Title: **What's Wrong With On-the-fly Partial Order Reduction (Extended Version)**

Author: Stephen F. Siegel

Notes: This is an extended version of the paper appearing in the *Computer-Aided Verification — 31st International Conference, CAV 2019*.

Verified Software Laboratory
Department of Computer and Information Sciences
University of Delaware
Newark DE 19716
USA
`http://vsl.cis.udel.edu`

What's wrong with on-the-fly partial order reduction (extended version)

Stephen F. Siegel[0000−0001−9359−3332]

University of Delaware, Newark DE, USA
siegel@udel.edu

Abstract. Partial order reduction and on-the-fly model checking are well-known approaches for improving model checking performance. The two optimizations interact in subtle ways, so care must be taken when using them in combination. A standard algorithm combining the two optimizations, published over twenty years ago, has been widely studied and deployed in popular model checking tools. Yet the algorithm is incorrect. Counterexamples were discovered using the Alloy analyzer. A fix for a restricted class of property automata is proposed.

Keywords: Model checking · partial order reduction · on-the-fly · Spin

1 Introduction

Partial order reduction (POR) refers to a family of model checking techniques used to reduce the size of the state space that must be explored when verifying a property of a program. The techniques vary, but all share the core observation that when two independent operations are enabled in a state, it is often safe to ignore traces that begin with one of them. A large number of POR techniques have been explored, differing in details such as the range of properties to which they apply. This paper focuses on *ample set* POR [4], an approach which applies to stutter-invariant properties and is used in the model checker Spin [8].

In the automata-theoretic view of model checking, the negation of the property to be verified is represented by an ω-automaton. The basic algorithm computes the product of this automaton with the state space of the program. The language of the product is empty if and only if the program cannot violate the property. *On-the-fly* model checking refers to an optimization of this basic algorithm in which the enumeration of the reachable program states, computation of the product, and language emptiness check are interleaved, rather than occurring in sequence.

These two optimizations must be combined with care, because they interact in subtle ways.[1] A standard algorithm for on-the-fly ample set POR is described in [12] and in further detail in [13]. I shall refer to this algorithm as the *combined*

[1] Previous work, for example, has dealt with problems, distinct from those discussed in this paper, that arise when combining nested depth first search and POR [7, 14].

algorithm. Theorem 4.2 of [13] asserts the soundness of the combined algorithm. A proof of the theorem is also given in [13].

The proof has a gap. This was pointed out in [16, §5], with details in [15]. The gap was rediscovered in the course of developing mechanized correctness proofs for model checking algorithms; an explicit counterexample to the incorrect proof step was also found ([2, §8.4.5] and [3, §5]). The fact that the proof is erroneous, however, does not imply the theorem is wrong. To the best of my knowledge, no one has yet produced a proof or a counterexample for the soundness of the combined algorithm.

In this paper, I show that the combined algorithm is not sound; a counterexample is given in Sec. 3.1. I found this counterexample by modeling the combined algorithm in Alloy and using the Alloy analyzer [11] to check its soundness. Sec. 4 describes this model. Spin's POR is based on the combined algorithm, and in Sec. 5, Spin is seen to return an incorrect result on a Promela model derived from the theoretical counterexample.

There is a small adjustment to the combined algorithm, yielding an algorithm that is arguably more natural and that returns the correct result on the previous counterexample; this is described in Sec. 6. It turns out this one is also unsound, as demonstrated by another Alloy-produced counterexample. However, in Sec. 7, I show that this variation is sound if certain restrictions are placed on the property automaton.

2 Preliminaries

Definition 1. *A* finite state program *is a triple* $P = \langle T, Q, \iota \rangle$*, where* Q *is a finite set of* states, $\iota \in Q$ *is the* initial state*, and* T *is a finite set of* operations*. Each operation* $\alpha \in T$ *is a function from a set* $\mathsf{en}_\alpha \subseteq Q$ *to* Q*.*

Fix a finite state program $P = \langle T, Q, \iota \rangle$.

Definition 2. *For* $q \in Q$*, define* $\mathsf{en}(q) = \{\alpha \in T \mid q \in \mathsf{en}_\alpha\}$.

Definition 3. *An* execution *of* P *is an infinite sequence of operations* $\alpha_1\alpha_2\cdots$ *that* generates *the sequence of states* $\xi = q_0q_1q_2\cdots$ *such that* $q_0 = \iota$ *and for* $i \geq 0$*,* $q_i \in \mathsf{en}_{\alpha_{i+1}}$ *and* $q_{i+1} = \alpha_{i+1}(q_i)$*. An* admissible *sequence is any segment of an execution.*

Definition 4. *A* Büchi automaton *is a tuple* $\mathcal{B} = \langle S, \Delta, \Sigma, \delta, F \rangle$*, where* S *is a finite set of* automaton states*,* $\Delta \subseteq S$ *is the set of* initial states*,* Σ *is a finite set called the* alphabet*,* $\delta \subseteq S \times \Sigma \times S$ *is the* transition relation*, and* $F \subseteq S$ *is the set of* accepting states*. The* language *of* $\mathcal{B}$*, denoted* $\mathcal{L}(B)$*, is the set of all* $\xi \in \Sigma^\omega$ *generated by infinite paths in* $\mathcal{B}$ *that pass through an accepting state infinitely often.*

Fix a finite set AP of *atomic propositions* and let $\Sigma = 2^{\mathsf{AP}}$.
Fix an *interpretation mapping* for P, i.e., a function $L\colon Q \to \Sigma$.

Definition 5. *The* language of P, *denoted* $\mathcal{L}(P)$, *is the set of all infinite words* $L(q_0)L(q_1)\cdots \in \Sigma^\omega$, *where* $q_0q_1\cdots$ *is the sequence of states generated by an execution of* P.

Definition 6. *A language* $L \subseteq \Sigma^\omega$ *is* stutter-invariant *if, for any* $a_0, a_1, \ldots \in \Sigma$ *and positive integers* $i_0, i_1 \ldots$, $a_0a_1\cdots \in L \Leftrightarrow a_0^{i_0}a_1^{i_1}\cdots \in L$, *where* a^i *denotes the concatenation of* i *copies of* a.

Definition 7. *Let* $\mathcal{B} = \langle S, \Delta, \Sigma, \delta, F\rangle$, *be a Büchi automaton with alphabet* Σ. *The* product of P and $\mathcal{B}$ *is the Büchi automaton*

$$P \otimes \mathcal{B} = \langle Q \times S, \{\iota\} \times \Delta, T \times \Sigma, \delta_\otimes, Q \times F\rangle,$$

where

$$\delta_\otimes = \{(\langle q, s\rangle, \langle \alpha, \sigma\rangle, \langle q', s'\rangle) \mid \sigma = L(q) \wedge \langle s, \sigma, s'\rangle \in \delta \wedge q' = \alpha(q)\}.$$

Note 1. A transition from product state $x = \langle q, s\rangle$ can be viewed as taking place in two steps. First, a transition $s \xrightarrow{L(q)} s'$ in $\mathcal{B}$ executes, leading to an "intermediate state" $x' = \langle q, s'\rangle$. Then a program transition $q \xrightarrow{\alpha} q'$ executes, culminating in $y = \langle q', s'\rangle$. While this is a good mental model, the product automaton does not necessarily contain a transition from x to x' or from x' to y. The intermediate state x' is not even necessarily reachable in the product. The transition in the product goes directly from x to y with label $\langle \alpha, L(q)\rangle$.

It is well-known that

$$\mathcal{L}(P) \cap \mathcal{L}(\mathcal{B}) = \emptyset \Leftrightarrow \mathcal{L}(P \otimes \mathcal{B}) = \emptyset.$$

In the context of model checking, $\mathcal{B}$ is used to represent the negation of a desirable property; the program P satisfies the property if, and only if, no execution of P is accepted by $\mathcal{B}$, i.e., $\mathcal{L}(P) \cap \mathcal{L}(\mathcal{B}) = \emptyset$. The automaton $\mathcal{B}$ may be generated from a (negated) LTL formula, but that assumption is not needed here.

The goal of "offline" (not on-the-fly) partial order reduction is to generate some subspace P' of P with the guarantee that

$$\mathcal{L}(P') \cap \mathcal{L}(\mathcal{B}) = \emptyset \Leftrightarrow \mathcal{L}(P) \cap \mathcal{L}(\mathcal{B}) = \emptyset$$

The emptiness of $\mathcal{L}(P' \otimes \mathcal{B}) = \mathcal{L}(P') \cap \mathcal{L}(\mathcal{B})$ can be decided in various ways, such as a nested depth first search (NDFS) [5].

3 On-the-fly partial order reduction

In on-the-fly model checking, the state space of the product automaton is enumerated directly, without first enumerating the program states. Adding POR to the mix means that at each state reached in the product automaton, some subset of enabled transitions will be explored. The goal is to ensure that if the

language of the full product automaton is nonempty, then the language of the resulting reduced automaton must be nonempty.

To make this precise, fix a finite state program $P = \langle T, Q, \iota \rangle$, a set AP of atomic propositions, an interpretation $L\colon Q \to \Sigma = 2^{\mathsf{AP}}$, and Büchi automaton $\mathcal{B} = \langle S, \Delta, \Sigma, \delta, F \rangle$. Let $\mathcal{A} = P \otimes \mathcal{B}$.

Definition 8. *A function* $\mathsf{amp}\colon Q \times S \to 2^T$ *is an* ample selector *if* $\mathsf{amp}(q,s) \subseteq \mathsf{en}(q)$ *for all* $q \in Q, s \in S$. *Each* $\mathsf{amp}(q,s)$ *is an* ample set.

An ample selector determines a subautomaton $\mathcal{A}' = \mathsf{reduced}(\mathcal{A}, \mathsf{amp})$ of $\mathcal{A}$: $\mathcal{A}'$ is defined exactly as in Definition 7, except that the transition relation has the additional restriction that $\alpha \in \mathsf{amp}(q, s')$:

$$\mathcal{A}' = \langle Q \times S, \{\iota\} \times \Delta, T \times \Sigma, \delta', Q \times F \rangle \tag{1}$$

$$\delta' = \{(\langle q, s\rangle, \langle \alpha, \sigma\rangle, \langle q', s'\rangle) \in (Q \times S) \times (T \times \Sigma) \times (Q \times S) \mid \sigma = L(q) \wedge \langle s, \sigma, s'\rangle \in \delta \wedge \alpha \in \mathsf{amp}(q, s') \wedge q' = \alpha(q)\}. \tag{2}$$

Definition 9. *An ample selector* amp *is* POR-sound *if the following holds:*

$$\mathcal{L}(\mathsf{reduced}(\mathcal{A}, \mathsf{amp})) = \emptyset \Leftrightarrow \mathcal{L}(P) \cap \mathcal{L}(\mathcal{B}) = \emptyset.$$

The goal is to define some constraints on an ample selector that guarantee it is POR-sound. Before stating the constraints, we need two more concepts:

Definition 10. *An* independence relation *is an irreflexive and symmetric relation* $I \subseteq T \times T$ *satisfying the following: if* $(\alpha, \beta) \in I$ *and* $q \in \mathsf{en}_\alpha \cap \mathsf{en}_\beta$, *then* $\alpha(q) \in \mathsf{en}_\beta$, $\beta(q) \in \mathsf{en}_\alpha$, *and* $\alpha(\beta(q)) = \beta(\alpha(q))$.

Fix an independence relation I. We say α and β are *dependent* if $(\alpha, \beta) \notin I$.

Definition 11. *An operation* $\alpha \in T$ *is* invisible with respect to L *if, for all* $q \in \mathsf{en}_\alpha$, $L(q) = L(\alpha(q))$.

Note 2. The definition in [13] is slightly different. Given an LTL formula ϕ over AP, let AP' be the set of atomic propositions occurring syntactically in ϕ. The definition in [13] says α is *invisible in* ϕ if, for all $p \in \mathsf{AP}'$ and $q \in \mathsf{en}_\alpha$, $p \in L(q) \Leftrightarrow p \in L(\alpha(q))$. However, there is no loss of generality using Definition 11, since one can define a new interpretation $L'\colon Q \to 2^{\mathsf{AP}'}$ by $L'(q) = L(q) \cap \mathsf{AP}'$. Then α is invisible for ϕ if, and only if, α is invisible with respect to L', and the results of this paper can be applied without modification to P, AP', and L'.

We now define the following constraints on an ample selector amp:[2]

C0 For all $q \in Q$, $s \in S$: $\mathsf{en}(q) \neq \emptyset \implies \mathsf{amp}(q,s) \neq \emptyset$.

C1 For all $q \in Q$, $s \in S$: in any admissible sequence in P starting from q, no operation in $T \setminus \mathsf{amp}(q,s)$ that is dependent on an operation in $\mathsf{amp}(q,s)$ can occur before some operation in $\mathsf{amp}(q,s)$ occurs.

[2] I am using the numbering from [4]. In [13], **C2** and **C3** are swapped.

C2 For all $q \in Q$, $s \in S$: if $\mathsf{amp}(q,s) \neq \mathsf{en}(q)$ then $\forall \alpha \in \mathsf{amp}(q,s)$, α is invisible.
C3 There is a depth-first search of $\mathcal{A}' = \mathsf{reduced}(\mathcal{A}, \mathsf{amp})$ with the following property: whenever there is a transition in $\mathcal{A}'$ from a node $\langle q, s\rangle$ on the top of the stack to a node $\langle q', s'\rangle$ on the stack, $\mathsf{amp}(q,s') = \mathsf{en}(q)$.

Condition **C3** is the interesting one. The combined algorithm of [13] enforces it using a DFS (the outer search of the NDFS) of the reduced space and the following protocol: given a new state $\langle q, s\rangle$ that has just been pushed onto the stack, first iterate over all Büchi transitions $\langle s, L(q), s'\rangle$ departing from s and labeled by $L(q)$. For each of these, a candidate ample set for $\mathsf{amp}(q,s')$ that satisfies the first three conditions is computed; this computation does not depend on s'. If any operation in that candidate set leads back to a state on the search stack (a "back edge"), a different candidate is tried and the process is repeated until a satisfactory one is found. If no such candidate is found, $\mathsf{en}(q)$ is used for the ample set.

Hence the process for choosing the ample set depends on the current state of the search. If $y_1 \neq y_2$, it is not necessarily the case that $\mathsf{amp}(x,y_1) = \mathsf{amp}(x,y_2)$, because it is possible that when $\langle x, y_1\rangle$ was encountered, a back edge existed for a candidate, but when $\langle x, y_2\rangle$ was encountered, there was no back edge.

3.1 Counterexample

Theorem 4.2 of [13] can be expressed as follows: if $\mathcal{L}(\mathcal{B})$ is stutter-invariant and the language of an LTL formula, and amp satisfies **C0**–**C3**, then amp is POR-sound.

A counterexample to this claim is given in Fig. 1. The program consists of two states, A and B, and two operations, α and β. There is a single atomic proposition, p, which is *false* at A and *true* at B. Note that α and β are independent. Also, α is invisible, and β is not.

The property automaton, $\mathcal{B}_1$, is shown in Fig. 1 (center top). It has two states, numbered 0 and 1. State 1 is the sole accepting state. The language consists of all infinite words of the following form: a finite nonempty prefix of $\varnothing$s followed by an infinite sequence of $\{p\}$s. This language is stutter-invariant, and is the language of the LTL formula $(\neg p) \wedge ((\neg p)\mathbf{U}\ \mathbf{G}p)$.

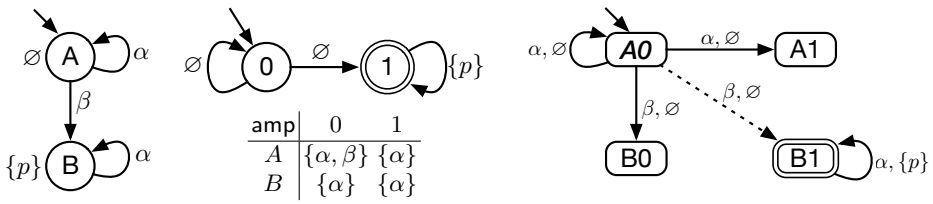

Fig. 1. Counterexample to combined theorem. Left: program and interpretation. Center: property automaton $\mathcal{B}_1$ and ample selector function. Right: the reachable product state space; dashed edges are in the full, but not reduced, space.

The ample selector is specified by the table (center bottom). Notice that $\mathsf{amp}(A,1) \neq \mathsf{en}(A)$, but the other three ample sets are full. **C0** holds because the ample sets are never empty. **C1** holds because β is independent of α. **C2** holds because α is invisible. The reachable product space is shown in Fig. 1 (right). In any DFS of $\mathsf{reduced}(\mathcal{A}, \mathsf{amp})$, the only back edge is the self-loop on $A0$ labeled $\langle \alpha, \varnothing \rangle$. Since $\mathsf{amp}(A,0)$ is full, **C3** holds. Yet there is an accepting path in the full space, but not in the reduced space.

4 Alloy models of POR Schemes

Alloy is a "lightweight formal methods" language and tool. It has been used in a wide variety of contexts, from exploring software designs to studying weak memory-consistency models. An Alloy model specifies *signatures*, each of which defines a type, relations on signatures, and constraints on the signatures and relations. Constraints are expressed in a logic that combines elements of first order logic and relational logic, and includes a transitive closure operator. An *instance* of a model assigns a finite set of *atoms* to each signature, and a finite set of tuples (of the right type) to each relation, in such a way that the constraints are satisfied. The Alloy analyzer can be used to check that an assertion holds on all instances in which the sizes of the signatures are within some specified bounds. The analyzer converts the question of the validity of the assertion into a SAT problem and invokes a SAT solver. Based on the result, it reports either that the assertion holds within the given bounds, or it produces an instance of the model violating the assertion.

I developed an Alloy model to search for counterexamples to various POR claims, such as the one in Sec. 3.1. The model encodes the main concepts of the previous two sections, including program, operations, interpretation, invisibility and independence, property automaton, the product space, ample selectors and the constraints on them, and a language emptiness predicate. The model culminates in an assertion which states that an ample selector satisfying the four constraints is POR-sound.

I was not able to find a way to encode stutter-invariance. In the end, I developed a small set of Büchi automata based on my own intuition of what would make interesting tests. I encoded these in Alloy and used the analyzer to explore all possible programs and ample selectors for each.

The first part of the model is a simple encoding of a finite state automaton. The following is a listing of file `ba.als`:

```
1 module ba  -- module for simple model of Büchi automata
2 sig Sigma {} -- alphabet of BA, valuation on atomic props
3 sig BState {} -- a state in the Büchi Automaton
4 one sig Binit extends BState {} -- initial state of BA
5 sig AState in BState {} -- accepting states of BA
6  -- a transition has a source state, label, and destination state...
7 sig BTrans { src: one BState, label: one Sigma, dest: one BState }
```

The alphabet is some unconstrained set `Sigma`. The set of states is represented by signature `BState`. There is a single initial state, and any number of accepting states. Each transition has a source and destination state, and label. Relations declared within a signature declaration have that signature as an implicit first argument. So, for example, `src` is a binary relation of type `BTrans` × `BState`. Furthermore, the relation is many-to-one: each transition has exactly one `BState` atom associated to it by the `src` relation.

The remaining concepts are incorporated into module `por_v0`:

```
module por_v0  -- on-the-fly POR variant 0, corresponding to [13]
open ba  -- import the Büchi automata module
sig Operation {} -- program operation
sig PState { -- program state
  label: one Sigma,  -- the set of propositions which hold in this state
  enabled: set Operation,  -- the set of all operations enabled at this state
  nextState: enabled -> one PState,  -- the next-state function
  ample: BState -> set Operation  -- ample(q,s)
}{ all s: BState | ample[s] in enabled } -- ample sets subsets of enabled
fun amp[q: PState, s: BState] : set Operation { q.ample[s] }
one sig Pinit extends PState {}  -- initial program state
fact { -- all program states are reachable from Pinit
  let r = {q, q': PState | some op: Operation | q.nextState[op]=q'} |
    PState = Pinit.*r
}
sig ProdState { -- state in the product of program and property automaton
  pstate: PState,  -- the program state component
  bstate: BState,  -- the property state component
  nextFull: set ProdState,  -- all next states in the full product space
  nextReduced: set ProdState  -- all next states in the reduced product space
}
one sig ProdInit extends ProdState {} -- initial product state
pred transitionInProduct[q,q': PState, op: Operation, s,s': BState] {
  q->op->q' in nextState
  some t : BTrans | t.src = s and t.dest = s' and t.label = q.label
}
pred nextProd[x: ProdState, op: Operation, x': ProdState] {
  transitionInProduct[x.pstate, x'.pstate, op, x.bstate, x'.bstate]
}
pred independent[op1, op2 : Operation] {
  all q: PState | (op1+op2 in q.enabled) implies (
    op2 in q.nextState[op1].enabled and
    op1 in q.nextState[op2].enabled and
    q.nextState[op1].nextState[op2] = q.nextState[op2].nextState[op1])
}
pred invisible[op: Operation] {
  all q: PState | op in q.enabled => q.nextState[op].label = q.label
}
fact C0 { all q: PState, s: BState | some q.enabled => some amp[q,s] }
fact C1 {
  all q: PState, s: BState | let A=amp[q,s] |
```

```
42      let r = { q1, q2: PState | some op: Operation-A |
43                q1->op->q2 in nextState } |
44        all q': q.*r, op1: q'.enabled-A, op2: A | independent[op1, op2]
45  }
46  fact C2 {
47    all q: PState, s: BState | let A = amp[q,s] |
48      A != q.enabled implies all op: A | invisible[op]
49  }
50  fact C3' {
51    let r = { x, x' : ProdState | x->x' in nextReduced and
52              amp[x.pstate, x'.bstate] != x.pstate.enabled } |
53      no x: ProdState | x in x.^r
54  }
55  fact { -- generate all reachable product states, etc.
56    nextFull = {x,y: ProdState | some op: Operation | nextProd[x,op,y]}
57    nextReduced = {x,y: ProdState |
58      some op: amp[x.pstate, y.bstate] | nextProd[x,op,y]}
59    ProdState = ProdInit.*nextFull
60    all x,y: ProdState | (x.pstate=y.pstate && x.bstate=y.bstate) => x=y
61    ProdInit.pstate = Pinit and ProdInit.bstate = Binit
62    all x: ProdState, op: Operation, q': PState, s': BState |
63      transitionInProduct[x.pstate, q', op, x.bstate, s'] implies
64        some y: ProdState | y.pstate = q' and y.bstate = s'
65  }
66  pred nonemptyLang[r: ProdState->ProdState] { -- r reaches accepting cycle
67    some x: ProdInit.*r | (x.bstate in AState and x in x.^r)
68  }
69  assert PORsoundness { -- if full space has a lasso, so does the reduced
70    nonemptyLang[nextFull] => nonemptyLang[nextReduced]
71  }
```

The `facts` are constraints that any instance must satisfy; some of the facts are given names for readability. A `pred` declaration defines a (typed) predicate.

Most aspects of this model are self-explanatory; I will comment only on the less obvious features. The relations `nextFull` and `nextReduced` represent the next state relations in the full and reduced spaces, respectively. They are declared in `ProdState`, but specified completely in the final `fact` on lines 56–58. Strictly speaking, one could remove those predicates and substitute their definitions, but this seemed more convenient. Line 60 asserts that a product state is determined uniquely by its program and property components. Line 61 specifies the initial product state.

Line 59 insists that only states reachable (in the full space) from the initial state will be included in an instance (`*` is the reflexive transitive closure operator). Lines 62–64 specify the converse. Hence in any instance of this model, `ProdState` will consist of exactly the reachable product states in the full space.

The encoding of **C1** is based on the following observation: given $q \in Q$ and a set A of operations enabled at q, define $r \subseteq Q \times Q$ by removing from the program's next-state relation all edges labeled by operations in A. Then "no operation dependent on an operation in A can occur unless an operation in A

occurs first" is equivalent to the statement that on any path from q using edges in r, all enabled operations encountered will either be in A or independent of every operation in A.

Condition **C3** is difficult to encode, in that it depends on specifying a depth-first search. I have replaced it with a weaker condition, which is similar to a well-known cycle proviso in the offline theory:

C3′ In any cycle in $\mathsf{reduced}(\mathcal{A}, \mathsf{amp})$, there is a transition from $\langle q, s\rangle$ to $\langle q', s'\rangle$ for which $\mathsf{amp}(q, s') = \mathsf{en}(q)$.

Equivalently: if one removes from the reduced product space all such transitions, then the resulting graph should have no cycles. This is the meaning of lines 50–54 (^ is the strict transitive closure operator).

The next step is to create tests for specific property automata. This example is for the automaton $\mathcal{B}_1$ of Fig. 1:

```
1  module ba1
2  open ba
3  one sig X0, X1 extends Sigma {}
4  one sig B1 extends BState {}
5  one sig T1, T2, T3 extends BTrans {}
6  fact {
7    AState = B1  -- B1 is the sole accepting state
8    T1.src=Binit && T1.label=X0 && T1.dest=Binit
9    T2.src=Binit && T2.label=X0 && T2.dest=B1
10   T3.src=B1 && T3.label=X1 && T3.dest=B1
11 }
```

The final step is a test that combines the modules above:

```
1  open por_v0
2  open ba1
3  checkPORsoundness for exactly 2 Sigma, exactly 2 BState,
4    exactly 3 BTrans, 2 Operation, 2 PState, 4 ProdState
```

It places upper bounds on the numbers of operations, program states, and product states while checking the soundness assertion. Using the Alloy analyzer to check the assertion above results in a counterexample like the one in Fig. 1. The runtime is a fraction of a second. The Alloy instance uses two uninterpreted atoms for the elements of `Sigma`; I have simply substituted the sets $\varnothing$ and $\{p\}$ for them to produce Fig. 1. As we have seen, this counterexample happens to also satisfy the stronger constraint **C3**.

5 Spin

The POR algorithm used by Spin is described in [10] and is similar to the combined algorithm. We can see what Spin actually does by encoding examples in Promela and executing Spin with and without POR.

```
bit p = 0;
active proctype p0() { p=1 }
active proctype p1() { bit x=0; do :: x=0 od }
never {
  B0: do :: !p :: !p -> break od
  accept_B1: do :: p od
}
```

Fig. 2. Promela representation of counterexample using $\mathcal{B}_1$ of Fig. 1

Fig. 2 shows an encoding of the example of Fig. 1. Transition α corresponds to the assignment `x=0`, where `x` is a variable local to `p1`. Transition β corresponds to the assignment `p=1`, where `p` is a shared variable. Applying Spin with the following commands allows one to see the structure of the program graphs for each process, as well as each step in the search of the full space:

```
spin -a test1.pml; cc -o pan -DCHECK -DNOREDUCE pan.c; ./pan -d; ./pan -a
```

I did this with Spin version 6.4.9, the latest stable release. The output indicates that 4 states and 5 transitions are explored, and one state is matched—exactly as in Fig. 1 (right). As expected, the output also reports a violation—a path to an accepting cycle that corresponds to the transition from $A0$ to $B1$ followed by the self-loop on $B1$ repeated forever.

Repeat this experiment without the `-DNOREDUCE`, however, and Spin finds no errors. The output indicates that it misses the transition from $A0$ to $B1$.

6 Ignoring the intermediate states

An interesting aspect of the combined algorithm is that the ample set is a function of an intermediate state. I.e., given a product state $x = \langle q, s\rangle$, the ample set is determined by the intermediate state $x' = \langle q, s'\rangle$ obtained after executing a property transition. This introduces a difference between the on-the-fly scheme and offline schemes, where there is no notion of intermediate state. It also introduces other complexities. For example, it is possible that x' was reached earlier in the search through some other state $\langle q, s_2\rangle$, because of a property transition $s_2 \xrightarrow{L(q)} s'$. How does the algorithm guarantee that the ample set selected for x' will be the same as the earlier choice? This issue is not addressed in [13] or [10].

These problems go away if one simply makes the ample set a function of the source product state x. The intermediate states do not have to play a role. Specifically, given an ample selector amp, define $\mathsf{reduced}_2(\mathcal{A}, \mathsf{amp})$ as in (1) and (2), except replace "$\alpha \in \mathsf{amp}(q, s')$" in (2) with "$\alpha \in \mathsf{amp}(q, s)$". Perform the same substitution in **C3** and call the resulting condition $\mathbf{C3}_1$. The weaker version of $\mathbf{C3}_1$ is simply:

$\mathbf{C3}'_1$ In any cycle in $\mathsf{reduced}_2(\mathcal{A}, \mathsf{amp})$ there is a state $\langle q, s\rangle$ with $\mathsf{amp}(q, s) = \mathsf{en}(q)$.

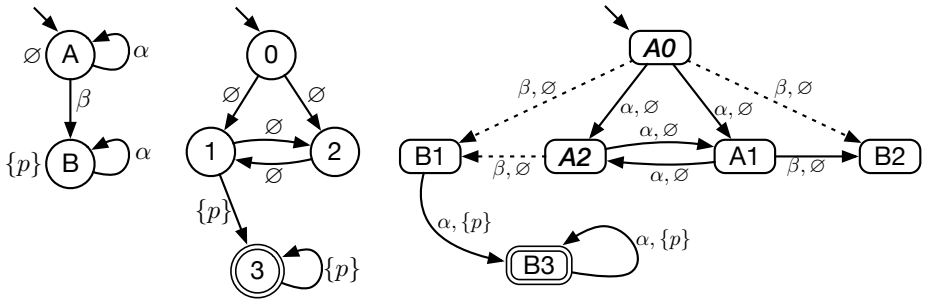

Fig. 3. Counterexample to V1 with $\mathcal{B}_2$ (center). $A0$ and $A2$ have proper ample set $\{\alpha\}$.

Conditions **C0**–**C2** are unchanged. I refer to this scheme as V1, and to the original combined algorithm as V0. The Alloy model of V0 in Sec. 4 can be easily modified to represent V1.

Using V1, the example of Fig. 1 is no longer a counterexample. In fact, Alloy reports there are no counterexamples using $\mathcal{B}_1$, at least for small bounds on the program size. Fig. 5 gives detailed results for this and other Alloy experiments.

Unfortunately, Alloy does find a counterexample for a slightly more complicated property automaton, $\mathcal{B}_2$, which is shown in Fig. 3.

The program is the same as the one in Sec. 3.1. Automaton $\mathcal{B}_2$ has four states, with state 3 the sole accepting state. The language is the same as that of $\mathcal{B}_1$: all infinite words formed by concatenating a finite nonempty prefix of $\varnothing$s and an infinite sequence of $\{p\}$s. If the prefix has odd length, the accepting run begins with the transition $0 \to 1$, otherwise it begins with the transition $0 \to 2$.

In the ample selector, only $A0$ and $A2$ are not fully enabled:

amp	0	1	2	3
A	$\{\alpha\}$	$\{\alpha, \beta\}$	$\{\alpha\}$	$\{\alpha, \beta\}$
B	$\{\alpha\}$	$\{\alpha\}$	$\{\alpha\}$	$\{\alpha\}$.

C0–**C2** hold for the reasons given in Sec. 3.1. $\mathbf{C3}_1$ holds for any DFS in which $A2$ is pushed onto the stack before $A1$. In that case, there is no back edge from $A2$; there will be a back edge when $A1$ is pushed, but $A1$ is fully enabled.

7 What's right

In this section, I show that POR scheme V1 of Sec. 6 is sound if one introduces certain assumptions on the property automaton. The following definition is similar to the notion of *stutter invariant (SI) automaton* in [6] and to that of *closure under stuttering* in [9]. The main differences derive from the use of Muller automata in [6] and *Büchi transition systems* in [9], while we are dealing with ordinary Büchi automata.

Definition 12. *A Büchi automaton $\mathcal{B} = \langle S, \{s_{init}\}, \Sigma, \delta, F\rangle$, is in* SI normal form *if it has a single initial state s_{init} with no incoming edges, and for each $s \in S \setminus \{s_{init}\}$, there is some $a_s \in \Sigma$ such that the following all hold:*

1. *Every edge terminating in s is labeled a_s.*
2. *s has exactly one outgoing edge with label a_s.*
3. *If $s \notin F$ then $\langle s, a_s, s\rangle \in \delta$.*
4. *If $\langle s, a_s, s\rangle \notin \delta$, then there exists $s^\sharp \in S \setminus F$ such that (i) $\langle s, a_s, s^\sharp\rangle \in \delta$ and (ii) for all $a \in \Sigma$ and $s' \in S$, $\langle s, a, s'\rangle \in \delta \Leftrightarrow \langle s^\sharp, a, s'\rangle \in \delta$.*

Lemma 1. *Let $\mathcal{B}$ be a Büchi automaton in SI normal form. Suppose $a, b \in \Sigma$ and $a \neq b$. Both of the following hold:*

1. *If $s_1 \xrightarrow{a} s_2 \xrightarrow{b} s_3$ is a path in $\mathcal{B}$, then for some $s_2' \in S$, $s_1 \xrightarrow{a} s_2 \xrightarrow{a} s_2' \xrightarrow{b} s_3$ is a path in $\mathcal{B}$.*
2. *If $s_1 \xrightarrow{a} s_2 \xrightarrow{a} s_3 \xrightarrow{b} s_4$ is a path in $\mathcal{B}$, then $s_1 \xrightarrow{a} s_2 \xrightarrow{b} s_4$ is a path in $\mathcal{B}$. Moreover, if s_3 is accepting, then s_2 is accepting.*

Following the approach of [6], one can show that the language of an automaton in SI normal form is stutter-invariant. Moreover, any Büchi automaton with a stutter-invariant language can be transformed into SI normal form without changing the language. The conversion satisfies $|S'| \leq O(|\Sigma||S|)$, where $|S|$ and $|S'|$ are the number of states in the original and new automaton, respectively. For details and proofs, see Appendix A. An example is given in Fig. 4; the language of $\mathcal{B}_3$ (or $\mathcal{B}_4$) consists of all words with a finite number of $\{p\}$s.

Theorem 1. *Suppose $\mathcal{B}$ is in SI normal form and* $\mathsf{amp}\colon Q \times S \to 2^T$ *is an ample selector satisfying* **C0**–**C2** *and* $\mathbf{C3}_1'$. *Then* amp *is POR-sound.*

The remainder of this section is devoted to the proof of Theorem 1. The proof is similar to the proof of the offline case in [4].

Let θ be an accepting path in the full space $\mathcal{A}$. An infinite sequence of accepting paths $\pi_0, \pi_1, \ldots$ will be constructed, where $\pi_0 = \theta$. For each $i \geq 0$, π_i will be decomposed as $\eta_i \circ \theta_i$, where η_i is a finite path of length i in the *reduced space*, θ_i is an infinite path, η_i is a prefix of η_{i+1}, and $\circ$ denotes concatenation. For $i = 0$, η_0 is empty and $\theta_0 = \theta$.

Assume $i \geq 0$ and we have defined η_j and θ_j for $j \leq i$. Write

$$\theta_i \;=\; \langle q_0, s_0\rangle \xrightarrow{\langle \alpha_1, \sigma_0\rangle} \langle q_1, s_1\rangle \xrightarrow{\langle \alpha_2, \sigma_1\rangle} \cdots \tag{3}$$

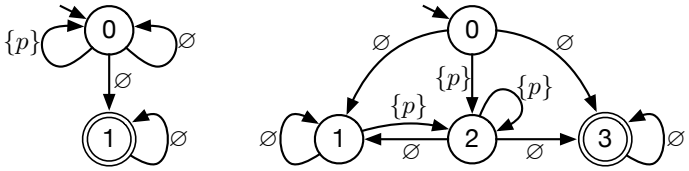

Fig. 4. Property automaton $\mathcal{B}_3$ and result of transformation to SI normal form, $\mathcal{B}_4$.

where $\sigma_k = L(q_k)$ for $k \geq 0$. Then η_{i+1} and θ_{i+1} are defined as follows. Let $A = \mathsf{amp}(q_0, s_0)$. There are two cases:

Case 1: $\alpha_1 \in A$. Let η_{i+1} be the path obtained by appending the first transition of θ_i to η_i, and θ_{i+1} the path obtained by removing the first transition from θ_i.

Case 2: $\alpha_1 \notin A$. Then there are two sub-cases:

Case 2a: Some operation in A occurs in θ_i. Let n be the index of the first occurrence, so that $\alpha_n \in A$, but $\alpha_j \notin A$ for $1 \leq j < n$. By **C1**, α_j and α_n are independent for $1 \leq j < n$. By repeated application of the independence property, there are paths in P

$$\begin{array}{ccccccccccccc}
q_0 & \xrightarrow{\alpha_1} & q_1 & \xrightarrow{\alpha_2} & q_2 & \xrightarrow{\alpha_3} & \cdots & \xrightarrow{\alpha_{n-2}} & q_{n-2} & \xrightarrow{\alpha_{n-1}} & q_{n-1} & & \\
\downarrow{\scriptstyle \alpha_n} & & \downarrow{\scriptstyle \alpha_n} & & \downarrow{\scriptstyle \alpha_n} & & & & \downarrow{\scriptstyle \alpha_n} & & \downarrow{\scriptstyle \alpha_n} & & \\
q'_1 & \xrightarrow{\alpha_1} & q'_2 & \xrightarrow{\alpha_2} & q'_3 & \xrightarrow{\alpha_3} & \cdots & \xrightarrow{\alpha_{n-2}} & q'_{n-1} & \xrightarrow{\alpha_{n-1}} & q_n & \xrightarrow{\alpha_{n+1}} & q_{n+1} \xrightarrow{\alpha_{n+2}} \cdots
\end{array}$$

By **C2**, α_n is invisible, whence $L(q'_{j+1}) = \sigma_j$ for $0 \leq j \leq n-2$, and $\sigma_{n-1} = \sigma_n$. Hence the admissible sequence

$$q_0 \overset{\alpha_n}{\to} q'_1 \overset{\alpha_1}{\to} q'_2 \overset{\alpha_2}{\to} q'_3 \to \cdots \overset{\alpha_{n-2}}{\to} q'_{n-1} \overset{\alpha_{n-1}}{\to} q_n \overset{\alpha_{n+1}}{\to} q_{n+1} \overset{\alpha_{n+2}}{\to} q_{n+2} \to \cdots \tag{4}$$

generates the word

$$\sigma_0\sigma_0\sigma_1\sigma_2 \cdots \sigma_{n-2}\sigma_n\sigma_{n+1}\sigma_{n+2} \cdots . \tag{5}$$

Now the projection of θ_i onto $\mathcal{B}$ has the form

$$s_0 \xrightarrow{\sigma_0} s_1 \xrightarrow{\sigma_1} s_2 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_{n-2}} s_{n-1} \xrightarrow{\sigma_n} s_n \xrightarrow{\sigma_n} s_{n+1} \xrightarrow{\sigma_{n+1}} s_{n+2} \xrightarrow{\sigma_{n+2}} \cdots$$

since $\sigma_{n-1} = \sigma_n$. By Lemma 1, there is a path in $\mathcal{B}$

$$s_0 \xrightarrow{\sigma_0} s_1 \xrightarrow{\sigma_0} s'_1 \xrightarrow{\sigma_1} s_2 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_{n-2}} s_{n-1} \xrightarrow{\sigma_n} s_n \xrightarrow{\sigma_{n+1}} s_{n+2} \xrightarrow{\sigma_{n+2}} \cdots \tag{6}$$

which accepts the word (5). Composing (4) and (6) therefore gives a path through the product space. Removing the first transition (labeled $\langle \alpha_n, \sigma_0 \rangle$) from this path yields θ_{i+1}. Appending that transition to η_i yields η_{i+1}.

Case 2b: No operation in A occurs in θ_i. By **C0**, A is nonempty. Let $\beta \in A$. By **C2**, every operation in θ_i is independent of β. With an argument that is similar to the one for Case 2a, we can see there is a path in the product space for which the projection onto the program component has the form

$$q_0 \overset{\beta}{\to} q'_1 \overset{\alpha_1}{\to} q'_2 \overset{\alpha_2}{\to} q'_3 \to \cdots$$

and the projection onto the property component has the form

$$s_0 \xrightarrow{\sigma_0} s_1 \xrightarrow{\sigma_0} s_1' \xrightarrow{\sigma_1} s_2 \xrightarrow{\sigma_2} \cdots .$$

Removing the first transition from this path yields θ_{i+1}. Appending that transition to η_i yields η_{i+1}. This completes the definitions of η_{i+1} and θ_{i+1}.

Let η be the limit of the η_i. Clearly η is an infinite path through the reduced product space, starting from the initial state. We must show that it passes through an accepting state infinitely often. To do so, we must examine more closely the sequence of property states through which each θ_i passes.

Let $i \geq 0$, and s_0 the final state of η_i. Say θ_i passes through states $s_0 s_1 s_2 \cdots$. Then the final state of η_{i+1} will be s_1, and the state sequence of θ_{i+1} is determined by the three cases as follows:

$$\begin{array}{rll} \text{Case 1:} & s_1 s_2 \cdots & \\ \text{Case 2a:} & s_1 s_1' s_2 \cdots s_n s_{n+2} \cdots & (s_{n+1} \in F \implies s_n \in F) \\ \text{Case 2b:} & s_1 s_1' s_2 \cdots & \end{array} \tag{7}$$

We first claim that for all $i \geq 0$, θ_i passes through an accepting state infinitely often. This holds for θ_0, which is an accepting path by assumption. Assume it holds for θ_i. In each case of (7), we see that the state sequence of θ_{i+1} has a suffix which is a suffix of the state sequence of θ_i, so the claim holds for θ_{i+1}.

Definition 13. *For any path $\xi = s_0 \to s_1 \to \cdots$ through $\mathcal{B}$ which passes through an accepting state infinitely often, define the* accepting distance of ξ, *written* $\mathsf{AD}(\xi)$, *to be the minimum $k \geq 1$ for which s_k is accepting.*

Lemma 2. *Let $i \geq 0$ and say the state sequence of θ_i is $s_0 s_1 s_2 \cdots$. If s_1 is not accepting then one of the following holds:*

- *Case 1 holds and $\mathsf{AD}(\theta_{i+1}) < \mathsf{AD}(\theta_i)$, or*
- *Case 2a or 2b holds and $\mathsf{AD}(\theta_{i+1}) \leq \mathsf{AD}(\theta_i)$.*

Proof. If s_1 is not accepting then there is some $k \geq 2$ for which s_k is accepting. The result follows by examining (7). In Case 1, the accepting distance decreases by 1. In Case 2a, the accepting distance is either unchanged (if $k \leq n$) or decreases by 1 (if $k > n$). In Case 2b, the accepting distance is unchanged. □

Lemma 3. *For an infinite number of $i \geq 0$, Case 1 holds for θ_i.*

Proof. Suppose not. Then there is some $i \geq 0$ such that Case 2 holds for all $j \geq i$. Let α_1 be the first program operation of θ_i. Then α_1 is the first program operation of θ_j, for all $j \geq i$. Furthermore, for all $j \geq i$, α_1 is not in the ample set of the final state of η_j. Since the product space has only a finite number of states, this means there is a cycle in the reduced space for which α_1 is enabled but never in the ample set, contradicting $\mathbf{C3}_1'$. □

v	BA	`Sigma`	`BState`	`BTrans`	`Operation`	`PState`	`ProdState`	time (s)	Result
V0	$\mathcal{B}_1$	2	2	3	≤ 2	≤ 2	≤ 4	0.3	✗
V1	$\mathcal{B}_1$	2	2	3	≤ 3	≤ 5	≤ 10	42.3	✓
V0	$\mathcal{B}_2$	2	4	6	≤ 2	≤ 2	≤ 6	0.4	✗
V1	$\mathcal{B}_2$	2	4	6	≤ 2	≤ 2	≤ 6	0.3	✗
V0	$\mathcal{B}_3$	2	2	4	≤ 3	≤ 5	≤ 10	256.3	✓
V1	$\mathcal{B}_3$	2	2	4	≤ 3	≤ 5	≤ 10	280.7	✓
V0	$\mathcal{B}_4$	2	4	9	≤ 3	≤ 4	≤ 16	39.5	✓
V1	$\mathcal{B}_4$	2	4	9	≤ 3	≤ 4	≤ 16	37.7	✓
V0	$\mathcal{B}_5$	≤ 3	≤ 4	≤ 6	≤ 3	≤ 4	≤ 16	2264.9	✓
V1	$\mathcal{B}_5$	≤ 3	≤ 4	≤ 6	≤ 3	≤ 4	≤ 16	1653.9	✓

Fig. 5. Bounded verification of soundness of POR schemes V0 and V1 on various Büchi automata using Alloy. $\mathcal{B}_5$ represents all automata in SI normal form within the bounds. Each run resulted in either a counterexample (✗) or not (✓).

We now show that η passes through an accepting state infinitely often. Note that, if $\mathsf{AD}(\theta_i) = 1$, an accepting state is added to η_i to form η_{i+1}. Suppose η does not pass through an accepting state infinitely often. Then there is some $i \geq 0$ such that for all $j \geq i$, $\mathsf{AD}(\theta_j) > 1$. By Lemma 2, $(\mathsf{AD}(\theta_j))_{j \geq i}$ is a nonincreasing sequence of positive integers, and by Lemma 3, this sequence strictly decreases infinitely often, a contradiction. This completes the proof of Theorem 1.

Remark 1. The proof goes through with minor modifications for V0 in place of V1. Let $A = \mathsf{amp}(q_0, s_1)$ instead of $\mathsf{amp}(q_0, s_0)$. In Case 2a (similarly in 2b), note the first transition $s_0 \xrightarrow{\sigma_0} s_1$ in the path in $\mathcal{B}$ remains in the new path (6).

8 Summary of Experimental Results and Conclusion

We have seen that standard ways of combining POR and on-the-fly model checking are unsound. This is not only a theoretical issue—the defect in the algorithm is realized in Spin, which can produce an incorrect result. A modification (V1) seems to help, but is still not enough to guarantee soundness for any Büchi automaton with a stutter-invariant language. However, any such automaton can be transformed into a normal form for which algorithm V1 is sound.

Alloy proved useful for reasoning about the algorithms and generating small counterexamples. A summary of the Alloy experiments and results is given in Fig. 5. These were run on an 8-core 3.7GHz Intel Xeon W-2145 and used the plingeling SAT solver [1].[3] In addition to the experiments already discussed, Alloy found no soundness counterexamples for property automata $\mathcal{B}_3$ or $\mathcal{B}_4$, using V0 or V1. In the case of $\mathcal{B}_4$, this is what Theorem 1 predicts. For further confirmation of Theorem 1, I constructed a general Alloy model of Büchi automata in SI normal

[3] All artifacts needed to reproduce the experiments reported in this paper can be downloaded from http://vsl.cis.udel.edu/cav19.

BA	POR	states(stored)	transitions	time(s)	Result
$\mathcal{B}_3$	N	18,964,912	116,510,960	25.8	✓
$\mathcal{B}_3$	Y	4,742,982	13,823,705	3.6	✓
$\mathcal{B}_4$	Y	4,719,514	12,503,008	3.4	✓

Fig. 6. Spin verification of starvation-freedom for 5-process Peterson. Using the SI normal form $\mathcal{B}_4$ instead of the smaller $\mathcal{B}_3$ has little impact on performance.

form, represented by $\mathcal{B}_5$ in the table. Alloy confirms that both V0 and V1 are sound for all such automata within small bounds on program and automata size.

It is possible that the use of the normal form, while correct, cancels out the benefits of POR. A comprehensive exploration of this issue is beyond the scope of this paper, but I can provide data on one non-trivial example. I encoded an n-process version of Peterson's mutual exclusion algorithm in Promela, and used Spin to verify starvation-freedom for one process in the case $n = 5$. If p is the predicate that holds whenever the process is enabled, a trace violates this property if p holds only a finite number of times in the trace, i.e., if the trace is in $\mathcal{L}(\mathcal{B}_3) = \mathcal{L}(\mathcal{B}_4)$. Fig. 6 shows the results of Spin verification using $\mathcal{B}_3$ without POR, and using $\mathcal{B}_3$ and $\mathcal{B}_4$ with POR. The results indicate that POR significantly improves performance on this problem, and that using the normal form $\mathcal{B}_4$ in place of $\mathcal{B}_3$ actually *improves* performance further by a small amount.

It is likely that V1 is sound for other interesting classes of automata. Observe, for example, that $\mathcal{B}_2$ of Fig. 3 has states u where the language of the automaton with u considered as the initial state is *not* stutter-invariant. If we restrict to automata in which every state has a stutter-invariant language, is V1 sound? I have neither a proof nor a counterexample. (This is certainly not true of V0, as $\mathcal{B}_1$ is a counterexample.) To explore this question, it would help to find a way to encode the stutter-invariant property—or a suitable approximation—in Alloy.

Finally, the proof of Theorem 1 is complicated and might also be flawed. Recent work mechanizing such proofs [3] represents an important advance in raising the level of assurance in model checking algorithms. It would be interesting to see if the proof of this theorem is amenable to such methods. However, constructing such proofs requires far more effort than the Alloy approach described here. One possible approach moving forward is to use tools such as Alloy when prototyping a new algorithm, to get feedback quickly and root out bugs. Once Alloy no longer finds any counterexamples, one could then expend the considerable effort required to construct a formal mechanized proof.

Acknowledgements. I am grateful to Ganesh Gopalakrishnan and Julian Brunner for fruitful conversations on partial order reduction, to Gerard Holzmann for help with Spin, and to the anonymous reviewers for suggestions that improved this paper. This material is based upon work by the RAPIDS Institute, supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program. Funding was also provided by the U.S. National Science Foundation under award CCF-1319571.

References

1. Biere, A.: CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In: Balyo, T., Heule, M., Järvisalo, M. (eds.) Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2017-1, pp. 14–15. University of Helsinki (2017)
2. Brunner, J.: Implementation and Verification of Partial Order Reduction for On-The-Fly Model Checking. Master's thesis, Technische Universität München, Department of Computer Science (Jul 2014), https://www21.in.tum.de/~brunnerj/documents/ivporotfmc.pdf
3. Brunner, J., Lammich, P.: Formal verification of an executable LTL model checker with partial order reduction. Journal of Automated Reasoning **60**, 3–21 (2018). https://doi.org/10.1007/s10817-017-9418-4
4. Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
5. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. Formal Methods in System Design **1**(2), 275–288 (Oct 1992), https://doi.org/10.1007/BF00121128
6. Etessami, K.: Stutter-invariant languages, ω-automata, and temporal logic. In: Halbwachs, N., Peled, D. (eds.) Computer Aided Verification. pp. 236–248. Springer, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-48683-6_22
7. Holzmann, G., Peled, D., Yannakakis, M.: On nested depth first search. In: The Spin Verification System, DIMACS - Series in Discrete Mathematics and Theoretical Computer Science, vol. 32, pp. 23–31. AMS and DIMACS (1997), https://bookstore.ams.org/dimacs-32/
8. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, Boston (2004)
9. Holzmann, G.J., Kupferman, O.: Not checking for closure under stuttering. In: Grégoire, J.C., Holzmann, G.J., Peled, D.A. (eds.) The SPIN Verification System. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 32, pp. 17–22. American Mathematical Society (1997)
10. Holzmann, G.J., Peled, D.: An improvement in formal verification. In: Hogrefe, D., Leue, S. (eds.) Proceedings of the 7th IFIP WG6.1 Intl. Conference on Formal Description Techniques (Forte '94). IFIP Conference Proceedings, vol. 6, pp. 197–211. Chapman & Hall (1995), http://dl.acm.org/citation.cfm?id=646213.681369
11. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, revised edition edn. (2012)
12. Peled, D.: Combining partial order reductions with on-the-fly model-checking. In: Dill, D.L. (ed.) Computer Aided Verification. pp. 377–390. Springer, Berlin, Heidelberg (1994). https://doi.org/10.1007/3-540-58179-0_69
13. Peled, D.: Combining partial order reductions with on-the-fly model-checking. Formal Methods in System Design **8**(1), 39–64 (Jan 1996), https://doi.org/10.1007/BF00121262
14. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: Halbwachs, N., Zuck, L.D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 174–190. Springer, Berlin, Heidelberg (2005), https://doi.org/10.1007/978-3-540-31980-1_12
15. Siegel, S.F.: Reexamining two results in partial order reduction. Tech. Rep. UD-CIS-2011/06, U. Delaware (2011), http://vsl.cis.udel.edu/pubs/por_tr_2011.html

16. Siegel, S.F.: Transparent partial order reduction. Formal Methods in System Design **40**(1), 1–19 (2012). https://doi.org/10.1007/s10703-011-0126-0

A SI Normal Form

Proof (of Lemma 1). Part 1: if s_2 has a self-loop on a, then let $s_2' = s_2$. Otherwise, by Definition 12(4), there is a state $s_2^\sharp$ and transitions $s_2 \xrightarrow{a} s_2^\sharp$ and $s_2^\sharp \xrightarrow{b} s_3$, so we may take $s_2' = s_2^\sharp$.

Part 2: if s_2 has a self-loop on a, then according to Definition 12(2), $s_3 = s_2$, and we are done. Otherwise, there is a state $s_2^\sharp$ satisfying the conditions of Definition 12(4). In particular, there is a transition $s_2 \xrightarrow{a} s_2^\sharp$. By Definition 12(2), $s_2^\sharp = s_3$. By Definition 12(4), there is a transition $s_2 \xrightarrow{b} s_4$. □

Proposition 1. *The language of a Büchi automaton $\mathcal{B}$ in SI normal form is stutter-invariant.*

Proof. Follows from Lemma 1. The first part allows one to increase the stuttering at any point in a path, the second allows one to decrease it. Moreover, the decrease in stutter will preserve an accepting state, so applying such a transformation to an accepting run yields an accepting run. □

Theorem 2. *Given any Büchi automaton $\mathcal{B} = \langle S, \Delta, \Sigma, \delta, F\rangle$, there is a Büchi automaton $\mathcal{B}'$ in SI normal form with $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{B}')$. Moreover, the number of states in $\mathcal{B}'$ is at most $\mathcal{O}(|\Sigma||S|)$.*

The remainder of this section is devoted to the proof of Theorem 2. Define

$$S' = \{s_{init}\} \;\cup\; S \times \Sigma \times \{0\} \;\cup\; F \times \Sigma \times \{1\} \;\cup\; \Sigma \times \{2\}.$$

Let $F' = F \times \Sigma \times \{0\} \cup \Sigma \times \{2\}$. Let δ' be the union of the following sets:

$$\{s_{init} \xrightarrow{a} (q, a, 0) \mid \text{for some } q_0 \in \Delta,\ \langle q_0, a, q\rangle \in \delta\} \tag{8}$$

$$\{s_{init} \xrightarrow{a} (a, 2) \mid a^\omega \in \mathcal{L}(B)\} \tag{9}$$

$$\{(q, a, 0) \xrightarrow{b} (q', b, 0) \mid a \neq b \wedge \langle q, b, q'\rangle \in \delta\} \tag{10}$$

$$\{(q, a, 0) \xrightarrow{a} (q, a, 0) \mid q \notin F\} \tag{11}$$

$$\{(q, a, 0) \xrightarrow{a} (q, a, 1) \mid q \in F\} \tag{12}$$

$$\{(q, a, 1) \xrightarrow{a} (q, a, 1) \mid q \in F\} \tag{13}$$

$$\{(q, a, 1) \xrightarrow{b} (q', b, 0) \mid q \in F \wedge a \neq b \wedge \langle q, b, q'\rangle \in \delta\} \tag{14}$$

$$\{(a, 2) \xrightarrow{a} (a, 2) \mid a \in \Sigma\} \tag{15}$$

$$\{(q, a, 0) \xrightarrow{b} (b, 2) \mid a \neq b \wedge b^\omega \in \mathcal{L}(q)\} \tag{16}$$

$$\{(q, a, 1) \xrightarrow{b} (b, 2) \mid q \in F \wedge a \neq b \wedge b^\omega \in \mathcal{L}(q)\} \tag{17}$$

Let $\mathcal{B}' = \langle S', \{s_{init}\}, \Sigma, \delta', F'\rangle$.

Lemma 4. *$\mathcal{B}'$ is in SI normal form.*

Proof. From the definition of δ', it is clear that s_{init} has no incoming edges. In addition, for any state of the form (q, a, e) or $(a, 2)$, all incoming edges are labeled a and there is exactly one outgoing edge labeled a. Hence conditions (1) and (2) of Definition 12 hold.

If s is not accepting and not s_{init}, then either (i) $s = (q, a, 0)$, where $q \notin F$, or (ii) $s = (q, a, 1)$, where $q \in F$. In the first case, s has a self loop on a by (11). In the second case, there is a self loop on a by (13). Hence Definition 12(3) holds.

Finally, suppose s is a state with incoming edges labeled by a, and s has no self-loop labeled a. Then $s = (q, a, 0)$, where $q \in F$. Define $s^\sharp = (q, a, 1)$. We obtain the fourth condition in Definition 12 because the edges departing from s are given by equations (10), (12), and (16), and these correspond exactly to the edges departing from $s^\sharp$, which are given by (14), (13), and (17), respectively. □

By Proposition 1 and Lemma 4, $\mathcal{L}(\mathcal{B}')$ is stutter-invariant. Therefore, to show $\mathcal{L}(\mathcal{B}) = \mathcal{L}(\mathcal{B}')$, it suffices to show that $\mathcal{B}$ and $\mathcal{B}'$ accept the same *stutter-free* words [6].

Let $w = a_0 a_1 \cdots \in \Sigma^\omega$ be a stutter-free word. There are two possibilities:

- w has *type 1*: $a_i \neq a_{i-1}$ for all $i \geq 1$, or
- w has *type 2*: there is some $n \geq 0$ and $b \in \Sigma$ such that $a_i = b$ for all $i \geq n$, and $a_i \neq a_{i-1}$ for $1 \leq i \leq n$.

Lemma 5. *If $w \in \mathcal{L}(\mathcal{B})$ then $w \in \mathcal{L}(\mathcal{B}')$.*

Proof. Say

$$q_0 \xrightarrow{a_0} q_1 \to \cdots \xrightarrow{a_{i-1}} q_i \xrightarrow{a_i} q_{i+1} \to \cdots$$

is an accepting run for w in $\mathcal{B}$.

If w has type 1, we claim that

$$s_{init} \xrightarrow{a_0} (q_1, a_0, 0) \to \cdots \xrightarrow{a_{i-1}} (q_i, a_{i-1}, 0) \xrightarrow{a_i} (q_{i+1}, a_i, 0) \to \cdots \tag{18}$$

is an accepting run for w in $\mathcal{B}'$. Indeed, the first transition is in δ' by (8), and all other transitions are in δ' by (10). Since q_i is accepting if, and only if, $(q_i, a_i, 0)$ is accepting, (18) is an accepting run for w in $\mathcal{B}'$.

If w has type 2, we claim

$$s_{init} \xrightarrow{a_0} (q_1, a_0, 0) \to \cdots \xrightarrow{a_{n-1}} (q_n, a_{n-1}, 0) \xrightarrow{b} (b, 2) \xrightarrow{b} (b, 2) \xrightarrow{b} \cdots \tag{19}$$

is an accepting run for w. The first n transitions are in δ' exactly as in the first case. The first transition to $(b, 2)$ is in δ' because of (9) (if $n = 0$) or (16) (if $n \geq 1$). The remaining self-loops on b follow from (15). □

Lemma 6. *If $w \in \mathcal{L}(\mathcal{B}')$ then $w \in \mathcal{L}(\mathcal{B})$.*

Proof. Suppose w has type 1. No accepting run for w in $\mathcal{B}'$ can enter a state in $\Sigma \times \{2\}$, because such a run must culminate with a suffix of the form a^ω for some $a \in \Sigma$. That means no transition from (9), (15), (16), or (17) can occur in an accepting run. No transition in (11), (12), or (13) can occur, because then the preceding transition would have the same label, contradicting the assumption that w is stutter-free. The remaining classes of transitions never enter a state in $F \times \Sigma \times \{1\}$, hence no transition in (14) can occur. Hence the accepting run consists only of transitions in (8) and (10). Such a run clearly corresponds to an accepting run in $\mathcal{B}$ by projecting the state onto the first component.

Now supposed w has type 2, and consider an accepting run for w in $\mathcal{B}'$:

$$s_{init} \xrightarrow{a_0} \cdots \xrightarrow{a_{n-1}} s \xrightarrow{b} s' \xrightarrow{b} s'' \cdots$$

The first n transitions must lie in (8) and (10), exactly as in case 1. The definition of δ' permits only three cases where two consecutive b transitions can occur, so the suffix of this run starting from s has one of the following forms:

$$s \xrightarrow{b} (q, b, 0) \xrightarrow{b} (q, b, 1) \xrightarrow{b} \cdots \tag{20}$$

$$s \xrightarrow{b} (b, 2) \xrightarrow{b} (b, 2) \xrightarrow{b} \cdots \tag{21}$$

$$s \xrightarrow{b} (q, b, 0) \xrightarrow{b} (q, b, 0) \xrightarrow{b} \cdots \tag{22}$$

Note that a state of the form $(q, b, 1)$ cannot have an accepting run on b^ω: the state is non-accepting, but the only transition departing from it and labeled by b is a self-loop. This eliminates (20).

Consider case (21). The transition from s must lie in one of the sets (9), (16), or (17). In the first case, b^ω is accepted from some initial state of $\mathcal{B}$, and we are done. In the second case, s has the form $(q, a, 0)$ and b^ω is accepted from q; hence one obtains an accepting run for w in $\mathcal{B}$ by concatenating the run for the prefix of length n terminating in q with some accepting run on b^ω from q. The third case (17) is not possible because the only way to reach state $(q, a, 1)$ is through two consecutive a-transitions, violating the assumption that w is stutter-free.

Finally, consider case (22). Since $(q, b, 0)$ has exactly one outgoing edge labeled b, the run stays in state $(q, b, 0)$ forever. Hence every state in the run after the initial state is in $S \times \Sigma \times \{0\}$, and there is an accepting run in $\mathcal{B}$ by projection onto the first component. □