Dynamic Partial Order Reduction Under the Release-Acquire Semantics (Tutorial)

Parosh Aziz Abdulla(✉), Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo

Uppsala University, Uppsala, Sweden
parosh@it.uu.se

Abstract. We describe at a high-level the main concepts in the Release-Acquire (RA) semantics that is part of the C11 language. Furthermore, we describe the ideas behind an optimal dynamic partial order reduction technique that can be used for systematic analysis of concurrent programs running under RA.

This tutorial is based on the material presented in [5], which also contains the formal definitions of all the models, concepts, and algorithms.

1 Introduction

Concurrent programs are difficult to get correct. The main reasons are the large number of threads that may arise during a given execution of the program, and the intricate nature of interactions among these threads. *Model checking* has been of the most prominent approaches to program verification during the last three decades [16]. Given a formal model of a program and a property to be checked, a model checking algorithm checks *automatically* whether the program satisfies the property or not. A limiting factor in the application of model checking is the *state explosion problem* which occurs since the size of the state space of the program grows exponentially with the number of threads. *Stateless Model Checking* (SMC) [21] has been proposed as a way to reduce the problem at the price of sacrificing the completeness of the analysis. The aim is to exploit the hypothesis that bugs that arise only due to some (usually a small number) of the possible thread schedulings. In contrast to full model checking, SMC algorithms are run under two assumptions on the input program:

- Each thread in the program is assumed to be terminating. To enforce this condition, program loops are unfolded a certain *a priori* decided number of times.
- Each thread is assumed to be data-deterministic, and hence the only source of non-determinism lies in the thread schedulings. To enforce the condition we analyze the program for a given initial value per variable.

Under these two assumptions, SMC systematically explores the set of all thread schedulings that are possible during the runs of the program. The SMC exploration is derived by a special run-time scheduler which derives new thread

M. F. Atig and A. A. Schwarzmann (Eds.): NETYS 2019, LNCS 11704, pp. 3–18, 2019.
https://doi.org/10.1007/978-3-030-31277-0_1

schedulings whenever it detects that such decisions may affect the interaction between threads. In such a manner, it is guaranteed that the exploration covers all possible executions. This also means that the algorithm detects any unexpected program results, program crashes, or assertion violations. Due to the assumptions, we can in general consider SMC as an *under-approximate* verification framework in the sense that all reported errors are true bugs in the program, but certain bugs may remain undetected. Despite not being complete, SMC offers numerous advantages, namely:

- It is completely automatic.
- It has no false positives.
- It does not consume excessive memory.
- It can easily reproduce the concurrency bugs it detects.

Due to these advantages, SMC has along the years been implemented in numerous tools, such as VeriSoft [23], CDSChecker [18,31], Chess [30], Concuerror [14], rInspect [37], and Nidhugg [1], and successfully applied to realistic concurrent programs [22,27].

Notwithstanding, SMC still faces the state space explosion problem, albeit in a less severe manner. To circumvent this problem, different techniques have been proposed to reduce the number of explored executions. The most important one is *partial order reduction*. Partial order techniques were first developed for full model checking [11,12,15,20,32,36]. It was later integrated in the SMC framework, and called *dynamic partial order reduction* (DPOR) [2,3,19,34,35]. DPOR avoids redundant exploration of equivalent executions with the same order between conflicting instructions. Two executions are regarded as equivalent if they induce the same ordering between conflicting events. It is sufficient to explore one execution in each equivalence class. The equivalence classes are sometimes called *Mazurkiewicz traces* [29]. An important goal is to avoid exploring several traces that belong to the same equivalence class since, after all, such traces carry the same information with respect to the property to be verified. Ideally, we would like to design DPOR algorithms that are *optimal* in the sense that they explore *exactly* one interleaving in each equivalence class [2].

In this tutorial, we will consider two issues related to the principles of the DPOR approach:

- Mazurkiewicz traces distinguish executions based on the ordering of conflicting write operations, and on how reads are ordered wrt. the writes. Therefore, Mazurkiewicz traces induce an equivalence relation that is occasionally unnecessarily coarse for checking typical properties such as program crashes or assertion violations. For instance, it includes *coherence order*, i.e., the order in which write events on shared variables reach the memory. Coherence order is irrelevant for checking the above properties, and hence using Mazurkiewicz traces is a source of redundancy, inherently limiting the efficiency that can possibly be achieved in the analysis. In this tutorial, we will consider a weaker equivalence relation that is still sufficiently strong to guarantee full coverage of the state space, and that will therefore potentially achieve efficiency levels that are not possible with current techniques.

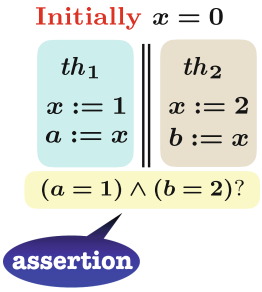

Fig. 1. A concurrent program $\mathcal{P}_1$ with two threads.

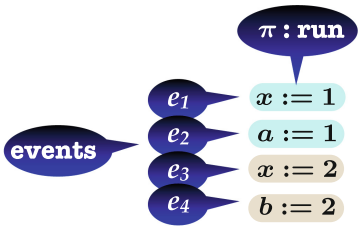

Fig. 2. A run of the program $\mathcal{P}_1$.

– The DPOR framework was originally deigned for concurrent program running under the classical model of Sequential Consistency (SC) [2,19,35]. However, nowadays most parallel software run on platforms that do not guarantee SC. More precisely, to satisfy demands on efficiency and energy saving, such platforms implement optimizations that lead to the relaxation of the inter-component synchronization, hence offering only *weak consistency* guarantees. In recent years, DPOR has also been adapted to hardware-induced relaxed memory models, such as TSO and PSO [1,37], and language-level concurrency models, such as the C/C++ memory model [26,31]. In this tutorial, we will describe how to extend DPOR to a particular consistency model, namely the Release-Acquire fragment of C11 [28].

2 Sequential Consistency

We will introduce concurrent programs, define the notions of runs and traces, and describe (optimal) DPOR algorithms.

2.1 Concurrent Programs

Fig. 1 depicts a program $\mathcal{P}_1$ with two threads, namely th_1 and th_2. The threads operate on a set X of shared variables, (in this example, one shared variable x), and a set of local variables (here one variable in each thread, namely a and b in th_1 and th_2 respectively). The code of each thread consists of two instructions, one *writing* the values 1 resp. 2 to the shared variable x, and one *reading* the value of x, and storing the value in the local variables a resp. b.

2.2 Runs and Traces

Fig. 2 depicts a run π of $\mathcal{P}_1$ under the *Sequential Consistency (SC) semantics*. A run is a sequence of *events* each corresponding to the execution of one instruction by a thread. Under SC, instructions are executed in *program-order*, i.e., in the

same order as they occur in the code of the thread. A *run* is the interleaving of the executions of the different threads. In particular, the read and write instructions are executed atomically. This means that when a write instruction is executed by a thread, its effect will be immediately visible to all the other threads, and a read instruction on a variable x will get its value from the latest write instruction on x. For instance, in the run π of Fig. 2, the value of x read by thread th_1 is 1 since the latest write instruction on x assigned the value 1 to x. Notice that we represent a read event by the value that it reads from the shared variable.

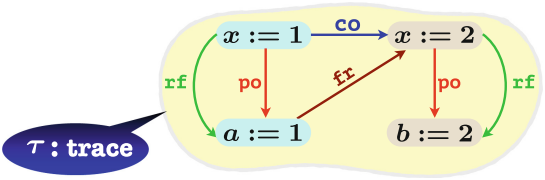

Fig. 3. The trace corresponding to π in Fig. 2.

Next, we define the notion of a *trace* that we will use to represent sets of runs. Traces have three advantages in our framework, namely exactness, efficiency, and abstraction:

- They provide sufficient information for checking different program properties, and therefore they are an *exact* representation of program runs.
- The provide a more compact representation than program runs, and therefore they are more *efficient* when performing verification.
- They are *abstract* in the sense that they can easily be adapted to different memory models.

A *trace* τ, corresponding to the run π in Fig. 2 is depicted in Fig. 3. We write $\pi \models \tau$ to denote that τ corresponds to π. A trace is a graph where the nodes represent events. The edges between the nodes represent four different relations. The *program-order* relation $\mathtt{po}$ is the order in which the events are executed by a given thread. The *read-form* relation $\mathtt{rf}$ defines the write event from which a read event gets its value. The *coherence-order* relation is defined as $\mathtt{co} = \cup_{x \in X} \mathtt{co}^x$, where $\mathtt{co}^x$ is a total order defining the order in which the write events are carried out on the variable x. Finally, the relation $\mathtt{fr}$ gives the write event which overwrites the value that is read by a read event. More precisely, if e_1 reads from e_2 and e_3 is the immediate successor of e_2 in the coherence-order relation, then e_1 precedes e_3 in the read-from relation. The relation $\mathtt{fr}$ can be derived from the relations $\mathtt{rf}$ and $\mathtt{co}$ in the sense that $\mathtt{fr} = \mathtt{co}^{-1} \circ \mathtt{rf}$.

Different memory models can be defined by requiring the acyclicity of different fragments of the above relations. For a trace τ, we write $\tau \models SC$ to denote that $acyclic(\mathtt{po} \cup \mathtt{rf} \cup \mathtt{cofr})$ holds, i.e., SC is defined by the constraint that the union of the four relations should be acyclic. We write $\pi \models SC$ to denote that both $\pi \models \tau$ and $\tau \models SC$, i.e., a run is in SC if its trace satisfies the SC condition.

Fig. 4. (Optimal) Partial Order Reduction.

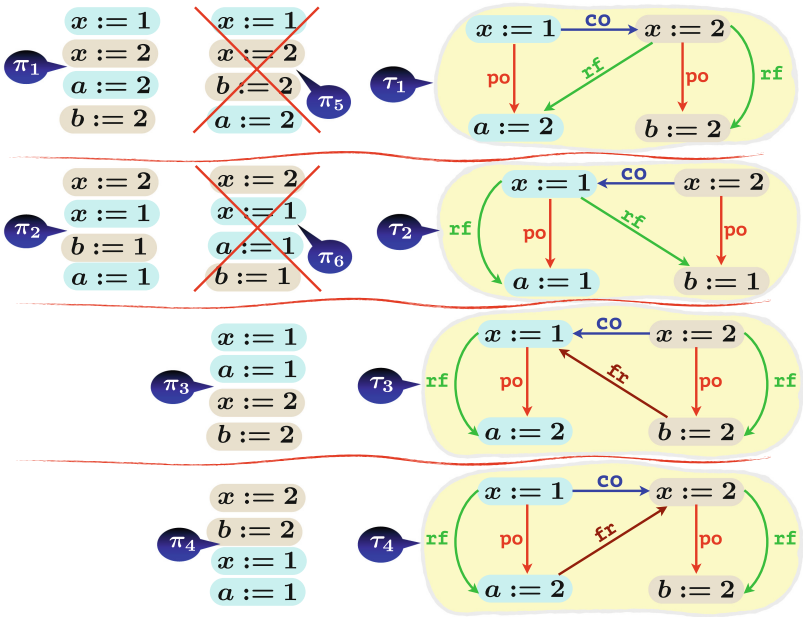

Fig. 5. The runs of the program in Fig. 1, together with their traces.

2.3 DPOR

A DPOR algorithm is represented as a black-box in Fig. 4. The algorithm is given a concurrent program as input and the algorithms generates at least one run per trace. The algorithm analyzes the traces on-the-fly to check for unexpected program results, program crashes, or assertion violations. An important goal for any DPOR algorithm is to generate as a few runs as possible per trace, thus increasing the efficiency of the analysis. An optimal DPOR algorithm generates exactly one run per each trace [2,3]. Fig. 5 shows all the six runs of the program $\mathcal{P}_1$ of Fig. 1, together with the corresponding traces. The runs π_1 and π_5 have the same trace τ_1, and the runs π_2 and π_6 have the same trace τ_2. A possible outcome of an optimal DPOR algorithm is the set $\{\pi_1, \pi_2, \pi_3, \pi_4\}$. The runs π_5 and π_6 are not generated. The algorithm may as well generate π_5 instead of π_1 but not both. The same applies to π_2 and π_6.

We can take one step further by observing that two traces may have identical program-order and read-from relations and differ only in their coherence-order.

In Fig. 5 this applies to the traces τ_3 and τ_4. The coherence-order relation is used to define the memory model, and it has no bearing on the set of assertions that are satisfied by the program. The latter are solely decided by the program-order and the read-from relations. Therefore, we can only generate runs that have different program-order and read-from relations without sacrificing the precision of the analysis. A *super-optimal* DPOR algorithm will only generate one of the runs π_3 and π_4 since their traces have identical program-order and read-from relations [5]. Therefore their traces τ_3 and τ_4 can be "merged" into one trace τ_5 which has the same program-order and read-from relations as τ_3 and τ_4 (Fig. 6).

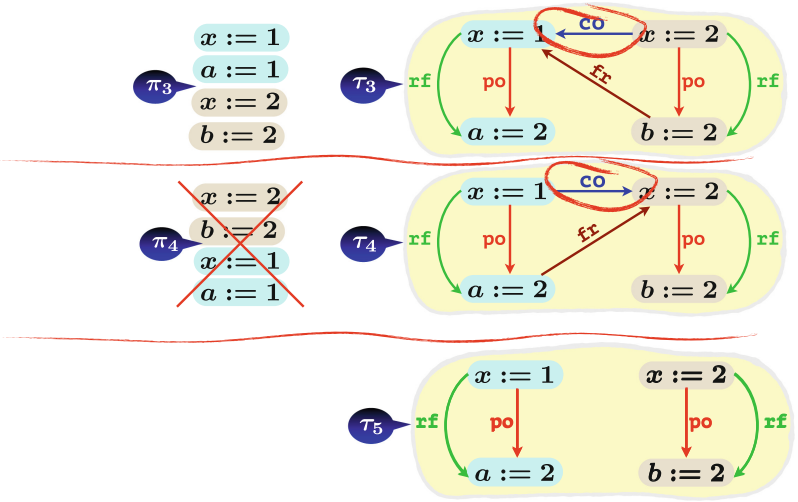

Fig. 6. Super-Optimal Partial Order Reduction.

3 The Release-Acquire Semantics

We will recall the Release-Acquire semantics, and describe the main ideas of a DPOR framework for the analysis of concurrent program running under RA. To that end, we introduce a saturation procedure, and use it to define operations for adding new events to traces, and finally descries the DPOR algorithm.

3.1 Semantics

The Release-Acquire semantics (RA) is defined by the constraint

$$\forall x \in X.\ acyclic(\mathtt{po} \cup \mathtt{rf} \cup \mathtt{co}_x \cup \mathtt{fr}_x)$$

In other words, for each variable $x \in X$, the union of the program-order and reads-form relations, together with the restriction of the coherence-order and

from-read relations to x should be acyclic. Fig. 7 shows a program $\mathcal{P}_2$ with four threads, and a trace τ of $\mathcal{P}_2$ that satisfies RA (denoted $\tau \models RA$). In this case, $\tau \not\models SC$ due to the cycle $(x := 1) \xrightarrow{\text{po}} (y := 2) \xrightarrow{\text{co}^y} (y := 1) \xrightarrow{\text{po}} (x := 2) \xrightarrow{\text{co}^x} (x := 1)$. The cycle does not violate the RA semantics since it contains coherence-order relations on different variables (x and y).

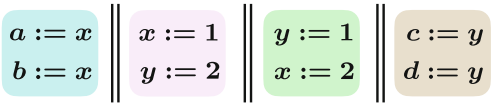

Fig. 7. A concurrent program $\mathcal{P}_2$ with four threads.

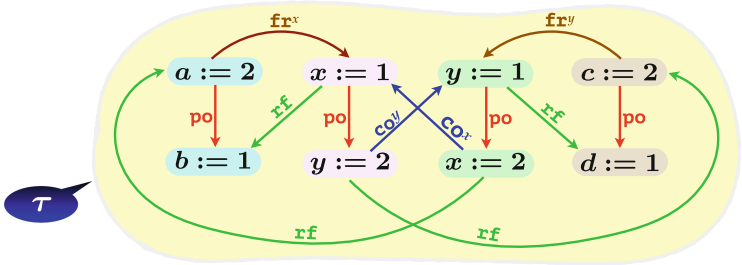

Fig. 8. A trace of the program in the RA-semantics.

3.2 DPOR

We will describe a super-optimal DPOR algorithm for concurrent programs under the RA-semantics. The aim of such an algorithms is to satisfy the following criteria for any input program $\mathcal{P}$:

- *Soundness*: If the algorithm generates a run π then (i) π is a run of $\mathcal{P}$, and (ii) $\pi \models \tau$ for some trace with $\tau \models RA$.
- *Completeness*: For any trace τ where $\pi \models \tau$ for some run π of $\mathcal{P}$ and $\tau \models RA$, the algorithm generates a run π' of $\mathcal{P}$ such that $\pi' \models \tau$.
- *Optimality:* The algorithms never generates two runs π_1 and π_2 such that there are traces τ_1 and τ_2, with $\pi_1 \models \tau_1$, $\pi_2 \models \tau_2$, and τ_1 and τ_2 have identical program-order and read-from relations.

Roughly, the DPOR algorithm operates as follows:

- It builds the traces one after one.
- For a given trace τ:
  - It builds τ incrementally.
  - It *extends* τ by one event e at a time.

We will describe how traces are extended by events. To do that, we will first define the notion of *saturation*.

3.3 Saturation

As mentioned above, coherence-order is not essential for checking assertion violations. On the other hand, coherence-order is part of the definition of the semantics, so it cannot be neglected completely; otherwise we may generate traces that do not satisfy the semantics, which makes the DPOR algorithm unsound. The idea is to *saturate traces*, i.e., add coherence-order edges by demand during the analysis, thus ensuring that we only add edges that are necessary to keep consistency wrt. the semantics. In fact, in the case of RA, saturation is sufficient to achieve optimality, i.e., never generating two traces with the program-order and read-from relations. Under the RA semantics, saturation is simple and be computed in polynomial time (in the size of the trace). To illustrate how, consider the trace in Fig. 9. The trace contains two write events w_1^x and w_2^x, and one read event r^x. The three events satisfy two conditions:

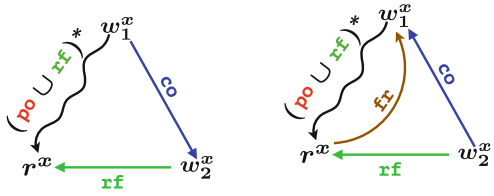

Fig. 9. Saturating a trace under the RA semantics. The coherence-order should point form the event w_1^x to w_1^x (left part of the figure); otherwise we create a cycle violating the RA semantics (right part).

- There is a sequence of po- and rf-edges leading from w_1^x to r^x.
- r^x reads from w_2^x.

In such a case, the saturation procedure adds a co-edge from w_1^x and w_2^x (the left part of Fig. 9). To see why this is necessary, consider the right part of Fig. 9. If the co-edge is put in the reverse direction, then we get a cycle that violates the RA-semantics. In general, for write events e_1 and e_2, and a read event e_3, all on the same variable x, we add an edge $e_1 \xrightarrow{\mathtt{co}^x} e_2$ whenever $e_1 \left(\xrightarrow{\mathtt{po} \cup \mathtt{rf}}\right)^+ e_3$ and $e_2 \xrightarrow{\mathtt{rf}} e_3$. The trace is *saturated* if the saturation rule does not add any new edges. A trace can be saturated in polynomial time, since we can compute the transitive closure of the relation $\mathtt{po} \cup \mathtt{rf}$ using, e.g., the Floyd-Warshall algorithm [17]. The trace in Fig. 10 is not saturated. Due to the path $(x := 2) \xrightarrow{\mathtt{rf}} (a := 2) \xrightarrow{\mathtt{po}} (b := 1)$, and $(x := 1) \xrightarrow{\mathtt{rf}} (b := 1)$, the saturation procedure will add the edge $(x := 2) \xrightarrow{\mathtt{co}^x} (x := 1)$. Analogously, it will add the edge $(y := 2) \xrightarrow{\mathtt{co}^y} (y := 1)$, thus obtaining the trace τ in Fig. 8 which is saturated.

In the DPOR algorithm, all the generated traces are saturated by construction. In Subsect. 3.4 we see how this can be achieved.

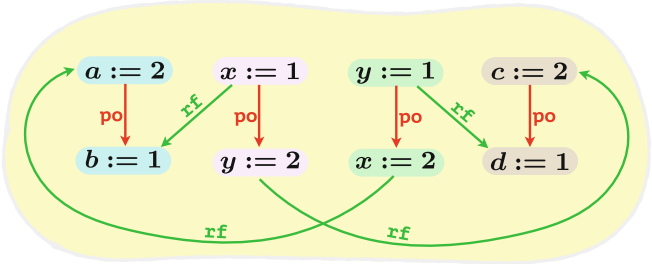

Fig. 10. Unsaturated trace.

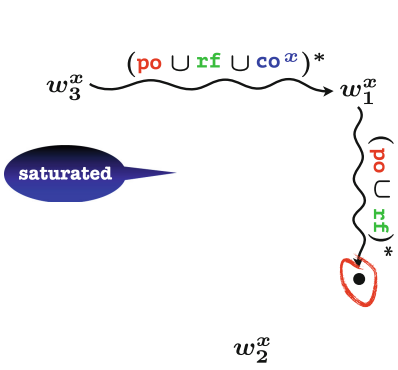

Fig. 11. Part of a trace before adding a new event.

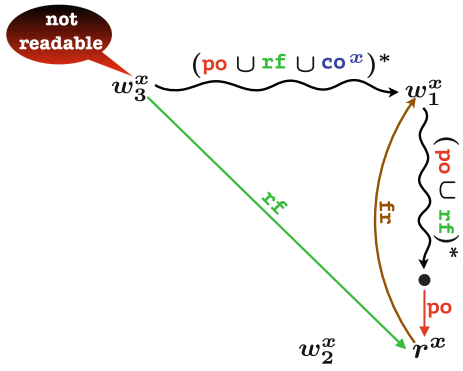

Fig. 12. Reading from the wrong write.

3.4 Adding Events to Traces

Suppose that we are given a trace τ that is saturated. We would like to add a new event e to τ such that the new trace remains saturated. We consider two cases, namely when e is a read resp. write event. Adding a read event amounts to two operations:

- Finding the write events, from which we can read the value of the new variable without violating the RA semantics.
- Adding the necessary (and only the necessary) coherence-order edges to maintain saturation.

To illustrate these ideas, consider a saturated trace partially shown in Fig. 11, with (among others) three write events w_1^x, w_2^x, and w_3^x. Assume that we are about to add a new read event r^x at the marked position. In this example the event r^x is not allowed to read from w_3^x, since this would create a cycle that violates the RA-semantics as shown in Fig. 12. We say that a write event e_1 on a variable x is *readable* if there is no other write event e_2 on x such that e_1 can reach e_2 and e_2 can reach the reading thread through the $(\mathtt{po} \cup \mathtt{rf} \cup \mathtt{co}^x)$-relation.

In Fig. 12, w_3^x is not readable for the read event since w_1^x is "blocking" its path to the reading thread.

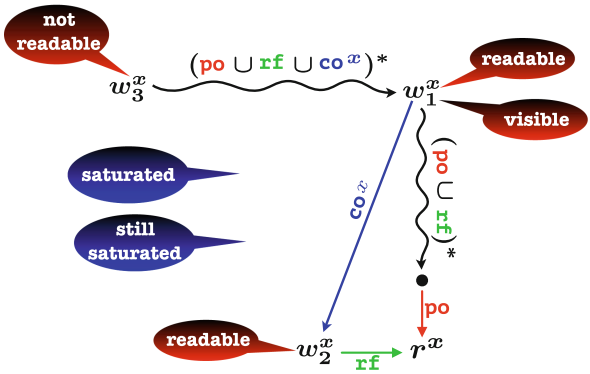

Fig. 13. Adding a new read event.

Assume that w_2^x is readable and that the new event r^x reads its value from w_2^x, as depicted in Fig. 13. Our objective is to make the new trace saturated (assuming that the original trace was saturated). To that end, we add all the new coherence-order edges that are necessary according to the RA semantics. This amounts to applying the saturation rule repeatedly until the trace becomes saturated. To find out where in the trace to apply the saturation rule, we consider the so called *visible events*. These are write events that are readable and can reach the read event through po- and rf-edges. The write event w_1^x is a visible event in Fig. 13. For each visible event, we add a coherence edge the write from which the new event reads (e.g., the edge $w_1^x \xrightarrow{\mathtt{co}^x} w_2^x$ in Fig. 13). If the original trace is saturated, and we add the edges from the visible events (as described above) then the new trace will be saturated.

Adding write events events is a much simpler operation. We only need to add one program-order edge, as depicted in Fig. 14, If the original trace is saturated, then the new trace will be saturated, without the need to add any extra edges.

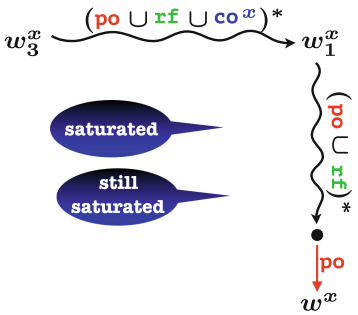

Fig. 14. Adding an new write event.

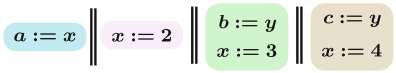

Fig. 15. A concurrent program with four threads.

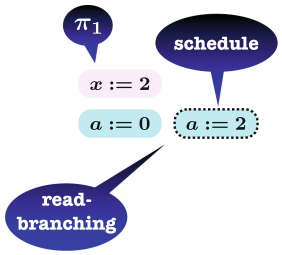

Fig. 16. A concurrent program with four threads.

3.5 Algorithm

Our DPOR algorithm generates systematically different runs of the input programs, and uses the operations described in the previous sub-section to build the corresponding traces. At any point of time, the algorithm will be in the middle of generating one particular run, while also scheduling (start segments) of other runs to be considered later. The algorithm is implemented recursively where a new call generates the next node in the recursion tree. The new node may correspond to the next event in the current run in which case we may also schedule parts of new runs to be considered when the recursive call has returned. The new node may also correspond to the start of the execution of one of the scheduled runs. Upon termination, the set of paths in the recursion tree represents the set of program runs that have been generated. The algorithm will not generate the full tree at the same time. Instead, it generates one path (corresponding to one run) at a time.

Figure 15 depicts a concurrent program with four threads. The DPOR algorithm will no-deterministically select a thread and run its next instruction (Fig. 16). In the current scenario it selects the instruction $x := 2$. Next, it selects the instruction $a := x$. Here, there are two choices. The read event may either read the initial value of x which is assumed to be 0, or read from the write event $x := 2$ that has already been generated. The choice is made non-deterministically. In the example, the algorithm will read from the initial value. Since, we only want to generate one run at a time, we *schedule* the event $a := 2$ (which corresponds to $a := x$ reading from $x := 2$), for future execution (Fig. 16). More precisely, we call the algorithm recursively with $a := 0$, and when the recursive call returns, we consider $a := 2$ to generate a new run. We call this *read-branching*, to hint that the tree branches on the different write events that a given instruction can read from. In the next step, the algorithm selects a possible next instruction, which in this case is $b := y$, for which it selects the only possible value, namely 0. Assume that the algorithm next selects the instruction $x := 3$ (Fig. 17). The instruction could have been used by the read instruction $a := x$. However, this was not made part of the schedule since the instruction $x := 3$ had not yet been detected. An event is called a *postponed write* if it is generated by the algorithm

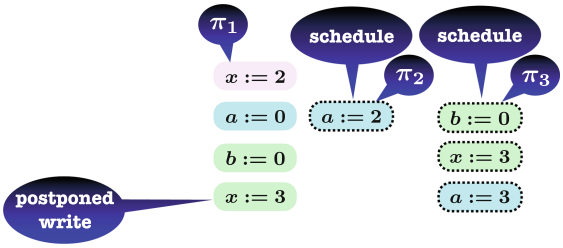

Fig. 17. Postponed writes.

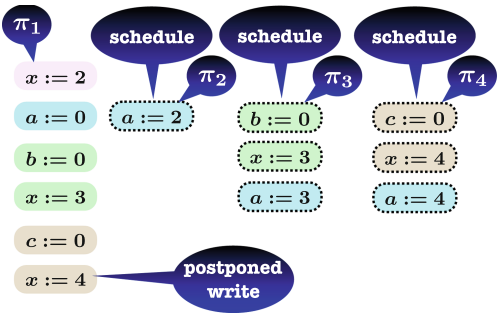

Fig. 18. A complete run π_1.

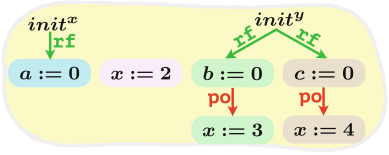

Fig. 19. The trace of π_1.

after a read event that may read it. To get the event $a := 3$, we also need to include the sequence of events that *happen before* it, i.e., the sequence of events that are needed to generate $a := 3$. The happens-before sequence is the sequence of events that proceed the given event by the po- and rf-relations. In our case, this sequence is $b := 0$ $x := 3$. The schedule then will contain the happen-before sequence as well as the event itself.

The run terminates after executing the instructions $c := 0$ (no branching needed), and $x := 4$ which is a postponed write for $a := x$ (the run π_1 Fig. 18). The trace corresponding to π_1 is shown in Fig. 19. The algorithm will now backtrack and considers the schedules. The second schedule $a := 2$ will induce the run π_2 shown in Fig. 20, with the trace of Fig. 21. In particular, it generates the postponed write instruction $x := 3$ for the read instruction $a := x$. However, this write instruction (together with its happens-before event $b := 0$) is already in the set of schedules, and therefore it will not be added to the set of schedules. This feature guarantees that we never generate two traces with identical program-order and read-from relations.

Finally, the two remaining schedules will be considered, resulting in runs two new runs (Fig. 22) whose traces are shown in Fig. 23.

The algorithm has generated all the four traces that belong to the program in Fig. 15 (completeness), has not generated any other traces (soundness), and all the generated traces have different program-order and read-from relations (optimality).

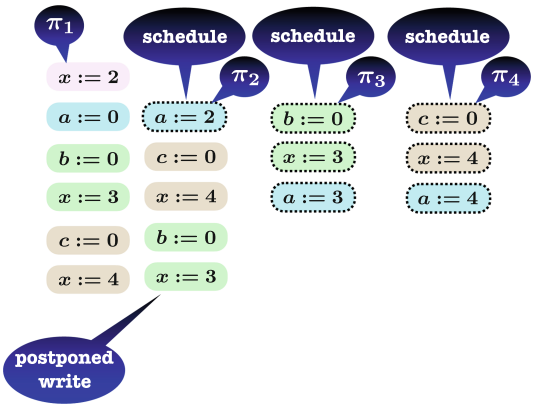

Fig. 20. Another run π_2.

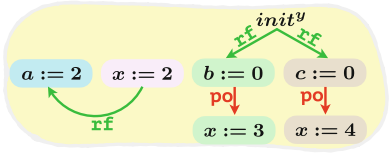

Fig. 21. The trace of π_2.

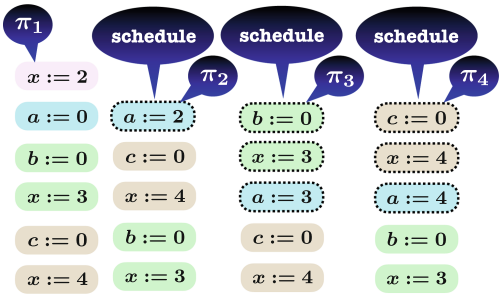

Fig. 22. Two more runs π_3 and π_4.

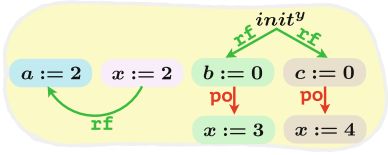

Fig. 23. The trace of π_3 and π_4.

4 Related Work

This tutorial is based on the material presented in [5].

Stateless model checking (SMC), coupled with (dynamic) partial order techniques, was initiated in the works of Verisoft [21,23] and CHESS [30], and has since been developed in several subsequent works, e.g., [2,19,22,27,33]. The method of [2] is optimal w.r.t. Mazurkiewicz traces under SC.

SMC has been applied to weak memory models such TSO, PSO, and POWER [1,4,18,37]. SMC has been adapted to (variants of) the C/C++11

memory model, which includes RA, and implemented in tools such as CDSChecker [31] and Rcmc [26].

Several recent DPOR techniques aim at using a weaker equivalence than Mazurkiewicz traces [13,24,25,31]. Maximal causality reduction (MCR) is a technique based on exploring the possible *values* that reads can see, instead of the possible value-producing writes, as in our approach. MCR has been developed for SC [24] and for TSO and PSO [25].

5 Conclusions and Future Work

We have presented the main ideas behind a DPOR algorithm for the analysis of concurrent programs running under the RA semantics. The algorithm is optimal in the sense that it generates at most one trace with a given program-order and read-from relation.

In this tutorial we only consider the RA semantics. It is interesting to extend the approach to other memory models. In particular, the *relaxed* fragment of C/C++11 is challenging since it contains speculative operations that are not covered in the current framework. However, we believe that speculations can be handled using a scheme similar to the postponed write mechanism that we described in this paper.

Other directions for future work include considering probabilistic and game-based models that describe the manner in which a given message will be read by a given process. It might then be possible to analyze the system using techniques for the analysis of probabilistic and game-based extensions of multi-dimension infinite-state systems, e.g., [6,8,9]. It is also relevant to check whether a given platform guarantees a given weak memory, using monitoring techniques such as the one described in [10]. Finally it is interesting to obtain efficient verification frameworks by integrating powerful abstraction techniques for infinite-state systems such as the one in [7].

References

1. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_28
2. Abdulla, P., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: Symposium on Principles of Programming Languages, (POPL), pp. 373–384. ACM, San Diego (2014)
3. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Source sets: a foundation for optimal dynamic partial order reduction. J. ACM **64**(4), 25:1–25:49 (2017). https://doi.org/10.1145/3073408
4. Abdulla, P.A., Atig, M.F., Jonsson, B., Leonardsson, C.: Stateless model checking for POWER. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 134–156. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_8

5. Abdulla, P.A., Atig, M.F., Jonsson, B., Ngo, T.P.: Optimal stateless model checking under the release-acquire semantics. PACMPL **2**(OOPSLA), 135:1–135:29 (2018). https://doi.org/10.1145/3276505
6. Abdulla, P.A., Bertrand, N., Rabinovich, A.M., Schnoebelen, P.: Verification of probabilistic systems with faulty communication. Inf. Comput. **202**(2), 141–165 (2005). https://doi.org/10.1016/j.ic.2005.05.008
7. Abdulla, P.A., Bouajjani, A., Cederberg, J., Haziza, F., Rezine, A.: Monotonic abstraction for programs with dynamic memory heaps. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 341–354. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_33
8. Abdulla, P.A., Bouajjani, A., d'Orso, J.: Deciding monotonic games. In: Baaz, M., Makowsky, J.A. (eds.) CSL 2003. LNCS, vol. 2803, pp. 1–14. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45220-1_1
9. Abdulla, P.A., Deneux, J., Mahata, P.: Multi-clock timed networks. In: 19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14–17 July 2004, Turku, Finland, Proceedings, pp. 345–354. IEEE Computer Society (2004). https://doi.org/10.1109/LICS.2004.1319629
10. Abdulla, P.A., Haziza, F., Holík, L., Jonsson, B., Rezine, A.: An integrated specification and verification technique for highly concurrent data structures. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 324–338. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_23
11. Abdulla, P.A., Jonsson, B., Kindahl, M., Peled, D.: A general approach to partial order reductions in symbolic verification. In: Hu, A.J., Vardi, M.Y. (eds.) CAV 1998. LNCS, vol. 1427, pp. 379–390. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0028760
12. Abdulla, P.A., Kindahl, M., Peled, D.A.: An improved search strategy for lossy channel systems. In: Togashi, A., Mizuno, T., Shiratori, N., Higashino, T. (eds.) Formal Description Techniques and Protocol Specification, Testing and Verification, FORTE X / PSTV XVII'97, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE X) and Protocol Specification, Testing and Verification (PSTV XVII), 18–21 November 1997, Osaka, Japan, IFIP Conference Proceedings, vol. 107, pp. 251–264. Chapman & Hall (1997)
13. Chalupa, M., Chatterjee, K., Pavlogiannis, A., Sinha, N., Vaidya, K.: Data-centric dynamic partial order reduction. Proc. ACM Program. Lang. **2**(POPL), 31:1–31:30 (2017). https://doi.org/10.1145/3158119
14. Christakis, M., Gotovos, A., Sagonas, K.: Systematic testing for detecting concurrency errors in Erlang programs. In: International Conference on Software Testing. Verification and Validation, (ICST), pp. 154–163. IEEE, Luxembourg (2013)
15. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.A.: State space reduction using partial order techniques. STTT **2**(3), 279–287 (1999)
16. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8
17. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, Cambridge (2009)
18. Demsky, B., Lam, P.: Satcheck: Sat-directed stateless model checking for SC and TSO. In: Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA), pp. 20–36. ACM, Pittsburgh (2015)
19. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Principles of Programming Languages, POPL, pp. 110–121. ACM, Long Beach (2005)

20. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Ph.D. thesis, University of Liége (1996). Also, volume 1032 of LNCS, Springer
21. Godefroid, P.: Model checking for programming languages using VeriSoft. In: Principles of Programming Languages, POPL, pp. 174–186. ACM Press, Paris (1997)
22. Godefroid, P., Hammer, B., Jagadeesan, L.: Model checking without a model: an analysis of the heart-beat monitor of a telephone switch using VeriSoft. In: Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 124–133 (1998)
23. Godefroid, P.: Software model checking: the VeriSoft approach. Formal Methods Syst. Des. **26**(2), 77–101 (2005)
24. Huang, J.: Stateless model checking concurrent programs with maximal causality reduction. In: Programming Language Design and Implementation, PLDI, pp. 165–174. ACM, Portland (2015)
25. Huang, S., Huang, J.: Maximal causality reduction for TSO and PSO. In: Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA), pp. 447–461. ACM, Amsterdam (2016)
26. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. In: POPL (2018, to appear). http://plv.mpi-sws.org/rcmc/
27. Kokologiannakis, M., Sagonas, K.: Stateless model checking of the linux kernel's hierarchical read-copy-update (tree RCU). In: Symposium on Model Checking of Software, SPIN, pp. 172–181. ACM, Santa Barbara (2017)
28. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 649–662. ACM (2016)
29. Mazurkiewicz, A.: Trace theory. In: Advances in Petri Nets (1986)
30. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: OSDI, pp. 267–280. USENIX Association (2008)
31. Norris, B., Demsky, B.: A practical approach for model checking C/C++11 code. ACM Trans. Program. Lang. Syst. **38**(3), 10:1–10:51 (2016)
32. Peled, D.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56922-7_34
33. Rodríguez, C., Sousa, M., Sharma, S., Kroening, D.: Unfolding-based partial order reduction. In: CONCUR 2015, pp. 456–469 (2015)
34. Saarikivi, O., Kähkönen, K., Heljanko, K.: Improving dynamic partial order reductions for concolic testing. In: Application of Concurrency to System Design, ACSD, pp. 132–141. IEEE, Hamburg (2012)
35. Sen, K., Agha, G.: A race-detection and flipping algorithm for automated testing of multi-threaded programs. In: Haifa Verification Conference. pp. 166–182 (2007), lNCS 4383
36. Valmari, A.: Stubborn sets for reduced state space generation. In: Rozenberg, G. (ed.) ICATPN 1989. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-53863-1_36
37. Zhang, N., Kusano, M., Wang, C.: Dynamic partial order reduction for relaxed memory models. In: Programming Language Design and Implementation (PLDI), pp. 250–259. ACM, Portland (2015)