Ten Years of Partial Order Reduction

Doron Peled*

Bell Laboratories

Abstract. Checking the properties of concurrent systems is an ever growing challenge. Along with the development of improved verification methods, some critical systems that require careful attention have become highly concurrent and intricate. Partial order reduction methods were proposed for reducing the time and memory required to automatically verify concurrent asynchronous systems. We describe partial order reduction for various logical formalisms, such as LTL, CTL and process algebras. We show how one can combine partial order reduction with other efficient model checking techniques.

1 Introduction

An important progress in formal verification was the introduction of model checking of finite state systems [6, 8, 36]. It allowed systems of certain types to be verified in a completely automatic way. Other techniques soon accompanied the basic model checking algorithms, allowing bigger and more complicated systems to be verified. Yet, it has been a constant challenge to verify concurrent systems with many independent components. The number of different states, representing the different values assigned to the variables of such systems, rapidly grows with the number of concurrent components. With the rapidly growing telecommunication and hardware industry, faster and cheaper computers became available; as a result, concurrent systems became more customary.

A seminal progress in attacking the intricacy of large concurrent systems was achieved by the use of binary decision diagrams (BDDs) [3, 4]. This data structure allows an efficient representation of states, such that certain logical operations can be performed on sets of states, rather than on a state-by-state basis. Symbolic model checking using BDDs was used to analyze systems with an impressively large number of states. The success of symbolic model checking was demonstrated mainly in verifying hardware systems. It was observed that BDDs tend to represent hardware circuits in a rather compact way. As a result, automatic verification technology has started to be integrated with hardware development and new industrial tools have been developed.

Even with the introduction of BDDs and symbolic model checking, software verification is still a challenging task. Concurrent programs tend to be less structured than hardware, as the basic units of software are cheaper to produce (e.g.,

* This survey was written while the author was visiting Carnegie Mellon University School of Computer Science Pittsburgh, PA 15213-3891, USA. Author's current address: Bell Laboratories, 700 Mountain Ave. Murray Hill, NJ 07974, USA

compare manufacturing a new adder circuit to writing a procedure for summing). An important difference between hardware and software is their mode of execution. Hardware is usually executed synchronously: all enabled concurrent units make progress at the same time, synchronized by some global clock. Software is usually executed asynchronously: concurrent units may execute independently and the result would be the same whether they execute simultaneously, or one at a time in any order.

Partial order reduction techniques [1, 5, 9, 12, 13, 14, 15, 17, 21, 23, 32, 33, 37, 38, 39, 40, 42] are based on this latter observation. Most formalisms, including logics such as LTL or CTL, and many process algebras, model the execution of concurrent systems as interleaved sequences, i.e., a total order between the occurrences of transitions. Thus, concurrently executed transitions create multiple executions that differ from each other only by their relative order of appearance. Since this order is usually uninteresting, or unobservable, most specifications do not distinguish between such executions. However, the existence of such different executions may contribute considerably to the state space explosion. Partial order reduction attempts to exploit the cases where the specification does not make such a distinction and allow performing model checking on smaller state spaces, based on a smaller number of executions.

2 Partial Order Reduction

A *finite transition system* is a fivetuple (S, S_0, T, AP, L) where S is a finite set of *states*, $S_0 \subseteq S$ are the *initial states*, T is a finite set of *transitions* such that each transition $\alpha \in T$ is a partial function $\alpha : S \mapsto S$, AP is a finite set of *propositions* and $L : S \mapsto 2^{AP}$ is the *assignment function*. An *execution* is an alternating sequence of states and transitions $s_0\alpha_0 s_1\alpha_1 \ldots$ such that $s_0 \in S_0$, and for each $i \geq 0$, $s_{i+1} = \alpha_i(s_i)$. Without loss of generality, we assume that an execution is always infinite. For each execution ξ we can define the following sequences:

- The states sequence $st(\xi) = s_0 s_1 s_2 \ldots$.
- The transitions sequence $tr(\xi) = \alpha_0\alpha_1\alpha_2 \ldots$.
- The propositions sequence $pr(\xi) = L(s_0)L(s_1)L(s_2) \ldots$.

A *segment* is a finite or infinite contiguous part of an execution.

A transition $\alpha \in T$ is *enabled* from a state s if $\alpha(s)$ is defined. That is, α can be applied to s, obtaining some successor state $s' = \alpha(s)$. Denote by $enabled(s)$ the set of states that are enabled from s. States based model checking techniques (including automata based algorithms) perform a search, often a depth first search (DFS), to explore the state space of the transition system. Then, some verification algorithms are applied to the state space. (In practice, these algorithms are usually applied to the state space *during* its construction. We defer the treatment of such on-the-fly algorithms to a later subsection.) The main principle of partial order reduction is to find a subset of the enabled transitions $ample(s) \subseteq enabled(s)$ that are used to generate the successors of a state s.

By choosing the subset of enabled transitions carefully, the correctness of the checked property (or the existence of a counterexample) is preserved between the full state space and the reduced one. It is important to notice that partial order reduction avoids generating the full state space, and constructs directly the reduced one.

Partial order reduction is based on several observations about the nature of concurrent computations and specification formalisms. The first observation is that concurrently executed transitions are often commutative. This is formalized in the definition of independence.

Definition 1. An *independence relation* $I \subseteq T \times T$ is symmetric and antireflexive. For each pair of independent transitions $(\alpha, \beta) \in I$ and state $s \in S$ such that $\alpha, \beta \in enabled(s)$, the following hold:

- $\alpha \in enabled(\beta(s))$ and $\beta \in enabled(\alpha(s))$. That is, independent transitions cannot disable each other.
- $\alpha(\beta(s)) = \beta(\alpha(s))$. That is, executing two enabled independent transition in any order result in the same global state.

Denote $D = (T \times T) \setminus I$. If $(\alpha, \beta) \in D$, we say that α and β are *dependent*. A refinement of this definition, allowing the independency between pairs of transitions to vary from state to state, can be used to further improve partial order reduction [13, 20] and will not be discussed here.

Consider a state s and two enabled independent transitions α and β. Let $r = \alpha(\beta(s))$. Then also $r = \beta(\alpha(s))$. If the specification only mentions the first and last states, there is no need to include both α and β in $ample(s)$. Otherwise, we need to consider the possibility that $L(\alpha(s))$ and $L(\beta(s))$ can be different from each other, and can even be distinct from $L(s)$ or $L(\alpha(\beta(s))$.

A second observation is that in many cases, only a few of the transitions can change, when executed, the truth values of the propositional variables [40].

Definition 2. A transition $\alpha \in T$ is *invisible* if for each $s, s' \in S$ such that $s' = \alpha(s)$, $L(s) = L(s')$.

When deciding the invisibility of a transition α is hard, one can conservatively assume that α is visible.

When a pair of independent transitions α, β are enabled at s and *at most one* of them is visible, we have one of the following cases:

α is invisible. $L(s) = L(\alpha(s))$, $L(\beta(s)) = L(\alpha(\beta(s))$.
β is invisible. $L(s) = L(\beta(s))$, $L(\alpha(s)) = L(\beta(\alpha(s))$.
α, β invisible. $L(s) = L(\alpha(s)) = L(\beta(s)) = L(\alpha(\beta(s))$.

In each one of these cases, there is at most one change when progressing from s to $r = \alpha(\beta(s))$. The difference between executing α before β or β before α in the first two cases amounts to stuttering, as defined below. Typical specifications cannot distinguish between two executions that are equivalent up to stuttering. This allows eliminating either α or β from $ample(s)$.

Definition 3. The *stutter removal operator* $\natural$ applied to a propositions sequence ρ results in a sequence $\natural(\rho)$ where each consecutive repetition of labeling is replaced by a single occurrence. Two propositions sequences σ, ρ are *equivalent up to stuttering* if $\natural(\sigma) = \natural(\rho)$. This is denoted by $\sigma \equiv_\natural \rho$.

For example, if $AP = \{p, q\}$, the finite sequences $\sigma = (p)(p,q)(p,q)(q)(q)(p,q)$ and $\rho = (p)(p)(p,q)(p,q)(p,q)(q)(p,q)$ are stuttering equivalent since $\natural(\sigma) = \natural(\rho) = (p)(p,q)(q)(p,q)$.

In the following sections we present reductions for several formalisms. In each case, the reduction is represented by a set of constraints that need to be enforced on selecting $ample(s)$ for a given state s. When $ample(s) = enabled(s)$, we say that s is *fully expanded.*

2.1 Reduction for LTL

Linear temporal logic (LTL) cannot distinguish between two stuttering equivalent sequences when disallowing the nexttime operator ('$\bigcirc$'). It is in fact argued that specifications *should* be closed under stuttering equivalence [24] and proved that LTL without the nexttime operator is exactly as expressive as stuttering closed first order monadic logic properties [34]. The following conditions for selecting the set $ample(s)$ when generating a reduced state space are based on DFS. We use the fact that during DFS, reaching a state that is already on the search stack implies closing a cycle. The partial order reduction generates a reduced state space such that for each execution in the full state space, there is a stuttering equivalent sequence in the reduced one.

C1 [13, 19, 32, 37] For every segment[2] starting from the state s, a transition that is dependent on some transition in $ample(s)$ cannot be executed before a transition from $ample(s)$.

To understand Condition **C1**, consider a suffix of an execution σ, starting at s. There are two possible cases:

Case 1. α is the first transition from $ample(s)$ on σ. Then, α is independent of all the transitions that precedes it on σ. By applying Definition 1 repeatedly, all the transitions on σ prior to α can be commuted with α, obtaining a segment σ'.

Case 2. No transition in $ample(s)$ occurs on σ. Then any $\alpha \in ample(s)$ is independent of all the transitions of σ. By Definition 1, one can form a segment σ' by executing α and then the transitions of σ.

Condition **C1** is quite abstract. Implementing it takes into account the particular mode of execution, e.g., shared variables, asynchronous or synchronous message passing [13, 14, 17, 40]. Consider for example an execution model with asynchronous message passing. Then the reduction can be implemented by searching

[2] Notice that the segment mentioned in **C1** are not necessarily constructed in the reduced state space.

for a set E of transitions belonging to a single process P. These transitions can be executed at the current location of P. To guarantee Condition **C1**, there should be no other transition α of type *receive* or *send*, originating at the same location of P and disabled due to an empty or full communication queue, respectively. The reason is that α is then dependent on the transitions in E (since $E \cup \{\alpha\}$ belong to the same process). By executing a sequence of independent transitions of other processes that end with a *send* or *receive* transition, respectively, α may become enabled.

In order for $pr(\sigma)$ and $pr(\sigma')$ will be stuttering equivalent (for both of the above cases) we enforce the following condition:

C2 [33] If s is not fully expanded then all of the transitions in $ample(s)$ are invisible.

Expanding $ample(s)$ from s instead of $enabled(s)$ can defer the execution of a transition $\beta \in enabled(s) \setminus ample(s)$. (Notice that β remains enabled in any state $\alpha(s)$ for $\alpha \in ample(s)$.) With only Conditions **C1** and **C2**, a transition can be deferred forever along a cycle. This may result in ignoring an execution that is not represented in the reduced state space by another stuttering equivalent execution, and can consequently lead to incorrect verification result. The following condition guarantee that no transition would be deferred forever.

C3 [32] If s is not fully expanded then for no transition $\alpha \in ample(s)$ it holds that $\alpha(s)$ is on the search stack.

There are different alternatives for condition **C2**, for example, Valmari [37] presented an algorithm for the following condition:

C3i For every cycle in the reduced state space there is at least one fully expanded node.

Another possibility is

C3ii [42] If a cycle contains a state where some transition $\alpha \in T$ is enabled, then it must also contain some state where α is taken.

It can be easily shown that **C3** implies **C3i**, which in turn implies **C3ii**. Using a stronger condition instead of a weaker one is less general and can be understood as an implementation of the weaker condition. When restricting the specification to safety properties, the following condition is sufficient:

C3iii [16] For at least one of the transitions $\alpha \in ample(s)$, $\alpha(s)$ is not on the search stack.

2.2 Reduction for CTL

The model for temporal logics such as CTL or CTL* is a branching structure. Even without the nexttime operator (the nexttime operator in these logics is

usually written as 'X'), two structures can have corresponding stuttering equivalent sequences but still be distinguished as they have different branching points. Thus, for branching temporal logics, we require that the partial order reduction generates a reduced state space that is *stuttering bisimilar* [2] to the full state space. Two states s and s' are related if the following conditions hold:

1. $L(s) = L(s')$,
2. for each infinite sequence σ starting from s there exists an infinite sequence σ' starting from s' such that σ and σ' can be partitioned into infinitely many finite blocks of consecutive states $B_0 B_1 \ldots$ and $B_0{}' B_1{}' \ldots$, respectively and the states in B_i are stuttering bisimilar to the states in $B_i{}'$ for each $i \geq 0$, and
3. similarly, for each sequence σ' from s' there exists a blockwise matching path σ from s.

It is shown in [2] that CTL and CTL* without the nexttime operator cannot distinguish between stuttering bisimilar structures. Stuttering bisimilarity between the full and reduced state space is achieved by adding the following constraint:

C4 [9] If s is not fully expanded, then $ample(s)$ contains exactly one transition.

2.3 Reduction for process algebra

The focus in process algebras is on the branching structure of states and the execution of transitions. The model for various process algebras usually impose labeling the transitions rather than the states. A transition labeled with τ is considered invisible, regardless of its effect on the state. Process algebras are usually based on simulation relations. Such relations associate corresponding pairs of states that have similar branching structure. Stuttering bisimulation was discussed above. Other relations for which we can apply partial order reduction are *branching bisimulation* [11, 29] and *weak bisimulation* [28].

The conditions **C1**–**C4** can be applied to produce a reduced structure that is branching bisimilar [9] and thus also weak bisimilar. One concern is that in process algebras transitions are often nondeterministic. To allow nondeterminism in partial order reduction, one can reformulate Condition **C4** as follows:

C4i [40] If s is not fully expanded, then $ample(s)$ consists of one deterministic transition.

Thus, nondeterministic transitions are allowed in ample sets of nodes that are fully expanded.

2.4 Reduction under fairness

In many systems, the execution of concurrent components is constrained by some fairness assumption. For example, it is natural to require that if a concurrent process *can* execute some transition, independently of other processes, then it is

eventually allowed to do so. Model checking under fairness is modified to check whether the fair executions satisfy the given specification [25].

For partial order reduction, the following 'weak' fairness (or 'justice' [26]) assumption is quite natural:

F if an operation α is enabled from some state of an execution, then some operation that is dependent on α must appear later in this execution.

The reduction is based now on the following equivalence relation between sequences:

Definition 4. Given an independence relation I, two finite transitions sequences u and v are *trace equivalent* [27], denoted $u \equiv_{tr} v$, if there exists a sequence $u = w_1, w_2, \ldots, w_n = v$ such that for each $1 \leq i < n$, there exists some $x, y \in T^*$ and independent transitions $(\alpha, \beta) \in I$ such that $w_i = x\alpha\beta y$ and $w_{i+1} = x\beta\alpha y$.

Thus, $u \equiv_{tr} v$ iff v can be obtained from u by repeatedly commuting adjacent transitions. The trace equivalence relation can be extended to infinite traces in the following way: $u \equiv_{tr} v$ iff for every finite prefix u' of u there exists a finite prefix v' of v such that $u'w \equiv v'$ for some sequence $w \in T^*$. The symmetric condition, replacing u with v, must also hold.

In fact, the origin of the term 'partial order reduction' is due to the use of trace equivalence. One can view trace equivalence as a partial order semantics. Consider the events obtained by taking the *occurrences* of transitions in an execution, e.g., the first appearance of α denoted $\langle \alpha, 1 \rangle$ and the second denoted $\langle \alpha, 2 \rangle$. Now, consider a partial order between occurrences of transitions on a trace equivalence class. (It can be easily checked that all trace equivalent sequences have the same occurrences.) Then define the order $\prec$ between occurrences such that $e_1 \prec e_2$ when e_1 preceded e_2 on all the equivalent sequences. This order can easily be checked to be a partial order, i.e., asymmetric, irreflexive and transitive. Occurrences that can appear in both orders in different equivalent sequences are unordered by $\prec$ and are considered concurrent.

Consider the case where the checked property φ is closed under trace equivalence. That is, it cannot distinguish between two executions by having $\sigma \models \varphi$ and $\rho \not\models \varphi$, while $tr(\sigma) \equiv_{tr} tr(\rho)$. Assuming **F**-fairness, **Case 2** of Section 2.1 cannot happen. Then it is sufficient to apply Conditions **C1** and **C3**; the obtained reduced state space includes at least one sequence for each trace equivalence class.

Checking that an LTL property is closed under trace equivalence [35] may be unnatural: LTL usually refers to the states, whereas trace equivalence relates executions according to their executed transitions. Instead, it is possible to connect trace equivalence to stuttering equivalence, supplying a condition that guarantees that every pair of sequences σ and ρ such that $tr(\sigma) \equiv_{tr} tr(\rho)$ satisfies that $st(\sigma) \equiv_{\natural} st(\rho)$. One way to enforce this is by requiring the following:

D1 [32] Extend the dependency relation D to include every pair of visible transitions.

It is important to note that the fairness assumption is still defined with respect to the original dependency relation and not the extended one. Extending the dependency relation limits the reduction. One way to relax Condition **D1** is to write the checked property, when possible, as a boolean combination $\varphi = \bigwedge_i \bigvee_j \varphi_{i,j}$. We can refine visibility such that $vis(\alpha, p)$ for $\alpha \in T$, $p \in AP$ holds when the truth value of p may change by executing α. Then we require the following:

D2 [32] Extend the dependency relation D to include every pair of transitions α, β such that $vis(\alpha, p)$ and $vis(\beta, q)$, and p, q both appear in some boolean component $\varphi_{i,j}$.

Weaker definitions, which require fewer dependencies to be added, appear in [18, 30].

Checking a property φ under a fairness condition that is stronger than **F**, e.g., strong fairness [26], can be done in the following way. The fairness condition is writen as a formula ψ, and dependencies are added to D according to **D2** (or a variant of it) as if the property $\psi \rightarrow \varphi$ is checked. However, the checked property is still φ, while the model checking algorithm checks the executions that satisfies ψ. Model checking algorithms that assume various fairness constraints appear in [25]. Typically, ψ is written as a large boolean combination, containing predicates related to the enabledness and execution of each transition. Thus, using Condition **D1** instead of **D2** would not result in any reduction.

2.5 On-the-fly model checking

In practice, model checking does not include a separate stage where the full or reduced state space is first generated before it is being analyzed. The analysis of the state space can coincide with its construction [7, 22, 41]. With this *on-the-fly* approach, if a counterexample is found, there is no further need to complete the construction of the state space. This observation has a potential of considerably reducing the memory size and time required for the verification. One way of performing on-the-fly model checking is to represent the state space as an automaton $\mathcal{A}$, recognizing the executions of the checked system. The checked property φ is also represented as an automaton. In fact, one generally uses an automaton $\mathcal{B}$ that accepts the sequences that do not satisfy φ (by a direct translation of $\neg\varphi$) [10, 41]).

The intersection of $\mathcal{A}$ and $\mathcal{B}$ is an automaton recognizing executions of the system that do not satisfy the specification. Such executions exist iff the property φ is not satisfied by the system, and can be presented as counterexamples. Specifically, with on-the-fly model checking one can combine the following [33, 39]:

- the construction of an automaton $\mathcal{A}$ that corresponds to the *reduced* state space,
- the intersecting with the automaton $\mathcal{B}$, and
- checking for the emptiness of the intersection.

The crucial change from the off-line partial order reduction presented in Section 2.1 is with respect to Condition **C3**. The cycles found during an on-the-fly construction are cycles of the automaton for $\mathcal{A} \cap \mathcal{B}$, rather than of the state space automaton $\mathcal{A}$. It can be shown [33] that relativizing **C3** to these cycles, i.e., fully expanding a state when it closes a cycle in the intersection of $\mathcal{A}$ and $\mathcal{B}$, still preserves the correctness of the algorithm. In fact, in the intersection, cycles are going to be larger (they may include several iterations of the cycles of the state space, coupled with different values for the property automaton). On the other hand, since the reduced state space, as represented by the automaton $\mathcal{A}$, contains fewer executions, some of the counterexamples may not be included, deferring the discovery of a counterexample.

2.6 Symbolic model checking

By combining various verification techniques, one may obtain the benefits offered by each one of them separately. Obvious candidates for such a combination are partial order reduction and symbolic model checking. The main problem is that the former is usually implemented using a DFS procedure that handles one state at a time, while the latter is described using a fixpoint computation that involves many states at the same time. This effects Condition **C3** (or one of its variants). One proposal for changing this condition for symbolic model checking is that the fixpoint computation can be seen as a breadth first search (BFS) [1]. With each successive fixpoint approximation, a new layer of states with further distance from the original ones are discovered. The cycle closing condition can then be relativized to BFS by pessimistically assuming that states in a new layer that also appeared in previous layers are closing a cycle [5].

Another solution is based on the observation that each cycle of the state space must be composed of several local cycles of the separate concurrent processes. The local structures of the processes are analyzed and at least one transition from each local cycle is selected. The selected transitions are called *sticky transitions*, and the following condition is imposed:

C3iv [23] If s is not fully expanded then no transition $\alpha \in \mathit{ample}(s)$ is sticky.

It can easily shown that Condition **C3iv** implies **C3ii**. With this new condition, the need to find when a cycle is closed during the state space exploration is eliminated. One can in fact combine Conditions **C3iv** with **C2** as follows:

C2+3 If s is not fully expanded then no transition $\alpha \in \mathit{ample}(s)$ can be sticky or visible.

Sticky transitions decrease the reduction and thus their number need to be minimized. One observation is that there are some dependencies between local cycles of different processes. If one local cycle includes only local operations and receiving messages, another local cycle that includes sending messages must also be included to form global cycle of the state space. Similarly, if one local cycle only decreases the value of a variable, a local cycle of another process that increases

it is also needed to complete a global cycle. Thus, local cycles that change some resource in a monotonic way can be exempted from the search for sticky transitions (but not at the same time with cycles that change it in the complementary way).

2.7 Reducing visibility

Experimental results [17] show that the reduction decreases rapidly with the number of visible transitions. One way to reduce the effect of visibility on partial order reduction is to let it dynamically decrease while checking the specification [21]. We will illustrate this with an example. Suppose that the property to be checked is $\varphi = \Box(p \rightarrow \Box q)$. The negation of the property is $\neg\varphi = \Diamond(p \wedge \Diamond\neg q)$. Once the automaton $\mathcal{B}$ constructed for $\neg\varphi$ has encounted a state where p holds, it may concentrate on checking $\Diamond\neg q$.

In this case, one may start the reduction by considering visible transitions with respect to all the propositions that appear in the formula. In this case, the relevant propositions are $\{p, q\}$. Then, once p occurs, we can then reduce the visible transitions to those that can affect the truth value of q, which is the only proposition that appears in $\Diamond\neg q$. Those transitions that can effect p but not q can now be considered invisible. An LTL translation algorithm that produces an automaton $\mathcal{B}$ that allows monotonically reducing the set of visible transitions as $\mathcal{B}$ executes appears in [10].

Acknowledgement The author would like to thank Marius Minea for carefully reading the paper and many useful comments.

References

1. R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani, Partial order reduction in symbolic state space exploration. In *Proceedings of the Conference on Computer Aided Verification (CAV'97)*, Haifa, Israel, June 1997.
2. M.C. Browne, E.M. Clarke, O. Grümberg, Characterizing finite Kripke structures in propositional temporal logic, *Theoretical Computer Science* 59 (1988), Elsevier, 115–131.
3. R.E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Transactions on Computers, C-35(8), 1986, 677–691.
4. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang, Symbolic model checking: 10^{20} states and beyond, Information and Computation, 98 (1992), 142–170.
5. C.T. Chou, D. Peled, Verifying a model-checking algorithm, *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1055, Springer, 1996, Passau, Germany. 241–257.
6. E.M. Clarke, E.A. Emerson, Design and synthesis of synchronous skeletons using branching time temporal logic, *Logic of Programs*, Yorktown Heights, NY, LNCS 131, Springer, 1981, 52–71.
7. C. Courcoubetis, M.Y. Vardi, P. Wolper, M, Yannakakis, Memory-efficient algorithms for the verification of temporal properties, *Formal methods in system design* 1 (1992) 275–288.

8. E.A. Emerson, E.M. Clarke, Characterizing correctness properties of parallel programs using fixpoints, Automata, Languages and Programming, LNCS 85, Springer, 1980, 169–181.
9. R. Gerth, R. Kuiper, W. Penczek, D. Peled, A partial order approach to branching time logic model checking, ISTCS'95, *3rd Israel Symposium on Theory on Computing and Systems*, IEEE press, 1995, Tel Aviv, Israel, 130-139. A full version was accepted to *Information and Computation.*
10. R. Gerth, D. Peled, M.Y. Vardi, P. Wolper, Simple on-the-fly automatic verification of linear temporal logic, *PSTV95, Protocol Specification Testing and Verification*, Chapman & Hall, 1995, Warsaw, Poland, 3–18.
11. R.J. van Glabbeek, W.P. Weijland, Branching time and abstraction in bisimulation semantics, *Information Processing 89.* Elsevier Science Publishers, 1989, 613–618.
12. P. Godefroid. Using partial orders to improve automatic verification methods. In Proc. *2nd Workshop on Computer Aided Verification*, LNCS 531, Springer, New Brunswick, NJ, 1990, 176–185.
13. P. Godefroid, D. Pirottin, Refining dependencies improves partial order verification methods, *5th Conference on Computer Aided Verification*, LNCS 697, Elounda, Greece, 1993, 438–449.
14. P. Godefroid, D. Peled, M. Staskauskas, Using partial order methods in the formal validation of industrial concurrent programs, 1996, ISSTA'96, *International Symposium on Software Testing and Analysis*, ACM Press, San Diego, California, USA, 261-269.
15. P. Godefroid, P. Wolper, A Partial approach to model checking, *6th Annual IEEE Symposium on Logic in Computer Science*, 1991, Amsterdam, 406–415.
16. G.J. Holzmann, P. Godefroid, D. Pirottin, Coverage preserving reduction strategies for reachability analysis, Proc. 12th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP, Orlando, Florida, 1992, 349–363.
17. G.J. Holzmann, D. Peled, An improvement in formal verification, *7th International Conference on Formal Description Techniques*, Berne, Switzerland, 1994, 177–194.
18. S. Jha, D. Peled, Generalized stuttering equivalence for linear temporal logic specification, Submitted for publication.
19. S. Katz, D. Peled, Verification of distributed programs using representative interleaving sequences, *Distributed Computing* 6 (1992), 107–120. A preliminary version appeared in Temporal Logic in Specification, UK, 1987, LNCS 398, 21–43.
20. S. Katz, D. Peled, Defining conditional independence using collapses, Theoretical Computer Science 101 (1992), 337-359, a preliminary version appeared in *BCS–FACS Workshop on Semantics for Concurrency*, Leicester, England, July 1990, Springer, 262–280.
21. I. Kokkarinen, A. Valmari, D. Peled, Relaxed visibility enhances partial order reduction, CAV'97, June 1997, Israel, LNCS 1254, 328–339.
22. R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, Princeton, New Jersey, 1994.
23. R.P. Kurshan, V. Levin, M. Minea, D. Peled, H. Yenigün, Static partial order reduction, 345–357, 1997.
24. L. Lamport, What good is temporal logic, in R.E.A. Mason (ed.), *Information Processing '83: Proc. of the IFIP 9th World Computer Congress,*, Paris, France, North-Holland, Amsterdam, 1983, 657–668.
25. O. Lichtenstein, A. Pnueli, Checking that finite-state concurrent programs satisfy their linear specification, Proceedings of the 11th Annual Symposium on Principles of Programming Languages, ACM Press, 1984, 97–107.

26. Z. Manna, A. Pnueli, How to cook a temporal proof system for your pet language. Proceedings of the Symposium on Principles on Programming Languages, Austin, Texas, 1983, 141–151.
27. A. Mazurkiewicz, Trace theory, *Advances in Petri Nets 1986*, Bad Honnef, Germany, LNCS 255, Springer, 1987, 279–324.
28. R. Milner, *A calculus of communicating system*, LNCS, Springer, 92.
29. R. de Nicola, F. Vaandrager, Three logics for branching bisimulation, *Logic in Computer Science '90*, IEEE, 1990, 118–129.
30. D. Peled, On projective and separable properties, *Theoretical Computer Science*, 186(1-2), 1997, 135-155.
31. D. Peled, A. Pnueli, Proving partial order properties, *Theoretical Computer Science*, 126(1994), 143–182.
32. D. Peled, All from one, one for all, on model-checking using representatives, *5th Conference on Computer Aided Verification*, Greece, 1993, LNCS, Springer, 409–423.
33. D. Peled, Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design* 8 (1996), 39–64. A preliminary version appeared in *Computer Aided Verification 94*, LNCS 818, Springer, Stanford, USA, 377-390.
34. D. Peled, Th. Wilke, Stutter-invariant temporal properties are expressible without the nexttime operator, *Information Processing Letters* 63 (1997), 243–246.
35. D. Peled, Th. Wilke, P. Wolper, An algorithmic approach for checking closure properties of ω-Regular Languages, *CONCUR'96, 7th International Conference on Concurrency Theory*, Piza, Italy, LNCS 1119, Springer, August 1996, 596–610. A full version accepted to *Theoretical Computer Science*.
36. J.P. Quielle, J. Sifakis, Specification and verification of concurrent systems in CESAR, Proceedings of the 5th International Symposium on Programming, 1981, 337–350.
37. A. Valmari, Stubborn sets for reduced state space generation, *10th International Conference on Application and Theory of Petri Nets*, Bonn, Germany, 1989, LNCS 483, Springer, 491–515.
38. A. Valmari, A stubborn attack on state explosion. *Formal Methods in System Design*, 1 (1992), 297–322.
39. A. Valmari, On-the-fly verification with stubborn sets, Proceedings of CAV '93, 5th International Conference on Computer-Aided Verification, Elounda, Greece, LNCS 697, Springer 1993, pp. 397-408.
40. A. Valmari, Stubborn set methods for process algebras, *POMIV'96, Partial Orders Methods in Verification*, American Mathematical Society, DIMACS, Princeton, NJ, USA, 1996, 213–232.
41. M.Y. Vardi, P. Wolper, An automata-theoretic approach to automatic program verification, *1st Annual IEEE Symposium on Logic in Computer Science*, 1986, Cambridge, England, 322–331.
42. B. Willems, P. Wolper, Partial-order methods for model-checking: from linear time to branching time, *11th Annual IEEE Symposium on Logic in Computer Science*, New Brunswick, NJ, USA, 1996, 294-303.