# Conditional Dynamic Partial Order Reduction and Optimality Results

Miguel Isabel
Complutense University of Madrid
Spain
miguelis@ucm.es

## ABSTRACT

Testing concurrent systems requires exploring all possible non-deterministic interleavings that the concurrent execution may have, as any of the interleavings may reveal an erroneous behaviour of the system. This introduces a combinatorial explosion on the number of states that must be considered, which leads often to a computationally intractable problem. In the present PhD thesis, this challenge will be addressed through the development of new Partial Order Reduction techniques (POR). The cornerstone of POR theory is the notion of *independence*, that is used to decided whether each pair of concurrent events $p$ and $t$ are in a race and thus both executions $p \cdot t$ and $t \cdot p$ must be explored. A fundamental goal of this thesis is to introduce notions of *conditional independence* –which ensure the commutativity of the considered events $p$ and $t$ under certain conditions that can be evaluated in the explored state– with a DPOR algorithm in order to alleviate the combinatorial explosion problem. The new techniques that we propose in the thesis have been implemented within the SYCO tool. We have carried out accompanying experimental evaluations to prove the effectiveness and applicability of the proposed techniques. Finally, we have successfully verified a range of properties for several case studies of Software-Defined Networks to illustrate the potential of the approach, scaling to larger networks than related techniques.

## CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

## KEYWORDS

Testing, Software Verification, Partial-Order Reduction

**ACM Reference Format:**
Miguel Isabel. 2019. Conditional Dynamic Partial Order Reduction and Optimality Results. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3293882.3338987

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*ISSTA '19, July 15–19, 2019, Beijing, China*
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3338987

## 1 INTRODUCTION

Due to increasing performance demands, application complexity and multi-core parallelism, concurrency is present everywhere in today's software applications. It is widely recognized that concurrent programs are difficult to develop, debug, test and analyze. This is even more so in the context of concurrent *imperative* languages that use a global memory (so called heap) to which the different tasks can have access. These accesses introduce additional hazards not present in sequential programs such as race conditions, data races, deadlocks, and livelocks. Therefore, software validation techniques urge especially in the context of concurrent programming.

Testing is the most widely-used methodology for software validation. However, due to the non-deterministic interleaving of tasks, traditional testing for concurrent programs is not as effective as for sequential programs. In order to ensure that all behaviors of the program are tested, the testing process, in principle, must systematically explore all possible ways in which the tasks can interleave. This is known as *systematic testing* [24] in the context of concurrent programs. Such full systematic exploration of all task interleavings produces the well known state explosion problem and is often computationally intractable (see, e.g., [25] and its references).

Partial-order reduction (POR) [14] is a general theory that helps mitigate this combinatorial explosion by formally identifying *equivalence* classes of redundant explorations. POR is based on the idea that two interleavings can be considered equivalent if one can be obtained from the other by swapping adjacent, non-conflicting *independent* execution steps. Such equivalence class is called a Mazurkiewicz trace [21], and POR guarantees that it is sufficient to explore one interleaving per equivalence class. For this purpose, a *happens-before* partial order among the events of the execution sequences is defined. This order relation induces a set of equivalent execution sequences, i.e., those sequences with the same happens-before order belong to the same equivalence class. As a result, only one of them needs to be explored.

## 2 STATE OF THE ART

Early POR algorithms [12, 14, 26] relied on static over-approximations to detect possible *future* conflicts. The Dynamic-POR (DPOR) algorithm, introduced by Flanagan and Godefroid [13] in 2005, was a breakthrough in the area because it does not need to look at the future. It keeps track of the independence races witnessed along its execution and uses them to decide the required exploration dynamically, without the need of static approximation. DPOR is nowadays considered one of the most scalable techniques for concurrent software testing. The key of DPOR algorithms is in the dynamic construction of two types of sets at each scheduling point: the *sleep set* that contains processes whose exploration has been



RIGHTSLINK

proven to be redundant (and hence should not be selected), and the *backtrack set* that contains the processes that have not been proven independent with previously explored steps (and hence may need to be explored). Source-DPOR (SDPOR) [1, 2] improves the precision computing backtrack sets (named there *source* sets).

An extension of the DPOR algorithm, named Optimal DPOR (ODPOR) [2], guarantees the optimality of the exploration, i.e., the proposed DPOR algorithm never explores redundant states (they are equivalent to others already explored). In addition to using source sets, a major extension is needed to achieve optimality: the use of *wakeup trees* to guide the initial steps in the exploration. By means of these extensions, ODPOR guarantees that redundant explorations are never even initiated, proving optimality for *any* number of processes w.r.t. an *unconditional independence* relation.

Both the original DPOR and its extension ODPOR are based on an *unconditional* dependency relation (also called unconditional happens-before relation) which determines the partial order of transitions, i.e., for two transitions be considered independent they must commute in all possible contexts. Let us consider the following three simple processes $\{p, q, r\}$ and the initial state $x = 0$:

$p$: `write(x=5)`, $q$: `write(x=5)`, $r$: `assert(x>0)`.

The transitions `write(x=5)` and `write(x=5)` are unconditionally independent but `write(x=5)` and `assert(x>0)` are not. The latter are independent only if $x > 0$.

Conditional independence was earlier introduced in the context of POR [19], where it was proven that only *uniform* conditional independence can be used, i.e., independence must hold along the whole trace. A notion of uniformity is needed because, unlike unconditional independence, pruning the DPOR search space by relying on whether conditions like $x > 0$ above, called *independence constraints (ICs)*, are satisfiable in the explored state is unsound in general. The ICs provide the conditions under which each pair of transitions commutes, i.e., both execution orders leads to the same state. The first algorithm that took advantage of independence constraints is [27], but this algorithm can only ensure optimality between *two* threads, (while DPOR ensures it among any number of threads) and infers ICs for *single* instructions. The first algorithm that has used notions of conditional independence within the state-of-the-art DPOR algorithm is Context-Sensitive DPOR (DPOR$_{cs}$) [4]. However, DPOR$_{cs}$ does not use ICs (it rather checks state equivalence dynamically during the exploration) and exploits conditional (context-sensitive) independence only *partially* to extend the sleep sets. ODPOR with Observers (ODPOR$^{ob}$) [3] introduces the notion of *observability*, where dependencies between execution steps $p$ and $t$ are conditional to the existence of future steps, called observers, which read the values modified by $p$ and $t$. For the previous example, when ODPOR$^{ob}$ explores the execution $r.p.q$, we have that $p$ and $q$ are considered as independent, since there is not a observer executed after them. This is because $r$ is executed before, so it does not read the value written by $p$ and $q$.

## 3 GOALS OF THE RESEARCH

The overall goal of this PhD thesis is to be able to explote notions of *conditional independence* –which ensure the commutativity of the considered events $p$ and $t$ under certain conditions that can be evaluated in the explored state– within DPOR algorithms in order to alleviate the combinatorial explosion problem described in Section 1. This goal is materialized in the following research challenges:

i) combine and exploit the notions of Context-Sensitive DPOR, Optimal DPOR with Observers and study their synergies to gain further pruning,
ii) propose (sufficient) conditions that ensure uniformity and enable new forms of pruning using ICs,
iii) integrate the notion of uniform conditional independence (which requires to look ahead) to prune the search space in a *dynamic* algorithm using ICs,
iv) statically synthesize ICs for *atomic blocks* of instructions in a pre-analysis using a SMT approach,
v) extend the DPOR framework, that ensures optimality for *any* number of processes, to exploit ICs during the exploration in order to both reduce the backtrack sets and expand the sleep sets as much as possible,
vi) carry out a thorough experimental evaluation to compare the different extensions, and
vii) apply the techniques to a realistic setting.

## 4 CURRENT STATE OF THE RESEARCH

At the Conference ISSTA 2019 [6], we have presented Optimal Context-Sensitive DPOR with Observers which addresses the first challenge. We have formulated Context-Sensitive DPOR over Optimal DPOR, which is named Optimal Context-Sensitive DPOR (ODPOR$_{cs}$), and it includes the extension of wakeup trees used to ensure optimality. Furthermore, we have also integrated the notion of observability into ODPOR$_{cs}$ and the resulting algorithm is called Optimal Context-Sensitive DPOR with Observers (ODPOR$^{ob}_{cs}$). To illustrate this, let us consider again the previous example. ODPOR$^{ob}_{cs}$ detects $p.q.r$ and $q.p.r$ as redundant, because in both sequences $r$ observes the same result (the assert holds). Consequently, ODPOR$^{ob}_{cs}$ only needs to explore one of them, whereas ODPOR$^{ob}$ explores both executions. Finally, we have implemented this new algorithm and perform an experimental evaluation that shows it can explore exponentially less sequences than DPOR$_{cs}$ and ODPOR$^{ob}_{cs}$.

At the Conference CAV 2018 [5], we have presented *Constrained Dynamic Partial Order Reduction* (CDPOR) which addresses the second, third and fourth challenges. As a result, we have introduced sufficient conditions –that can be checked dynamically– to soundly exploit ICs within the DPOR framework. Moreover, we have extended the state-of-the-art DPOR algorithm with new forms of pruning (by means of expanding sleep sets and reducing backtrack sets). We have also presented an SMT-based approach to automatically synthesize ICs for *atomic blocks*, whose applicability goes beyond the DPOR context. For `write(x)` and `write(x)`, it infers *true* as IC (that is, they are unconditional independent) and, for `write(x)` and `assert(x>0)` it infers $x > 0$. During the exploration, CDPOR detects $p$ and $r$ as dependent in the execution $p.r.q$ (because the condition does not hold in the initial state) and as independent in the execution $q.p.r$ (since after $q$, the IC holds), avoiding the exploration of $q.r.p$. Moreover, we have experimentally shown the exponential gains achieved by CDPOR.

However, CDPOR extends Source-DPOR only with sound ways of exploiting ICs and, although it has experimentally shown to



RIGHTSLINK

achieve exponential reductions, they have not been proven optimal w.r.t. the equivalence classes induced by a conditional happens-before relation. Currently, we are working on a new direction to address the fifth challenge. We aim at characterizing the properties of a *conditional happens-before* relation which allows defining the equivalence classes to be explored by an optimal DPOR algorithm as a disjunction of partial orders. Furthermore, we aim at defining a provably Optimal Constrained DPOR algorithm that varies from unconditional ODPOR in the construction of the sequences to be explored when races are detected. We plan to use a conditional happens-before relation that can be checked efficiently during the execution of the DPOR algorithm and is sufficiently accurate to detect redundancies that can only be captured using a conditional relation. Finally, we will perform an experimental evaluation against ODPOR and CDPOR to show the gains of this new approach.

## 5 EXPERIMENTS

The thesis will be backed up by a thorough experimental evaluation that addresses goal vi). Namely, in [5], we have reported on experimental results that compare the performance of three DPOR algorithms: SDPOR [1, 2], DPOR$_{cs}$ [4] and our proposal CDPOR [5]. We have implemented and experimentally evaluated CDPOR within the SYCO tool [4], a systematic testing tool for message-passing concurrent programs. SYCO can be used online through its web interface available at http://costa.fdi.ucm.es/syco. To generate the ICs, SYCO calls a new feature of the VeryMax program analyzer [10] which uses Barcelogic [9] as SMT solver.

As benchmarks, we have borrowed the examples from [4] (available online from the previous url) that were used to compare SDPOR with DPOR$_{cs}$ (see Table 1). They are classical concurrent applications: a concurrent sorting algorithm (*QS*), concurrent Fibonacci (*Fib*) and several distributed workers (*Pi, PS*). These benchmarks feature the typical concurrent programming methodology in which computations are split into smaller atomic subcomputations which concurrently interleave their executions, and which work on the same shared data. Therefore, the concurrent processes are highly interfering, and both inferring ICs and applying DPOR algorithms on them becomes challenging. We have executed each benchmark with size increasing input parameters. As it can be observed in the table, he results show that the gains of CDPOR increase exponentially in all examples respect to the size of the input. When compared with DPOR$_{cs}$, we achieve reductions up to 4 orders of magnitude for the largest inputs on which DPOR$_{cs}$ terminates. W.r.t. SDPOR, we achieve reductions of 4 orders of magnitude even for smaller inputs for which SDPOR terminates. In *QS* and *PS*, though the gains are linear for the small inputs, when the size of the problem increases both SDPOR and DPOR$_{cs}$ time out, while CDPOR can still handle them efficiently. As regards the time to infer ICs, we observe that in most cases it is negligible compared to the exploration time of the other methods. Let us also notice that the inference is a pre-process which does not add complexity to the actual DPOR algorithm.

In [6], we have reported on an experimental comparison of the performance of DPOR$_{cs}$, ODPOR$^{ob}$ and our proposal ODPOR$^{ob}_{cs}$. We have used two sets of benchmarks: the same set described above, and a subset of the synthetic examples used in [3] to compare Optimal DPOR and Optimal DPOR with Observers. In general, we obtain speedups with respect to both methods, although when the reduction in explored sequences is small, the overhead of the complex context-sensitive checks does not pay off. Furthermore, our proposal obtains gains, scaling by several orders of magnitude.

In summary, we conclude that our experimental results in [5] and [6] show exponential reductions of explored executions and our gains increase exponentially.

A potential hazard of using conditional independence within DPOR algorithms is that it needs to check independence with respect to states explored in the current execution sequence but not in the current state. It does not need to revisit states that have been completely explored and backtracked, but only those in the current execution sequence. There are several strategies to confront this challenge: *on-demand recomputing*, all states are recomputed following the same events order that led to them (and then no memory usage is needed); *full storage*, all states are stored to be used until the state is backtracked; and *state caching* [23], where states are stored until the memory is approaching full utilization. Our current implementation follows the second strategy. To the light of our experimental results, full storage performs efficiently, since the number of stored states is limited by the number of events in each execution sequence and it remains quite low for the experiments.

## 6 FOR THE VERIFICATION OF SOFTWARE-DEFINED NETWORKS

To address the vii) challenge, we want to apply our techniques for the verification of Software-Defined Network (SDN). SDN is a relatively recent networking paradigm which is now widely used in industry, with many companies—such as Google and Facebook—using SDN to control their backbone networks and data-centers. The core principle in SDN is the separation of control and data planes—there is a centralized *controller* which operates a collection of distributed interconnected switches. *Hosts* communicate with each other by sending packets to their *switches*. Each switch checks if its own *flow table* contains the destination of the packet. If it does, the packet is sent to the next switch or to the final host. Otherwise, it sends a *message* to the controller in order to receive information about the destination of the packet. The controller answers by means of messages and dynamically updates switches' policies depending on the observed flow of packets, which is a simple but powerful way to react to unexpected events in the network.

Network verification has become increasingly popular since SDN was introduced, because in this new paradigm the amount of detailed information available about network events is rich enough and can be centrally gathered to check for properties of the network behavior. Moreover, the controller itself is a program which can be analyzed and verified before deployment.

We have encoded all basic components of an SDN network (switches, hosts, controller) into the message-passing concurrent language ABS [17]. Furthermore, we have formalized the semantics of SDN networks that allows us to prove soundness of the equivalence between such semantics and the semantics of our encoding. Furthermore, we have overcome one of the most challenging aspects to encode SDN networks, which is the *barrier* messages, special instructions used by the controller to force switches to execute all their messages previously received. We have built a model checker for our SDN models on top of SYCO (choosing the appropriate configuration). This tool uses the DPOR algorithms proposed in this



RIGHTSLINK

**Table 1: Experimental evaluation comparing SDPOR, DPOR$_{cs}$ and CDPOR**

| | SDPOR | | | DPOR$_{cs}$ | | | CDPOR | | | | Speed-up | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Bench.** | **Tr** | **S** | **T** | **Tr** | **S** | **T** | **Tr** | **S** | **T** | **T$^{smt}$** | **G$^{s}$** | **G$^{cs}$** | **G$^{smt}$** |
| Fib(7) | >13$k$ | >160$k$ | 60.0 | 1 | 551 | 0.3 | 1 | 82 | 0.05 | 0.12 | >1364 | 6 | 1.4 |
| Fib(8) | >8$k$ | >101$k$ | 60.0 | 1 | 2k | 0.7 | 1 | 134 | 0.12 | | >527 | 6 | 3.0 |
| Fib(9) | >4$k$ | >51$k$ | 60.0 | 1 | 3k | 2.8 | 1 | 218 | 0.25 | | >242 | 12 | 7.5 |
| Fib(10) | >2$k$ | >27$k$ | 60.0 | 1 | 8k | 11.5 | 1 | 354 | 0.69 | | >88 | 17 | 14.3 |
| Fib(14) | >10 | >3$k$ | 60.0 | >1 | >4$k$ | 60.0 | 1 | 3k | 42.67 | | >2 | >2 | >1.5 |
| QS(13) | 5k | 91k | 29.5 | 1 | 29k | 7.9 | 1 | 50 | 0.03 | 11.99 | 1474 | 395 | 0.7 |
| QS(15) | >7$k$ | >157$k$ | 60.0 | 1 | 115k | 42.6 | 1 | 58 | 0.05 | | >1500 | 1064 | 3.6 |
| QS(20) | >4$k$ | >98$k$ | 60.0 | >1 | >148$k$ | 60.0 | 1 | 78 | 0.04 | | >1539 | >1539 | >5.0 |
| QS(25) | >3$k$ | >96$k$ | 60.0 | >1 | >133$k$ | 60.0 | 1 | 98 | 0.06 | | >1017 | >1017 | >5.0 |
| QS(200) | >5 | >2$k$ | 60.0 | >1 | >87$k$ | 60.0 | 1 | 798 | 4.45 | | >14 | >14 | >3.7 |
| Pi(8) | >10$k$ | >105$k$ | 60.0 | 264 | 5k | 1.7 | 1 | 26 | 0.02 | 0.05 | >4616 | 128 | 26.9 |
| Pi(9) | >11$k$ | >120$k$ | 60.0 | 2k | 19k | 7.0 | 1 | 29 | 0.02 | | >4000 | 465 | 108.9 |
| Pi(10) | >10$k$ | >128$k$ | 60.0 | 6k | 91k | 45.2 | 1 | 32 | 0.02 | | >3530 | 2655 | 683.7 |
| Pi(12) | >9$k$ | >122$k$ | 60.0 | >7$k$ | >128$k$ | 60.0 | 1 | 38 | 0.03 | | >2400 | >2400 | >810.9 |
| Pi(20) | >5$k$ | >101$k$ | 60.0 | >5$k$ | >115$k$ | 60.0 | 1 | 62 | 0.09 | | >723 | >723 | >454.6 |
| PS(5) | 35k | 156k | 43.2 | 8 | 142 | 0.1 | 1 | 22 | 0.01 | 0.59 | 5391 | 5 | 0.1 |
| PS(6) | >32$k$ | >141$k$ | 60.0 | 72 | 2k | 0.4 | 1 | 29 | 0.02 | | >4286 | 28 | 0.7 |
| PS(7) | >29$k$ | >130$k$ | 60.0 | 2k | 28k | 7.5 | 1 | 37 | 0.03 | | >2858 | 357 | 12.3 |
| PS(9) | >25$k$ | >109$k$ | 60.0 | >11$k$ | >165$k$ | 60.0 | 1 | 56 | 0.06 | | >1053 | >1053 | >92.9 |
| PS(11) | >23$k$ | >103$k$ | 60.0 | >9$k$ | >132$k$ | 60.0 | 1 | 79 | 0.09 | | >690 | >690 | >88.8 |

thesis to avoid exploring redundant executions and incorporates visualization tools to view the exploration and execution diagram.

To evaluate this approach, we have modelled and analysed several SDN scenarios: a controller with load balancer, a SSH controller, a learning switch with authentication, and a firewall with migration. We have found bugs related to programming errors in the controller [7], forwarding loops [15] and violation of safety policies [7, 20]. SYCO needs to explore all possible reorderings of dependent messages and packets, leading to a combinatorial explosion. Thanks to the use of conditional independence within DPOR algorithms, SYCO can handle networks larger than in related systems [20], but without requiring simplifications to the SDN models. This is achieved by means of ICs that are satisfied if two messages or packets are independent (for instance, they do not access to the same entries of the flow table) and they can be proven automatically by using SMT solvers, as in [5]. In our current experiments, we have declared by-hand ICs which are valid for any SDN model. It is part of our future work to infer them automatically.

## 7 RELATED AND FUTURE WORK

The work in [18, 27] generated for the first time ICs for processes with a single instruction following some predefined patterns. This is a problem strictly simpler than our inference of ICs both in the type of IC generated (restricted to the patterns) and on the single-instruction blocks they consider. Furthermore, our approach using an AllSAT SMT solver is different from the CEGAR approach in [8]. The ICs are used in [18, 27] for SMT-based bounded model checking, an approach to model checking fundamentally different from our stateless model checking setting. Consequently ICs are used in a different way, in our case with no bounds on number of processes, nor derivation lengths, but requiring a uniformity condition on independence in order to ensure soundness.

It remains as future work the combination of Constrained DPOR and Context-Sensitive ODPOR with Observers. To the best of our knowledge, it has not been studied yet. We believe the integration of both techniques can achieve exponential gains compared with these approaches. Data-centric DPOR (DCDPOR) [11] presents a new DPOR algorithm based on a different notion of dependency according to which the equivalence classes of derivations are based on the pairs read-write of variables. Let us consider again the running example with the three processes $\{p, q, r\}$ and the initial state $x = 0$. In DCDPOR, we have only three different observation functions: $(r, x)$ (reading the initial value), $(r, p)$ (reading the value that $p$ writes), $(r, q)$ (reading the value that $q$ writes). Therefore, this notion of relational independence is finer grained than the traditional one in DPOR. However, DCDPOR does not consider conditional dependency, i.e., it does not realize that $(r, p)$ and $(r, q)$ are equivalent, and hence only two explorations are required. In conclusion, our approaches and DCDPOR can complement each other and it is in our agenda to study this integration.

## ACKNOWLEDGMENTS

This work was funded partially by the Spanish MECD FPU Grant FPU15/04313, the MINECO/FEDER, UE project TIN2015-69175-C4-3-R, the Spanish MINECO project TIN2015-69175-C4-2-R, the Spanish MICINN/FEDER, UE projects RTI2018-094403-B-C31 and RTI2018-094403-B-C33, and by the CM project P2018/TCS-4314.

## REFERENCES

[1] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Source sets: A foundation for optimal dynamic partial order reduction. *J. ACM*, 64(4):25:1–25:49, 2017.

[2] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos F. Sagonas. Optimal Dynamic Partial Order Reduction. In *POPL*, pages 373–384, 2014.

[3] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. Optimal dynamic partial order reduction with observers. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, pages 229–248, 2018.

[4] Elvira Albert, Puri Arenas, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey. Context-sensitive dynamic partial order reduction. In *CAV*, pages 526–543, 2017.

[5] Elvira Albert, Miguel Gómez-Zamalloa, Miguel Isabel, and Albert Rubio. Constrained Dynamic Partial Order Reduction. In CAV, volume 10982 of - *Lecture Notes in Computer Science*, pages 392–410. Springer, 2018.



RIGHTSLINK

[6] Elvira Albert, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, Miguel Isabel, and Peter J. Stuckey. Optimal Context-sensitive Dynamic Partial Order reduction with Observers. In *ISSTA*, 2019.
[7] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon: towards verifying controller programs in software-defined networks. In *PLDI*, pages 282–293, 2014.
[8] Kshitij Bansal, Eric Koskinen, and Omer Tripp. Commutativity condition refinement, 2015.
[9] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. The barcelogic SMT solver. In *CAV*, pages 294–298, 2008.
[10] Cristina Borralleras, Daniel Larraz, Albert Oliveras, José Miguel Rivero, Enric Rodríguez-Carbonell, and Albert Rubio. VeryMax: Tool description for termCOMP 2016. In *WST*, 2016.
[11] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Kapil Vaidya, and Nishant Sinha. Data-centric dynamic partial order reduction. In *POPL 2018*, 2018.
[12] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron A. Peled. State space reduction using partial order techniques. *STTT*, 2(3):279–287, 1999.
[13] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121, 2005.
[14] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer, 1996.
[15] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin T. Vechev. Sdnracer: concurrency analysis for software-defined networks. In *POPL*, pages 402–415, 2016.
[16] Shiyou Huang and Jeff Huang. Speeding up maximal causality reduction with static dependency analysis. In *ECOOP*, pages 16:1–16:22, 2017.
[17] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proc. FMCO'10*, LNCS 6957, pp. 142-164. Springer, 2012.
[18] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *CAV*, pages 398–413, 2009.
[19] Shmuel Katz and Doron A. Peled. Defining conditional independence using collapses. *TCS*, 101(2):337–359, 1992.
[20] Rupak Majumdar, Sai Deep Tetali, and Zilong Wang. Kuai: A model checker for software-defined networks. In *FMCAD*, pages 163–170, 2014.
[21] Antoni W. Mazurkiewicz. Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986. pages 279–324, 1986.
[22] Huyen T. T. Nguyen, César Rodríguez, Marcelo Sousa, Camille Coti, and Laure Petrucci. Quasi-optimal partial order reduction. *CoRR*, abs/1802.03950, 2018.
[23] César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based partial order reduction. In *CONCUR*, pages 456–469, 2015.
[24] Koushik Sen and Gul Agha. Automated Systematic Testing of Open Distributed Programs. In *FASE*, pages 339–356, 2006.
[25] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. TransDPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. In FMOODS/FORTE, volume 7273 of Lecture Notes in Computer Science, pages 219-234. Springer, 2012.
[26] Antti Valmari. Stubborn Sets for Reduced State Space Generation. In *Advances in Petri Nets*, pages 491–515, 1990.
[27] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole partial order reduction. In *TACAS*, pages 382–396, 2008.



RIGHTSLINK