commutative region, but rather a class of commutative regions. Or perhaps a proof construction exists for a more restricted class of commutative systems that lends itself toward efficient and practical systems.

Many other questions remain, such as whether there are specs for which all practical implementations of the spec will not commute for the SIM-commutative regions of the spec. In other words, are there specs for which the only fully scalable implementation *must* have the equivalent of an oracle? We see the space between the commutativity of implementations of a spec, and the commutativity of the spec itself as a fruitful area to explore in future work.

6 Acknowledgments

This research was supported by NSF award CNS-1302359.

References

[1] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Transactions on Computer Systems (TOCS)*, 32(4):10, 2015.

[2] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages Systems*, 12(3):463–492, 1990.

```
Definition action : Type := tid * invocation * response.
Definition history : Type := list action.
Inductive mode : Type :=
  | Commute : mode
  | Oracle : mode
  | Replay : mode.

Record state := mkState {
    X_copy : history;
    Y_copy : tid -> history;
    X_performed : history;
    Y_performed : tid -> history;
    oracle_performed : history;
    md : mode
}.

Parameter sim_commutes :
    forall hd tl tl' Z,
      reordered (hd ++ tl) Y ->
      reordered tl' tl ->
      spec (Z++tl++X) ->
      spec (Z++tl'++X).

Section Conflict.
    Definition diff_histories_tid_set {A : Type} (ts1 ts2 : tid -> A) : Ensemble tid :=
        fun tid => ts1 tid <> ts2 tid.
    Definition diff_states_tid_set (s1 s2 : state) : Ensemble tid :=
        Union tid
            (diff_histories_tid_set s1.(Y_performed) s2.(Y_performed))
            (diff_histories_tid_set s1.(Y_copy) s2.(Y_copy)).
    Definition conflict_free_writes (t :tid) (s1 s2 : state) :=
        diff_states_tid_set s1 s2 = Singleton tid t /\
        s1.(md) = s2.(md) /\
        s1.(X_copy) = s2.(X_copy) /\
        s1.(X_performed) = s2.(X_performed) /\
        s1.(oracle_performed) = s2.(oracle_performed).
    Definition conflict_free_reads t i s :=
        forall (s1 s2 s1' s2': state) (a1 a2: action),
            s1.(Y_copy) t = s.(Y_copy) t ->
            s2.(Y_copy) t = s.(Y_copy) t ->
            s1.(Y_performed) t = s.(Y_performed) t ->
            s2.(Y_performed) t = s.(Y_performed) t ->
            s1.(md) = s.(md) ->
            s2.(md) = s.(md) ->
            machine_act s1 t i = (s1', a1) ->
            machine_act s2 t i = (s2', a2) ->
            a1 = a2 /\ s1'.(md) = s.(md) /\ s2'.(md) = s.(md).
End Conflict.

Definition machine_act (s : state) (t: tid) (i : invocation) : (state * action) :=
    let mode := next_mode s t i in
    match mode with
      | Oracle => get_oracle_response (state_with_md s Oracle) t i
      | Commute => get_commute_response (state_with_md s Commute) t i
      | Replay => match rev (s.(X_copy)) with
                  | [hd] => get_replay_response (state_with_md s Commute) t i

                  | _ => get_replay_response (state_with_md s Replay) t i
                  end
    end.
```

Figure 3: Definitions for proving the SCR

```
Lemma machine_correct :
    forall s h,
      generated s h ->
      spec h.

Lemma machine_conflict_free :
    forall s s' h t i r,
      generated s (h ++ X) ->
      spec ((t,i,NoResp) :: h ++ X) ->
      (exists h', reordered (h' ++ (t,i,r) :: h) Y) ->
      machine_act s t i = (s', (t,i,r)) ->
      conflict_free_step t s s'.

Theorem scalable_commutativity_rule :
    (* All achievable histories satisfy the spec and *)
    (* the machine never returns an invalid response *)
    (forall s h t i r,
        current_state_history s h ->
            spec h /\
            (List.In (t,i,r) h -> exists rtyp, r = Resp rtyp))
    (* If the machine's next step is in the middle of a (reordering of) *)
    (* a SIM-commutative region Y, then the machine's execution of the  *)
    (* step is conflict free.                                           *)
    /\ (forall s s' h t i r,
        current_state_history s (h ++ X) ->
        spec ((t,i,NoResp) :: h ++ X) ->
        (exists h', reordered (h' ++ (t,i,r) :: h) Y) ->
        machine_act s t i = (s', (t,i,r)) ->
            conflict_free_writes t s s'
            /\ conflict_free_reads t i s).
```

Figure 4: Theorems proven about the SCR in Coq