Combining Partial Order Reductions with On-the-fly Model-Checking

Doron Peled
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974, USA

Abstract

Partial order model-checking is an approach to reduce time and memory in model-checking concurrent programs. On-the-fly model-checking is a technique to eliminate part of the search by intersecting the (negation of the) checked property with the state space during its generation. We prove conditions under which these two methods can be combined in order to gain from both reductions. An extension of the model-checker SPIN, which implements this combination, is studied, showing substantial reduction over traditional search, not only in the number of reachable states, but directly in the amount of memory and time used.

1 Introduction

Partial order model-checking is an approach to reduce time and memory when checking that concurrent programs satisfy their linear temporal specification. The main idea behind partial order methods is the observation that when modeling the executions of a program as interleaved sequences of atomic actions (and indeed most assertion languages used are based on such modeling), concurrent activities are interleaved in many possible orders. The checked properties are in many cases insensitive to the interleaving order. This allows fixing some arbitrary order among them, which allows reducing the size of the checked state space. In the kernel of such algorithms for generating a reduced state space are routines for exploring from each generated state a *subset* of the successor states rather than *all of them*.

Partial order methods were at first restricted to checking a constrainted family of properties: the verification method of Katz and Peled [9] and the model-checking methods of Valmari [17] and Godefroid [4] were limited to dealing with safety properties, termination, local and stable properties. Later, Valmari [18] developed a way to check arbitrary nexttime-free temporal properties. Peled [16] generalized these ideas and showed how to gain more reduction by rewriting the checked formula, and how to do the model-checking under fairness assumptions.

We suggest here an algorithm that combines on-the-fly model-checking [11, 3, 2] with partial order reduction. That is, intersect the reduced state space during its generation with an automaton that represents the negation of the checked property. Then, besides the benefit of generating a reduced state space, the generation does not necessarily need to be completed: it might be that an error will be found before the end of the construction, or that parts of the (reduced) state space need not be present in the intersection. The method allows checking the class of stuttering-closed Büchi automata properties, which includes the nexttime-free temporal properties.

Godefroid and Wolper [5] proposed a method based on combining automata, one for each of the program's processes, with one for the checked formula. With this method, if the program can execute from some point two totally independent (i.e., concurrent) infinite tasks, one of them might be ignored by the reduction algorithm. Valmari [19] presented a different method for on-the-fly partial order reduction: in order to solve the above ignoring problem, the choice of a subset of successors taken from each node is based on an algorithm that prefers to include all the operations that can make a change to the current state of the property automaton.

Both of these methods handle specifications represented as automata over (illegal) sequences of the program *operations*. The edges of the specification automaton $\mathcal{B}$ are labeled with operations. The specification automaton in both of these methods synchronizes each of its transitions, labeled with some program operation, with the execution of that program operation in one or more processes. When a pair of program operations that can be executed concurrently in the checked program appear both on transitions of $\mathcal{B}$, their relative order can no longer be fixed arbitrarily. Thus, the checked property restricts the partial order reduction of the state space.

Our method allows combining a Büchi automaton that defines illegal sequences of program *states*. This is the kind of automaton obtained from a temporal specification, e.g., by using Wolper's translation algorithm [20] and is also the kind of automaton used to check properties with the model-checker SPIN [6]. In this setting, the transitions of the specification automaton $\mathcal{B}$ are labeled with sets of propositions rather than with program operations. Each transition of $\mathcal{B}$ is synchronized with global program states. Again, the partial order reduction must be restricted by the checked property in order to guarantee that the truth of the checked property is preserved between the full and the reduced state space. This is done here by restricting the reduction, not allowing to arbitrarily fix the order between any pair of operations that can both change at least one of the checked predicates. Unlike [5, 19], in our approach, we are not always deprived of the ability to fix an arbitrary order between two concurrent operations when both can cause a change to the state of the specification automaton $\mathcal{B}$ when synchronized with it.

Special care is taken w.r.t. the above ignoring problem. A way to avoid this problem is to disallow a reduced set of successors from some state, when one of them closes a cycle. This was first suggested in [16]. The applicability of it to the on-the-fly case is not trivial and is carefully proven here. It should be mentioned that a requirement such as the above is very subtle: an earlier condition for the ignoring problem appeared in [7] and required only that at least one of the selected operations does not close a cycle. This turns out to be insufficient for preserving temporal properties.

Our strategy is in essence the reverse of [19]. Namely, we prefer selecting operations that *do not* change the predicates or propositional values in the checked formula. The advantage is the use of a simpler algorithm than [19] for calculating subsets of successors. The reduction method presented here also introduces partial order reductions done on-the-fly under various fairness assumptions.

The reduction method suggested here is presented as a collection of constraints on the selection of an appropriate subset of the enabled operation from each given program state. The constraints are simple and easy to implement within state space model-checkers. The cost of the additional calculations in time and space required to find appropriate subsets of operations is very small, hence making the additional calculations pay-off very quickly. The algorithm described here was implemented as an extension to the model-checker SPIN [6]. This is the first implementation of a partial order model-checker with the full power of stuttering closed Büchi automata [1]. Experiments with various known algorithms and protocols show substantial reductions in space and time.

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*. The enabling condition $en_\alpha \subseteq Q$ of an operation $\alpha \in T$ is the set of states from which α can be executed. Each operation $\alpha \in T$ is a partial transformation $\alpha : Q \mapsto Q$ which needs to be defined at least for each $q \in en_\alpha$.

An *interleaving sequence* of a program is an infinite[1] sequence of operations $v = \alpha_1\alpha_2 \ldots$ that *generates* the sequence of states $\xi = q_0\, q_1\, q_2 \ldots$ from Q, such that (1) $q_0 = \iota$, (2) for each $0 \leq i < n$, $q_i \in en_{\alpha_{i+1}}$ holds, and $q_{i+1} = \alpha_{i+1}(q_i)$. The interleaving semantics of a program sometimes involves a restricting condition on interleaving sequences called *fairness*. Then, only sequences satisfying the assumed fairness conditions are considered to be executions of the program.

An *admissible* sequence is an interleaving sequence or any segment, i.e., a suffix of a prefix, of such a sequence. We represent an admissible sequence either as a set of states from Q (denoted using ξ, $\xi' \ldots$), or a sequence of executed operations $\alpha_1\, \alpha_2\, \alpha_3 \ldots$. (denoted using v, u, w, v', v_i, ...). The fact that ξ is the sequence of states obtained by executing the sequence of operations v from initial state ι is denoted by $states(v, \xi)$. The last state of a finite admissible sequence obtained by executing the sequence of operations v from ι is denoted by fin_v.

Definition 2.1 *An* independence relation *is a binary reflexive and symmetric relation $I \subseteq T \times T$ such that for each pair of operations $(\alpha, \beta) \in I$ (called* independent operations*) it must hold that for each $q \in Q$,*

- *If $q \in en_\alpha \cap en_\beta$ (i.e., α and β enabled from q), then $\beta(q) \in en_\alpha$ (due to symmetry, also $\alpha(q) \in en_\beta$).*
- *If $q \in en_\alpha \cap en_\beta$ then α and β are commutative as state transformers of Q. That is, $\alpha(\beta(q)) = \beta(\alpha(q))$.*

A dependency relation *D is the complement of an independence relation, i.e., $D = (T \times T) \setminus I$.*

Two strings $v, w \in T^*$ are considered equivalent [15], denoted $v \equiv_D w$, iff there exists a sequence of strings $u_0, u_1, \ldots, u_n$, where $u_0 = v$, $u_n = w$, and for each $0 \leq i < n$, $u_i = \bar{u}\alpha\beta\hat{u}$ and $u_{i+1} = \bar{u}\beta\alpha\hat{u}$ are admissible[2] for some $\bar{u}, \hat{u} \in T^*$, $\alpha, \beta \in T$, $(a, \beta) \notin D$. That is, w is equivalent to v iff it can be obtained from it by repeatedly commuting adjacent independent operations.

The definition of equivalence between finite strings is now extended to interleaving sequences [12]. Denote by $Pref(w)$ the set of finite prefixes of the (finite or infinite) string w. A relation '$\preceq_D$' is defined between pairs of finite or infinite strings over T as follows: $v \preceq_D v'$ iff $\forall u \in Pref(v) \exists w \in Pref(v') \exists z \in T^* (w \equiv_D z \wedge u \in Pref(z))$. That is, each finite prefix of v is a prefix of a permutation (obtained by commuting adjacent independent operations) of some prefix of v'. Extend now '$\equiv_D$' to infinite strings by defining $v \equiv_D v'$ for $v, v' \in T^\omega$ iff $v \preceq_D v'$ and $v' \preceq_D v$. It is easy to see that '$\equiv_D$' is an equivalence relation [12]. A *trace* is an equivalence class of admissible

[1] Finite executions can be avoided, for simplifying the representation, by adding a special operation that is enabled exactly when no other operation is enabled and does not change the state.

[2] The requirement of admissibility is essential here since the definition of independence does not rule out the case that $(\alpha, \beta) \in I$, β is disabled from some state q, and becomes enabled after α is executed from q.

finite [15] or infinite [12] sequences. Denote a trace σ by $[v]_D$, where v is any member of σ. The index D is omitted when clear from the context. It can be easily shown that for finite v, if $v \equiv_D v'$, then $fin_v = fin_{v'}$.

Concatenation of two traces $\sigma = [\alpha_0 \alpha_1 \ldots \alpha_n]$ and $\sigma' = [\beta_0 \beta_1 \ldots \beta_m \ldots]$, where σ is finite and σ' is either finite or infinite, is defined as $\sigma\sigma' = [\alpha_0 \alpha_1 \ldots \alpha_n \beta_0 \beta_1 \ldots \beta_m \ldots]$, provided that $\alpha_0\alpha_1 \ldots \alpha_n\beta_0\beta_1 \ldots \beta_m \ldots$ is admissible. Denote $\sigma \sqsubseteq_D \rho$ if $\sigma = [v]$, $\rho = [w]$ and $v \preceq_D w$. For finite traces, it means that there exists σ' such that $\rho = \sigma\sigma'$. A *run* π of a program P, defined with respect to some dependency relation D, is an infinite trace that contains interleaving sequences of P.

A nexttime-free LTL [14, 13] formula φ is constructed from propositional variables $p_0, p_1, p_2 \ldots$, the boolean connectives ('$\wedge$', '$\vee$', '$\neg$') and the modals '$\Box$' (always), '$\Diamond$' (eventually) and '$\mathcal{U}$' (until), but not the modal '$\bigcirc$' (next-time). Denote the propositions appearing in the checked formula φ by $\mathcal{P}$. An *interpretation mapping* is a function $\mathcal{F} : Q \mapsto 2^{\mathcal{P}}$, i.e., $\mathcal{F}(q)$ are the variables that are assigned a truth value T in q (the others are assigned the value F). We assume the existence of an interpretation mapping $\mathcal{F}_{(P,\mathcal{P})}$ for each pair of a program P and a set of propositional variables $\mathcal{P}$ (these indexes will be omitted when clear from the context). An interpretation mapping can be extended to sequences, mapping an execution sequence ξ of P into a *propositional sequence* $\mathcal{F}(\xi)$. The fact that a sequence ξ (more precisely, $\mathcal{F}(\xi)$) satisfies a temporal formula φ is denoted by $\xi \models \varphi$, and the fact that all the sequences of a program P satisfy φ is denoted by $P \models \varphi$.

A *state graph* of a program P, which can be used to represent the state space of P, is a graph $G = \langle \hat{s}, V, E \rangle$, where V is a finite set of nodes, $\hat{s} \in V$ is the *starting node*, and E is a finite set of edges, labeled with operations from T. For each node $s \in V$, $val(s)$ is a state of Q, and in particular, $val(\hat{s}) = \iota$. If $s \xrightarrow{\alpha} t \in E$, then $val(s) \in en_\alpha$ (we also say that α is *enabled from* s), and $val(t) = \alpha(val(s))$. A state graph *generates* a sequence of operations $\alpha_1\alpha_2 \ldots$ (or their corresponding sequence of states), if there exists a (finite or infinite) path starting with $\hat{s}$ whose edges are labeled with $\alpha_1\alpha_2 \ldots$ in this order.

An algorithm to generate the *full* state graph of a program can be obtained from the one in Figure 1, by replacing the underlined procedure call $ample(val(s))$ at line 3 by the set of *all* the operations enabled at the state $val(s)$ of the node s, denoted by $en(val(s))$. The flag $open(s)$ is a boolean flag that holds when s is not fully expanded, i.e., is still active, and thus is currently on the search stack.

3 Spawning Reduced State Graphs

We start the presentation with an algorithm **A1** that constructs reduced state spaces. It is related to the works of [9, 17, 18, 7, 16]. It uses a depth first search to expand the state space. We initially employ the following fairness assumption:

F if an operation α is enabled from some state of an interleaving sequence, then some operation that is dependent on α (possibly α itself) must appear later (or immediately) in this sequence.

Thus, we limit the following discussion to runs π that contain interleaving sequences satisfying **F**. Adding this fairness assumption slightly changes the algorithm and greatly simplifies its proof. Later, in Section 3.2, the fairness assumption will be removed and all the runs of the program P will be considered. Thus, two versions of the algorithm, one under the fairness requirement **F** or any stronger requirement, and one under no fairness assumption, will be presented.

3.1 Subsets Construction under Fairness

When algorithm **A1**, depicted in Figure 1, expands a node s, only a subset $ample(val(s))$ of the enabled operations $en(val(s))$ is used to generate successors for s (line 3 in Figure 1). Various algorithms to calculate the selected subset of successors from a given node s with value $val(s) = x$ can be found e.g., in the works of Katz and Peled [9], Valmari [17, 18], and Godefroid et al [4, 5, 7]. Such a subset $\mathcal{E}(x)$ must satisfy the following condition:

C1 No operation $\alpha \in T \setminus \mathcal{E}(x)$ that is dependent on some operation in $\mathcal{E}(x)$ can be executed in P after reaching the state $x = val(s)$ and before some operation in $\mathcal{E}(x)$ is executed.

It follows trivially that under the fairness assumption **F**, the condition **C1** guarantees that:

P1 For every run $\pi = [v][w]$, such that $v \in T^*$, $w \in T^\omega$, $fin_v = val(s) = x$, there exists $\alpha \in \mathcal{E}(x)$ s.t. $[\alpha] \sqsubseteq_D [w]$.

An *occurrence* of a state x in a run π is a string v with $fin_v = x$ such that $[v] \sqsubseteq_D \pi$. According to property **P1**, $\mathcal{E}(x)$ returns at least one immediate successor operation for each occurrence of x, in each partial order execution π of P (with respect to the dependency relation D). The algorithm **R1** to calculate $ample(x)$ appears in Figure 2. It uses the routine $check\_succ(x, i)$, which returns the operations enabled from the state x if they satisfy the property **C1** (at line 3 in Figure 2), or the empty set, otherwise. More details on how our implementation checks this condition are described in Section 5 and in [8]. A second condition [16] is enforced at lines 5–8 in Figure 2:

C2 If $\mathcal{E}(x)$ does not include *all* the operations enabled from $x = val(s)$, then no operation $\alpha \in \mathcal{E}(x)$, when applied to x, closes a cycle on the search stack (i.e., we do not allow that an open node with value $\alpha(x)$).

The correctness of our algorithm depends also on a simple property of the DFS algorithm:

P2 During the execution of a DFS algorithm, if the immediate successors of a node s are not open (i.e., not currently on the search stack), then when closing the node s, its immediate successors are already closed.

Notice that this does not hold if there is at least one successor of the currently expanded node s on the search stack: this successor is still open when the execution backtracks to s. The requirements **C1** and **C2** guarantee the following property, during and after the execution of the expansion algorithm **A1+R1**:

P3 Let s be a closed node and let $\pi = [v][\alpha w]$, with $\alpha \in T$, be a run of P, such that $fin_v = val(s)$. Then there exists a path $s_0 \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_n} s_n \xrightarrow{\alpha} t$, with $s_0 = s$, such that $[\beta_1\beta_2 \ldots \beta_n\alpha] = [\alpha\beta_1\beta_2 \ldots \beta_n] \sqsubseteq_D [\alpha w]$.

In other words, even if α is enabled from $x = val(s)$, the reduced state space G' may not contain an edge labeled by α exiting s. However, for each run π in which that state x occurs, there exists a path in G' that starts with s, and labeled with a sequence of operations $\beta_1\beta_2 \ldots \beta_n\alpha$, (where the operations $\beta_1\beta_2 \ldots \beta_n$ are independent of α) such that $\beta_1\beta_2 \ldots \beta_n\alpha$ is a possible continuation of π after the occurrence of x (i.e., $[v\,\beta_1\beta_2 \ldots \beta_n\alpha] \sqsubseteq_D \pi$). The proof of this property is by induction on the *order of closing nodes* by the expansion algorithm (at line 14 in Figure 1). When closing a node s, there are two possibilities:

1. All the operations enabled from $x = val(s)$ are expanded from s. This includes in particular the case where by applying one of them to x we reach an open node. In this case, the appropriate path, with a length of 1, exists trivially.

```
1   create_node(s, ι); set open(s); expand_node(s); /* initialization */
2   proc expand_node(s);
3      working_set(s) :=ample(val(s));
4      while working_set(s) ≠ φ do
5        α := some operation of working_set(s);
6        working_set(s) := working_set(s) \ {α};
7        succ_state := α(val(s)); /* the α-successor of val(s) */
8        if new(succ_state) then
9          create_node(s', succ_state); /* node s' has value succ_state */
10         set open(s'); /* set s' to open, i.e., on the search stack */
11         expand_node(s') fi; /* expand the successors of s' */
12       create_edge(s, α, s');
13     end while;
14     unset open(s); /* close s, i.e., remove it from search stack */
15  end expand_node.
```

Figure 1: Algorithm **A1**: an off-line reduced state space expansion algorithm

```
1   proc ample(x):set of T;
2      for i := 1 to num_proc /* repeat for every process */
3        E := check_succ(x, i); /* enabled P_i operations, if satisfy C1 */
4        if E ≠ φ then /* ...  otherwise, check_succ returns φ */
5          foreach α in E /* check cycle closing */
6            if exists s' with val(s') = α(x) and open(s') /* α closes a cycle */
7              then goto next_proc;
8          end foreach;
9          return(E) fi /* A subset satisfying C1 and C2 found */
10     next_proc: next i;
11     return(en(x)); /* cannot find a good subset */
12  end ample;
```

Figure 2: Routine **R1**: finding a subset of successors under the fairness **F**

2. A proper subset of operations enabled from x is calculated by $ample(x)$. By **C1** and hence **P1**, there exists an edge $s \xrightarrow{\gamma} s'$ such that γ is an immediate successor of this occurrence of x in the run π. By **C2**, none of the operations in $ample(x)$ applied to the state $x = val(s)$ closes a cycle. Thus, s' is not open. By **P2**, once completing the expansion of s, the node s' is already closed. Thus, the inductive hypothesis can be applied to s' to obtain a path that ends with α. Since both α and γ are immediately executed from the state $val(s)$ in π, they are interindependent and thus the edge $s \xrightarrow{\gamma} s'$ can be appended to the beginning of that path to form a longer path with the appropriate property.

The ability to use the reduced state graph is based on the following theorem [16]:

Theorem 3.1 *The reduced state graph G' generated by algorithm* **A1**+**R1** *satisfies the following properties: (1) all the sequences generated by G' are interleaving sequences of P, and (2) for each run π of the program P, there exists at least one sequence that corresponds to some path of G', starting from its initial node.*

The first property is trivial. The second is proved by constructing for an arbitrary interleaving sequence of P, inductively on the length of its prefixes, an equivalent

sequence which is generated by G', using **P3**. This allows applying LTL model-checking algorithms to G', instead of to the full state space G for properties φ that satisfy:

P4 If $\xi \equiv_D \xi'$ (i.e., ξ and ξ' belong to the same run), then $\xi \models \varphi$ iff $\xi' \models \varphi$.

This is of course unsatisfactory, as (1) the checked property φ may not satisfy **P4**, and (2) it can be difficult to check if φ satisfies **P4**. To allow checking arbitrary nexttime-free temporal properties, we employ the following definition [18]:

Definition 3.2 *An operation $\alpha \in T$ is* visible *in φ if it can change the truth value of some predicate that appears in the checked property φ. Denote the set of operations that are visible in φ by $a(\varphi)$.*

Theorem 3.3 *Let φ be a nexttime-free LTL property and the dependency relation D' used satisfies $D' \supseteq (a(\varphi) \times a(\varphi))$. If $v \equiv_{D'} v'$, with $states(v, \xi)$, $states(v', \xi')$, then $\xi \models \varphi$ iff $\xi' \models \varphi$.*

The proof of this theorem [16] is based on showing that the two propositional sequences $\mathcal{F}(\xi)$ and $\mathcal{F}(\xi')$, are equivalent up to stuttering (see also Theorem 3.6). Since φ is next-time free, it cannot distinguish between these two propositional sequences [13]. Now, in order to force the premise of the theorem, instead of using a dependency relation D obtained by analyzing the commutativity between the operations of the program P, we can use $D' = D \cup (a(\varphi) \times a(\varphi))$. Notice that extending the dependency relation while maintaining its symmetry and reflexivity preserves the conditions in Definition 2.1. In [16] it is shown how to avoid adding all the pairs $a(\varphi) \times a(\varphi)$ to the dependency relation. This is based on the fact that if φ is written as a boolean combination of smaller temporal formulas φ_i, then we need to add to D the dependencies $\bigcup_i (a(\varphi_i) \times a(\varphi_i))$, which can be fewer than $a(\varphi) \times a(\varphi)$. This can be formalized as an additional requirement:

C3 The dependence relation D used satisfies besides the conditions of Definition 2.1 also that $D \supseteq \bigcup_i (a(\varphi_i) \times a(\varphi_i))$, where φ can be equivalently written as some boolean combination of the temporal properties φ_i.

Notice that there may be various ways to rewrite φ as a boolean combination (the most trivial of which is to use φ itself), some of which may give bigger dependencies relations than others. For example, if $\varphi = \Box(p_1 \wedge p_2)$, then $a(\varphi)$ includes all the operations whose execution may change the value of the predicates p_1 or p_2. But φ is logically equivalent to $\Box p_1 \wedge \Box p_2$ with $a(\varphi_1) = a(\Box p_1)$ includes dependencies between the operations that can change p_1, and similarly, for $a(\varphi_2)$. Thus, a pair of operations such that the first changes only p_1 but not p_2, and the second changes p_2 but not p_1 is in $a(\varphi) \times a(\varphi)$; however, these operations are not necessarily dependent according to **C3**.

When model-checking under a fairness assumption such as **F**, or any stronger assumption, e.g., process justice or process fairness, the algorithm **A1** with the routine **R1** are appropriate. One just has to apply a model-checking algorithm that is tuned to the fairness assumption used, as shown in [14], to the reduced state space. In case one is using a fairness assumption ψ that is strictly stronger than **F**, one needs also to add dependencies between operations, so that if $\xi \equiv_D \xi'$, then $\xi \models \psi$ iff $\xi' \models \psi$. To achieve this, one can apply requirement **C3** also to ψ, in the same way it is applied to φ [16].

3.2 Subsets Construction without Fairness Assumption

We remove now the fairness assumption **F**. Thus, we consider now all the runs π of P. In this case, the condition **C1** does not guarantee anymore the property **P1**: there may

be a run π that has no immediate successors for an occurrence of $x = val(s)$ among the subset of successors $\mathcal{E}(x)$ chosen by the algorithm **R1**. Condition **C3** is replaced now with the following:

C3' If $\mathcal{E}(x)$ does not include all the operations enabled from x, then none of the operations in the selected subset of operations $\mathcal{E}(x)$ is visible.

As will be evident from the following theorems, requirement **C3'** enforces that visible operations cannot be commuted. This means effectively that the commutativity is restricted by the dependency relation $D \cup (a(\varphi) \times a(\varphi))$. Thus, this is a stronger requirement than **C3**, which allows adding dependencies to D after decomposing the property φ to subformulas, each one of which contributing a smaller number of dependencies. We modify algorithm **R1** into algorithm **R2**, which is the same as the one in Figure 2, except for replacing the frame at line 6 with the following:

if $visible(\alpha)$ or (exists s' with $val(s') = \alpha(x)$ and $open(s')$).

Instead of **P1**, the following weaker property now holds:

P1' For every run $\pi = [v][w]$, such that $x = fin_v = val(s)$, $\mathcal{E}(x)$ either

a. there exists some $\alpha \in \mathcal{E}(x)$ such that $[\alpha] \sqsubseteq_D [w]$ or
b. the operations in $\mathcal{E}(x)$ are invisible and independent of all the operations that appear in w.

Consider now a string w to be a (finite or infinite) vector of operations. Denote by $w(i)$ the $i+1st$ operation in w (the first operation is $w(0)$), by $w(i..j)$ the operations in the places i through j, and by $w(n..)$ the operations of w except the first n. Let v be a finite string over T of length n. A *selection function* for v is a function $r : \{0 \ldots n-1\} \mapsto \{\mathrm{T}, \mathrm{F}\}$. Denote by v_r the string remaining after deleting all the symbols $v(i)$ where $r(i) = \mathrm{F}$. Denote by $v_{\bar{r}}$ the string remaining after deleting the symbols $v(i)$ where $r(i) = \mathrm{T}$. Selection functions are also extended to infinite strings in a natural way. Let $r \angle m$ be the selection function r shifted to the left m places, i.e., $r \angle m(i) = r(i+m)$.

The property **P3'** replaces now **P3**, where the underlined consequence is replaced by:

P3' ... for $u = \beta_1 \beta_2 \ldots \beta_n$, there exists a selection function r such that (1) $[u\,\alpha] = [\alpha\, u_r\, u_{\bar{r}}]$, (2) there exists w' such that $[u_r\, w'] = [w]$ (and hence $[u_r] \sqsubseteq_D [w]$), (3) the operations $\beta_1 \beta_2 \ldots \beta_n$ are invisible, and (4) the operations in $u_{\bar{r}}$ are independent of the operations in w'.

The proof of **P3'** is similar in details to the proof of **P3**.

Definition 3.4 *Let $A \subset T$ be a set of letters. Let v, $w \in T^\omega$. Define $v \preceq_D^A w$ if there exists a selection function r for w such that $v \equiv_D w_r$, the operations of $w_{\bar{r}}$ are among A, and if $r(m) = \mathrm{F}$, then the operations in $w(m+1..)_{r \angle m+1}$ are independent of $w(m)$.*

That is, $v \preceq_D^A w$ iff it is possible to remove from w some operations from A, which are independent of all the non-removed operations of w that come after them, and obtain a string which is equivalent to v. The fact that the reduced state space G' can be used instead of the full state space G to check that $P \models \varphi$ (although without the fairness assumption **F**, it does not satisfy consequence (2) of Theorem 3.1), is based on the following theorems:

Theorem 3.5 *For every interleaving sequence v of the program P, there exists an interleaving sequence v' of P generated by the reduced state graph G' obtained by* **A1+R2** *such that $v \preceq_{D'}^A v'$, with $D' = D \cup (a(\varphi) \times a(\varphi))$ and $A \cap a(\varphi) = \phi$.*

The proof is by constructing inductively (over the length of prefixes of v) a sequence v' from v, using **P3'**.

Theorem 3.6 *Let φ be a nexttime-free LTL formula with a set of visible operations $a(\varphi)$. Let $A \subseteq T$, $A \cap a(\varphi) = \phi$ and $D' = D \cup a(\varphi) \times a(\varphi)$. Let $v, v' \in T^\omega$ such that $v \preceq^A_{D'} v'$ and let states(v, ξ), states(v', ξ'). Then $\xi \models \varphi$ iff $\xi' \models \varphi$.*

Sketch of proof. It can be shown that $\mathcal{F}(\xi)$ and $\mathcal{F}(\xi')$ are equivalent up to stuttering: for every prefix v' of v with n occurrences of operations from $a(\varphi)$ of v, there exists a prefix w' of w with n occurrences of visible operations, and vice versa. Because the visible operations are all interdependent, these occurrences are the same and appear in both prefixes in the same order. This relies on properties of traces [15]. Then it follows that the interpretation mapping $\mathcal{F}$ assigns the same subset of propositions to the states $fin_{v'}$ and $fin_{w'}$.

To summary, theorems 3.5 and 3.6 (or Theorems 3.1 and 3.3, under fairness, respectively) assert that the algorithm **A1** with **R2** (or with **R1**, resp.) constructs a state graph G' that generates for each propositional sequence (fair propositional sequence, resp.) of a program P a propositional sequence which is equivalent to it up to stuttering. To guarantee it, the dependency relation used does not allow to commute the visible operations, i.e., those that can change the value of the propositional variables (when fairness is assumed, certain commutativity between visible operations is allowed by **C3**). Since the checked property φ is restricted to be nexttime-free, it cannot distinguish between any two stuttering equivalent sequences [13], and thus φ holds in all the sequences generated by G' iff it holds in all the interleaving sequences of P.

4 Combining the Reduction with On-the-fly Intersection

One may view $P \models \varphi$ as language containment: let L_P be the language of the propositional sequences of P, obtained from the interleaving sequences of P using the interpretation mapping $\mathcal{F}$. Let L_φ be the language of the propositional sequences satisfying φ. Then $P \models \varphi$ is the same as $L_P \subseteq L_\varphi$ or equivalently $L_P \cap \overline{L_\varphi} = \phi$ [11] (where $\overline{L_\varphi}$ is the compliment of the language L_φ w.r.t. the alphabet $2^{\mathcal{P}}$). This view is very important to the correctness of our on-the-fly algorithm.

To implement a checker for the latter condition, one can transfer the property $\neg\varphi$ into a finite Büchi automaton $\mathcal{B}$ that generates exactly the sequences of the language $L_{\neg\varphi}$ (which is the same as $\overline{L_\varphi}$), as shown in [20]. Such an automaton $\mathcal{B}$ is a quintuple $\langle S, \Delta, \Sigma, \delta, F\rangle$, where S is the set of automaton states, $\Delta \subset S$ is the set of *initial states*, $\Sigma = 2^{\mathcal{P}}$ is the alphabet (the labels on the transitions), $\delta \subseteq S \times \Sigma \times S$ is the transition relation, and F is the set of *accepting* states. The automaton $\mathcal{B}$ *accepts* an infinite sequence ξ iff there exists an infinite path in $\mathcal{B}$, starting with some state in Δ, whose *edges* agree upon the propositional variables with the *states* of ξ, such that some state in F appears in this path infinitely many times. The specification may directly be given as an automaton that accepts exactly the sequences of the negation of the checked property φ over a set of propositional variables $\mathcal{P}$, instead of as an LTL formula [11, 6, 2].

A state graph G of P can be treated as an automaton that generates the propositional sequences of P. The state graph G can be constructed "on-the-fly", i.e., while intersecting it with the automaton $\mathcal{B}$. This allows sometimes to find a counter example for the checked property before the entire graph for G is generated, or to eliminate generating some subgraphs that do not synchronize with $\mathcal{B}$, gaining in memory and time.

Let $\mathcal{A}$ be the Büchi automaton that generates the intersection of the language L_P of G and the language $L_{\neg\varphi}$ of $\mathcal{B}$. A transition $\langle x, y\rangle \xrightarrow{\langle\alpha,\beta\rangle} \langle x', y'\rangle$ of $\mathcal{A}$ corresponds

to a transition $x \xrightarrow{\alpha} x'$ of G and a transition $y \xrightarrow{\beta} y'$ of $\mathcal{B}$, such that x and β agree on the atomic propositions; denote by $same\_label(x, \beta)$ the fact that the node x and the transition β agree, i.e., are labeled by the same subset of $\mathcal{P}$. The initial states of $\mathcal{A}$ are $\{\iota\} \times \Delta$, i.e., the initial state of the program, paired with any initial state of $\mathcal{B}$. A combined state $\langle x, y \rangle$ of $\mathcal{A}$ is accepting iff y is accepting in $\mathcal{B}$. Checking for emptiness of the language accepted by $\mathcal{A}$ is done by checking iff there exists a cycle, reachable from some initial state, that contains some accepting state. In this case, the intersection is not empty (the path from an initial state that traverses this cycle indefinitely, with labels from $\Sigma = 2^{\mathcal{P}}$ on its edges, belongs to both L_P and $L_{\neg\varphi}$), and $P \not\models \varphi$.

Combining partial order reduction with on-the-fly model-checking allows reduction in space and time by both methods. Let L' be the language of the reduced state graph G'. As seen in Theorem 3.5, the constructed reduced state space G' satisfies (1) $L' \subseteq L_P$, i.e., G' generates only interleaving sequences of P, and (2) for each $\xi \in L_P$ there exists $\xi' \in L'$ such that $states(v, \xi)$, $states(v', \xi')$ for some $v, v' \in T^\omega$, and $v \preceq_D^A v'$. It follows from (2) by theorem 3.6 that L' contains for each sequence ξ of L_P a sequence ξ' that has the same truth value, i.e., $\xi \models \varphi$ iff $\xi' \models \varphi$. Now, $L_P \cap L_{\neg\varphi} \neq \phi$ iff there exists some $\xi \in L_P \cap L_{\neg\varphi}$ iff there exists some $\xi', \xi \preceq_D^A \xi'$ such that $\xi' \in L' \cap L_{\neg\varphi}$. This proves that it is sufficient to check the nonemptiness of the intersection of the languages of the automata G' (rather than G) and $\mathcal{B}$.

In the intersection of the reduced state graph G', seen as an automaton, and the automaton $\mathcal{B}$, for each combined state $\langle x, y \rangle$, a $\mathcal{B}$ transition from the second component y of the state is paired with every operation in a *subset* $\mathcal{E}(x, y)$ of the program operations enabled from the program-state component x. Conditions **C1** and **C3'** are not changed, ignoring the new component y. Condition **C2** is now changed, taking into consideration the Büchi component y:

C2' For a state $\langle x, y \rangle$, if $\mathcal{E}(x, y)$ is a proper subset of the enabled process i operations at state x, no transition $\alpha \in \mathcal{E}(x, y)$, applied to x produces a state $\langle \alpha(x), y \rangle$ of $\mathcal{A}$ that is still open.

The procedure $ample(x, y)$ that calculates such a subset is similar to $ample(x)$ in Figure 2. It uses the additional paramenter y in checking for the new condition **C2'**. The only change is the frame at line 6 which is replaced by the following (the underlined condition is not needed under fairness assumption **F**):

if $visible(\alpha)$) or (exists node s' with $val(s') = \langle \alpha(x), y \rangle$ and $open(s')$)

It is tempting to reduce the correctness of the on-the-fly algorithm to the on-line version **A1** that was presented in the previous section by projecting each combined state into its first (i.e., program) component. However, this does not work: for each program-state there might be several combined states with different second component. However, one can still make a similar reduction, from the on-the-fly version into a non-deterministic variant **A2** of **A1**, described below. In *italics* we give the actual choice done when the algorithm is combined with the property automaton:

1. Choose a number n, which is the maximal number of nodes that can have the same program-state value. *This number is actually the number of states of the Büchi automaton $\mathcal{B}$.*
2. When checking in **R2** whether an operation α in a subset $\mathcal{E}(x)$ of the enabled operations closes a cycle (line 6 in Figure 2), if indeed a cycle would be closed by α because of reaching some state x that is the value of some open node (this excludes the case that x is also the value of some closed node, which allows

eliminating the cycle), and there are less than n nodes with value x, decide non-deterministically if to open a new node with value x or to discard the subset $\mathcal{E}(x)$ and choose another subset. *Actually, closing a cycle is done exactly when the* combined *state of the program and the Büchi automaton already exists and is open (i.e., is in the search stack).*

3. The algorithm **A2** *must* create an additional node s' for a state $x' = \alpha(x)$ for $x = val(s)$ if a proper subset of the enabled operations that contains α was returned by **R2** and there exist only *open* nodes with value x' (i.e., s' is needed to eliminate closing a cycle). However, **A2** can non-deterministically choose to create a new node even in case that a closed node with value x' exists, or in case that $ample(x)$ returns the set of all the enabled operations from x. *Actually, a new node is created iff there exists no node with the value of the new* combined *state.* If it was decided not to create a new node for x', and there exist multiple nodes with the value x', choose non-deterministically to connect the edge labeled α from s to one of them. This non-deterministic choice replaces lines 8-12 in Figure 1. *Actually, the connection choice is made to accord also with the value of the Büchi component.*[3]

The above changes to **A1** maintain the arguments in the proof and hence in the correctness of property **P3'**. It is easy to repeat the proof of Theorem 3.5 with respect to the new version of the algorithm, namely **A2+R2**.

The language of an automaton $\mathcal{A}$ is nonempty iff there exists an accepting state, accessible from an initial state, which is reachable from itself. The algorithm in [2] seeks in DFS order the accepting states. It stores them in its search stack, and then for each one of them, in reversed order (i.e., treating the last accepting state that was found first), it looks for a cycle. The cycle search is also done using a DFS algorithm. The execution of the two DFS algorithms, DFS1 and DFS2, are interleaved. The adaptation of this algorithm to partial order model-checking is by using the procedure $ample(x, y)$ as described above. This algorithm **A3** appears in Figure 3.

DFS1 (lines 1–20) is used to generate the intersection of the reduced state space and the checked automaton. Its correctness is based on using a projection from combined states to their first component, i.e., the program-states: the obtained projected structure is the same as the one produced by the above non-deterministic algorithm **A2**. When calling $ample(x, y')$ at line 4, the value of y' is a successor of the current $\mathcal{B}$ state y. That is, we apply first a Büchi transition β to y, such that the label of the transition β has the same subset of $\mathcal{P}$ as the current program-state x. Then the program operations $\mathcal{E}(x, y')$ returned by $ample(x, y')$ are combined with β such that for each $\alpha \in \mathcal{E}(x, y')$, $\langle \alpha, \beta \rangle$ is the $\mathcal{A}$ transitions taken from $\langle x, y \rangle$ to generate the new combined state $z = \langle \alpha(x), \beta(y) \rangle$ (line 5).

DFS2 (lines 21–30) searches for cycles through accepting states in the subgraph that was already generated by DFS1. DFS2 is only executed from an accepting node s after backtracking to it. Thus, from properties of the DFS algorithm, either a cycle through s was generated by DFS1, or all the nodes of $\mathcal{A}$ reachable from s where already generated by DFS1. This guarantees that the ability of the combined algorithm to find cycles through s, as proved in [2], is preserved. Alternatively, one can trade gain in space for time by regenerating the successors of $\langle x, y \rangle$ in DFS2 using $ample(x, y)$ and thus avoiding to store the edges (lines 8 and 12). The algorithm returns $true$ iff $P \not\models \varphi$, i.e., a reachable cycle (which is a counter example to $P \models \varphi$) was found.

[3]The intersection of G' with $\mathcal{B}$ might cause that a successor x' of the current program state x in G' will not be generated as no transition of $\mathcal{B}$ agrees with the label of x, even when there are program operations enabled from x. This (beneficially) eliminates expanding a node for x' and its successors. It corresponds in **A2** to choosing to close existing cycles rather than generating new multiple nodes.

```
1   proc DFS1(s):boolean
2     ⟨x, y⟩ := val(s); /* x = program-state, y = Büchi component */
3     forall β ∈ Σ, y' ∈ S s.t. same_label(x, β) and y --β--> y' ∈ δ do
4       forall α ∈ ample(x, y')
5         z := ⟨α(x), y'⟩;
6         if new(z) then
7           create_node(s', z);
8           create_edge(s, s');
9           set open(s'), unchecked2(s'); /* s' not participated in DFS2 */
10          if DFS1(s') then return true fi
11        else s' := node(z) ; /* s' is an existing node with value z */
12          create_edge(s, s') fi;
13      od
14    od;
15    if accepting(s) then /* if backtracked to an accepting node */
16      seed:=s; /* seek for a cycle through s */
17      return DFS2(s) fi /* do secondary search */
18    unset open(s); /* remove s from search stack and backtrack */
19    return false
20  end DFS1;

21  proc DFS2(t):boolean
22    forall t' ∈ succ(t) do /* use successors of t generated by DFS1 */
23      if t'= seed then return true ; /* a cycle was found */
24      if unchecked2(t') then /* if t' did not participate in DFS2 */
25        unset unchecked2(t'); /* mark that t' participated in DFS2 */
26        if DFS2(t') then return true fi
27      fi
28    od;
29    return false
30  end DFS2;
```

Figure 3: Algorithm **A3**: On-the-fly cycle-detection

In [2], an algorithm for reachable cycle detection that can deal with certain fairness assumptions is presented. It is based on finding *num_proc* cycles, one for satisfying the fairness condition for each process. This algorithm can also be combined with partial order model-checking. In this case, conditions **C1**, **C2'** and **C3** restrict the selection of subsets of operations returned by calling $ample(x, y)$. Certain fairness assumptions, such as strong fairness, require a substantial change of the algorithm in [2], and will be presented in the full version.

This can be used to check fairness assumptions which are at least as strong as (i.e., imply) **F**. However, as explained in [16], one has to add additional dependencies, so that the fairness assumption will satisfy that if $\xi \equiv_D \xi'$, then $\xi \models \psi$ iff $\xi' \models \psi$. That is, two equivalent sequences will be either both fair or both unfair. To achieve this, condition **C3** is applied also to the fairness assumption ψ the same way as it is applied to the checked property φ.

5 Implementation and Experimental Results

An instance of this algorithm was implemented by Gerard Holzmann as an extension of the model-checker SPIN [8]. SPIN [6] is a protocol validator that checks protocols and specifications written in the language PROMELA. The main criterion for choosing the particular implementation details was to avoid adding significant overhead in time and memory for making the reduction over the classical full search. In this way, the reduction can start to pay-off quickly, as no compensation for the additional cost is needed. This motivates the following implementation details.

A dependency relation was defined between pairs of PROMELA operations. It includes dependencies between operations referring to the same global objects, e.g., global variables, synchronous queues (but excluding asynchronous queues, where reads and writes can be commuted), operations of the same process, and operations that refer to an object that appears in the checked assertion (making all visible operations dependent as required in Theorems 3.3 and 3.6). However, no explicit representation of the dependency relation is calculated or stored. Its definition is merely used to prove that the actual algorithm used to calculate a subset of successors (*check_succ* at line 3 in Figure 2) indeed enforces the condition **C1**.

During pre-model-checking compilation of the checked PROMELA protocol and specification, each program control-state l of each process-definition is analyzed and annotated by one of three types of labels. These labels correspond to whether at run time, when this process is at control-state l, the set of enabled operations of this process satisfies condition **C1**:

safe It is already known at compile time that this process' set of enabled operations can be chosen.

$\langle$*maybe*, $C\rangle$ the set of this processes' enabled operations can be chosen only if the precomputed condition C (which is one out of a small number of conditions) holds during run time. For example, if the control-state l includes only receive operations, such a condition C can be that all of their receiving queues are non-empty.

not_safe The set of this process' enabled operations cannot be chosen.

The results of checking the liveness property $\Box\Diamond at(m)$, i.e., that a label m is visited infinitely often (which was implemented by introducing a global variable that changes its value just before and after label m) for various protocols is summarized in the following table. Memory is given in Megabytes and time in seconds. The first line in each pair of line shows the measurements for the full search, while the second line shows the measurements for the reduced search. All measurements were made on a 40MHz SGI 4D/480S R3k processor, with 128 Mbyte of Memory.

Alg.	States	Trans.	Mem.	Time	Comment
leader	335919	1858549	66.6	124.5	leader election
	829	1521	1.2	0.1	in ring
sorting	287736	1787327	46.4	102.6	distributed sorting
	1541	2911	1.3	0.2	
urp	6491	24241	2.2	1.8	AT&T universal
	3301	7001	1.8	0.6	receiver
snoopy	287230	1342296	35.6	101.8	cache coherence
	131873	292519	17.5	29.0	
pftp	1601373	6492515	254.5	695.9	file copy
	419076	905644	68.1	105	

Acknowledgement. The author is grateful for Gerard Holzmann, who implemented the partial order reduction on SPIN, and Ramesh Bharadwaj for helping in debugging the system. Both of them and R. P. Kurshan gave many helpful comments.

References

[1] J. R. Büchi, On a decision method in restricted second order arithmetic, in E. Nagel et al. (eds.), Proceeding of the International Congress on Logic, Methodology and Philosophy of Science, Stanford, CA, Stanford University Press, 1960, 1–11.

[2] C. Courcoubetis, M. Vardi, P. Wolper, M, Yanakakis, Memory-efficient algorithms for the verification of temporal properties, Formal methods in system design 1 (1992) 275–288.

[3] J. C. Fernandez, L. Mounier, C. Jard, T. Jeron, On-the-fly verification of finite transition systems, Formal Methods in System Design 1 (1992), Kluwer, 251–273.

[4] P. Godefroid, Using partial orders to improve automatic verification methods, Computer Aided Verification 1990, DIMACS, Vol 3, 1991, 321–339.

[5] P. Godefroid, P. Wolper, A Partial Approach to Model Checking, $6th$ LICS, 1991, Amsterdam, 406–415.

[6] G. J. Holzmann, Design and Validation of Computer Protocols, Prentice Hall Software Series, 1992.

[7] G. J. Holzmann, P. Godefroid, D. Pirottin, Coverage preserving reduction strategies for reachability analysis, Proc. IFIP, Symp. on Protocol Specification, Testing, and Verification, June 1992, Orlando, U.S.A., 349-364.

[8] G. J. Holzmann, D. Peled, STREM: A static reduction method, Manuscript, 1994, available from the authors.

[9] S. Katz, D. Peled, Verification of distributed programs using representative interleaving sequences, Distributed Computing 6 (1992), 107–120.

[10] 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.

[11] R. P. Kurshan, Reducibility in analysis of coordination, Lecture Notes in Communication and Information, Springer, 103, 19–39.

[12] M. Z. Kwiatkowska, Fairness for non–interleaving concurrency, Phd. Thesis, Faculty of Science, University of Leicester, 1989.

[13] L. Lamport, What good is temporal logic, IFIP Congress, North Holland, 1983, 657–668.

[14] O. Lichtenstein, A. Pnueli, Checking that finite-state concurrent programs satisfy their linear specification, $11th$ ACM POPL, 1984, 97–107.

[15] A. Mazurkiewicz, Trace semantics, Advances in Petri Nets 1986, LNCS 255, Springer, 1987, 279–324.

[16] D, Peled, All from one, one for all, on model-checking using representatives, $5th$ international conference on Computer Aided Verification, Greece, 1993, LNCS, Springer, 409–423.

[17] A. Valmari, Stubborn sets for reduced state space generation, $10th$ International Conference on Application and Theory of Petri Nets, Vol. 2, 1–22, Bonn, 1989.

[18] A. Valmari, A Stubborn attack on state explosion, in E.M. Clarke, R.P. Kurshan (eds.), CAV'90, DIMACS, Vol 3, 1991, 25–42.

[19] A. Valmari, On-The-Fly Verification of stubborn sets, $5th$ CAV, Greece, 1993, LNCS 697, Springer, 397–408.

[20] P. Wolper, M.Y. Vardi, A.P. Sistla, Reasoning about infinite computation paths, Proceedings of $24th$ IEEE symposium on foundation of computer science, Tuscan, 1983, 185–194.