**Summary of Optimal Stateless Model Checking for Reads-From Equivalence under Sequential Consistency**

Authors: Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson,Magnus Lång,Tuan Phong Ngo, Konstantinos Sagonas, Konstantinos Sagonas

Publication: Paper published in Proceedings of the ACM on Programming Languages, October 2019 Article No.: 150
https://doi.org/10.1145/3360576

Review by: Tuppe Omkar Vijaykumar

Authors present a novel approach for stateless model checking(SMC) of multi-threaded programs under sequential consistency based on reads-from(rf) equivalence. rf equivalence, which is coarser than Mazurkiewicz traces, respects order between events of each individual thread and also between events different threads. Authors claim that presented SMC algorithm is both optimal and efficient. Optimality is ensured by constructing only one execution in each equivalence class and efficiency follows from the point that algorithm spends at most polynomial time per equivalence class. Given SMC algorithm is inspired from SMC algorithm under Release-Acquire (RA) semantics.

Given algorithm presents new test for consistency of a rf trace, which is divided into three phases (i) *Saturation phase*: Assigns ordering based on program order and reads-from relation (ii) *Witness construction*: Orders various write events under sequential consistency. (iii) *Decision Procedure*: Decision procedure which ensures whether given trace is consistent or not.

Authors define execution of a statement in program execution as a event.A write event is a tuple $e = \langle\, id, t, W, x, v \rangle$ where $id \in \mathbb{N}$ is an event identifier, t $\in \mathbb{T}$is a thread,x $\in \mathbb{X}$ is a variable, v $\in \mathbb{V}$ is a value. And a read event is tuple $e = \langle\, id, t, R, x, e' \rangle$, where $e'$is write event or initializer event. A trace $\tau$ is set of events from each thread $t$ from some run of $t$ such that source of every read-event $e_R$ is in $\tau$. A linearization of a trace $\tau$ is a ordering of events of $\tau$, where each event of a thread are totally-ordered by their event identifier and events of different threads are unordered. An execution $\mathbb{E}$ is an linearization of trace $\tau$ where each read-event reads from last preceding write event.

For a given trace $\tau$, for two events reads-from rf relation holds when read event $e$ reads from write event $e'$ and the relation is denoted as $e'$ [rf] $e$ i.e $e$ reads from $e'$. And relation $e$ $[\leq_r \cap$ rf$]$ $e'$ says that event $e$ *happens-before* event $e'$

Now, let us look at given SMC algorithm, ReadsFrom-SMC.

**Algorithm 1:** ReadsFrom-SMC

1 *ReadsFrom-SMC* $(\tau, E)$:

2 **extend** $E$ to complete execution $E \cdot \hat{E}$ where each event of $\hat{E}$ is unmarked

3 $\tau' := \tau \cdot \hat{E}$

4 **foreach** *read event* $e_R \in \hat{E}$ **do**

5 *schedules(pre(*$\tau'$*,*$e_R$*))*

6 **foreach** $e_R, e_W \in \tau' : e_W.var = e_R.var$ *and* $e_W \neq e_R.src$ *and* $(e_R \in \hat{E}$ *or* $e_W \in \hat{E})$ *and* $unmarked(e_R)$ *and* $MayRead(\tau', e_R, e_W)$ **do**

7 $\tau'' := pre(\tau', e_R)$

8 $\pi := predecs(\tau', e_W) \cap post(\tau', e_R)$

9 $\sigma := e_R[src := e_W] \cdot mark(\pi)$

10 $E'' :=$ GetWitness$(\tau'' \cdot \sigma, E, \hat{E})$

11 **if** $E'' \neq \langle\rangle$ *and* $\neg\exists\langle\sigma', -\rangle \in schedules(\tau'') : \sigma' \equiv \sigma$ **then**

12 **add** $\langle\sigma, E''\rangle$ to $schedules(\tau'')$

13 **foreach** *read event* $e_R \in \hat{E}$ *starting from the end* **do**

14 $\tau'' := pre(\tau', e_R)$

15 **foreach** $\langle\sigma, E''\rangle \in schedules(\tau'')$ **do**

16 ReadsFrom-SMC$(\tau'' \cdot \sigma, E'')$

17 **erase** $schedules(\tau'')$

SMC Algorithm maintains a set named *schedules(*$\tau''$*)*, and it's each item is of form $\langle\sigma.E\rangle$, where $\sigma$ is consistent extension of trace $\tau''$, and witness(feasible execution of given trace) $E$ for extension $\sigma$. Algorithm requires consistent trace $\tau$ and corresponding execution $E$ as parameters. Then it extends, this trace $\tau$ to consistent and complete trace $\tau'$ by concatenating $\tau$ feasible execution $\hat{E}$, where $E \cdot \hat{E}$ is arbitrary complete execution. Then algorithm tries to generate new traces by changing the source of read events present in the trace. It considers only those read events, which are unmarked i.e this read event is not explored in earlier calls to ReadsFrom-SMC algorithm. Line (6) looks for such feasible read and write events, where either read event $e_R$ or write event $e_W$ appear in $\hat{E}$. Then, for such every feasible $(e_R, e_W)$ pair it tries to generate new consistent trace $(\tau'' \cdot \sigma)$, by constructing a execution under SC(sequential consistency) as witness to trace $(\tau'' \cdot \sigma)$.

Here trace $\tau''$ is prefix of trace $\tau'$ upto event $e_R$, excluding event $e_R$. And $\sigma$ $e_R$ concatenated with minimal cut(smallest trace) of event $e_W$ in trace $\tau'$, including $e_W$.Then algorithm calls itself recursively with these newly generated traces and their witnesses. Initially, ReadFrom-SMC is called with empty trace(empty trace is consistent) and empty execution.**So**, by construction itself algorithm satisfies soundness property, *i.e* each expored trace is consistent trace.

**Algorithm 2:** ConstructWitness.

*ConstructWitness* $(\tau, E)$:

  shb := Saturate($\leq_r \cup$ rf) where $\leq_r$ and rf are extracted from $\tau$

  cohb := shb

  **for** *write events* $e_W, e'_W \in \tau : e_W.var = e'_w.var$ *and* $(e_W, e'_W) \notin$ *cohb* *and* $(e'_w, e_W) \notin$ *cohb* **do**

    **if** $e_W, e'_w \in E$ **then**

      cohb := cohb $\cup$ $(e_W, e'_W)$

    **else**

      cohb := cohb $\cup$ $(e'_W, e_W)$

    cohb := Saturate(cohb)

  $E'$ := TopologicalSort(cohb)

  **return** $\langle$ "consistent", $E'\rangle$

Now let us look into the algorithm GetWitness, which tries to generate witness for new trace and verifies whether given trace $\tau$ is consistent or not. GetWitness($\tau$,$E$) has three phases:

**Phase 1.Saturation Phase:** This phase orders events of trace based on program order and reads-from relation and by following two rules.

Rule 1. If $e_W$ [shb] $e_R$ and $e'_W$ [rf] $e_R$, then $e_W$ [shb] $e'_W$.

Rule 2. If $e'_W$ [shb] $e_W$ and $e'_W$ [rf] $e_R$, then $e_R$ [shb] $e_W$.

Here, shb, read as *saturated-happens-before*, is transitive relation which extends relation [$\leq_r \cap$ rf]. Then, this phase checks for cycles in the relation shb. If cycle is present, it returns inconsistent. If no cycles, then it moves to next phase.

**Phase 2. Witness Construction Phase:** This phase constructs new transitive relation cohb by extending shb and orders each write event corresponding to same variable based on order in which they appear in provided execution $E$, as shown in Algorithm 2. Then, this phase performs Topological Sort on this cohb relation and generates feasible execution as witness for given trace $\tau$. If this phase fails to generate witness, then it moves to next phase.

**Phase 3 Decision Procedure:** This phase tries to generate the witness for given trace $\tau$ by following brute-force like algorithm, which is polynomial in time of program size, bur exponential in size of number of threads. This phase generates a graph, where each vertex is mapping of threads to one of its events, hence exponential number vertices. Then, it adds to special vertices $v_init$ and $v_target$, where $v_init$ means start of each thread *i.e* it maps to event with $id$=1 and $v_target$ denotes termination of all threads. Then, it add edges between those vertices which represents events from some thread which will be executed sequentially. Then, it looks for path from $v_init$ to $v_target$. If such path exists, then trace $\tau$ is consistent and this path is witness. If no such path exists, then declares that trace is inconsistent.

Now, let us try to simulate a example on this algorithm. Consider program with three threads as shown in Figure 1.

First call of SMC alogirthm will be with empty trace and execution.

*i.e* ReadFrom-SMC($\langle\rangle, \langle\rangle$). Then we extend the execution by concatenation $\hat{E}$ := y=2, b=2, y=3, a=0, x=4. So, generated new consistent and complete trace (line 3 of algorithm 1) $\tau'$ := y=2, b=2, y=3, a=0, x=4. Then, we look for different write event sources for all read events in trace $\tau'$, and these pairs are $(b = 2, y = 3)$, $(a = 0, x = 4, )$. Next, we try to get witness for new trace for $(b = 2, y = 3)$. This, new trace(line 7-10) will be y=2, b=2, y=3, and we call GetWitness($\tau$, $E$), where $\tau := \langle y = 2, b = 2, y = 3\rangle$ and $E := \langle y = 2, b = 2, y = 3, a = 0, x = 4\rangle$.
This call to GetWitness($\tau$, $E$), will generate relation as shown in Figure 2.

| | | |
|---|---|---|
| ***y = 2*** | ***y = 3*** | ***x = 4*** |
| ***b = y*** | ***a = x*** | |

Figure 1: simple program

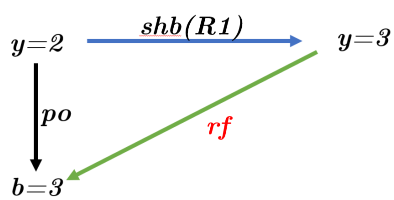


Figure 2: cohb relation for

Then, we perform TopologicalSort on this cohb relation and construct the witness, $E := y = 2, b = 2, y = 3, a = 0, x = 4$. Next we add this witness to $schedules(\tau'' := y = 2, b = 2, y = 3)$. Same steps are followed for other pair $(a = 0, x = 4, )$. Then, we recursively call RedsFrom-SMC() for to this newly generated traces. After, exploring all traces by this SMC algorithm, we get traces as shown in Figure 3.

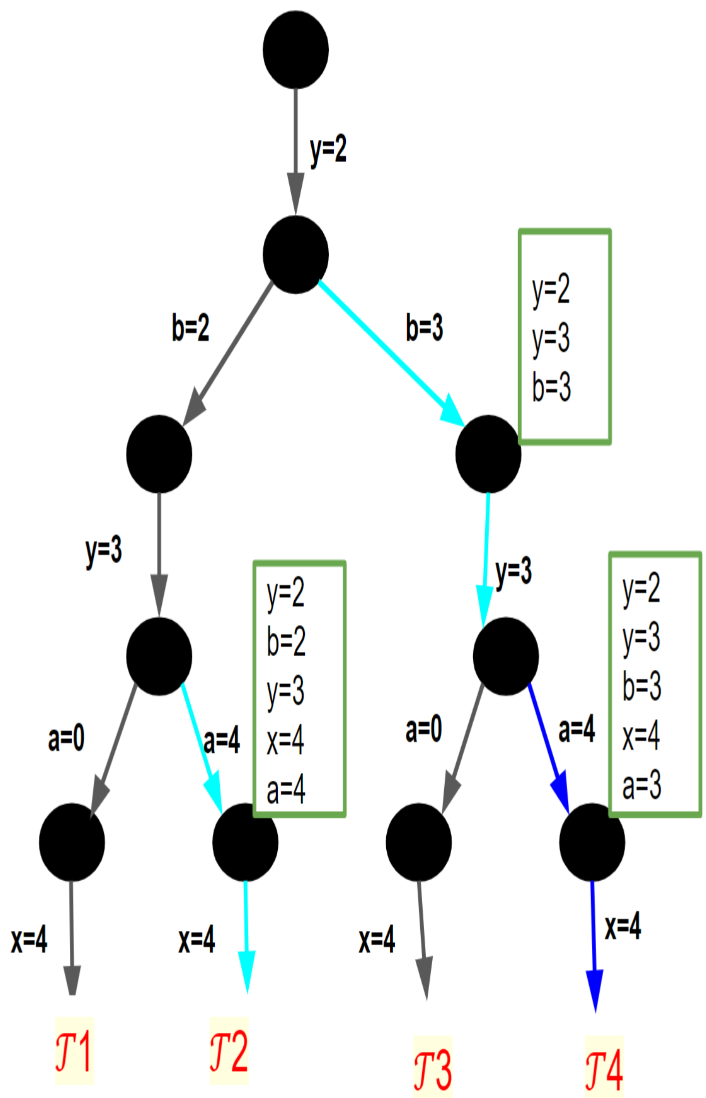


Figure 3: Four different complete traces generated by algorithm. Green box shows content of *schedules() for corresponding read event*