TSINGHUA UNIVERSITY

The Proof in Commuter

XINGGAO YANG

LAST EDIT:April 25, 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:

$$s.h = \{inv_1, res_1, inv_2, res_2, ...\}$$

$$head(s.h) = s.h.get\_the\_head()$$

$$tail(s.h) = s.h.remove(head(s.h))$$

  And thus, in each procedure, m_{replay} gets the head of s.h and remove it after that.

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

3.1 CONSTRUCTIVE IMPLEMENTATION

1. Now here, I need to emphasis that it is a "constructive" proof, thus, the state is changed: it is not just the state set $s.refstate$ needed by the reference implement but alao the portion of history by $s.h$, which is a list of invocation and responce with yield or continue removed. In a word, the state needed or modified by the constructed method contains two conponent: the history $s.h$ hold the portion of H to be replayed and the reference state $s.refstate$ to record the state of the **implement**.

2. All the stuff we know is that: The history $H = X||Y$ constructed by the specification with the invocation and response already known; the reference implementation. Notice that the history is irrelevant with the implementation, thus the state is not specified. All the implementation needed to satisfy is the response! For example, in the invocation part of history$\{inc(), inc(), dec(), iszero()\}$ with specification set $\{inc, dec, iszero\}$, , there is no counter now, the implement just need to response $\{1, 2, 1, false\}$ or some values else depending on the initial state defined by the reference. It is the responce that is indistinguishable, but not the states defined by the reference implementation.

3. Hence, let's start the procedure of the method:

   - In the former condition statement, the new invocation(response) is checked: whether it is a invocation(responce) the same as the history, or a diverged one. As I mentioned above, the responce value is all that the user who invokes the API able to acknowledge, hence the method simply returns the value defined in the specification, **the reference state is not necessary to be modified here.**

- When the invocation diverges, the history gets invalid, we don't know the responce and an emulation is needed. However, in the former method just finished the replay job, the state of the implementation hadn't been modified. As it is noticed, the responce stuff is the only information the user notified by the API, and the API needn't to modify the state inside of it if it knows how to responce with the history well defined by the specification. However, the API lost the "guidance" as soon as it met a diverged invocation. This time, the API is obliged to modify the state from the beginning of the history with the history "H'" hold the invocations from the start to the last invocation before $s.h$ of the history "H". And then, execute the diverged invocation, modify the state and return the emulate responce generated by the reference implementation. When the invocation diverges, the portion of history hold by $s.h$ becomes useless, and is discard from here.
- Although, the responces are the same with the history(in the replay mode) or correct(defined by the specification), the method are not commute: in the replay mod, two methods may modify the component $s.h$ the same time causing conflict. Thus the history are distinguished by the threads, and in the replay modes, the method will modify its own history, without confliction.
- Here, some may ask two questions:
  (a) Where is "commute"? I mentioned nothing related to condition "commute". In the conflict-free method, the history is divided into different thread, thus, the invocations are executed in the sequence the same with the history of the thread, but may interleave with each other in different threads, which are commute.
  (b) The reference state is still shared, what if it conflicts? The answer is intuitive, the only circumstances we need to modify it, is when the invocation diverges from the history. In the history, which is replay mode, we just modify the history $s.h[t]$ which is conflict free. And when the invocation diverges, the history is useless, and we cannot guarantee the "commute" feature anymore, and method may cause conflict when it modify the reference state of course depending on the reference implementation. The constructive proof only guarantee conflict-free in the history part, not in the diverged part.