Summary of Optimal Stateless Model Checking under the Release-Acquire Semantics

Author: PAROSH AZIZ ABDULLA, MOHAMED FAOUZI ATIG, BENGT JONSSON, TUAN PHONG NGO

Publication: Proceedings of the ACM on Programming Languages October 2018 Article No.: 135
https://doi.org/10.1145/3276505

Authors present stateless model checking(SMC) algorithm for concurrent programs under Release-Acquire (RA) semantics based on the program order and read-from relation. Optimality of algorithm is achieved by exploring each program order and read-from relation exactly once. Optimal partial order reduction(DPOR) technique is applied to proposed SMC algorithm to reduce the number of explored executions. DPOR treats two executions equivalent if they are associated with same *Shasha-Snir trace*, and hence associates exactly one execution to the generated trace. *Shasha-Snir trace* considers three relations between events; (i) program order (po), (ii) coherence order (co), (iii) read-from relation (rf).

Presented DPOR algorithm tries to further reduce the number of traces by following weaker equivalence based on only (po) and (rf) relations. As algorithm is based on the *weak traces*, it needs a way to extend existing *weak trace* to some RA-consistent trace. This is achieved by *saturation* operation, which extends given *weak trace* to some corresponding *Shasha-Snir trace* by adding required coherence edges. Algorithm generates new RA-consistent weak traces from explored consistent *weak trace* by considering feasible alternative write events for every read events while maintaining consistency.

A trace τ is tuple $\langle E, po, rf, co\rangle$, where is E is set of events including initializer events, and po, rf, co are binary relations on E. $e[po]e'$ means $e.thread = e'.thread$ and $e.id < e'.id$. co relation is union of coherence order of each variable, i.e. $\text{co} = \cup_{x\in\mathbb{X}} \text{co}^x$. If e_w $[\text{co}^x]$ e'_w then event e_w happens before event e'_w. po relation totally orders events of individual threads. Each co^x may or may not be totally ordered. We say that a trace $\tau = \langle E, po, rf, co\rangle$ is totally ordered if each co^x relation is strict total order. And total trace τ is a *Shasha-Snir trace*. A transition from configuration γ to configuration γ' is denoted by $\gamma \overset{l}{\rightsquigarrow} \gamma'$, where label l is one of three forms: (i) indicates a operation performed by thread which updates local state of thread, (ii) $\langle th, W, x, v\rangle$, thread th writes value v to variable x and updates program counter, and (iii) $\langle th, R, x, v\rangle$, thread th reads value v from variable x in some register and updates program counter. A run p from configuration γ is given by $\gamma_0 \overset{l_1}{\rightsquigarrow} \gamma_2 \overset{l_1}{\rightsquigarrow} ... \overset{l_n}{\rightsquigarrow} \gamma_n$, where $\gamma_0 = \gamma$, and γ_n indicates that all threads are terminated. A run p can be associated with a trace τ, where each non-initializer event of trace τ correspond to exactly one label of run p. Set of runs follow Release-Acquire semantics if their associated total traces do not from cycles over relation $\text{po} \cup \text{rf} \cup \text{co}^x \cup \text{fr}^x$, for each variable $x \in \mathbb{X}$. Relation e $[\text{fr}^x]$ e' holds if value read by e is overwritten by e'.

Let $[[\gamma]]^{Total}_{RA}$ be the set of total RA-consistent traces generated from configuration γ. DPOR algorithm geneates weak traces, where a *weak trace* τ is defined as a tuple $\langle E, po, rf, \phi\rangle$. So weak trace is obtained from a trace by removing coherence edges. A weak trace τ is RA-consistent if it can be obtained by a RA-consistent total trace by removing its coherence edges. We define set of RA-consistent weak traces from configuration γ as, $[[\gamma]]^{weak}_{RA} := \left\{ weak(\tau) \mid \tau \in [[\gamma]]^{Total}_{RA} \right\}$. Given DPOR algorithm explores every *weak trace* in $[[\gamma]]^{weak}_{RA}$.

To ensure consistency of generated weak traces a new semantics called *saturated semantics* is followed by the algorithm. Saturated trace is a extension of a weak trace obtained by adding coherence edges which occur in some RA-consistent extension of given weak trace. Saturated semantics allows adding new read or write event to a trace without violating consistency. A trace τ is *saturated*, if for any two write events e_W, e'_W and a read event e_R on same variable x, relations e_W $[\text{po} \cup \text{rf} \cup \text{co}^x]^+$ e_R and e'_W $[\text{rf}^x]$ e_R hold, then e_W $[\text{po} \cup \text{rf} \cup \text{co}^x]^+$ e'_W. Adding a new write event e to a saturated trace(called as S-write operation) is easy, we just need to ensure that e is last event in program order of its thread. Adding a new read event(called S-Read operation) to a saturated is more involved, so two sets called *readable events* and *visible events* are defined to ensure consistency.

Set of readable events $\mathcal{R}$ for a thread th and a variable x is set of of all write events e on

variable x such that there is no write event e' on variable x and a event $e'' \in th$, satisfying relation e [po $\cup$ rf $\cup$ co^x]$^+$ e' and e' [po $\cup$ rf]* e''. So, $\mathcal{R}$ is set of events from which a read event of thread th can obtain its value without violating RA-semantics. For a thread th and a variable x *visible events* $\mathcal{V}$ is set of events $e \in \mathcal{R}$ that precedes some event e' belonging to thread th, i.e. e [po $\cup$ rf]* e'. In other words, $\mathcal{V}$ gives set of events from which coherence edges should be added to write event e if new read event reads from e. Hence, while adding a new read event e_R to a saturated trace it must be ensured that: (i) e is last event in program order of its thread, (ii) e reads from $e_W \in \mathcal{R}$, and (iii) coherence edges should be added from $e'_W \in \mathcal{V}$ to e_W. Authors prove that obtaining readability set and visibility set can be done in polynomial time, so it does add any overhead. Now, let us look at DPOR algorithm DfVisit based on depth-first exploration.

Procedure DfVisit(γ, τ, π) generates all RA-consistent weak traces from given configuration γ, saturated trace τ, and sequence of observation π. Initially, τ and π are empty. Swapping of read events and their corresponding postponed write events is done by following observation sequence π.This Procedure checks if there exists a write event $\langle th, E, x, v$ which can be added to trace τ under saturated semantics. If such write event is available, then procedure adds this event e to τ to obtain τ' and configuration γ'. Then it invokes itself recursively with parameters $(\gamma', \tau', \pi \cdot e)$. After returning from this call, it calls procedure DeclarPostponed$(\tau', \pi \cdot e)$ which finds read event $e_R \in \pi$ which can read from e. If no such write event present, then procedure looks for a new read event e. If there exists such read event procedure allows read event e to read from every possible write event e' to obtain configuration γ' and trace τ'. For every such $\langle e, e' \rangle$ pair procedure invokes itself with parameters $(\gamma', \tau', \pi \cdot \langle e, e' \rangle)$. Then, for all generated possible schedules of read event e procedure RunSchedule is called to allow e read from respective consistent write events.

Algorithm 1: DfVsit(γ, τ, π)

Input: γ is a configuration, $\tau = \langle E$, po, rf, co $\rangle$ is trace, π is an observation sequence

if $\exists \gamma \xrightarrow{\langle th, W, x, v \rangle} \gamma'$ **then**
  let e be $\left\langle |E^{th}| + 1, th, W, x, V \right\rangle$ and τ' such that $\left\langle \gamma, \tau \right\rangle \xrightarrow{e}_S \left\langle \gamma', \tau' \right\rangle$
  DfVisit$(\gamma', \tau', \pi \cdot e)$
  DeclarPostponed$(\tau', \pi \cdot e)$
else if $\exists \gamma \xrightarrow{\langle th, R, x, v \rangle} *$ **then**
  let e be $\left\langle |E^{th}| + 1, th, R, x, V \right\rangle$
  Schedules$(e) \leftarrow \varnothing$;Swappable$(e) \leftarrow true$
  for $e', \gamma', \tau' : \langle \gamma, \tau \rangle \xrightarrow{\langle e, e' \rangle} \langle \gamma', \tau' \rangle$ **do**
    DfVisit$(\gamma', \tau', \pi \cdot \langle e, e' \rangle)$
  for $\beta \in$ *Schedules*(e) **do**
    RunSchedule$(\gamma, \tau, \pi, \beta)$

Algorithm 2: DeclarPostponed(τ, π)

Input: $\tau = \langle E$, po, rf, co $\rangle$ is trace and π is an explored observation sequence

let e be *last(π)* and x be $e.var$
for $i \leftarrow |\pi| - 1$ *to 1* **do**
  let e' be $\pi[i].event$
  if $e' \in E^{R,x} \wedge \neg(e'[po \cup rf]^+ e) \wedge$ *Swappable* (e') **then**
    $\beta \leftarrow \epsilon$
    for $j \leftarrow i + 1$ *to* $|\pi|$ - *1* **do**
      let e'' be $\pi[j].event$
      if $(e''[po \cup rf]^+ e)$ **then**
        $\beta \leftarrow \beta \cdot \pi[j]$
    if $\nexists \beta' \in Schedules(e').\beta' \approx \beta \cdot e \cdot \langle e', e \rangle$ **then**
      Schedules$(e') \leftarrow$ Schedules$(e') \cup \beta \cdot e \cdot \langle e', e \rangle$
    break

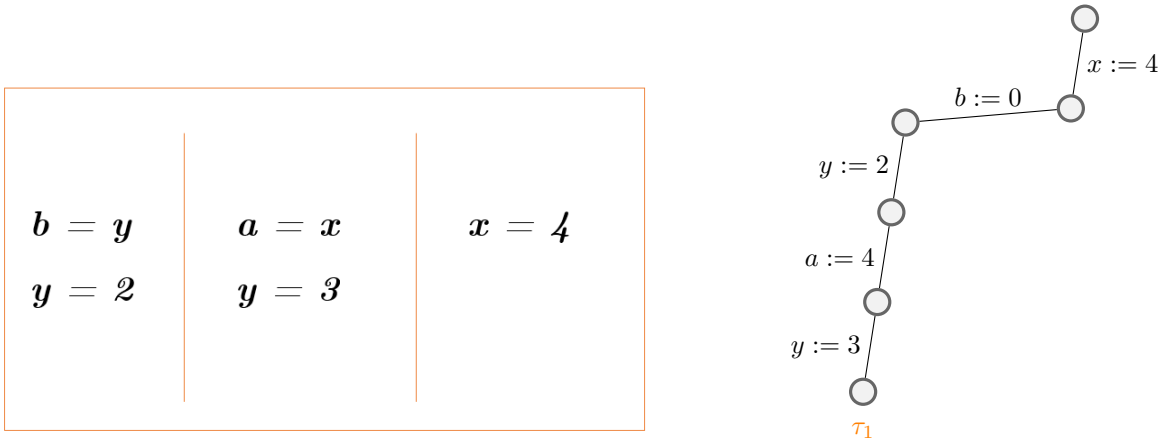
Here procedure DeclarPostponed finds closest read event e' to the last event e of sequence π,

Algorithm 3: RunSchedule$(\gamma, \tau, \pi, \beta)$

Input: γ is a configuration, τ is trace and π is an explored observation sequence, and β is a schedule

```
1 if β ≠ ϵ then
2     let present β in the form of α · β′ and e be α.event
3     choose γ′, τ′ : ⟨γ, τ⟩ --α-->_S ⟨γ′, τ′⟩
4     if e.type = R then
5         Swappable(e) ← false
6     RunSchedule(γ′, τ′, π · α, β′)
7 else
8     DfVsit(γ, τ, π)
```

which can read from e. This event e' should not precede event e and it must be swappable. Once such event e' is found, procedure creates a schedule β for event e', by adding all events after e' in π which precede e in $[\text{po} \cup \text{rf}]^+$. Then adds this schedule β to Schedules(e'). RunSchedule procedure visits each event in schedule β one-by-one and invokes itself recursively. Then, it calls DfVsit(γ, τ, π) procedure for further exploration of trace. Now, let us see an example to illustrate the DPOR algorithm. Consider program with three threads shown in figure 1 (a).



(a) (b)

Figure 1: (a) Program $\mathcal{P}$ with three threads, (b) Trace τ_1 of $\mathcal{P}$

Initially DfVsit(γ, τ, π) is called with empty trace and observation sequence. Then algorithm looks for a write event, here event $x := 4$ and performs S-write. Then recursively calls itself, with $\tau' = \{x := 4\}$. Then, again we look for write event, as there is no write event in program order, so algorithm picks a read event say $b := y$ and allows it to reads from every possible write event, first consider that $b := y$ reads from $init_y$. Then recursively calls itself. And, this continues until we get a complete trace τ_1 as shown in figure 1 (b). After visiting $y := 3$, DeclarPostponed() will be called with $\tau' = \pi = \{x := 4, b := 0, y := 2, a := 4, y := 3\}$.

In DeclarPostponed we look for read event which can read from postponed event $y := 3$, i.e event $b := 0$. Then schedule $\beta_1 = \{a := 4, y := 3, b := 3\}$ will be created for $b := 0$ based on saturated semantics for trace τ_1 shown in figure 2(a), and then β_1 will be added to Schedules($b := y$). Now, DfVsit() will be invoked again for $a := 0$ (line 8 DfVsit), and after it event $y := 3$ will be visited again. So, now we obtained trace τ_2, shown in figure 2(b). For trace τ_2 for $b := y$, we create schedule $\beta_2 = \{a := 0, y := 3, b := 3\}$. Then for event $y := 2$, again DeclarPostponed is invoked. This call will not create a schedule for $b := y$ as $b := y$ precedes $y := 2$ in program order. Then call to RunSchedule() is made for read event $b := y$.

In RunSchedule() we generate new traces τ_3 and τ_4 from schedules β_1, and β_2 respectively, shown in figure 3. After, exploring τ_4, new traces will not be generated, as we covered all possible read event, write event pairs under RA-semantics. Hence, given DPOR algorithm will generate 4 RA-consistent weak traces for program $\mathcal{P}$.

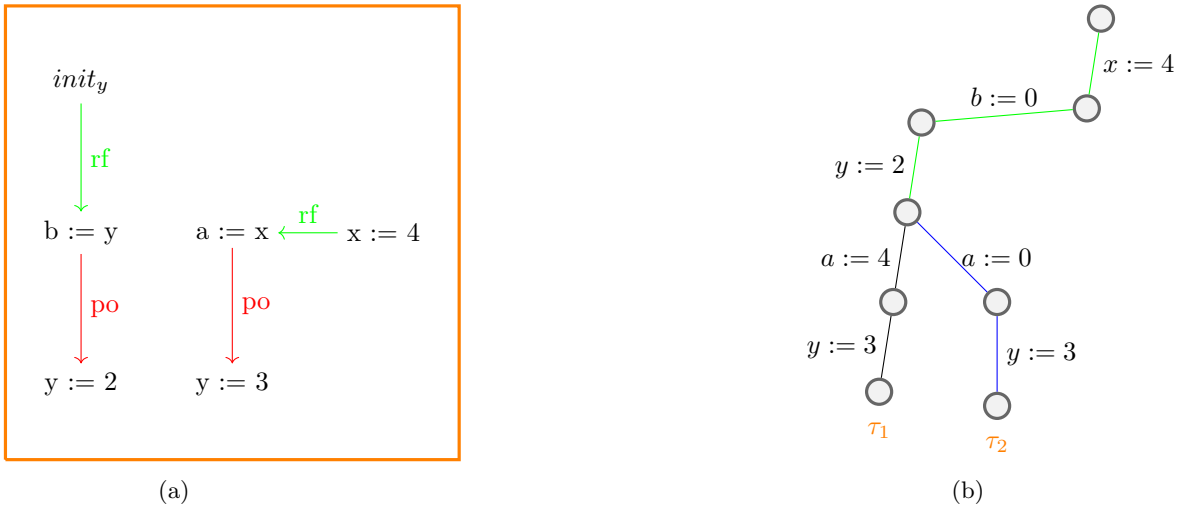


Figure 2: (a) po, rf relations for trace τ_1 (b) Trace τ_2

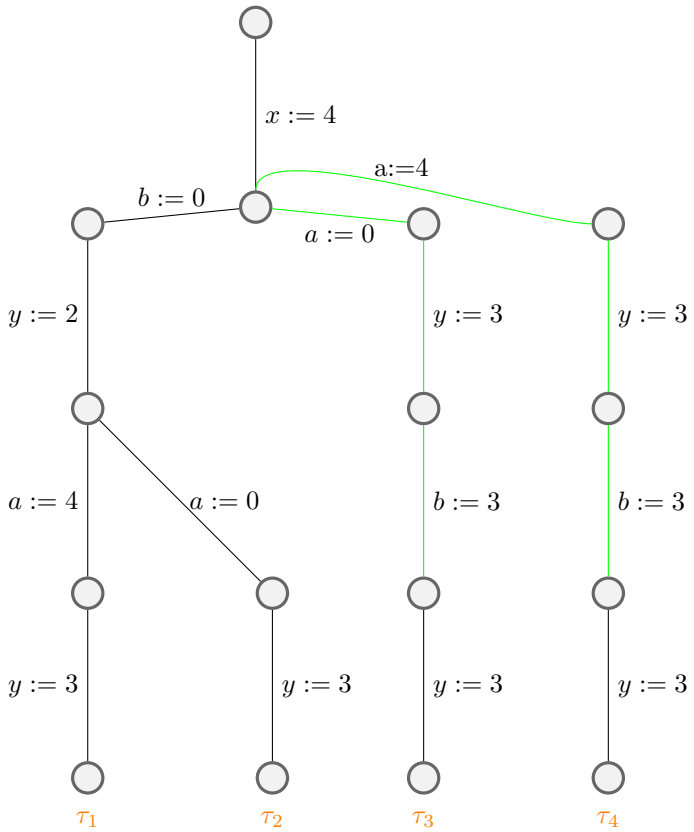


Figure 3: Traces generated from program $\mathcal{P}$