Partial-order verification in SPIN can be more efficient

Hans van der Schoot

Department of Computer Science
University of Ottawa
Ottawa, Ontario K1N 6N5, Canada

vdschoot@csi.uottawa.ca

Abstract

Partial-order reduction methods form a collection of state exploration techniques set to relieve the state-explosion problem in concurrent program verification. One such method is implemented in the verification tool SPIN. Its use often reduces significantly the memory and time needed for verifying local and termination properties of concurrent programs and, moreover, for verifying that concurrent programs satisfy their linear temporal logic specifications (i.e. for LTL model-checking). This paper builds on SPIN's partial-order reduction method to yield an approach which enables further reductions in space and time for verifying concurrent programs.

Keywords

Concurrency, program correctness, model-checking, partial-order reduction, temporal logic

1 Introduction

Partial-order reduction methods [2-4, 7, 10, 19, 20, 24-26] form a collection of state exploration techniques set to relieve the state-explosion problem in concurrent program verification. The main observation underlying these methods is that in many cases the properties verified are insensitive to the interleaving order of concurrent program operations. Therefore, following just an arbitrary order among concurrent operations, by exploring from each generated state only a *subset* of the successor states instead of all of them, provides a means for reducing the size of the state space that needs to be analyzed.

Partial-order reduction methods have proved adequate for verifying local and termination properties of concurrent programs [2, 3, 10, 24] and, moreover, for verifying linear-time temporal logic (LTL) properties [4, 7, 19, 20, 25]. The latter is usually referred to as LTL model-checking and captures arbitrary safety and liveness properties of concurrent programs [27]. Experiments have indicated that in many cases partial-order methods can substantially reduce the space and time needed for verification. Also, these methods have been shown to combine well with verification on-the-fly [4, 20, 25] and with verification under certain fairness conditions [19, 20].

Notwithstanding these results, this paper proposes an enhancement of partial-order reduction methods, in particular of the method described in [20, 7]. An approach is presented which builds on the concepts of [20, 7] to enable a further reduction in space and time for verifying local and termination properties, and for (on-the-fly) LTL model-checking. The idea behind the additional reduction is surprisingly simple: rather than fixing an arbitrary interleaving order among concurrent program operations, as do partial-order reduction methods, one can abstain from any order altogether by executing them collectively as sets. This then mimics a truly concurrent execution of these operations. There are several reasons for taking up the partial-order reduction method in [20, 7]. First, it is implemented as an extension to the model-checker SPIN [7]. SPIN is a verification tool for programs specified in the language Promela [6] and is increasingly being used for teaching and for industrial applications. In addition, this partial-order reduction method is advocated as the most advanced in terms of the properties that can be checked, the way fairness is treated, and the low overhead and high overall performance of its implementation [7, 20].

The idea of executing sets of concurrent operations was first introduced by Rubin and West [21] for a protocol model, known as the CFSM model, in which processes communicate asynchronously over FIFO queues. They proposed a reduced reachability analysis technique for verifying the absence of deadlocks and unspecified receptions in protocols with two processes. This technique has evolved under the name fair reachability analysis for the verification of two additional properties, viz. the absence of non-executable transitions and unbounded communication [5, 13], and for protocols with more processes in a restricted communication topology [13, 14]. Itoh and Ichikawa [9] similarly employed the execution of sets of concurrent operations in the same model to verify protocols with an arbitrary number of processes and arbitrary communication topology, but with restricted process structures. The idea was ultimately generalized by Özdemir and Ural [17, 18] to protocols in the CFSM model with no structural constraints at all. The proposed reduction technique is called simultaneous reachability analysis and can be used to verify the absence of deadlocks, non-executable transitions, unspecified receptions and buffer overflows. An improvement of this technique has already followed as well to further reduce the space and time requirements for verifying the same four properties [22, 23]. Özdemir [16] also adapted simultaneous reachability analysis for verifying (on-the-fly) arbitrary safety properties of concurrent programs in which processes synchronize on common actions (i.e. operations with the same label). The relation between his approach and the one proposed here is discussed at the end of this paper.

The paper is organized as follows. Section 2 gives preliminary concepts and notations, most of which are adopted directly from [20] for ease of presentation. Section 3 outlines the partial-order model-checking method as described in [20]. Section 4 explains the proposed enhancement and states the main theoretical results. The proofs of the corresponding lemmas and theorems are given in the Appendix. Section 5 reports on the experimental results of a comparative study. Finally, Section 6 summarizes our contribution and advocates additional characteristics of our approach including its pertinence to partial-order reduction methods in general.

2 Preliminaries

A *finite-state program* P is a triple $\langle T, Q, \iota\rangle$, where T is a finite set of *operations*, Q is a *finite* set of *states* and $\iota \in Q$ is the *initial state* of P. The enabling condition $\mathrm{en}_\alpha \subseteq Q$ of an operation $\alpha \in T$ is the set of states at which α can be executed; α is said to be *enabled at* a state $x \in \mathrm{en}_\alpha$. The set of all operations enabled at x is denoted by $\mathrm{en}(x)$ and the set of operations enabled at x of a distinguished process P_i in the program is denoted by $\mathrm{en}_i(x)$. Each operation $\alpha \in T$ is viewed as a partial transformation $\alpha : Q \mapsto Q$ which must be defined at least for each $x \in \mathrm{en}_\alpha$.

A *computation* of a program is a finite or infinite sequence of operations $\mathtt{v} = \alpha_1\alpha_2\ldots$ of length $|\mathtt{v}|$ (ω when $\mathtt{v}$ is infinite) that *generates* the sequence of states $\xi = x_0x_1x_2\ldots$ from Q, such that (1) $x_0 = \iota$, (2) for all $0 \leq i < |\mathtt{v}|$, $x_i \in \mathrm{en}_{\alpha_{i+1}} \wedge x_{i+1} = \alpha_{i+1}(x_i)$, and (3) either ξ is infinite or its last state $x_{|\mathtt{v}|}$ satisfies $x_{|\mathtt{v}|} \notin \bigcup_{\alpha\in T}\mathrm{en}_\alpha$. A computation or any segment thereof can thus be seen as a sequence of executed operations from T or as a sequence of states from Q. For a sequence of operations $\mathtt{v}$, $\mathrm{op}(\mathtt{v})$ denotes the set of operations in $\mathtt{v}$ and, when $\mathtt{v}$ is finite, $\mathrm{fin}_\mathtt{v}$ denotes the last state generated by $\mathtt{v}$.

The *full state graph* of a program $P = \langle T, Q, \iota\rangle$ is a directed graph $G = \langle V, E\rangle$, with $V = Q$ and E a finite set of edges labeled by operations from T, such that $x \xrightarrow{\alpha} x' \in E$ iff $x \in \mathrm{en}_\alpha$ and $x' = \alpha(x)$. The full state graph represents the computations of P. It *generates* a sequence of operations $\alpha_1\alpha_2\ldots$ (or the corresponding sequence of states) if there exists a (finite or infinite) path starting with ι whose edges are labeled $\alpha_1\alpha_2\ldots$. A standard algorithm for computing the full state graph of a program implements a depth-first search (DFS). It recursively expands *all* successor states of *all* states encountered during the search, starting at the initial state of the program, by executing *all* enabled operations at these states.

A *dependency relation* of a program is a reflexive and symmetric relation $D \subseteq T \times T$ such that for each pair of operations $(\alpha, \beta) \notin D$ (i.e. α and β are *independent*) it holds that for all $x \in Q$: (1) if $x \in \mathrm{en}_\alpha$, then $x \in \mathrm{en}_\beta$ iff $\alpha(x) \in \mathrm{en}_\beta$, and (2) if $x \in \mathrm{en}_\alpha \cap \mathrm{en}_\beta$, then $\alpha(\beta(x)) = \beta(\alpha(x))$. The dependency relation D induces an equivalence relation among sequences of operations. First, for *finite* sequences $\mathtt{v}$ and $\mathtt{v}'$, $\mathtt{v}$ is equivalent to $\mathtt{v}'$, denoted by $\mathtt{v} \equiv_D \mathtt{v}'$, iff $\mathtt{v}$ can be obtained from $\mathtt{v}'$ by repeatedly permuting adjacent independent operations [15]. The relation '$\equiv_D$' is then extended to *infinite* sequences as follows [20]. Let $\mathrm{Pref}(\mathtt{w})$ be the set of finite prefixes of a (finite or infinite) sequence $\mathtt{w}$, and define $\mathtt{v} \preceq_D \mathtt{v}'$ iff $\forall \mathtt{u} \in \mathrm{Pref}(\mathtt{v})\ \exists \mathtt{w} \in \mathrm{Pref}(\mathtt{v}')\ \exists \mathtt{z} \in T^*\ (\mathtt{w} \equiv_D \mathtt{z} \wedge \mathtt{u} \in \mathrm{Pref}(\mathtt{z}))$. For infinite sequences $\mathtt{v}$ and $\mathtt{v}'$, $\mathtt{v} \equiv_D \mathtt{v}'$ iff $\mathtt{v} \preceq_D \mathtt{v}'$ and $\mathtt{v}' \preceq_D \mathtt{v}$. Equivalence classes resulting from '$\equiv_D$' are called *traces*. A trace σ is denoted by $[\mathtt{v}]_D$, where $\mathtt{v}$ is any member of σ (the index D is omitted when clear from the context). When $\mathtt{v}$ is finite, all sequences of operations equivalent to $\mathtt{v}$ yield the same last state and hence $\mathrm{fin}_\mathtt{v}$ may be used also to denote $\mathrm{fin}_{[\mathtt{v}]}$. Concatenation of a finite trace $\sigma = [\alpha_1\alpha_2\ldots\alpha_n]$ and a finite or infinite trace $\sigma' = [\beta_1\beta_2\ldots\beta_m \ldots]$ is defined as $\sigma\sigma' = [\alpha_1\alpha_2\ldots\alpha_n\beta_1\beta_2\ldots\beta_m\ldots]$. A trace σ is *subsumed* by a trace ρ, denoted as $\sigma \trianglelefteq_D \rho$, if $\sigma = [\mathtt{v}]$, $\rho = [\mathtt{v}']$ and $\mathtt{v} \preceq_D \mathtt{v}'$. A *run* π of a program P with respect to (wrt) a dependency relation D is a trace containing computations of P. Thus, a run is finite iff none of its sequences can be extended with another operation.

LTL formulas assert properties of infinite sequences of states. An LTL formula (in the context of a program P) is constructed from boolean propositions on program states, the boolean operators '$\wedge$', '$\vee$' and '$\neg$', and the temporal operators '$\Box$' (always), '$\Diamond$' (eventually), 'U' (until) and '$\bigcirc$' (next-time). Without the operator '$\bigcirc$' an LTL formula is called *nexttime-free* and is then *stuttering closed*, meaning that it cannot distinguish between two *stuttering equivalent* sequences [11]. Two sequences are stuttering equivalent (wrt a formula φ) if one sequence can be obtained from the other by replacing in it every finite adjacent number of occurrences of the same (wrt φ) program state with a single occurrence. It has been shown that both safety and liveness properties of concurrent programs can be expressed by nexttime-free LTL formulas, and that any such formula can be formalized as a nondeterministic Büchi automaton [27]. A Büchi automaton is a tuple $B = \langle S, \Sigma, \Delta, s^0, F\rangle$ with a set of states S, an alphabet Σ, a transition relation $\Delta \subseteq S \times \Sigma \times S$, an initial state $s^0 \in S$, and a set of acceptance states $F \subseteq S$. For an LTL formula φ wrt a program P, each transition in the corresponding Büchi automaton B_φ is labeled by a boolean proposition on the states of P. B_φ *accepts* an infinite sequence of states ξ of P iff there is an infinite path p in B_φ starting from s_φ^0 such that (1) for all $i \geq 1$, the label of the i-th edge/transition in p holds true in the i-th state of ξ, and (2) some state of F_φ appears infinitely often in p. Note that also finite sequences can be considered, viewing each finite sequence as an infinite sequence by letting its last state repeat forever [20, 25]. Accordingly, a (finite or infinite) computation v *satisfies* φ iff B_φ accepts the infinite sequence of states generated by v. A set vis(φ) of *visible* operations is associated with φ containing the program operations that can change the truth value of some proposition in φ [25].

3 Partial-order model-checking

In [20], Peled presents a method that exploits partial-order reductions for nexttime-free LTL model-checking. The method comes in four different "modes", depending on whether model-checking is done off-line or on-the-fly (i.e. after or during the expansion algorithm, respectively), and with or without certain fairness assumptions. In this paper, the focus is primarily on the off-line and on-the-fly versions *without* fairness assumptions. Incorporating fairness assumptions is briefly discussed in Section 6, which also addresses alternative methods for partial-order model-checking [4, 25, 26].

3.1 Model-checking off-line

Partial-order model-checking is described in [20] by three constraints on selecting an appropriate *subset* of enabled operations to be executed at a given program state. For the off-line version without fairness assumptions, when a state x is expanded and at least one operation is enabled at x, a non-empty subset ample(x) of en(x) is used to generate successors for x such that [20]:

- **C1** No operation $\alpha \in T \setminus$ ample(x) that is dependent on an operation in ample(x) can be executed after reaching the state x and before an operation in ample(x) is executed;
- **C2** If ample(x) is a *proper* subset of en(x), then no operation $\alpha \in$ ample(x) is such that the state $\alpha(x)$ is on the DFS stack (i.e. executing α at x does not close a cycle);
- **C3** If ample(x) is a *proper* subset of en(x), then no operation in it is visible (wrt the checked formula φ).

The algorithm given in [20] for calculating "ample sets" enforces **C1** to **C3** as follows. Based on the fact that enabled operations of a single process cannot be independent, it aims at finding some process in the program P whose set of operations enabled at the state x satisfy **C1** to **C3**. As soon as such a process P_i is found, en$_i$(x) is returned as ample(x). If no such process exists, the algorithm returns the set en(x) of all operations enabled at x for en(x) trivially qualifies as an ample set. Most of the information required for checking **C1** and **C3** is gathered efficiently by a static analysis of the program before state exploration. This is explained in detail in [7]. Condition **C2** is checked during state exploration by inspecting the current contents of the DFS stack.

Off-line partial-order model-checking in [20, 7] thus proceeds as a modified DFS using the above algorithm for calculating ample sets to determine for each generated state the subset of successor states that need be explored. This is proved to yield a *reduced* state graph G' which generates for each sequence of states corresponding to a computation of a program P at least one sequence stuttering equivalent to it. Hence, when a property φ is stuttering closed, φ holds in all the sequences generated by G' iff it holds for all the computations of P. Algorithms for LTL model-checking [12] can then be applied directly to G' rather than to the full state graph of P.

3.2 Model-checking on-the-fly

When model-checking is performed on-the-fly, a program P is examined *during* rather than after the construction of its state space. This involves in practice computing the synchronous product $G \times B_{\neg\varphi}$ of the full state graph G of P and a Büchi automaton $B_{\neg\varphi}$ formalizing the negation of the checked formula φ [1, 6, 27]. Each transition of this product is of the form $\langle x, y\rangle \xrightarrow{\langle\alpha,\mathbf{P}\rangle} \langle x', y'\rangle$, where $x \xrightarrow{\alpha} x'$ is an edge/transition of G and $y \xrightarrow{\mathbf{P}} y'$ a transition of $B_{\neg\varphi}$ such that proposition $\mathbf{P}$ is true in program state x. Its initial state is $\langle\iota, s^0\rangle$ and the acceptance states are those composite states whose second components are acceptance states in $B_{\neg\varphi}$. $G \times B_{\neg\varphi}$ accepts exactly those sequences of P that satisfy $\neg\varphi$ and hence φ can be proved by establishing emptiness of the synchronous product [27]. This entails detecting acceptance cycles, infinite paths from the initial state $\langle\iota, s^0\rangle$ in which some (composite) acceptance state is repeated infinitely often. A memory efficient algorithm for on-the-fly detection of acceptance cycles, implementing a nested DFS, was proposed in [1]. The presence of such a cycle signifies the existence of a sequence satisfying $\neg\varphi$ which serves as a counter example to φ. A clear advantage of on-the-fly model-checking is that a counter example may be found before completing the synchronous product. Another advantage is the possible reduction in space and time since the product can be smaller than the program state space itself (the checked formula acts as a constraint on the program's behavior through the required accordance of proposition labels).

In [20] Peled combines his partial-order reduction method with on-the-fly model-checking in order to gain from both. That is, for a stuttering-closed property φ it is proved to be sufficient to check emptiness of the synchronous product of the *reduced* state graph G' and $B_{\neg\varphi}$. This is done again by seeking acceptance cycles, but using a slight modification of the nested DFS algorithm in [1]. The modification increases time efficiency and is further needed to ensure compatibility with partial-order reduction methods [11], i.e. to guarantee that the algorithm indeed finds an acceptance cycle if there exists one in $G' \times B_{\neg\varphi}$ (this was not yet recognized in [7, 20], but the authors proposed the correction in [11]). It also increasesThe calculation of ample sets itself also undergoes a minor change so that it applies to composite states of the product $G' \times B_{\neg\varphi}$ instead of single program states. Precisely, when expanding a composite state $\langle x, y\rangle$ of $G' \times B_{\neg\varphi}$, a non-empty subset ample(x, y) of en(x) is used to generate successors for $\langle x, y\rangle$ satisfying the earlier conditions **C1** and **C3** and the new condition **C2′** [20]:

C2′ If ample(x, y) is a *proper* subset of en(x), then no operation $\alpha \in$ ample(x, y) is such that the state $\langle\alpha(x), y\rangle$ *of* $G' \times B_{\neg\varphi}$ is on the DFS stack.

Conditions **C1** and **C3** remain unaffected as the dependency relation D and the set vis(φ) of visible program operations are irrespective of the state of the Büchi automaton. Checking **C2** entails inspecting the DFS stack and must be adapted in particular because each program state may yield several composite states that differ in the state of the Büchi automaton. That is, in comparison with the off-line construction of G' the on-the-fly construction of $G' \times B_{\neg\varphi}$ may postpone the closing of cycles [20]. The adapted condition **C2′** appears sufficient to guarantee that the modified version [11] of the nested DFS algorithm in [1], with the calculation of ample sets to determine successor states, detects at least one acceptance cycle on-the-fly if one or more such cycles exist in the full state graph G.

4 The proposed enhancement

For many concurrent programs the reduced state graph G' can be substantially smaller than the full state graph, as witnessed by the experimental results reported in [7, 20]. Yet, it is recognized that even G' may still manifest a notable amount of redundancy that can be eliminated. A simplified example explains this. Consider a program with two processes P_1 and P_2, where both P_1 and P_2 terminate after executing a single operation, say a and b, respectively. Also assume that the operations a and b are independent and invisible (i.e. P_1 and P_2 execute autonomously and no particular temporal property is checked). Obeying conditions **C1**, **C2** and (trivially) **C3** on ample sets, it is easy to construct the reduced state graph for this program. It generates only one of the two possible orderings of a and b, namely ab. This amounts to a reduction of one node and two edges compared to the full state graph. Yet, even the generation of just this one order appears redundant. The fact that a and b are independent and invisible allows one to avoid an order altogether by mimicking a truly concurrent execution of these operations, i.e. by executing them collectively. A further reduction of the state graph is then achieved since the intermediate state reached after executing a at the initial state is no longer generated. We employ this idea of executing concurrently multiple enabled operations at a given program state to enhance the method in [20, 7] in terms of the space and time requirements (i.e. the number of stored nodes and explored edges, respectively) for LTL model-checking. The initial focus is on off-line model-checking.

4.1 Leap sets

The key behind the approach proposed here lies in a simple observation: a program state may have multiple pairwise *disjoint* ample sets. For any two ample sets, say A_1 and A_2, of a program state x, it holds that A_1 and A_2 are disjoint iff A_1 and A_2 do not both contain an operation from the same process iff all operations in A_1 are independent of all operations in A_2. This follows from condition **C1** and the fact that two operations from the same process cannot be independent. In the context of [20], a program state x has multiple disjoint ample sets particularly when there is more than one process P_i whose set $\mathrm{en}_i(x)$ satisfies conditions **C1** to **C3** (cf. Section 3).

To exploit this observation, we adopt an algorithm for calculating ample sets that differs in two ways from the one in [20]. First, it enforces only conditions **C1** and **C3** on ample sets. The reason for omitting **C2** will become clear in Section 4.3. Secondly, it returns the set of *all* sets $\mathrm{en}_i(x)$ satisfying **C1** and **C3**, or the empty set if no such $\mathrm{en}_i(x)$ exists. Thus, only nontrivial ample sets (wrt **C1** and **C3**) are calculated. The algorithm is a straightforward adaptation of the one in [20] for finding just one ample set and does not introduce significant extra overhead, especially since **C1** and **C3** are already checked for the most part statically by a prescan of the processes during program compilation [7, 20].

Let q be the number of ample sets returned by the modified algorithm ($0 \leq q \leq$ the number of processes in the program). When $q > 0$, each ample set $\mathrm{ample}_j(x)$ ($1 \leq j \leq q$) is a subset of $\mathrm{en}(x)$ satisfying **C1** and **C3**, and hence all its operations are invisible and independent of all operations in the other ample sets. This allows enabled operations from different ample sets to be executed concurrently at x. Precisely, any collection of enabled operations forming an element of the product $\prod_{j=1}^{q} \mathrm{ample}_j(x)$ can be executed concurrently at x. Accordingly, define the set of *leap sets* in a state x as follows:

$$
\begin{aligned}
\mathrm{leap}(x) \;&=_{def} \prod_{j=1}^{q} \mathrm{ample}_j(x) && \text{if } q > 0 \\
&=_{def} \{\ \{\alpha\} \mid \alpha \in \mathrm{en}(x)\ \} && \text{if } q = 0
\end{aligned}
$$

When $q = 0$, all operations enabled at x are thus considered individually, as is the case in [20]. These operations appear here in the form of singleton sets.

4.2 Deadlock detection

Akin to a DFS using ample sets, one can perform a DFS that governs the execution of leap sets to determine successors for each state generated during the search. Such a reduced search essentially leaps through the full state space of a program (hence the name leap set) and is proved to be sufficient for detecting all deadlocks (i.e. program states at which no operations are enabled).

Define the *leap graph* of a program P as a directed graph $G_\ell = \langle V_\ell, E_\ell \rangle$, with $\iota \in V_\ell \subseteq V$ and E_ℓ a finite set of edges labeled by *sets of* operations from T, i.e. leap sets, such that $x \xrightarrow{L} x' \in E_\ell$ iff $L \in \mathrm{leap}(x)$, $x \in \mathrm{en}_{\alpha_i}$ for all $\alpha_i \in L$, and $x' = \alpha_1(\alpha_2(\ldots\alpha_{|L|}(x)\ldots))$ with $\alpha_1\alpha_2\ldots\alpha_{|L|}$ any permutation of the operations in L. A leap graph *generates* a sequence of leap sets $L_1L_2\ldots$ (or the corresponding sequence of states) if there exists a (finite or infinite) path starting with ι whose edges are labeled $L_1L_2\ldots$. A permutation of the operations in a leap set L is called a *linearization* of L. Since all linearizations of L belong to the same finite trace (its operations are mutually independent) and thus lead to the same state, $\mathrm{lin}(L)$ is used to denote any such linearization. This notation is extended to (finite or infinite) sequences of leap sets in a natural way, i.e. a linearization of $\vartheta = L_1L_2\ldots$ is defined as $\mathrm{lin}(\vartheta) = \mathrm{lin}(L_1)\mathrm{lin}(L_2)\ldots$.

Lemma 1 For every *finite* run $[\mathtt{v}][\mathtt{w}]$ of a program P, with $\mathtt{v}, \mathtt{w} \in T^*$ and $\mathrm{fin}_{\mathtt{v}} = x$, there exists a leap set $L \in \mathrm{leap}(x)$ such that $[\mathrm{lin}(L)] \trianglelefteq_D [\mathtt{w}]$. □

Theorem 1 For every *finite* run $[\mathtt{v}]$ of a program P, there exists a sequence of leap sets ϑ generated by the leap graph G_ℓ of P such that $\mathtt{v} \equiv_D \mathrm{lin}(\vartheta)$. □

Theorem 1 directly implies that for every finite computation $\mathtt{v}$ of a program P, $\mathrm{fin}_{\mathtt{v}}$ is a node in the leap graph G_ℓ of P. G_ℓ can thus be used to detect all deadlocks of P, i.e. all its valid and invalid end-states (in the terminology of SPIN). It should be noted that the detection of deadlocks does not require the specification of a temporal formula, meaning that condition **C3** on ample sets actually becomes void. This often allows the calculation of more disjoint ample sets and thus larger leap sets, which generally favors the size of the leap graph.

4.3 Off-line LTL model-checking

Unfortunately, the leap graph as defined above does not lend itself for verifying properties more complex than deadlock-freedom, such as LTL properties. This is due to a phenomenon already recognized in [25] as the "ignoring problem": some operations may be ignored completely by the reduced search as their execution is indefinitely deferred along a cycle. To avoid it, Peled enforces condition **C2** which ensures that no operation in ample(x) closes a cycle on the DFS stack in case ample(x) is a proper subset of en(x) [20] (cf. Section 3.1). We adopt the same principle for leap sets. Let op(leap(x)) denote the set of all operations of all leap sets in state x. If op(leap(x)) is a proper subset of en(x), then for no leap set $L \in$ leap(x) it should hold that the execution of L at x leads to a state that is already on the stack. Otherwise, leap(x) is augmented by adding for *one arbitrary* $L' \in$ leap(x) all sets $L' \cup \{\alpha\}$, where α ranges over the set of operations enabled at x but not included in op(leap(x)). Formally, let $L' \in$ leap(x) be any leap set in x and define

$$
\begin{aligned}
\mathrm{xleap}(x) \; &=_{def} \mathrm{leap}(x) \\
&\qquad \text{if } \mathrm{op}(\mathrm{leap}(x)) = \mathrm{en}(x) \vee \nexists\, L \in \mathrm{leap}(x)\text{: } x \xrightarrow{L} x' \text{ and } x' \text{ is on the DFS stack} \\
&=_{def} \mathrm{leap}(x) \cup \{L' \cup \{\alpha\} \mid \alpha \in \mathrm{en}(x) \setminus \mathrm{op}(\mathrm{leap}(x))\} \\
&\qquad \text{otherwise.}
\end{aligned}
$$

It is important to note that we *extend* leap(x) rather than returning all singleton sets of enabled operations (which would reflect the calculation of ample sets in [20]). In this way the calculation of leap(x) is never wasted, not even when it turns out that the execution of some leap set leads back to a state on the DFS stack. It follows readily from the construction and condition **C1** that for each element of xleap(x) it still holds that all its operations are mutually independent wrt the dependency relation D and, in addition, that at most one of these operations is visible.

Analogous to the leap graph G_ℓ of a program P, an *extended leap graph* G_ℓ^* of P can be defined based on the execution of elements from the set xleap(x). Henceforth, the term leap set refers to an element of xleap(x) instead of leap(x) (for some x) and, as before, lin(ϑ) to any linearization of a (finite or infinite) sequence ϑ of such leap sets. As for computations of a program (cf. Section 2), a sequence of leap sets ϑ from the initial state of the program (i.e. lin(ϑ) is a computation) is said to satisfy a formula φ iff the Büchi automaton B_φ accepts the sequence of states extracted from ϑ. Since any leap set contains at most one visible operation from vis(φ), all such operations in ϑ appear in the same order in each linearization of ϑ, while the invisible operations correspond to stuttering steps. Therefore, either all or none of the linearizations of ϑ satisfy φ and, moreover, ϑ itself satisfies φ iff lin(ϑ) satisfies φ.

Let $D' = D \cup (\mathrm{vis}(\varphi) \times \mathrm{vis}(\varphi))$ be the dependency relation D of a program P augmented with dependencies between all the visible operations for the checked formula φ. This makes φ *equivalence robust* [19, 20]: all sequences equivalent wrt D' (i.e. belonging to the same run wrt D')

contain the same visible operations and in the same order, and thus φ has the same truth value for each of them. Surely, all linearizations of a sequence of leap sets are equivalent wrt D'. The dependency relation D' is used to show that a DFS governing the execution of leap sets preserves the order of visible operations in computations of a concurrent program (Lemma 2 and Lemma 3). This in turn is the key to proving the main result (Theorem 2): nexttime-free LTL model-checking can be conducted using the extended leap graph G_ℓ^* of a program P. The proofs (see Appendix) are essentially similar to the proofs of the corresponding claims in [20]. The following concepts and notations are hence taken directly from [20].

For a finite or infinite sequence of operations $\mathtt{v}$, let $\mathtt{v}(i)$ denote the i-th operation in $\mathtt{v}$, $\mathtt{v}(i\ldots j)$ the i-th through j-th operations, and $\mathtt{v}(i{+}1\ldots)$ the operations in $\mathtt{v}$ except the first i. A *selection function* for $\mathtt{v}$ is a function $c : \{1,\ldots, |\mathtt{v}|\} \mapsto \{\mathbf{T}, \mathbf{F}\}$ mapping each operation in $\mathtt{v}$ to either $\mathbf{T}$ or $\mathbf{F}$. Denote by $\mathtt{v}_c$ ($\mathtt{v}_{\bar{c}}$) the sequence remaining after removing all operations $\mathtt{v}(i)$ with $c(i) = \mathbf{F}$ ($c(i) = \mathbf{T}$). Also, denote by $c \ll m$ the selection function c shifted to the left m places: $(c \ll m)(i) = c(i + m)$. Define $\mathtt{v} \preceq_D^A \mathtt{w}$ iff there is a selection function c for $\mathtt{w}$ such that (1) $\mathtt{v} \equiv_D \mathtt{w}_c$, (2) $\mathrm{op}(\mathtt{w}_{\bar{c}}) \in A \subset T$, and (3) for all $1 \le m \le |\mathtt{w}|$, if $c(m) = \mathbf{F}$, then all operations in $\mathrm{op}(\mathtt{w}(m{+}1\ldots)_{c \ll m})$ are independent wrt D of $\mathtt{w}(m)$. That is, $\mathtt{v} \preceq_D^A \mathtt{w}$ iff a sequence equivalent to $\mathtt{v}$ can be obtained from $\mathtt{w}$ by removing from $\mathtt{w}$ some operations in A that are independent of all the non-removed operations of $\mathtt{w}$ that appear after them [20].

Lemma 2 Let x be a state removed from the DFS stack during the construction of the extended leap graph G_ℓ^* of a program P and let $[\mathtt{v}][\alpha\mathtt{w}]$ be a run wrt D' of P, where $\alpha \in T$ and $\mathrm{fin}_{\mathtt{v}} = x$. Then, there exist a sequence $x \xrightarrow{L_1} x_1 \xrightarrow{L_2} \ldots \xrightarrow{L_k} x_k \xrightarrow{L_{k+1}} x'$ ($k \ge 0$) generated by G_ℓ^*, and a selection function c for $\mathtt{u} = \mathrm{lin}(L_1 L_2 \ldots L_k (L_{k+1} \setminus \{\alpha\}))$, such that

(1) $\alpha \in L_{k+1}$,

(2) no operation in $L_1, L_2, \ldots, L_k$ is visible,

(3) all operations in $L_1, L_2, \ldots, L_k, L_{k+1} \setminus \{\alpha\}$ are independent wrt D' of α,

(4) $\mathtt{u}\alpha \equiv_{D'} \alpha \mathtt{u}_c \mathtt{u}_{\bar{c}}$,

(5) $\exists \mathtt{w}'$: $\mathtt{u}_c \mathtt{w}' \equiv_{D'} \mathtt{w}$ (i.e. $[\mathtt{u}_c] \trianglelefteq_{D'} [\mathtt{w}]$), and

(6) the operations in $\mathtt{u}_{\bar{c}}$ are independent wrt D' of the operations in $\mathtt{w}'$. □

Lemma 2 implies that for each operation α that becomes enabled along some computation of a program P, the extended leap graph G_ℓ^* of P also generates a sequence along which α becomes enabled. Thus, aside from our objective, G_ℓ^* can be used to detect all "dead" operations of P, i.e. operations that are not executed along any computation. Again condition **C3** on ample sets can thereby be ignored, as was the case for deadlock detection, and this similarly favors the size of the extended leap graph.

Lemma 3 For every run $[\mathtt{v}]_{D'}$ of a program P, there exists a sequence of leap sets ϑ generated by the extended leap graph G_ℓ^* of P such that $\mathtt{v} \preceq_{D'}^{T\setminus \mathrm{vis}(\varphi)} \mathrm{lin}(\vartheta)$. □

Theorem 2 Let φ be a nexttime-free LTL formula. For every run $[\mathtt{v}]_{D'}$ of a program P, there exists a sequence of leap sets ϑ generated by the extended leap graph G_ℓ^* of P such that $\mathtt{v}$ satisfies φ iff ϑ satisfies φ. □

Theorem 2 thus allows applying (off-line) LTL model-checking algorithms [12] to the extended leap graph of a program, as opposed to its full state graph or the reduced state graph from [20], for properties that are nexttime-free.

4.4 On-the-fly LTL model-checking

We now turn to the aptness of the extended leap graph G_ℓ^* for on-the-fly LTL model-checking. In analogy with Section 3.2, each transition of the synchronous product $G_\ell^* \times B_{\neg\varphi}$ is of the form $\langle x, y\rangle \xrightarrow{\langle L, \mathbf{P}\rangle} \langle x', y'\rangle$, where $x \xrightarrow{L} x'$ is an edge/transition of G_ℓ^* and $y \xrightarrow{\mathbf{P}} y'$ a transition of $B_{\neg\varphi}$ such that proposition **P** is true in program state x. The following theorem (cf. [20]) proves that it is equally sufficient to check the emptiness of this product in order to verify a nexttime-free LTL formula φ (i.e. a stuttering-closed property).

Theorem 3 Let φ be a nexttime-free LTL formula. The synchronous product $G \times B_{\neg\varphi}$ is empty iff the synchronous product $G_\ell^* \times B_{\neg\varphi}$ is empty. □

Compared to the off-line construction of G_ℓ^*, the closing of cycles may be retarded when constructing the product $G_\ell^* \times B_{\neg\varphi}$ for the aforementioned reason that a program state can occur in multiple composite states distinguished by their second components (cf. Section 3.2). Since this affects solely condition **C2** and the definition of leap(x) in Section 4.1 does not rely on **C2** (the ample sets used to construct leap(x) satisfy just **C1** and **C3**), only the definition of xleap(x) in Section 4.2 needs to be adapted. Similar to condition **C2′** in [20], composite states are accounted for as follows:

$$\begin{aligned}
\text{xleap}(x, y) \;&=_{def} \text{leap}(x) \\
&\qquad \text{if } \text{op}(\text{leap}(x)) = \text{en}(x) \vee \nexists\, L \in \text{leap}(x)\colon x \xrightarrow{L} x' \text{ and } \langle x', y\rangle \text{ is on the DFS stack} \\
&=_{def} \text{leap}(x) \cup \{L' \cup \{\alpha\} \mid \alpha \in \text{en}(x) \setminus \text{op}(\text{leap}(x))\} \\
&\qquad \text{otherwise.}
\end{aligned}$$

The adaptation of the algorithm for on-the-fly detection of acceptance cycles in the partial-order context [7, 20, 11] now simply consists in using xleap(x, y) instead of ample(x, y) to determine the subset of (not necessarily immediate) successors of x that need be explored next. Suffice it to say that this adaptation is indeed adequate for effectively combining the execution of leap sets with on-the-fly model-checking. That is, the adapted algorithm returns true if the verified program does not satisfy the checked property, and false otherwise. The proof of correctness is merely identical to the one for the on-the-fly partial-order reduction algorithm in [20], taking into account also the modification suggested in [11] of the nested DFS algorithm in [1] (cf. Section 3.2). It involves a reduction to a nondeterministic variant of the off-line algorithm preserving the validity of Lemma 2 and hence of Lemma 3 and Theorem 2 (see [20] for the precise details).

It is appropriate here to remark that an alternative definition can be given for xleap(x, y), and likewise for xleap(x) in Section 4.3, that does not require inspection of the DFS stack. By *always* extending leap(x) in case op(leap(x)) is a proper subset of en(x), i.e. even if no $L \in$ leap(x) leads to a state on the stack, all the previous results still hold (in particular, the sequence of leap sets sought in Lemma 2 is then guaranteed to be of length one). Although this generally increases the size of the extended leap graph G_ℓ^* of a program, it may reduce the time for constructing this graph (and the product $G_\ell^* \times B_{\neg\varphi}$) especially when it turns out that in many expension steps the number of elementary leap sets (i.e. those in leap(x)) and the current DFS stack are large. In addition, for on-the-fly cycle detection one can directly use the algorithm in [1] without any modification.

5 Experiments

The off-line versions of the proposed approach, the partial-order reduction method of [7, 20] and the non-reduced, conventional state exploration algorithm are all implemented in C under UNIX as part of a reachability analysis tool called RELIEF. This tool takes protocols specified as sets of processes that communicate by exchanging messages over bounded FIFO channels. A few hundred such protocols were analyzed with RELIEF in order to compare the performance of the three (off-line) methods in terms of the sizes of the resulting state graphs, i.e. the number of stored states/nodes and explored transitions/edges. To eliminate potential bias we conducted experiments on randomly generated protocols: 300 protocols were generated with a (non-service oriented) automatic protocol synthesizer [16, 18] included in the reachability analysis tool. The number of processes in these protocols (ranging from two to six), as well as the various attributes of the processes themselves (the numbers of process states and the number of send and receive operations) all disperse quite well [16, 18]. The number of global states for each protocol lies between 1000 and 300,000 states. All experiments were conducted on a SPARC 10 station with 48 Mbytes of RAM.

Table 1: Average percentage of reduction per number of processes and per concurrency level.

Methods		**Number of processes**					**Concurrency level**		
		2	**3**	**4**	**5**	**6**	**[0, 1]**	**(1, 2]**	**(2, 3]**
B vs. A	States	36.11	49.91	60.99	68.93	76.29	38.19	61.82	83.48
	Transitions	56.33	68.27	76.76	82.15	87.59	57.44	77.53	92.90
C vs. A	States	54.92	64.03	70.92	75.01	82.35	53.22	72.27	89.94
	Transitions	63.97	73.63	80.29	84.21	89.58	63.56	81.36	94.65
C vs. B	States	29.44	28.19	25.46	19.57	25.56	24.32	27.37	39.10
	Transitions	17.49	16.89	15.19	11.54	16.04	14.38	17.04	24.65

Table 1 compares the performance of our approach (method C) against both the reduced search in [7, 20] (method B) and the conventional state exploration (method A). The figures indicate average percentages of reduction, and are arranged by the number of processes in a protocol and by the concurrency level of a protocol. The latter is defined as the size of a leap set in leap(x) averaged over all states x in the *full* state graph, which is a conceivable measure for the degree of parallelism in a protocol. Investigating the reductions per concurrency level is motivated by the intuitive expectation that method C as well as method B yield better reductions for protocols with increasing degrees of parallelism (so-called "loosely-coupled" protocols). The first two rows show average reductions of these methods over the conventional state exploration, while the third row compares method C directly to method B by normalizing the reductions for the former with respect to those for the latter. Overall the results show that the approach described in this paper can yield notable extra reductions over the partial-order reduction method in [7, 20], especially with respect to the space requirements, and may hence broaden the applicability of state exploration based verification to more complex concurrent programs and communication protocols.

6 Conclusions

In this paper, we have proposed an enhancement of SPIN's partial-order reduction method [7, 20] to enable a further reduction in space and time for verifying local and termination properties of concurrent programs (deadlock-freedom and local state or code reachability), and for model-checking (off-line and on-the-fly) nexttime-free LTL properties including arbitrary safety and liveness properties of concurrent programs. The concept underlying the enhancement is quite intuitive: rather than fixing an arbitrary interleaving order among concurrent program operations, as do partial-order reduction methods, we abstain from any order altogether by executing leap sets that mimic a truly concurrent execution of these operations. Although the partial-order reduction method in [7, 20] and the proposed enhancement cannot be strictly compared in the sense that one does not subsume the other, the experiments indicate that our approach generally yields significant extra reductions and can therefore widen the applicability of verification by state space exploration to more complex concurrent programs and protocols. We must recall that the experimental results pertain to the off-line construction of the reduced state spaces. An extension of the reachability analysis tool RELIEF for on-the-fly verification with leap sets and ample sets is anticipated in the near future. Nevertheless, there is no reason to believe that similar comparative results will then not be obtained. In order to get an even more comprehensive picture of the performance of our reduction approach with respect to SPIN's partial-order method, it would of course be beneficial also to add it to SPIN.

The discussion has thus far been confined to (partial-order) model-checking *without fairness*. When the interleaving semantics of a program involves fairness, the computations of a program are limited to those satisfying the assumed fairness conditions. It is well-known that checking a property φ under a fairness condition ψ can be done by model-checking the implication $\psi \Rightarrow \varphi$. Unfortunately, condition **C3** on visible operations must then be applied also to ψ, which often introduces many dependencies (wrt D') and may thus yield little or no reduction at all [4, 20]. However, by rewriting a temporal formula as a conjunction of sub-formulas and treating each sub-formula individually when adding dependencies, Peled takes notable advantage of a certain class of fairness conditions (including process justice and process fairness) that can be made equivalence robust in this way by adding relatively few or no dependencies among visible operations [19, 20]. In effect, these fairness assumptions then emerge as "low-cost filters" on runs, enabling the calculation of ample sets with respect to a smaller set of runs using a slight modification of condition **C3**. This may decrease the size of ample sets and result in smaller state graphs. We advocate that this advantage applies equally well to our approach. One simply adopts the same adjustment to **C3** and constructs leap sets as before from ample sets that respect the modified condition. We refer to [20] for details, which should convince the interested reader of this claim.

Alternative methods exist for partial-order model-checking [4, 25, 26] which vary, among others, in the way they select an appropriate subset of the enabled operations at a given program state to determine successor states [2, 26]. Such a set is called a *persistent set* in [2] and a *stubborn set* in [26]. These different methods are amply compared in [20]. The point here is that the proposed enhancement for partial-order model-checking can be based on ample sets, persistent sets or stubborn sets, even when these sets include enabled operations from multiple processes (in contrast to the ample sets considered in Section 3). Leap sets can be constructed as before provided that the ample sets, persistent sets or stubborn sets employed are pairwise disjoint. As shown in Section 4.1, this is a necessary and sufficient condition to enforce that all operations in each employed set are mutually independent of all operations in the other sets.

As pointed out earlier, Özdemir [16] already investigated the idea of executing sets of concurrent operations for verifying (on-the-fly) arbitrary safety properties of concurrent programs in a framework where processes synchronize on common actions. Following the approach in [3], the negation of the checked safety property is represented by a finite state automaton on finite words (i.e. sequences of program operations) with only one accepting state, and this automaton is viewed as an additional process of the program. A so-called simultaneous product of all these processes is then defined that preserves the reachability of process (or local) states. Thus, the single accepting state of the property automaton is reachable only if it is reachable in the simultaneous product, hence guaranteeing a conclusive verification result. Apart from not handling liveness properties which requires the scrutiny of infinite computations, the approach in [16] differs from ours in the way sets of concurrently executable operations are constructed. The algorithm used in [16] searches for connected components in an undirected graph whose nodes and edges correspond to program operations and dependencies between these operations, respectively, similar as in [26]. In contrast, the generation of ample sets involves only local computations [20] allowing a simpler algorithm for constructing ample sets and thus leap sets. Other advantages of our approach stem from the use of a property automaton over sequences of program states instead of program operations (as in [16, 4, 26]). As Peled argues [20], this is the kind of automaton which is derived from a temporal formula and used by SPIN, and which also eases the expression and manipulation of fairness constraints [4, 20].

Acknowledgement

This work is supported in part by the Natural Sciences and Engineering Research Council of Canada under grant number STR0149338, the Ontario Graduate Scholarship Program and the "Fonds pour la formation de chercheurs et l'aide à la recherche (FCAR)". The author would like to thank Tuong Nguyen for his invaluable effort in implementing the reachability analysis tool RELIEF (v.4.0), and in conducting the experiments.

References

[1] C. Courcoubetis, M. Vardi, P. Wolper, M. Yannakakis, "Memory efficient algorithms for the verification of temporal properties," *Formal Methods in System Design*, 1: 275-288, 1992.

[2] P. Godefroid, "Using partial orders to improve automatic verification methods," in *Proc. 2nd Workshop on Computer Aided Verification (CAV'90)*, LNCS 531, 1991, pp. 176-185.

[3] P. Godefroid, P. Wolper, "Using partial orders for the efficient verification of deadlock freedom and safety properties," *Formal Methods in System Design*, 2(2): 149-164, 1993.

[4] P. Godefroid, P. Wolper, "A partial approach to model checking," *Information and Computation*, 110(2): 305-326, 1994.

[5] M.G. Gouda, J.Y. Han, "Protocol validation by fair progress state exploration," *Computer Networks and ISDN Systems*, 9: 353-361, 1985.

[6] G. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991.

[7] G. Holzmann, D. Peled, "An improvement in formal verification," in *Proc. 7th Conference on Formal Description Techniques (FORTE'94)*, Bern, Switzerland, 1994, pp. 177-191.

[8] G. Holzmann, D. Peled, M. Yannakakis, "On nested depth-first search," in *Proc. 2nd Workshop on the SPIN Verification System (SPIN'96)*, , New Brunswick, NJ, 1996, pp. 81-89.

[9] M. Itoh, H. Ichikawa, "Protocol verification using reduced reachability analysis," *Trans. of the IECE of Japan*, E66(2): 88-93, 1983.

[10] S. Katz, D. Peled, "Verification of distributed programs using representative interleaving sequences," *Distributed Computing*, 6: 107-120, 1992.

[11] L. Lamport, "What good is temporal logic?," in *Proc. 9th IFIP Congress – Information Processing '83*, pp. 657-668.

[12] O. Lichtenstein, A. Pnueli, "Checking that finite state concurrent programs satisfy their linear specification," in *Proc. 12th ACM Symposium on Principles of Programming Languages*, New Orleans, LA, 1985, pp. 97-107.

[13] H. Liu, R.E. Miller, "Generalized fair reachability analysis for cyclic protocols," in *IEEE/ACM Trans. on Networking*, 4(2): 192-204, 1996.

[14] H. Liu, R.E. Miller, H. van der Schoot and H. Ural, "Deadlock detection by fair reachability analysis: from cyclic to multi-cyclic protocols (and beyond?)," in *Proc. 16th IEEE Conference on Distributed Computing Systems*, Hong Kong, 1996, pp. 605-612.

[15] A. Mazurkiewicz, "Trace theory," in *Advances in Petri Nets 1986, Part II*, LNCS 255, 1986, pp. 279-324.

[16] K. Özdemir, "Verifying the safety properties of concurrent systems via simultaneous reachability," Ph.D. Thesis, Department of Computer Science, University of Ottawa, 1995.

[17] K. Özdemir, H. Ural, "Deadlock detection in CFSM models via simultaneously executable sets," in *Proc. 6th Conference on Communication and Information*, Peterborough, Ontario, Canada, 1994, pp. 673-688.

[18] K. Özdemir, H. Ural, "Protocol validation by simultaneous reachability analysis," TR-95-09, Department of Computer Science, University of Ottawa, 1995 (submitted for publication).

[19] D. Peled, "All from one, one for all: on model checking using representatives," in *Proc. 5th Conference on Computer Aided Verification (CAV'93)*, LNCS 697, 1993, pp. 409-423.

[20] D. Peled, "Combining partial order reductions with on-the-fly model checking," *Formal Methods in System Design*, 8: 39-64, 1996.

[21] J. Rubin, C.H. West, "An improved protocol validation technique," *Computer Networks and ISDN Systems*, 6: 65-73, 1982.

[22] H. van der Schoot, H. Ural, "On improving simultaneous reachability analysis for the efficient verification of deadlock-freedom," TR-95-20, Department of Computer Science, University of Ottawa, Ontario, Canada, 1995.

[23] H. van der Schoot, H. Ural, "A uniform approach to tackle state explosion in verifying progress properties for networks of CFSMs," TR-96-13, Department of Computer Science, University of Ottawa, Ontario, Canada, 1996.

[24] A. Valmari, "Stubborn sets for reduced state space generation," in *Advances in Petri Nets 1990*, LNCS 483, 1991, pp. 491-515.

[25] A. Valmari, "A stubborn attack on state explosion," *Formal Methods in System Design*, 1: 297-322, 1992.

[26] A. Valmari, "On-the-fly verification with stubborn sets," in *Proc. 5th Conference on Computer Aided Verification (CAV'93)*, LNCS 697, 1993, pp. 397-408.

[27] P. Wolper, M.Y. Vardi, A.P. Sistla, "Reasoning about infinite computation paths," in *Proc. 24th IEEE Symposium on Foundations of Computer Science*, Tucson, AZ, 1983, pp. 185-194.

Appendix (proofs)

Lemma 1 For every *finite* run $[\mathtt{v}][\mathtt{w}]$ of a program P, with $\mathtt{v}, \mathtt{w} \in T^*$ and $\mathrm{fin}_{\mathtt{v}} = x$, there exists a leap set $L \in \mathrm{leap}(x)$ such that $[\mathrm{lin}(L)] \trianglelefteq_D [\mathtt{w}]$.

Proof: If $\mathrm{leap}(x) = \{\ \{\alpha\} \mid \alpha \in \mathrm{en}(x)\ \}$, then the appropriate leap set exists trivially, viz. the singleton set containing the first operation of $\mathtt{w}$. If $\mathrm{leap}(x) = \prod_{j=1}^{q} \mathrm{ample}_j(x)$, then from **C1** and the fact that $[\mathtt{v}][\mathtt{w}]$ is a finite run it follows that for each process P_i with $\mathrm{en}_i(x) = \mathrm{ample}_j(x)$ there exists an operation from $\mathrm{ample}_j(x)$ in $\mathtt{w}$. Let α_j denote the first operation from $\mathrm{ample}_j(x)$ in $\mathtt{w}$. Clearly, $\{\alpha_1, \alpha_2, \ldots, \alpha_q\} \in \mathrm{leap}(x)$. By **C1**, each α_j is independent of all operations appearing in $\mathtt{w}$ before it and hence they can all be permuted to the front of $\mathtt{w}$. Thus, $[\mathrm{lin}(\{\alpha_1, \alpha_2, \ldots, \alpha_q\})] \trianglelefteq_D [\mathtt{w}]$. □

Theorem 1 For every *finite* run $[\mathtt{v}]$ of a program P, there exists a sequence of leap sets ϑ generated by the leap graph G_ℓ of P such that $\mathtt{v} \equiv_D \mathrm{lin}(\vartheta)$.

Proof: By repeated application of Lemma 1. □

Lemma 2 Let x be a state removed from the DFS stack during the construction of the extended leap graph G_ℓ^* of a program P and let $[\mathtt{v}][\alpha\mathtt{w}]$ be a run wrt D' of P, where $\alpha \in T$ and $\mathrm{fin}_{\mathtt{v}} = x$. Then, there exist a sequence $x \xrightarrow{L_1} x_1 \xrightarrow{L_2} \ldots \xrightarrow{L_k} x_k \xrightarrow{L_{k+1}} x'$ $(k \geq 0)$ generated by G_ℓ^*, and a selection function c for $\mathtt{u} = \mathrm{lin}(L_1 L_2 \ldots L_k (L_{k+1} \setminus \{\alpha\}))$, such that (1) $\alpha \in L_{k+1}$, (2) no operation in $L_1, L_2, \ldots, L_k$ is visible, (3) all operations in $L_1, L_2, \ldots, L_k, L_{k+1} \setminus \{\alpha\}$ are independent wrt D' of α, (4) $\mathtt{u}\alpha \equiv_{D'} \alpha \mathtt{u}_c \mathtt{u}_{\bar{c}}$, (5) $\exists \mathtt{w}'$: $\mathtt{u}_c \mathtt{w}' \equiv_{D'} \mathtt{w}$ (i.e. $[\mathtt{u}_c] \trianglelefteq_{D'} [\mathtt{w}]$), and (6) the operations in $\mathtt{u}_{\bar{c}}$ are independent wrt D' of the operations in $\mathtt{w}'$.

Proof: By induction on the order in which states are removed from the DFS stack during the leap graph construction. When removing a state x from the stack, there are two possibilities:

- $\alpha \in \mathrm{op}(\mathrm{xleap}(x))$

  Remark that this case covers in particular the induction basis where x is the first state removed from the stack: each set in $\mathrm{leap}(x) \subseteq \mathrm{xleap}(x)$ applied to x leads to a state that is already on the stack (any other state would have been removed before x – a characteristic of DFS) and thus, by the definition of $\mathrm{xleap}(x)$, $\mathrm{op}(\mathrm{xleap}(x)) = \mathrm{en}(x)$. Hence, $\alpha \in \mathrm{op}(\mathrm{xleap}(x))$ since $\alpha \in \mathrm{en}(x)$. Two alternatives must now be considered:

  i) $\alpha \in \mathrm{op}(\mathrm{leap}(x))$

     Let $L \in \mathrm{leap}(x) \subseteq \mathrm{xleap}(x)$ such that $\alpha \in L$. By definition of $\mathrm{leap}(x)$ as the cross-product of ample sets satisfying **C1** and **C3**, all operations in L are invisible and mutually independent wrt D. Thus, they are also mutually independent wrt D'. It follows directly that $x \xrightarrow{L} x'$ is a sequence satisfying properties (1), (2) and (3).

  ii) $\alpha \notin \mathrm{op}(\mathrm{leap}(x))$

     Since $\alpha \in \mathrm{op}(\mathrm{xleap}(x))$, it must be the case that $\mathrm{leap}(x) \subset \mathrm{xleap}(x)$ and some leap set, say $L' \in \mathrm{leap}(x)$, is selected to form $\mathrm{xleap}(x)$. Again, all operations in L' are mutually independent wrt D' by definition of $\mathrm{leap}(x)$. Moreover, they are also independent wrt D' of α by **C1** and **C3** and the fact that $\alpha \in \mathrm{en}(x)$. Let $L = L' \cup \{\alpha\}$, then by construction $L \in \mathrm{xleap}(x)$ and $x \xrightarrow{L} x'$ is a sequence satisfying properties (1), (2) and (3).

Finally, for any sequence $\mathtt{u}$ define a selection function c such that $c(i) = \mathbf{T}$ if $\mathtt{u}(i) \in \mathrm{op}(\mathtt{w})$; $c(i) = \mathbf{F}$ otherwise. Here, $\mathtt{u} = \mathrm{lin}(L \setminus \{\alpha\})$ and properties (4) to (6) follow readily for both cases.

- $\alpha \notin \mathrm{op}(\mathrm{xleap}(x))$

  Clearly, in this case $\alpha \in \mathrm{en}(x) \setminus \mathrm{op}(\mathrm{leap}(x))$ and therefore it must hold that $\mathrm{xleap}(x) = \mathrm{leap}(x) = \prod_{j=1}^{q} \mathrm{ample}_j(x)$ and no leap set in leap(x) applied to x closes a cycle on the DFS stack. Let $L_1 \in \mathrm{leap}(x)$ and $x \xrightarrow{L_1} x_1$. Since L_1 does not close a cycle, x_1 is not on the stack but once added, it will be removed before x itself is removed (a characteristic of DFS). This implies that the induction hypothesis can be applied to x_1, viz. there exists a sequence of leap sets $L_2 \ldots L_k L_{k+1}$ from x_1 with respective selection function c for $\mathtt{u}' = \mathrm{lin}(L_2 \ldots L_k(L_{k+1} \setminus \{\alpha\}))$ (defined as above) satisfying (1) to (6). Now, since $\alpha \notin L_1$ and each operation in L_1 is invisible and independent wrt D' of all operations in $\alpha\mathtt{w}$ before its own occurrence (if any) in $\mathtt{w}$ (by **C1** and **C3**), it is not difficult to see that $L_1 L_2 \ldots L_k L_{k+1}$ is a sequence of leap sets from x satisfying (1) to (3). To prove (4), let $\mathtt{u}^1 = \mathrm{lin}(L_1)$ and $\mathtt{u} = \mathrm{lin}(L_1 L_2 \ldots L_k(L_{k+1} \setminus \{\alpha\})) = \mathtt{u}^1\mathtt{u}'$. We derive $\mathtt{u}\alpha = \mathtt{u}^1\mathtt{u}'\alpha \equiv_{D'} \mathtt{u}^1\alpha\mathtt{u}'_c\,\mathtt{u}'_{\bar{c}} \equiv_{D'} \mathtt{u}^1_c\,\mathtt{u}^1_{\bar{c}}\alpha\mathtt{u}'_c\,\mathtt{u}'_{\bar{c}}$, using the induction hypothesis and the fact that the operations in a leap set are mutually independent wrt D'. Since in particular the operations in L_1 are independent wrt D' of α and the operations in $\mathtt{u}^1_{\bar{c}}$ are independent wrt D' of all operations in $\mathtt{u}'_c$, it follows directly that $\mathtt{u}^1_c\,\mathtt{u}^1_{\bar{c}}\alpha\mathtt{u}'_c\,\mathtt{u}'_{\bar{c}} \equiv_{D'} \alpha\mathtt{u}^1_c\,\mathtt{u}'_c\,\mathtt{u}^1_{\bar{c}}\,\mathtt{u}'_{\bar{c}} = \alpha\mathtt{u}_c\mathtt{u}_{\bar{c}}$. Properties (5) and (6) are derived similarly. □

Lemma 3 For every run $[\mathtt{v}]_{D'}$ of a program P, there exists a sequence of leap sets ϑ generated by the extended leap graph G_ℓ^* of P such that $\mathtt{v} \preceq_{D'}^{T \setminus \mathrm{vis}(\varphi)} \mathrm{lin}(\vartheta)$.

Proof: Recall that all linearizations of each sequence generated by the extended leap graph G_ℓ^* belong to a run wrt D' of P. For any run $[\mathtt{v}]_{D'}$ of P, while reading $\mathtt{v}$, we describe a traversal of G_ℓ^* starting from ι that yields a sequence ϑ of leap sets such that $\mathtt{v} \preceq_{D'}^{T \setminus \mathrm{vis}(\varphi)} \mathrm{lin}(\vartheta)$. The following variables are used in the process:

- r the sequence of operations read so far from $\mathtt{v}$;
- t the sequence of leap sets labeling the edges of G_ℓ^* traversed so far;
- l a linearization of t, projected on the set of operations that have not yet been read from $\mathtt{v}$ (i.e. removed from lin(t) are the operations already read from $\mathtt{v}$);
- s the current node of G_ℓ^*.

The variables r, t and l are initialized to the empty sequence ε, and s to ι. Whenever the next operation $\alpha \in T$ is read from $\mathtt{v}$, the following updates are made:

1. $r := r\alpha$;
2. **if** $l = \mathtt{w}\alpha\mathtt{w}'$, for some $\mathtt{w}, \mathtt{w}' \in T^*$ such that all operations in $\mathtt{w}$ independent (wrt D') of α, **then** $l := \mathtt{w}\mathtt{w}'$;
3. **else** choose a sequence $L_1 L_2 \ldots L_k L_{k+1}$ from the node s, ending with a node s', such that $\alpha \in L_{k+1}$ and $[(\mathrm{lin}(t)\mathtt{u})_c\alpha] = [\mathrm{lin}(t)_c\alpha\mathtt{u}_c] \trianglelefteq_{D'} [\mathtt{v}]$, with $\mathtt{u} = \mathrm{lin}(L_1 L_2 \ldots L_k(L_{k+1} \setminus \{\alpha\}))$ and the selection function c , defined for arbitrary sequence $\mathtt{z}$, such that $c(i) = \mathbf{T}$ if $\mathtt{z}(i) \in \mathrm{op}(\mathtt{v})$; $c(i) = \mathbf{F}$ otherwise. Make the following updates:
   (a) $s := s'$
   (b) $l := l\mathtt{u}_c$
   (c) $t := tL_1 L_2 \ldots L_k L_{k+1}$.

The following properties are now inductively proved to be invariant while reading v:

(1) $[r][l] = [\mathrm{lin}(t)_c]$,

(2) $[\mathrm{lin}(t)_c] \trianglelefteq_{D'} [\mathrm{v}]$,

(3) if the condition of Step 2 does not hold when checking it, then all the operations in $\mathrm{op}(l)$ are independent wrt D' of α, and

(4) the choice of the sequence $L_1L_2\ldots L_kL_{k+1}$ required by Step 3 can always be made when executing Step 3.

Initially, (1) to (4) trivially hold, as $r = l = t = \varepsilon$ and the algorithm is just before executing Step 1. At each step of the algorithm, $[r][\alpha] \trianglelefteq_{D'} [\mathrm{v}]$ since α is the next operation from v read after r, and $[r][l] \trianglelefteq_{D'} [\mathrm{v}]$ by the induction hypotheses (1) and (2). Therefore, l cannot have operations dependent wrt D' on α before the first (if any) occurrence of α in it, which proves (3). When α does not occur in l, the existence of the sequence $L_1L_2\ldots L_kL_{k+1}$ from node s is guaranteed by Lemma 2, proving (4). It is then easy to check that both (1) and (2) are preserved by taking either Step 2 or Step 3 of the algorithm.

Let ϑ be the finite or infinite sequence of leap sets collected into t along a finite or infinite traversal, respectively, of G_ℓ^* upon reading v. From (1), for each $r \in \mathrm{Pref}(\mathrm{v})$, $r \preceq_{D'} \mathrm{lin}(t)_c$ and thus $r \preceq_{D'} \mathrm{lin}(\vartheta)_c$. Also, from (2), for each $\mathrm{z} \in \mathrm{Pref}(\mathrm{lin}(\vartheta))$ we have $\mathrm{z}_c \preceq_{D'} \mathrm{v}$. Thus, $\mathrm{v} \equiv_{D'} \mathrm{lin}(\vartheta)_c$ and by using Lemma 1.(4)-(6) and the definition of '$\preceq_{D'}^{T\setminus \mathrm{vis}(\varphi)}$' it follows that $\mathrm{v} \preceq_{D'}^{T\setminus \mathrm{vis}(\varphi)} \mathrm{lin}(\vartheta)$. □

Theorem 2 Let φ be a nexttime-free LTL formula. For every run $[\mathrm{v}]_{D'}$ of a program P, there exists a sequence of leap sets ϑ generated by the extended leap graph G_ℓ^* of P such that v satisfies φ iff ϑ satisfies φ.

Proof: From Lemma 3, there exists a sequence of leap sets ϑ generated by the leap graph G_ℓ^* of P such that $\mathrm{v} \preceq_{D'}^{T\setminus \mathrm{vis}(\varphi)} \mathrm{lin}(\vartheta)$. We prove that v satisfies φ iff ϑ satisfies φ, for which it is sufficient to prove that v satisfies φ iff $\mathrm{lin}(\vartheta)$ satisfies φ. This follows from the fact that ϑ "captures" the same visible operations and in the same order as they appear in $\mathrm{lin}(\vartheta)$ (any leap set contains at most one visible operation). That is, the sequences of states extracted from ϑ and $\mathrm{lin}(\vartheta)$ are stuttering equivalent wrt φ and thus φ cannot distinguish between them (φ is stuttering closed because it is a nexttime-free LTL formula). It remains to be shown that v and $\mathrm{lin}(\vartheta)$ yield stuttering equivalent sequences. Since $\mathrm{v} \preceq_{D'}^{T\setminus \mathrm{vis}(\varphi)} \mathrm{lin}(\vartheta)$, this is readily derived from the definition of '$\preceq_{D'}^{T\setminus \mathrm{vis}(\varphi)}$' and the inclusion $\mathrm{vis}(\varphi) \times \mathrm{vis}(\varphi) \subseteq D'$. □

Theorem 3 Let φ be a nexttime-free LTL formula. The synchronous product $G \times B_{\neg\varphi}$ is empty iff the synchronous product $G_\ell^* \times B_{\neg\varphi}$ is empty.

Proof: By Theorem 2, for every computation of a program P there exists a sequence of leap sets generated by the extended leap graph G_ℓ^* of P such that either both satisfy φ or both do not satisfy φ. Thus, $G \times B_{\neg\varphi}$ is non-empty iff there exists some computation v of P satisfying $\neg\varphi$ iff there exists some sequence of leap sets ϑ in G_ℓ^* satisfying $\neg\varphi$ iff $G_\ell^* \times B_{\neg\varphi}$ is non-empty. □