TSINGHUA UNIVERSITY

The Proof in Commuter

XINGGAO YANG

LAST EDIT:April 13, 2017

1 SI & SIM COMMUTE

Definition 1.1 (HISTORY)**.** History H is a sequence of invocations and responses of all threads.

Definition 1.2 (SPECIFICATION)**.** Specification $\mathscr{S}$ is a prefix-closed set of well-formed histories, distinguishes whether or not a history is "correct."
Accoring to the thesis, it is a set of pairs consist of invocations and responses, depending on the system and not concerned with how it is constructed. The only usage of it is to justify the pair in the history could exist, no matter what the other pair is.

Definition 1.3 (REORDER OF A SEQUENCE)**.** H' is a reordering of an action sequence, when for each thread, H and H' contain the same invocations and responses in the same order, but interleave threads differently.
That is, the quantum that commutes is the actions in different threads, and the prefix of the squence is the whole actions set.

Theorem 1.1 (SI-COMMUTE)**.** *Consider a history $H = X||Y$. Y SI-commutes in H when given any reordering of Y' of Y and any action sequence Z,*

$$X||Y||Z \in \mathscr{S} \quad \text{iff} \quad X||Y'||Z \in \mathscr{S}.$$

Thus both action squences Y and Y' set all the states to the same, making the order of Y indistinguishable from Z, the latter sequnces. **Notice here, that not every part of Y is sef-commute, even if the whole sequence commutes**.

Theorem 1.2 (SIM-COMMUTE)**.** *An action sequence Y SIM-commutes in history $H = X||Y$ when for any prefix P of any reordering of Y (including $P = Y$), P SI-commutes in $X||P$.*

This is "more strict" than "SI-Commute", that we can regard sequence Y as a set, not only all the reorder of actions lead to the same state, but all the sub-sequences from the beginning of any reorder of Y self-commute.

2 IMPLEMENTATION

Definition 2.1 (IMPLEMENTATION)**.** An implementation m is a funtion in $S \times I \longmapsto S \times R$. Given an old state and an invocation, the implementation produces a new state and a response. And express it as:

$$m(s_{i-1}, invo_i) =< s_i, resp_i >$$

- It is just one step, which invoke an invocation to generate a new state.
- I regard YIELD here as asynchronous, that hang up the thread, waiting the OS to finish its job and put the thread to the ready state. YIELD and CONTINUE here is representation of thread switch.
- It is the only mathmatic expression in the paper.
- The history is the overall implementation sequence with invocations and responses write together, exclude the CONTINUEs and YIELDs.

Definition 2.2 (access conflict)**.** Two implementation steps have an access conflict when they are on different threads and one writes a state component that the other either writes or reads. A set of implementation steps is conflict-free when no pair of steps in the set has an access conflict.

Theorem 2.1 (RULE)**.** *Assume a specification $\mathscr{S}$ with a correct reference implementation M. Consider a history $H = X||Y$ where Y SIM-commutes in H, and where M can generate H. then there exists a correct implememtation m of $\mathscr{S}$ whose steps in the Y region of H are conflict-free.*

```
m_ns(s, a) ≡
  If head(s.h) = a:
    r ← CONTINUE
  else if a = YIELD and head(s.h) is a response
          and thread(head(s.h)) = thread(a):
    r ← head(s.h)                                   // replay s.h
  else if s.h ≠ EMULATE:                            // H complete or input diverged
    H′ ← an invocation sequence consistent with s.h
    For each invocation x in H′:
      ⟨s.refstate, _⟩ ← M(s.refstate, x)
    s.h ← EMULATE                                   // switch to emulation mode
  If s.h = EMULATE:
    ⟨s.refstate, r⟩ ← M(s.refstate, a)
  else:                                             // replay mode
    s.h ← tail(s.h)
  Return ⟨s, r⟩
```

Figure 4-1: Constructed *non-scalable* implementation m_{ns} for history H and reference implementation M.

(a) non-scalable

```
m(s, a) ≡
  t ← thread(a)
  If head(s.h[t]) = COMMUTE:                        // enter conflict-free mode
    s.commute[t] ← TRUE; s.h[t] ← tail(s.h[t])
  If head(s.h[t]) = a:
    r ← CONTINUE
  else if a = YIELD and head(s.h[t]) is a response
          and thread(head(s.h[t])) = t:
    r ← head(s.h[t])                                // replay s.h
  else if s.h[t] ≠ EMULATE:                         // H complete/input diverged
    H′ ← an invocation sequence consistent with s.h[∗]
    For each invocation x in H′:
      ⟨s.refstate, _⟩ ← M(s.refstate, x)
    s.h[u] ← EMULATE for each thread u
  If s.h[t] = EMULATE:
    ⟨s.refstate, r⟩ ← M(s.refstate, a)
  else if s.commute[t]:                             // conflict-free mode
    s.h[t] ← tail(s.h[t])
  else:                                             // replay mode
    s.h[u] ← tail(s.h[u]) for each thread u
  Return ⟨s, r⟩
```

Figure 4-2: Constructed scalable implementation m for history H and reference implementation M.

(b) scalable

Figure 1: Generate an implementation

3 PROOF

Here the author construct a procedure, that the implementation with a new state and new response that are conflict free, and the implementation is also correct in the specification. Some expresion is needed additional guess or explain:

- **head() & tail()** : Get the ket or bra of the expression:

  $$m(s_{i-1}, invo_i) =< s_i, resp_i >$$

  And thus, head$(s_{i-1}) = resp_i$ and tail$(s_{i-1}) = s_i$

- **thread()**: Get the thread of an invocation or response.

3.1 NON_SCALABLE IMPLEMENTATION

There are three branches in the beginning, and I will comment it separately.

1. Notice that there's no different threads here
2. And here comes a question, as a shows up as the second parameter of m_{ns} how could it be YIELD.
3. The propose here is to run the implementation backwards, to find the location that H complere or input diverged, and get the s.refstate and new response. Meanwhile, set s.h to EMULATE.

3.2 SCALABLE IMPLEMENTATION