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 an automaton representing 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 reduction from both methods. 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. We also describe how to apply partial-order model-checking under given fairness assumptions.

1 Introduction

Partial order model-checking is an approach for reducing time and memory when checking that concurrent programs satisfy their temporal logic specifications. When modeling the executions of a program as interleaved sequences of atomic actions, concurrent activities are interleaved in many possible orders. Partial order methods exploit the observation that 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, there 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 [13], the model-checking methods of Valmari [24], and of Godefroid [7] were limited to dealing with safety properties, termination, local and stable properties. Later, Valmari [25] extended his method to handle arbitrary nexttime-free temporal properties. Peled [22] generalized these ideas and showed how to gain more reduction by rewriting the checked formula, and how to apply the model-checking under fairness assumptions.

We suggest here an algorithm that combines on-the-fly model-checking [15, 6, 5] with partial order reduction. That is, it intersects 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 construction enjoys the advantage that it may not be required to be completed; a violation of the checked property may be found before the end of the construction, and parts of the (reduced) state space may not be necessary in the intersection. The method allows checking the class of stuttering-closed [17] Büchi automata properties, which includes the nexttime-free temporal properties. The reduction method presented here also shows how to apply partial order reductions on-the-fly under various fairness assumptions.

Other algorithms that exploit partial-order reductions while doing on-the-fly construction where suggested in [9] and [26]. These algorithms use a different model than ours to represent the specification, namely an automaton over sequences of operations, rather than a state-based specification. These algorithms also differ in the way the subsets of operations are constructed, and the way that validity of the checked property is enforced. Treatment of fairness conditions is also different: we take advantage of the fairness conditions when calculating the subsets of operations. A comparison with these work appear in Section 5.

Our reduction method is presented as a collection of constraints on the selection of an appropriate subset of the enabled operations from each given program state. The constraints are simple and easy to impose on state-space based model-checkers. The cost of the additional calculations required in order to find appropriate subsets of operations in time and space 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 [10]. This is the first implementation of a partial order model-checker with the full power of stuttering closed Büchi automata [4]. Experiments with various known algorithms and protocols show substantial reductions in space and time.

In section 2, some background is given. Section 3 presents an algorithm for constructing reduced state space for off-line model-checking of temporal properties. Section 4 shows how to combine the reduction of the state space with on-the-fly model-checking of temporal properties. Section 5 compares this work to other related methods. Finally, section 6 discusses an implementation of the on-the-fly partial-order model-checking and gives some experimental results.

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* or an *execution* of a program is an infinite 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. $q_0 = \iota$, and

2. for each $i \geq 0$, $q_i \in en_{\alpha_{i+1}}$ holds, and $q_{i+1} = \alpha_{i+1}(q_i)$.

We restrict the executions to infinite sequences only for simplifying the presentation. It is easy to change the definitions and proofs to deal with both finite an infinite sequences.

The interleaving semantics of a program sometimes involves a restricting condition on its interleaving sequences called *fairness*. Then, the set of executions of the program is limited to the sequences satisfying the assumed fairness condition.

An *admissible* sequence is an interleaving sequence or any segment of it, i.e., a suffix of a prefix, of such a sequence. We represent an admissible sequence either as a sequence 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, $\ldots$). The fact that ξ is the sequence of states obtained by executing the sequence of operations v from the 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 an irreflexive 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 $q \in en_\alpha$ (i.e., α is enabled from q), then $q \in en_\beta$ iff $\alpha(q) \in en_\beta$.*
- *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 with respect to the given alphabet T, i.e., $D = (T \times T) \setminus I$.*

Two strings $v,\, w \in T^*$ are considered equivalent [20], 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 $u_{i+1} = \bar{u}\beta\alpha\hat{u}$ for some $\bar{u},\, \hat{u} \in T^*$, $\alpha,\, \beta \in T$, $(a,\, \beta) \in I$. 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 infinite strings [16]. 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 strings from $T^* \cup T^\omega$ (i.e., finite or infinite) 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 [16]. A *trace* is an equivalence class of finite [20] or infinite [16] sequences of operations. 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 $fin_v = fin_{v'}$. Thus, we will denote $fin_{[v]}$ by fin_v. The operations that appear in a string v or a trace σ are denoted by $op(v)$ or $op(\sigma)$, respectively.

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 $\sigma\sigma' = [\alpha_0\, \alpha_1 \ldots \alpha_n \beta_0\, \beta_1 \ldots \beta_m \ldots]$. 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 $\rho = \sigma\, \sigma'$. We say that σ *is subsumed* by ρ. 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.

The conditions for independence in Definition 2.1 can be relaxed in several ways. One possibility is to replace the first condition by if $q \in en_\alpha \cap en_\beta$, then $\alpha(q) \in en_\beta$. This allows α and β to be independent even if at some cases they can appear in one order but not in the other. This change slightly complicates the definitions (and hence the presentation), as now $v\alpha\beta w \equiv_D v\beta\alpha w$ iff both $v\alpha\beta w$ and $v\beta\alpha w$ are admissible sequences. The theorems in this paper are not affected. For example, sending and receiving messages over a buffer cannot be independent according to Definition 2.1: it is sometimes possible to do a send before a receive but not vice versa (when the message queue is initially empty), or to do only a receive before a send (when message the queue is full). The relaxed definition allows considering these two operations as independent.

Two sequences are *equivalent up to stuttering* if when we replace in both sequences every finite adjacent number of occurrence of the same program state with a single occurrence, we obtain the same sequence. A property is stuttering closed if it cannot distinguish between stuttering equivalent sequences.

A nexttime-free LTL [18, 17] formula φ is constructed from the 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 from the modal '$\bigcirc$' (next-time). Without the nexttime modal, a linear temporal formula cannot distinguish between two sequences that are equivalent up to stuttering [17]. 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 the truth value $\mathbf{T}$ in q (the others are assigned the truth value $\mathbf{F}$). We assume the existence of an interpretation mapping $\mathcal{F}_{\langle P, \mathcal{P} \rangle}$ 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 temporal property φ is said to be *equivalence robust* if for every pair of sequences ξ and ξ', if $\xi \equiv_D \xi'$ (i.e., ξ and ξ' belong to the same run), then $\xi \models \varphi$ iff $\xi' \models \varphi$.

A *state graph* of a program P, which can be used to represent the executions 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 $val(t) = \alpha(val(s))$. If $s \neq t$, then $val(s) \neq val(t)$. (Later we will generalize $val(s)$ to include additional information, and allow two nodes to represent the same program state.) 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 $\alpha_1\alpha_2\ldots$.

An algorithm to generate the *full* state graph of a program can be obtained from the one in Figure 1, by replacing the underlined call to the routine $ample(val(s))$ at line 5 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. After a node s is removed from the search stack, it becomes *closed*, hence $open(s)$ does not hold anymore. A state q is *new* if there is no node with that value. In this case, $new(q)$ holds.

3 Spawning Reduced State Graphs

With off-line model-checking, the state space is first constructed and then some verification algorithms are applied to it. We start the presentation with an algorithm that constructs off-line reduced state spaces. It uses a modified depth-first-search (DFS) to expand the reduced state space of the checked program. The reduced state space preserves the checked property. Thus, a model-checking algorithm, such as described in [18] can be applied directly to the reduced state space.

Model-checking under fairness assumptions is usually more complicated than without such an assumption [18, 3]. However, using partial order reductions, it is easier to explain the reduction principles under one particular fairness assumption. We thus *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**. The **F**-fair sequences were shown [16, 21] to be exactly the set of maximal traces with respect to the subsumption relation '$\preceq$'. Furthermore, **F** is equivalence robust [16]. Thus, **F** can be also viewed of as a property of infinite traces, rather than sequences. Adding this fairness assumption slightly changes the algorithm and greatly simplifies its proof. Later, it is explained how to model-check under stronger fairness assumptions. In Section 3.2, the fairness assumption **F** is removed and all the runs of the program P are considered. Thus, two versions of the algorithm, one under the fairness requirement **F** (or any stronger assumption), and one under no fairness assumption, will be presented.

Notice that the off-line model-checking algorithm that is subsequently applied to the reduced state space in order to check the property must correspond to the assumed fairness condition. The algorithms in [18] cover various frequently used notions of fairness. Similar algorithms that handle other notions of fairness can be formed from them using slight modifications. Section 4 will present an on-the-fly algorithm for model-checking using partial-order reduction.

3.1 Reduction under Fairness

The reduced state space construction algorithm is described here using three constraints **C1**–**C3** on the selection of a subset of the enabled operations from each given program state. When the algorithm **A1**, depicted in Figure 1, expands a node s with $val(s) = x$, only a non-empty subset $ample(x)$ of the enabled operations $en(x)$ is used to generate successors for s (line 5 in Figure 1). An algorithm **R1** that calculates $ample(x)$ appears in Figure 2. Such a subset $ample(x)$ must satisfy the following condition:

- **C1** No operation $\alpha \in T \setminus ample(x)$ that is dependent on some operation in $ample(x)$ can be executed in P after reaching the state x and before some operation in $ample(x)$ is executed.

The condition **C1** is based on the notion of a *faithful decomposition* of operations, first defined in [13] and used in a proof method for verification of concurrent programs. It was used for model-checking in [22, 8]. It follows that under the fairness assumption **F**, the condition **C1** guarantees that:

Lemma 3.1 *Let* $\mathrm{ample}(x)$ *be a set satisfying condition* **C1**. *For every* **F**-*fair run* $\pi = [v][w]$, *such that* $v \in T^*$, $w \in T^\omega$, *where* $fin_v = x$, *there exists an operation* $\alpha \in \mathrm{ample}(x)$ *s.t.* $[\alpha] \sqsubseteq_D [w]$.

Proof. Take the maximal prefix w' of w that does not contain any operation from $ample(x)$. According to **C1**, it contains only operations independent of those in $ample(x)$. According to the fairness assumption **F** (recall that **F** is equivalence robust, thus, in particular, w must satisfy F), w' must be a proper prefix of w. Let α be the first operation in w after w'. Then, $\alpha \in ample(x)$, hence all the operations in w' are independent of α. Thus, $w'\alpha \equiv_D \alpha w'$ and hence $[\alpha] \sqsubseteq_D [\alpha w'] = [w'\alpha] \sqsubseteq_D [w]$. ∎

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 Lemma 3.1, $ample(x)$ returns at least one immediate successor operation for each occurrence of x, for each run π of P (with respect to the dependency relation D). The Algorithm **R1** uses the routine $check\_succ(x, i)$. This routine checks if the operations of process P_i that are enabled from the state x satisfy the property **C1** (at line 3 in Figure 2). If they do, it returns these operations. Otherwise, it returns the empty set. More details on how our implementation checks this condition are described in Section 6 and in [12].

A second condition [22] is enforced at lines 5–8 in Figure 2. It is needed to prevent some pathological cases where the execution of operations is indefinitely deferred along a cycle.

```
create_node(s, ι);
set open(s);
expand_node(s);

proc expand_node(s);
    working_set(s) := ample(val(s));
    while working_set(s) ≠ φ do
      α := some operation of working_set(s);
      working_set(s) := working_set(s) \ {α};
      succ_state := α(val(s)); /* the α-successor of val(s) */
      if new(succ_state) then
        create_node(s', succ_state); /* node s' has value succ_state */
        set open(s'); /* set s' to open, i.e., on the search stack */
        expand_node(s') fi; /* expand the successors of s' */
      create_edge(s, α, s');
    end while;
    unset open(s); /* close s, i.e., remove it from search stack */
end expand_node.
```

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

C2 If *ample*(x) is a *proper* subset of the operations enabled from $x = val(s)$, then for no operation $\alpha \in ample(x)$ it holds that $\alpha(x)$ is on the search stack.

In other words, if one of the operations α closes a cycle, the above condition does not allow selecting as an ample set a proper subset of the enabled operations from x. That is, $\alpha(x)$ is already on the search stack, i.e., it is open.

A routine *ample* that preserves the requirements **C1** and **C2**, guarantee the following property, throughout the execution of the expansion algorithm **A1**:

Theorem 3.2 *Let s be a closed node and let $\pi = [v][\alpha w]$, with $\alpha \in T$, be an* **F***-fair 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 $\beta_1\beta_2\ldots\beta_n$ are independent of α and $[\beta_1\beta_2\ldots\beta_n\alpha] = [\alpha\beta_1\beta_2\ldots\beta_n] \sqsubseteq_D [\alpha w]$.*

In other words, even if α is enabled from $x = val(s)$, the reduced state space G' may omit the edge labeled by α exiting from s; however, for each **F**-fair 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., $[v\,\beta_1\beta_2\ldots\beta_n\alpha] \sqsubseteq_D \pi$).

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

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

Proof. By induction on the *order of closing nodes* by the expansion algorithm (at line 16 in Figure 1). When closing a node s, there are two possibilities:

1. $\alpha \in ample(x)$. In this case, the appropriate path, with the length of 1, exists trivially.

2. $\alpha \notin ample(x)$. Thus, a proper subset of operations enabled from x, not including α, is calculated by $ample(x)$. By **C1** and hence Lemma 3.1, 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. It is a property of the DFS algorithm that if none of the edges from a node s closes a cycle, once completing the expansion of s, then all of its immediate successors are already closed. Thus, the inductive hypothesis can be applied to s' to obtain a path that ends with α. The operation γ can be commuted to the front of $[\alpha w]$ (i.e., $[\alpha w] = [\gamma w']$ for some $w' \in T^*$). Thus, both α and γ can immediately follow $[v]$ in π. Hence α and γ are interindependent and thus the edge $s \xrightarrow{\gamma} s'$ can be appended to the beginning of this path to form a longer path with the appropriate property. ∎

The ability to use the reduced state graph is based on the following theorem [22]:

Theorem 3.3 *The reduced state graph G' generated by algorithm* **A1** *with a routine ample that preserves* **C1** *and* **C2** *satisfies the following properties:*

1. *all the sequences generated by G' are interleaving sequences of P, and*

2. *for each interleaving sequence v of the program P, there exists at least one sequence v' that corresponds to some path of G', starting from its starting node $\hat{s}$, such that $v \equiv_D v'$.*

Proof. The first property is trivial from the DFS construction. The second is proved by constructing for an arbitrary interleaving sequence v of P an equivalent sequence which is generated by G'. Let $G' = \langle \hat{s}, V, E \rangle$. The state graph G' is treated as an automaton over infinite words. We describe a traversal of the state graph G', starting from $\hat{s}$, that produces a sequence equivalent to v, while reading v. Denote the following variables used for an execution of the automaton G' over the word v:

r The sequence of operations read so far from v.

t The sequence of operations labeling the edges of G' traversed so far.

l The operations traversed on the edges of G' but has not yet read on the input during the execution.

s The current state of G'.

The variables r, t and l are initialized to the empty word ϵ, while s is initialized to $\hat{s}$. Whenever the next letter $\alpha \in T$ is read from the input word v, the following updates are made:

1. let $r := r\,\alpha$;
2. if $l = u\alpha w$ for some $u,\ w \in T^*$ with $op(u)$ independent of α, then let $l = u\,w$,
3. else, choose a sequence $\beta_1\beta_2\ldots\beta_n\alpha$ from the node s, ending with a state s' such that $[t\beta_1\beta_2\ldots\beta_n\alpha] = [t\alpha\beta_1\beta_2\ldots\beta_n] \sqsubseteq_D [v]$. let
   (a) $s := s'$;
   (b) $l = l\,\beta_1\beta_2\ldots\beta_n$.
   (c) $t := t\,\beta_1\beta_2\ldots\beta_n\,\alpha$.

Now the following invariants can be inductively proved to hold before and after reading each letter of v:

(1) $[r]\,[l] = [t]$.

(2) $[t] \sqsubseteq [v]$.

(3) If the condition of Step 2 does not hold when checking it, then all the operations in $op(l)$ are independent of α.

(4) The choice of sequence $\beta_1\beta_2\ldots\beta_n\,\alpha$ required by Step 3 can always be made when executing Step 3.

Initially, (1) and (2) trivially hold, since $r = l = t = \epsilon$. Moreover, (3) and (4) trivially hold because the execution is initially just before executing Step 1. At each step of the execution, $[r]\,[\alpha] \sqsubseteq_D [v]$ (as α is the next letter of v that is read after r) and $[r]\,[l] \sqsubseteq_D [v]$ (by the inductive hypotheses (1) and (2)). Thus, l cannot contain operations that are dependent on α before the first (if any) occurrence of α in it. This proves (3). In case that α does not occur in l, the existence of the sequence $\beta_1\beta_2\ldots\beta_n$ from s is guaranteed by Theorem 3.2, proving (4). It is now easy to check that both (1) and (2) are preserved by taking either Step 2 or Step 3 in the above algorithm.

Let v' be the infinite sequence of operations collected into the variable t during an infinite execution of the above automaton over an input v. Now, by (1), for every prefix r of v, $r \preceq_D t$, hence $r \preceq_D v'$. On the other hand, by (2), for arbitrarily long prefixes t of v', $t \preceq_D v$. Thus, $v \equiv_D v'$. ∎

This allows applying LTL model-checking algorithms to G', instead of to the full state space G, for properties φ that are equivalence robust. This is of course unsatisfactory, as (a) the checked property φ may not be equivalence robust, and (b) it can be difficult to check if φ is equivalence robust [22]. To allow checking arbitrary nexttime-free temporal properties, we employ the following definition [25]:

Definition 3.4 *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 $a(\varphi)$.*

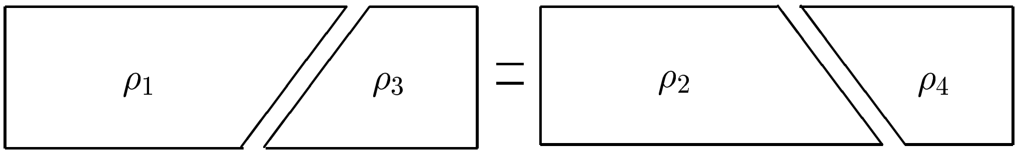
Theorem 3.5 *Let φ be a nexttime-free LTL property and the dependency relation D' used satisfies $D' \supseteq (a(\varphi) \times a(\varphi))$. If $v \equiv_{D'} v'$, with $states(v, \xi)$ and $states(v', \xi')$, then $\mathcal{F}(\xi)$ and $\mathcal{F}(\xi')$ are equivalent up to stuttering.*

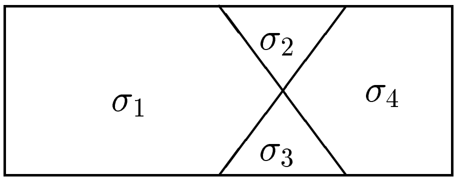
Proof. The proof is symmetric w.r.t. v and v'. Denote by $v\mid_A$ the projection of the string v on the operations in A (i.e., the sequence of operations remained after erasing all other operations). Then, for each prefix $\hat{v}$ of v there exists at least one prefix $\tilde{v}$ of v' such that $\hat{v}\mid_{a(\varphi)} = \tilde{v}\mid_{a(\varphi)}$. This stems from the fact that the visible operations in $a(\varphi)$ cannot be interpermuted because $D \supseteq a(\varphi) \times a(\varphi)$.

Now it will be shown that $\mathcal{F}(fin_{\hat{v}}) = \mathcal{F}(fin_{\tilde{v}})$. First, from the definition of a run as an equivalence class of execution sequences, there exists a finite trace σ that subsumes both $\rho_1 = [\hat{v}]$ and $\rho_2 = [\tilde{v}]$. Thus, there exists ρ_3, ρ_4 such that $\rho_1\,\rho_3 = \rho_2\,\rho_4 = \sigma$. The Levi Lemma for traces [20, Page 307] states that

> If $\rho_1\,\rho_3 = \rho_2\,\rho_4$, then there exist traces σ_1, σ_2, σ_3, σ_4 such that $\rho_1 = \sigma_1\,\sigma_2$, $\rho_2 = \sigma_1\,\sigma_3$, $\rho_3 = \sigma_3\,\sigma_4$ and $\rho_4 = \sigma_2\,\sigma_4$. Furthermore, $(op(\sigma_2) \times op(\sigma_3)) \cap D = \phi$, i.e., the operations in σ_2 are independent of the operations in σ_3. See Figure 3.

It follows that $op(\sigma_2) \cap a(\varphi) = \phi$ and $op(\sigma_3) \cap a(\varphi) = \phi$, i.e., the operations in σ_2 and σ_3 cannot change the values of the propositions appearing in the checked property φ. To see this, observe that it is not possible that both $op(\sigma_2)$ *and* $op(\sigma_3)$ contain visible operations, since





$\rho_1 = \sigma_1\,\sigma_2,\ \rho_2 = \sigma_1\,\sigma_3$
$\rho_3 = \sigma_3\,\sigma_4,\ \rho_4 = \sigma_2\,\sigma_4$
op(σ_2) are independent of op(σ_3)

Figure 3: The Levi Lemma for traces

we assumed visible operations to be interdependent. Assume without loss of generality that $op(\sigma_2)$ does not contain visible operations. Since $\rho_1 = \sigma_1\,\sigma_2$, both ρ_1 and σ_1 contain the same visible operations, occurring in the same order. Since $\rho_2 = \sigma_1\,\sigma_3$ also contains the same visible operations as ρ_1 and hence σ_1, it follows that σ_3 cannot contain any visible operation.

Thus, $\mathcal{F}(fin_{\rho_1}) = \mathcal{F}(fin_{\sigma_1\,\sigma_2}) = \mathcal{F}(fin_{\sigma_1}) = \mathcal{F}(fin_{\sigma_1\,\sigma_3}) = \mathcal{F}(fin_{\rho_2})$. Hence, for any two prefixes $\hat{v}$ and $\tilde{v}$ of the same execution sequence v such that $\hat{v}\mid_{a(\varphi)} = \tilde{v}\mid_{a(\varphi)}$, it holds that $\mathcal{F}(\hat{v}) = \mathcal{F}(\tilde{v})$. ∎

The specifications we use are stuttering closed, i.e., cannot distinguish between two sequences that are the same up to stuttering. It follows immediately that:

Theorem 3.6 *Let φ be a nexttime-free LTL property and the dependency relation D' that is used satisfies $D' \supseteq (a(\varphi) \times a(\varphi))$. If $v \equiv_{D'} v'$, with states(v, ξ), states(v', ξ'), then $\xi \models \varphi$ iff $\xi' \models \varphi$.*

Since φ is next-time free, it cannot distinguish between these two propositional sequences [17]. 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 [22] 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_1 and φ_2 are equivalence robust, so are $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$ and φ_1. Thus, if φ is written as a boolean combination of some temporal formulas φ_i, then we need to add to D the dependencies $\bigcup_i(a(\varphi_i) \times a(\varphi_i))$. This union is a subset of $a(\varphi) \times a(\varphi)$ and can be in some cases much smaller.

This can be formalized as an additional requirement:

C3 The dependence relation D used satisfies (besides the conditions of Definition 2.1) 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, some of which may give bigger dependency 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 $a(\varphi_1) = a(\Box p_1)$, which 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 $a(\varphi) \times a(\varphi)$, but is not necessarily dependent according to **C3**.

In order to express the checked property as a boolean combination, one can apply various rewriting rules to the given formula φ. For example:

$$\begin{aligned}
\Box(\eta \wedge \mu) &= \Box\eta \wedge \Box\mu \\
\Diamond(\eta \vee \mu) &= \Diamond\eta \vee \Diamond\mu \\
\Diamond\Box(\eta \wedge \mu) &= \Diamond\Box\eta \wedge \Diamond\Box\mu \\
\Box\Diamond(\eta \vee \mu) &= \Box\Diamond\eta \vee \Box\Diamond\mu \\
(\eta_1 \wedge \eta_2)\mathcal{U}\mu &= \eta_1\mathcal{U}\mu \wedge \eta_2\mathcal{U}\mu \\
\eta\mathcal{U}(\mu_1 \vee \mu_2) &= \eta\mathcal{U}\mu_1 \vee \eta\mathcal{U}\mu_2
\end{aligned}$$

Notice that rewriting a formula may cause an exponential blowup in its size. This should not be in general a problem, as the checked properties are relatively small. Moreover, it is not necessary to completely rewrite a formula; the rewriting rules can be used to obtain the additional dependencies needed without explicitly obtaining the new form of the formula. A recursive procedure can use these rules to obtain the separated component one at a time, and add dependencies accordingly. For example, the last rewriting rule above can be used to infer that if the formula is of the form $\eta\mathcal{U}(\mu_1 \vee \mu_2)$, rewriting it as $\eta\mathcal{U}\mu_1 \vee \eta\mathcal{U}\mu_2$ means that the subformula η is copied twice. Instead, this rule should be used to separate $\eta\mathcal{U}\mu_1$, adding new dependencies to D, and then separating $\eta\mathcal{U}\mu_2$. After doing this, the original formula can still be used for the actual model-checking.

To summary, when model-checking under a fairness assumption such as **F**, or any stronger assumption, e.g., process justice or process fairness, one can use Algorithm **A1** with a routine *ample* that preserves **C1**, **C2** and **C3**, e.g., **R1** in Figure 2. However, notice that **R1** is only an example of an algorithm that preserves these requirements. One just has to apply to the reduced state space a model-checking algorithm that is appropriate for to the fairness assumption used, as shown in [18].

As pointed out earlier, the fairness condition **F** itself is equivalence robust [16] and thus no additional dependencies are required when it is assumed. In case one is using a fairness assumption ψ that is strictly stronger than **F**, one needs also to add dependencies between operations to make it equivalence robust. This can be achieved by applying requirement **C3** also to the fairness formula ψ, in the same way it is applied to φ [22].

The fairness assumption **F** can be enforced by the off-line algorithm by seeking a strongly connected component such that if some operation α is enabled in some state of it then some operation dependent on α is executed in it. (The off-line algorithms do not work directly on the reduced state space, but rather construct from it a graph that contains in each node also information of the temporal properties that hold in it, see [18].)

3.2 Reduction without Fairness Assumption

The fairness assumption **F** is now removed. Thus, we consider all the runs π of P, instead of the **F**-fair ones. In this case, the condition **C1** does not guarantee that the conclusion of Lemma 3.1 holds: there may be a run π that has no immediate successors for an occurrence of a state $x = val(s)$ among the subset of successors $ample(x)$ even if conditions **C1**, **C2** and **C3** hold.

Condition **C3** is replaced now with the following:

C3' If $ample(x)$ is a *proper* subset of the operations enabled from x, then none of the operations in it is visible.

To preserve **C3'**, we modify algorithm **R1**, which appear in Figure 2 into algorithm **R2** by replacing the frame at line 6 with the following:

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

Let $D' = D \cup (a(\varphi) \times a(\varphi))$, i.e., the dependency of the checked program P, augmented with dependencies between all visible operations. Instead of Lemma 3.1, the following property now holds:

Lemma 3.7 *For every run $\pi = [v][w]$, w.r.t. the dependency D', such that $x = fin_v = val(s)$, $ample(x)$ satisfies either*

a. there exists some $\alpha \in ample(x)$ such that $[\alpha] \sqsubseteq_{D'} [w]$ or

b. the operations in $ample(x)$ are invisible and independent (according to D') of all the operations that appear in w.

Proof. Consider two cases:

1. $ample(x) = en(x)$. Then $ample(x)$ includes in particular the first operation α of w.

2. $ample(x) \subset en(x)$. Using **C1**, no operation dependent, *according to* D, on an operation from $ample(x)$ can appear in w before some operation from $ample(x)$ occurs. Moreover, by **C3'**, the operations of $ample(x)$ are in this case invisible, thus the above statement also holds when replacing D by D'. ∎

Notice that since we don't assume the property **F** here, it might happen that no operation from $ample(x)$ ever occurs in w.

According to Lemma 3.7 and the following theorems, requirement **C3'** effectively imposes the dependency relation $D' = D \cup (a(\varphi) \times a(\varphi))$. That is, it doesn't allow commuting operations that are visible. This is a stronger requirement than **C3** (which allowed adding dependencies to D after decomposing the property φ to subformulas, each one of which contributing a smaller number of dependencies).

Let w be a (finite or infinite) string of operations. Denote by $w(i)$ the $i+1st$ operation in w (the first operation is $w(0)$), by $w(i..j)$ the $i+1th$ through the $j+1th$ operations, and by $w(n..)$ the operations of w except the first n.

Definition 3.8 *Let v be a finite string over T of length n. A* selection function *for v is a function $r : \{0 \ldots n-1\} \mapsto \{\text{T}, \text{F}\}$. Denote by v_r the string remaining after deleting all the symbols $v(i)$ with $r(i) = \text{F}$. Denote by $v_{\bar{r}}$ the string remaining after deleting the symbols $v(i)$ with $r(i) = \text{T}$.*

For example, if $v = abcabcabc$, and r a selection function such that $r(i) = \text{F}$ for $i = 0, 3, 6$, then $v_r = bcbcbc$, and $v_{\bar{r}} = aaa$. Selection functions are also extended to infinite strings in a natural way. Denote by $r\angle m$ the selection function r shifted to the left m places, i.e., $r\angle m(i) = r(i+m)$.

The Theorem 3.9 replaces now Theorem 3.2:

Theorem 3.9 *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 the operations in the sequence $u = \beta_1\beta_2\ldots\beta_n$ are* invisible *and independent, according to D', of α and a selection function r, satisfying*

1. *$u\,\alpha \equiv_{D'} \alpha\, u_r\, u_{\bar{r}}$,*
2. *there exists w' such that $u_r\, w' \equiv_{D'} w$ (and hence $[u_r] \sqsubseteq_{D'} [w]$), and*
3. *the operations in $u_{\bar{r}}$ are independent, according to D', of the operations in w'.*

The proof of Theorem 3.9 is similar in details to the proof of Theorem 3.2.

Definition 3.10 *Let $A \subset T$ be a set of operations. 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 for each m in the domain of the selection function r, if $r(m) = \text{F}$, then $op(w(m+1..)_{r\angle m+1})$ are independent of $w(m)$.*

That is, $v \preceq_D^A w$ iff it is possible to obtain a string which is equivalent to v by removing from w some operations from A, which are independent of all the non-removed operations of w that come after them. For example, let $\Sigma = \{a, b, c\}$, $v = abc(ab)^\omega$, $D = \Sigma \times \Sigma$, $w = (cba)^\omega$ and $A = \{c\}$. Then, $v \preceq_D^A w$. To see that, let r be a selection function such that $r(i) = \text{F}$ for exactly the positive indexes i that are divisible by 3. Thus, $w_r = c(ba)^\omega$, i.e., every c, except the first one, was removed. But then, $v \equiv_D w_r$. Notice that every operation c that is removed is independent of the rest of the operations of w occurring to its right *that were not removed.* In this case, these are the occurrences of the operations a and b, except the first a and b.

The fact that the reduced state space G' can be used instead of the full state space G to check that $P \models \varphi$ (albeit the fact that without the fairness assumption **F**, it does not satisfy consequence (2) of Theorem 3.3), is based on the following theorems:

Theorem 3.11 *The reduced state graph G' generated by algorithm* **A1** *with a routine ample that preserves* **C1**, **C2** *and* **C3'** *satisfies the following properties:*

1. *all the sequences generated by G' are interleaving sequences of P, and*
2. *for each interleaving sequence v of the program P, there exists at least one sequence v' that corresponds to some path of G', such that $v \preceq^A_{D'} v'$, where $A \cap a(\varphi) = \phi$.*

Notice that since $A \cap a(\varphi) = \phi$, only invisible operations can be removed from v' to result in a sequence equivalent to v.

Proof. The proof is by constructing inductively (over the length of prefixes of v) a sequence v' from v, using Theorem 3.9. The details are similar to the proof of Theorem 3.3. However, now, the variable t contains also occurrences that do not appear in v. There are a few changes that are required from the proof of Theorem 3.3:

- At line 3(b), only the occurrences that appear in v among the sequence $\beta_1\beta_2\ldots\beta_n$ are appended to l. That is, no operation occurs in l more times than in v after this assignment.
- A selection function c is constructed together with t at line 3(c). Each time an occurrence of some operation γ is appended to t more times than it occurs in v, its position in t is mapped by the selection function c to $\textsc{f}$. Otherwise, c maps this position to $\textsc{t}$.
- The invariants are relativized to c. Namely,
    1. $[r][l] = [t_c]$.
    2. $[t_c] \sqsubseteq [v]$.
- For the infinite sequence v' collected in t in an infinite run, and for c', the infinite selection function collected in the variable c, the same arguments as in 3.3 holds, i.e., $v \equiv_{D'} v'_{c'}$. Thus, it follows from Definition 3.10 that $v \preceq^A_{D'} v'$. ∎

Theorem 3.12 *Let φ be a nexttime-free LTL formula with a set of visible operations $a(\varphi)$. Let $A \subseteq T \setminus a(\varphi)$. Let v, $v' \in T^\omega$ such that $v \preceq^A_{D'} v'$ and let states(v, ξ) and states(v', ξ') hold. Then $\xi \models \varphi$ iff $\xi' \models \varphi$.*

Proof. It can be shown that $\mathcal{F}(\xi)$ and $\mathcal{F}(\xi')$ are equivalent up to stuttering. The details are similar to the proof of Theorem 3.5. The difference in the proof lies with the existence of additional occurrences of invisible operations in v', which do not occur in v'. Now, by the definition of the relation '$\preceq^A_{D'}$', v and v' have the same visible operations, and because

$D' = D \cup (a(\varphi) \times a(\varphi))$, they also appear in both sequences in the same order. Now the proof proceeds as in Theorem 3.5 by selecting two prefixes $\hat{v}$ and $\tilde{v}$ of v and v', respectively, such that $\hat{v} \mid_{a(\varphi)} = \tilde{v} \mid_{a(\varphi)}$. Again, from the definition of '$\preceq_{D'}$', the occurrences of operations that were removed from v' are independent of all the operations that appear after them and were not removed. Thus, if such occurrences appear in $\tilde{v}$, they can be commuted to its end, obtaining $[\tilde{v}] = [\tilde{v}'w]$, with w containing only removed occurrences. The rest of the proof of Theorem 3.5 is repeated w.r.t. $\tilde{v}'$ instead of $\tilde{v}$. Then, we observe that $\mathcal{F}(fin_{\tilde{v}}) = \mathcal{F}(fin_{\tilde{v}'})$, as the operations that occur in w are invisible. ∎

To summary, Theorems 3.11 and 3.12 assert that the algorithm **A1** with a routine *ample* that satisfies **C1**, **C2** and **C3'**, e.g., **R2**, 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. Since the checked property φ is restricted to be nexttime-free, it cannot distinguish between any two stuttering equivalent sequences [17], and thus φ holds in all the sequences generated by G' iff it holds in all the interleaving sequences of P.

Checking a property φ under a fairness condition ψ that is not stronger than the fairness **F** (defined in the previous subsection) can be done as in the non-fair case. Then, one should apply condition **C3'** to ψ in order to force it to become equivalence robust. However, this typically makes all the program operations interdependent, resulting in no reduction. In practice, the typical fairness assumptions [19] are usually stronger than **F**, although some of them are not equivalence robust.

4 Reduction with On-the-fly Model-Checking

Model-checking on-the-fly means that the verification algorithm starts to examine the checked program *while constructing its state space*, not waiting for this construction to be completed. Thus, if the property does not hold, a counter example can be encountered before completing the construction. Another advantage is that some parts of the state space that are not important to the verification of the checked property can be eliminated in this way (even if the property holds in the checked program). Before presenting the partial-order version of the on-the-fly model-checking we first review some background of on-the-fly model-checking.

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 the LTL formula φ. Then $P \models \varphi$ is the same as $L_P \subseteq L_\varphi$ or equivalently $L_P \cap \overline{L_\varphi} = \phi$ [15] (where $\overline{L_\varphi}$ is the compliment of the language L_φ w.r.t. the alphabet $2^{\mathcal{P}}$).

To implement a verifier 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 [27]. 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. There is a growing practice [15, 10, 5] according to which the specification is directly given as an automaton that accepts exactly the sequences of $\neg\varphi$, i.e., the negation of the checked property, over a set of propositional variables $\mathcal{P}$.

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 a Büchi automaton that generates the intersection of the language L_P of G and the language $L_{\neg\varphi}$ of $\mathcal{B}$. 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 the propositions $\mathcal{P}$. The product of the automata G and $\mathcal{B}$, accepting the intersection of their languages, is obtained as follows: a transition $\langle x,\, y\rangle \xrightarrow{\langle\alpha,\beta\rangle} \langle x',\, y'\rangle$ of $\mathcal{A}$ corresponds to a pair of transitions:

1. The transition $y \xrightarrow{\beta} y'$ of $\mathcal{B}$, such that x and β agree on the atomic propositions, i.e., $same\_label(x, \beta)$, and
2. The transition $x \xrightarrow{\alpha} x'$ of G.

Such a combination of transitions is depicted in Figure 4; the combined transition is obtained in two steps: first, the $\mathcal{B}$ state is changed, and then the G component is changed. 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 if there exists a cycle, reachable from some initial state, that contains some accepting state. In this case, the intersection is not empty i.e., the path from an initial state that traverses this cycle indefinitely 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 $L_{G'}$ be the language of a reduced state graph G' that satisfies the properties in Theorem 3.11.

Theorem 4.1 *The intersection of the automaton G with $\mathcal{B}$ is empty iff the intersection of the reduced automaton G' and the property automaton $\mathcal{B}$ is empty, i.e., $L_P \cap L_{\neg\varphi} = \phi$ iff $L_{G'} \cap L_{\neg\varphi} = \phi$.*

Proof. The constructed reduced state space G' satisfies:

1. $L_{G'} \subseteq L_P$, i.e., G' generates only interleaving sequences of P, and

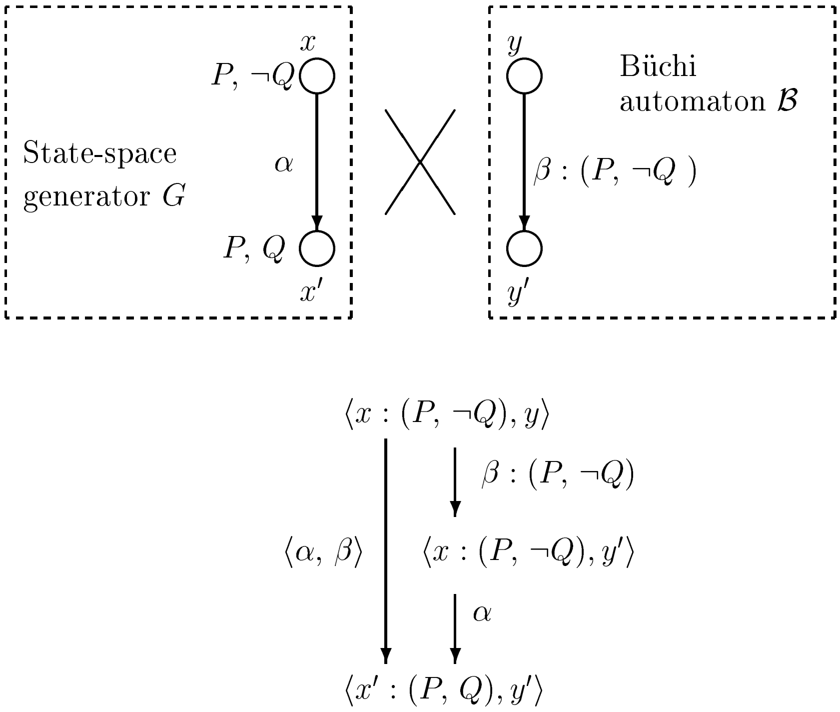


Figure 4: Combining transitions of the program and the property automaton

2. for each $\xi \in L_P$ there exists $\xi' \in L_{G'}$ such that $states(v,\ \xi)$ and $states(v',\ \xi')$ for some $v,\ v' \in T^\omega$, and $v \preceq_D^A v'$.

It follows by theorem 3.12 that $\xi \models \varphi$ iff $\xi' \models \varphi$. Thus, $L_P \cap L_{\neg\varphi} \neq \phi$ iff there exists some $\xi \in L_P \cap L_{\neg\varphi}$ iff there exists some ξ', $\xi \preceq_D^A \xi'$ such that $\xi' \in L_{G'} \cap L_{\neg\varphi}$. ∎

This proves that it is sufficient to check the nonemptiness of the intersection of the languages of the automata G' (rather than G) and $\mathcal{B}$, obtained by taking their product $\mathcal{A}'$. A transition of this product $\langle x,\ y\rangle \xrightarrow{\langle\alpha,\beta\rangle} \langle x',\ y'\rangle$ is formed from a transition $x \xrightarrow{\alpha} x'$ of G', where α is in a subset $ample(x,\ y)$ of the program operations enabled from the program-state component x, and a transition $y \xrightarrow{\alpha} y'$ of $\mathcal{B}$. Conditions **C1** and **C3'** from Section 3.2 are not changed, ignoring the new component y. Condition **C2** is now changed, involving the Büchi component y:

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

The routine $ample(x,\ y)$ that calculates such a subset is similar to $ample(x)$ in Figure 2. It uses the additional parameter y in checking for the new condition **C2'**. Since each node s contains both a program state and a property automaton state, $val(s)$ is generalized to return such a pair. The only change to **R2** is the frame at line 6 in Figure 2, which is replaced by the following:

$$(*)\ \boxed{\textsf{if } \underline{\mathit{visible}(\alpha))\ \textsf{or}}\ \textsf{(exists node } s' \textsf{ with } \mathit{val}(s') = \langle \alpha(x),\, y\rangle \textsf{ and } \mathit{open}(s') \textsf{)}}\,.$$

(The underlined part is not necessary under fairness assumptions that are stronger than **F**, as will be discussed below.)

The language of automaton $\mathcal{A}'$ is nonempty iff there exists an accepting state, accessible from an initial state, which is reachable from itself. To check emptiness of an automaton on-the-fly, i.e., while constructing it, one can use the algorithm in [5]. This algorithm 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 second DFS algorithm. The executions of the two DFS algorithms, DFS1 and DFS2, are interleaved. The adaptation of this algorithm to partial order model-checking is by using the routine $\mathit{ample}(x,\, y)$ to calculate a *subset* of the enabled operations in x, as described above. Algorithm **A2** appears in Figure 5.

DFS1 (lines 1–20) is used to generate the intersection of the reduced state space with the checked automaton. When calling $\mathit{ample}(x,\, y')$ 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 $\mathit{ample}(x,\, y')$ are combined with β such that for each $\alpha \in \mathit{ample}(x,\, y')$, $\langle\alpha,\, \beta\rangle$ is the label of an $\mathcal{A}'$ transition 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 (line 17). 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 were already generated by DFS1. One can trade gain in space for time by regenerating the successors of $\langle x,\, y\rangle$ in DFS2 using $\mathit{ample}(x,\, y)$ 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.

Theorem 4.2 *The algorithm* **A2** *will return* true *if the program P does not satisfy the property φ, and* false *otherwise.*

Proof. To reason about the correctness of the algorithm **A2**, consider first an algorithm **A2'** that consists of DFS1 without calling DFS2. That is, with lines 15-17 removed. This algorithm generates an automaton $\mathcal{A}'$, with initial states $\langle x\,, y\rangle$ such that x is an initial state of the program P, and y an initial state of $\mathcal{B}$. The accepting states are pairs such that the second component is an accepting state in $\mathcal{B}$. We claim that the generated automaton $\mathcal{A}'$ is the product of a reduced state space automaton G', satisfying the properties of Theorem 3.11, and the property automaton $\mathcal{B}$.

```
proc DFS1(s):boolean
  ⟨x, y⟩ := val(s); /* x = program-state, y = Büchi component */
  forall β ∈ Σ, y' ∈ S s.t. same_label(x, β) and y --β--> y' ∈ δ do
    forall α ∈ ample(x, y')
      z := ⟨α(x), y'⟩;
      if new(z) then
        create_node(s', z);
        create_edge(s, s');
        set open(s') and unchecked2(s'); /* s' did not participate in DFS2 */
        if DFS1(s') then return true fi
      else s' := node(z) ; /* s' is an existing node with value z */
        create_edge(s, s') fi;
    od
  od;
  if accepting(s) then /* if backtracked to an accepting node */
    seed:=s; /* seek for a cycle through s; seed is global */
    if DFS2(s) then return true fi /* do secondary search */
  set closed(s); /* remove s from search stack and backtrack */
  return false
end DFS1;

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

Figure 5: Algorithm **A2**: On-the-fly cycle-detection

It is tempting to reduce the correctness of this claim to the off-line construction **A1** with a routine *ample* that preserves **C1**, **C2** and **C3'**, that was presented in Section 3.2 by projecting each combined state $\langle x, y\rangle$ into its first (i.e., program) component x. However, this does not work: for each program-state there might be several combined states with a different second component. Thus, it might happen that whereas in the off-line algorithm, some proper subset $\mathcal{E}$ of operations was rejected from being an ample set, as some $\alpha \in \mathcal{E}$ would have closed a cycle, falsifying **C2**, the on-the-fly version may choose $\mathcal{E}$ without closing a cycle. The reason is that by executing α, some old program state occurs, but this time it is accompanied with a different Büchi component, creating a new node.

However, one can still make a similar reduction from the on-the-fly case to an off-line-construction; this time, we use a non-deterministic variant **A3** of **A1**, described below. The algorithm **A3** may decide non-deterministically to create multiple nodes for the same program state. Thus, condition **C2** may be satisfied by generating a new copy of an existing node, preventing a cycle with an already existing node. *All* the executions of the algorithm **A3** (each execution can create a different reduced state space, due to non-deterministic choices in the code of **A3**) would maintain the correctness claims of **A1**. Algorithm **A3** differ from **A1** as follows:

1. Each node of **A3** is a pair $\langle x, y\rangle$ such that $0 \leq y \leq n$ for some positive integer n. The initial nodes have ι as their first components.
2. The edges generated by **A3** are of the form $\langle x, y\rangle \xrightarrow{\alpha} \langle x', z\rangle$, where $x' = \alpha(x)$. Moreover, there can be multiple edges labeled by α generated from $\langle x, y\rangle$. The choice of the second component is made non-deterministically, with one exception: there can not be an edge from a node with a second component n to a node with a second component other than n.
3. The routine *ample* used by **A2** checks at line 6 in Figure 2 whether applying the operation α to the current state x closes a cycle. If so, according to condition **C2**, α cannot be part of an ample set that is a proper subset of the enabled operations. The algorithm **A3** can generate multiple nodes with first component x. Thus, if a node with first component $x' = \alpha(x)$ already appeared, **A3** can non-deterministically make one of the following decisions about choosing α as a part of an ample set that does not contain all the enabled operations:
   - decide to discard α as part of such an ample set, or
   - decide to use α in such an ample set, generating edges of the form $\langle x, y\rangle \xrightarrow{\alpha} \langle x', z\rangle$ that do not close a cycle.
4. The algorithm **A3** can choose up to m such ample sets for some positive integer m.

Now, theorems 3.9 and 3.11 hold for every execution of the algorithm **A3**, i.e., for every set of non-deterministic choices. The proofs remain the same. This allows us to pick any particular set of non-deterministic choices. Thus, any graph G' constructed by **A3** satisfies Theorem 4.1.

We describe now the connection between the graphs generated by Algorithm **A2'**, and the graphs generated by Algorithm **A3**. We will use the Büchi automaton $\mathcal{B}$ to decide upon the nondeterministic choices of Algorithm **A3**. These executions of **A3** will be said to be *controlled by* $\mathcal{B}$. Let n be the number of nodes of $\mathcal{B}$, identified as $\{0, 1, \ldots, n-1\}$, and m the

	off-line	on-the-fly
no fairness	**C1, C2, C3'**	**C1, C2', C3'**
fairness assumption ψ s.t. $\psi \rightarrow \mathbf{F}$	**C1, C2, C3**	**C1, C2', C3**

Figure 6: Summary of effective conditions according to cases

maximal number of edges from any node of $\mathcal{B}$. The choices of **A3** from any node $\langle x,\, y\rangle$ with $y < n$ are done now according to lines 3–4 of DFS1 in Figure 5. If from such a node $\langle x,\, y\rangle$ there is no transition β of $\mathcal{B}$ such that $same\_label(x,\, \beta)$, then **A3** picks up an ample set, and for each operation α in it, generates an edge $\langle x,\, y\rangle \xrightarrow{\alpha} \langle \alpha(x),\, n\rangle$. **A3** treats nodes of the form $\langle x,\, n\rangle$ in the same way. That is, the executions of **A3** that are controlled by $\mathcal{B}$ simulate the choices of DFS1, except for the cases where a node does not have a possible successor in DFS1 due to $\mathcal{B}$. Furthermore, nodes with second component n may lead only to nodes with second component n. The initial nodes are pairs $\langle \iota,\, y\rangle$, where $y \in \Delta$.

Thus, the Algorithm **A2'** generates exactly the restriction of the graphs generated by Algorithm **A3**, when controlled by $\mathcal{B}$, to nodes with second component smaller than n. Since Theorem 3.11 holds for every execution of **A3**, it holds in particular to the executions controlled by $\mathcal{B}$. Let G' be a graph generated by **A3** with non-deterministic choices controlled by $\mathcal{B}$, and $\mathcal{A}'$ its restriction to nodes with second component smaller than n. Consider G' as an acceptor of infinite sequences, and $\mathcal{A}'$ as a Büchi automaton, with accepting states $\langle x, y\rangle$ such that y is an accepting state of $\mathcal{B}$. It is easy to see that $L(\mathcal{A}') = L(G') \cap L(\mathcal{B})$. By Theorem 4.1, $L(G') \cap L(\mathcal{B})$ is non-empty iff there is a counter-example to the checked property. Checking the non-emptiness of a Büchi automaton can be done by searching for a reachable accepting state that appears on a cycle [5].

Returning from the reduced DFS algorithm **A2'** to the on-the-fly cycle detection algorithm **A2**, DFS1 and DFS2 are now searching for such a cycle in $\mathcal{A}'$ before completing the construction of $\mathcal{A}'$. In [5] it is proved that such an pair of DFS algorithms will find such a cycle iff one exists. ∎

Consider now model-checking on-the-fly under fairness assumptions. Fairness assumptions can be translated into constraints over strongly connected components [18], or equivalently, over cycles. Thus, model-checking under fairness can be done by incorporating the *generalized* cycle detection, as described in [5]. Notice that in case that the fairness ψ that is assumed, which is stronger than **F**, is *not equivalence robust*, the condition **C3** must be applied separately to both the checked property φ *and* the fairness assumption ψ. This can add dependencies, and achieve less reduction. The various conditions that are in effect for the different possibilities of fairness or no-fairness assumption, and off-line or on-the-fly reduction are summarized in a table in Figure 6.

5 Comparison with Other Work

The off-line algorithm combines and adapts several ideas that appeared in various contexts: the condition **C1** is related to the faithful decomposition of Katz and Peled [13] (an ample set is obtained from a faithful decomposition by removing the disabled operations) and later adopted for model-checking [25, 22, 8]. Condition **C2** was used first in [22]. The idea of visible operations which stands behind **C3'** appeared first in [25]. Condition **C3**, which handles the fairness case, appeared in [22]. The combinations presented in Section 3 forms a surprisingly simple and easy to implement set of constraints that can be enforced at very little cost on state-space based model checkers. Alternative algorithms for calculating subsets of the enabled operations appeared e.g., in [24, 7]. These algorithms perform a deeper search for subsets of operations. Thus, they can in some cases produce smaller subsets. However, by doing so they incur more overhead. A comparison between an implementation of the strategy to calculate subsets suggested here and the one in [7], when used to check for deadlocks, appears in [12].

Godefroid and Wolper [9] proposed a method based on combining automata, one for each of the program's processes, with one for the checked formula. The method is intended for specifications expressing the behavior of the program as sequences of operations rather than as sequences of states, since the synchronization of the program with the checked property is according to the operations labeling the edges of the processes and the property automata.

The method of [9] suggests an alternative solution to condition **C2**, which tackles the problem of handling the case where from some point two interindependent infinite tasks execute forever. The problem is to avoid "ignoring" some operation along a loop, which may lead to failing to preserve the correctness of the checked property. The method of [9] uses a generalized Büchi acceptance condition, where instead of one set of accepting conditions there can be multiple such sets. The property automaton (and each automaton with a non-empty accepting conditions) requires then to have at least two *disjoint* sets of accepting sets. Let us call the additional restriction on the acceptance conditions the *multiple acceptance condition.*

Figure 7 gives the motivation behind the multiple acceptance condition. It depicts two processes A and B that execute autonomously, i.e., without any interaction, and a property automaton that has only one accepting state, i.e., a singleton accepting state. The checked property is described as an automaton over the alphabet of program operations. (The self loops are kept implicit in [9], and correspond to the operations that do not change the state. They were explicitly marked in our example.) The automaton has only one accepting state, namely S_2, and hence does not satisfy the multiple acceptance condition. The property checks the absence of sequences in which f occurs but d does not occur. The reduced intersection, as appears on the right of Figure 7 is obtained by applying the algorithm in [9]. It does not preserve the checked property, e.g., it misses the bad sequences where a and b execute infinitely many times where f occurs but d does not.

Imposing the multiple acceptance condition can cause the loss of ability to reduce the state space: in order to apply it to the above property automaton, one needs to find at least one additional disjoint accepting state such that accepting sequences must pass through at least

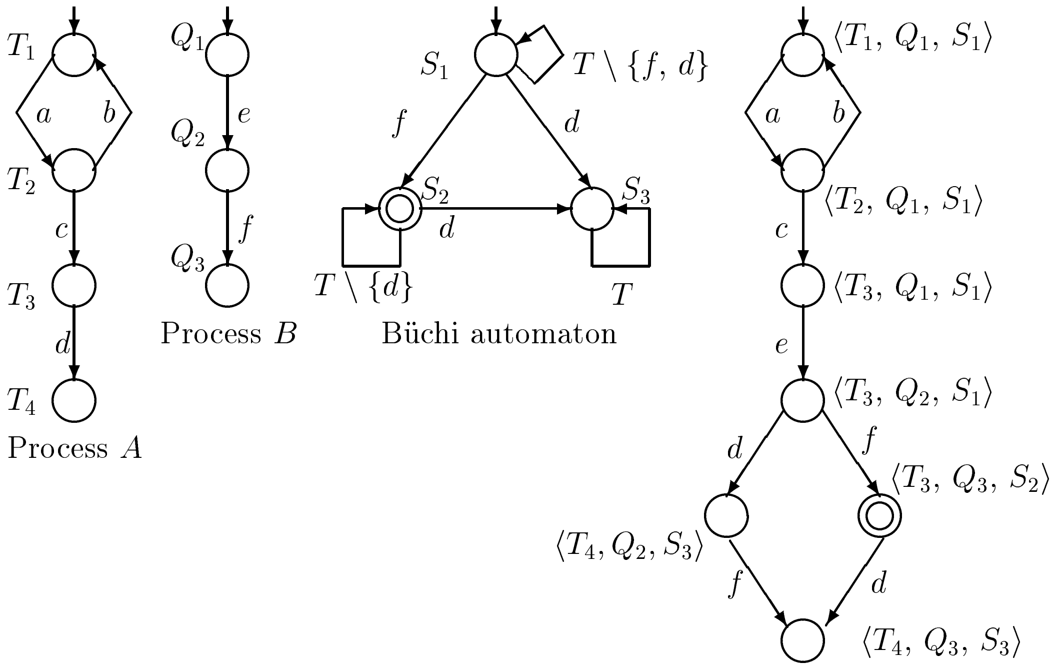


Figure 7: An Incorrect Reduced Intersection

both accepting states. Trying to obtain such an additional state by splitting state S_2 into a loop that includes two states S_2' and S_2'' with edges from each one to the other does not work; these edges will be labeled by all the program operations except d, and thus it would make all the program operations visible. This would force all the program operations to become interdependent, resulting in *no reduction at all.* This problem is inherent to the checked property and not only to the automaton used: it can be easily checked that every automaton for the above checked property with only the operations d and f visible (i.e., changing the state of the property automaton) cannot satisfy the multiple acceptance requirement.

In [9], the problem of automatically transforming property automata into generalized Büchi automata that satisfy the multiple acceptance condition is left open. Our algorithm solves the ignoring problem by imposing the simple constraint **C2**. Thus, it can be used directly in conjunction with traditional Büchi automata specifications, e.g., the ones used in SPIN [10], or the ones obtained by translating temporal logic properties [27]. The applicability of **C2** to solve the ignoring problem in the on-the-fly case is carefully studied here. The on-the-fly construction may defer the closing of cycles. Thus, it is to some extent surprising that only a trivial adjustment is needed to the simple condition **C2** in order to adapt it to the on-the-fly case, as shown in Section 4. It should be mentioned that the requirement **C2** is very subtle: an earlier attempt for solving the ignoring problem appeared in [11] 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, as can be seen from the example in Figure 7, where the weaker condition of [11] is satisfied.

Valmari [26] presented a different method for on-the-fly partial order reduction called the *stubborn set* method. A stubborn set is a subset $\mathcal{E}$ of the program operations T that includes at least one operation that is enabled at the current state x. The operations of $\mathcal{E}$ which are enabled at x are used to generate the successors of x. In terms of our previous definitions, a stubborn set $\mathcal{E}$ can be described as a set that must satisfy the following conditions:

- if $\alpha \in \mathcal{E}$ is enabled in x, then at least one operation from $\mathcal{E}$ must be executed before α can become enabled.
- if $\alpha \in \mathcal{E}$ is disabled in x, then there is a set of operations $\mathcal{D} \subseteq \mathcal{E}$ such that at least one operation from $\mathcal{D}$ must be executed before α can become enabled.

In order to solve the above problem, an additional constraint requires that $\mathcal{E}$ includes the set of visible operations $a(\varphi)$.

The algorithm to calculate a stubborn set for a given state x involves constructing a graph with nodes labeled by program operations and program processes, and edges corresponding to the dependencies between the operations (and the processes they belong to). Then, the algorithm searches this graph to find strongly connected components that include the set of visible operations.

Our strategy is in essence the reverse of [26]. Namely, we prefer selecting a subset of invisible operations, and doing only local calculations to generate the ample sets. The advantage is the use of a simpler algorithm than [26] for calculating subsets of successors. This allows us to make a substantial part of the computations related to the reduction statically, before expanding the state space, resulting in an implementation with very low overhead.

Both of the methods [9] and [26] handle specifications represented as automata over (illegal) sequences of the program *operations*. 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 the translation algorithm [27]. This is also the kind of automaton used to check properties with the model-checker SPIN [10]. 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 (rather than between operations that change the state of the property automaton, as in the other approaches).

The algorithm [26] does not treat fairness assumptions. The reduction method [9] suggests to encode fairness assumptions ψ as generalized Büchi accepting conditions (allowing more than one set of accepting states) in each process. This encoding might not be straightforward for some fairness assumptions such as weak or strong fairness, since the fairness conditions of one process may also depend on the states of other processes. That is, the fact that at least one operation in a process is enabled may also depend on the enabledness of a matching communication in another process.

Our algorithm takes advantage of the fairness assumptions ψ in case it is stronger than the fairness assumption **F** to weaken the conditions enforced on choosing successors from each node. Then it runs a specialized cycle detection algorithm for ψ. Such a specialized model-checking algorithm [18] imposes an overhead which is linear or quadratic in the size of the fairness formula, for weak or strong fairness, respectively. An alternative is to check the implication $\psi \to \varphi$, where φ is the checked property. However, this may result in overhead which is exponential in the size of the fairness formula ψ, caused by the translation of the implication $\psi \to \varphi$, into a Büchi automaton.

6 Implementation and Experimental Results

An instance of this algorithm was implemented by Gerard Holzmann as an extension of the model-checker SPIN [12]. SPIN [10] 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 message queues (but excluding asynchronous message 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 Theorem 3.12). However, no explicit representation of the dependency relation is calculated or stored. That is, the dependency relation is merely used to explain and formally reason about the correctness of the algorithm that calculates a subset of successors, i.e., *check_succ* at line 3 in Figure 2.

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*, $\mathcal{C}\rangle$ The set of this processes' enabled operations can be chosen only if the precomputed condition $\mathcal{C}$ (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 $\mathcal{C}$ can be that all of their receiving queues are non-empty.

Alg.	States	Trans.	Mem.	Time
sieve	670,399	4,434,585	99,166,040	218.0
	995	1,738	1,242,968	0.3
urp	2,165	5,522	1,664,888	0.4
	575	721	1,468,280	0.2
abp	378	806	1,077,256	0.1
	378	806	1,081,352	0.1
snoopy	367,867	1,911,998	46,290,360	118.7
	131,702	327,270	18,650,552	29.7
pftp	893,575	3,583,154	143,255,096	311.3
	210,919	469,850	35,472,952	43.2
dtp	754,225	2,845,271	112,375,352	190.8
	233,191	437,211	36,156,984	34.8
accounting	71,053	220,441	8,343,560	10.1
	10,634	15,119	2,158,600	1.0

Figure 8: Experimental results

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 the label m is visited infinitely often (which was implemented by introducing a global variable that changes its value twice just before label m) for various protocols is summarized in the table in Figure 8. Memory is given in Megabytes and time in seconds. The first line in each pair of lines shows the measurements for the full search, while the second line shows the measurements for the reduced search. All measurements were made on a SPARC 10 station, with 128 Mbyte of Memory.

The program *sieve* is the Sieve of Eratosthenes algorithm for finding the first N prime number using $N+1$ parallel processes. The protocol *urp* is AT&T universal receiver protocol. The protocol *abp* is the alternating bit protocol. The protocol *snoopy* is a cache coherence protocol. The protocol *pftp* is a file transfer protocol [10]. The protocol *dtp* is data transfer protocol, and *accounting* is a distributed accounting algorithm.

Acknowledgements. The author is grateful to Gerard Holzmann, who implemented the partial order reduction on SPIN, and Ramesh Bharadwaj for helping in debugging the system. Both of them, R. P. Kurshan, Wojciech Penczek, and the reviewers of the journal *Formal Methods in Systems Design* gave many helpful comments.

References

[1] S. Aggarwal, C. Courcoubetis, P. Wolper, Adding Liveness Properties to Coupled Finite State Machines, ACM Transactions on Programming Languages and systems 12 (1990), 303–339.

[2] K. Apt, N. Francez, S. Katz, Appraising fairness in languages for distributed programming, Distributed Computing, Vol 2 (1988), 226–241.

[3] E.M. Clarke, E.A. Emerson, and A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal-logic specifications, ACM Transactions on Programming Languages and Systems, 8(1986), 244–263.

[4] 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.

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

[6] 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.

[7] P. Godefroid, Using partial orders to improve automatic verification methods, in E.M. Clarke, R.P. Kurshan (eds.), Computer Aided Verification 1990, DIMACS, Vol 3, 1991, 321–339.

[8] P. Godefroid, D. Pirottin, Refining Dependencies Improves Partial-Order Verification Methods, $5th$ International Conference on Computer Aided Verification, Elounda, Greece, Lecture Notes in Computer Science 697, Springer–Verlag, 1993, 438-449.

[9] P. Godefroid, P. Wolper, A Partial Approach to Model Checking, $6th$ LICS, 1991, Amsterdam, 406–415, also in Information and Computation 110(2), 305–326.

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

[11] 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.

[12] G. J. Holzmann, D. Peled, An Improvement in Formal Verification, $7th$ International Conference on Formal Description Techniques, Berne, Switzerland, 1994, 177–194.

[13] S. Katz, D. Peled, Verification of distributed programs using representative interleaving sequences, Distributed Computing 6 (1992), 107–120, A preliminary version, titled An efficient verification method for parallel and distributed programs, appeared in: Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Noordwijkerhout, The Netherlands, May/June 1988, Lecture Notes in Computer Science 354, Springer, 489–507.

[14] 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.

[15] R. P. Kurshan, Reducibility in analysis of coordination, Lecture Notes in Communication and Information, Springer, 103, 19–39, 1987.

[16] M. Z. Kwiatkowska, Event Fairness and Non-Interleaving Concurrency, Formal Aspects of Computing 1 (1989), 213–228.

[17] L. Lamport, What good is temporal logic, IFIP Congress, North Holland, 1983, 657–668. in Computer Science 115,

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

[19] Z. Manna, A. Pnueli, How to cook a temporal proof system for your pet language. 9*th* ACM Symposium on Principles on Programming Languages, Austin, Texas, 1983, 141–151.

[20] A. Mazurkiewicz, Trace Theory, in: W. Brauer, W. Reisig, G. Rozenberg (eds.) Advances in Petri Nets 1986, Bad Honnef, Germany, Lecture Notes in Computer Science 255, Springer, 1987, 279–324.

[21] D. Peled, A. Pnueli, Proving partial order properties, Theoretical Computer Science 126 (1994), 143–182.

[22] D, Peled, All from one, one for all, on model-checking using representatives, 5*th* international conference on Computer Aided Verification, Greece, 1993, Lecture Notes in Computer Science, Springer, 409–423.

[23] R. S. Street, Propositional Dynamic Logic of Looping and Converse, Information and Control 54 (1982), 121–141.

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

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

[26] A. Valmari, On-The-Fly Verification of stubborn sets, $5th$ CAV, Greece, 1993, Lecture Notes in Computer Science 697, Springer, 397–408.

[27] 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.