Preserving LTL Properties in Sweep-Line State Space Exploration with Partial-Order Reduction

Sami Evangelista[1 [0000−0002−7666−583X]],
Lars M. Kristensen[2 [0000−0002−1465−5791]], and Laure Petrucci[1 [0000−0003−3154−5268]]

[1] LIPN, CNRS UMR 7030
Université Sorbonne Paris Nord
[2] Western Norway University of Applied Sciences
Dept. of Computer Science, Electrical Engineering, and Mathematical Sciences
sami.evangelista@univ-paris13.fr, lmkr@hvl.no,
laure.petrucci@lipn.univ-paris13.fr

Abstract. Model checking often requires the combined use of multiple reduction techniques to mitigate the state explosion problem. The contribution of this paper is a generalised sweep-line algorithm that enforces the conditions necessary for partial-order reduction in the form of stubborn sets to preserve LTL_{-X} properties. The core idea in our algorithm is to compute strongly connected components and detect cycles within each state space layer explored by the sweep-line method, while leveraging persistent states as anchor states for enforcing cross-layer partial-order reduction conditions. We have implemented our new algorithm and conducted an experimental evaluation on set of benchmarks from the Petri Nets Model Checking Contest. The results show our LTL_{-X} preserving combination of sweep-line exploration with partial-order reduction even in a conservative implementation may substantially reduce peak memory usage. Furthermore, the overhead is mostly similar to plain partial-order preserving LTL_{-X} reduction.

Keywords: Model checking · Verification · Temporal Logic · State explosion

1 Introduction

The sweep-line method [3,10] and partial-order reduction [11] have been developed to alleviate the inherent state explosion problem in model checking. The sweep-line method exploits a progress measure on states to partition the state space into layers and delete states from memory on-the-fly during state space exploration. This reduces the number of states that must be stored simultaneously in memory. Partial-order reduction analyses dependencies between transitions (actions/events) and explores in each encountered state only a subset of the enabled transitions. This reduces the interleaved executions considered and implies that only a *reduced state space*, i.e., a subset of the full state space is explored. It has been demonstrated [14] that the sweep-line method and partial-order reduction are complementary, i.e., using both methods simultaneously yields better reduction than when either method is used alone. In many cases, the sweep-line method must be combined with partial-order reduction to effectively alleviate state explosion.

Linear-time Temporal Logic (LTL) is being widely used for expressing behavioural properties in model checking [1] and is often combined with partial-order reduction to

improve efficiency. Model checking of safety properties with the sweep-line method was described in [10]. To the best of our knowledge, earlier research [14] on the combination of the sweep-line method and partial-order reduction has not considered the preservation of LTL_{-X} properties (linear-time temporal logic without the next operator) for the generalized sweep-line method [10], i.e., covering both monotonic and non-monotonic progress measures. In earlier work [5], we developed an automata-based algorithm for LTL model checking with the sweep-line method combining the MAP algorithm [2] with nested depth-first to detect acceptance cycles [8]. The combined use of the sweep-line method and partial-order reduction for LTL_{-X} properties was, however, not investigated in [5].

Partial-order reduction generally requires certain conditions to be fulfilled by the set of transitions selected for exploration in each encountered state. The conditions to be enforced depend upon the class of properties considered and must ensure that the property under verification is *preserved*, i.e., that it holds in the reduced state space if and only if it holds in the full state space. In the context of this paper, we are concerned with how partial-order reduction conditions for LTL_{-X} can be enforced with sweep-line state space exploration.

The sweep-line method when not combined with other reduction methods explores the full state space. Hence, when combined with partial-order reduction it will explore the same reduced state space as the partial-order method prescribes as per the conditions being enforced. This means that the sweep-line method does not impose specific additional conditions in terms of the property being preserved. The primary aspect is therefore how to ensure that the conditions on the selected transitions of the partial-order reduction are being enforced. This is the focus of this paper, where we consider partial-order reduction in the framework of stubborn sets [15,16] and the conditions imposed on stubborn sets for the preservation of LTL_{-X} properties. A secondary aspect is how properties can be verified (assuming that they are preserved) given that the sweep-line method: 1) only stores a subset of the state space being explored in memory at a time; and 2) prescribes a least-progress first exploration order. For LTL_{-X}, we addressed this secondary aspect already in [5] by developing a full LTL model checking algorithm compatible with the sweep-line exploration. In the context of the present paper, the algorithm of [5] would be used on the sweep-line explored space space reduced using stuborn sets preserving LTL_{-X}.

The rest of this paper is organised as follows. Section 2 introduces basic state space notations and the example state space that we use throughout the paper for illustration purposes. Section 3 introduces the sweep-line method, and Sect. 4 introduces the stubborn set method. In Sect. 5 and Sect. 6 we develop our new sweep-line algorithm enforcing stubborn set conditions for preserving LTL_{-X} properties. Section 7 introduces our initial prototype implementation and presents results from our experimental evaluation. Finally, in Sect. 8 we sum up conclusions and discuss future work. We assume familiarity with the basic principles of explicit-state model checking.

2 Background

To maintain independence from any particular modeling language for concurrent systems, our algorithms are presented at the level of state spaces.

Definition 1. *A* **state space** *is a tuple* (S, T, Δ, ι)*, where* S *is a finite set of* **states***,* T *is a finite set of* **transitions***,* $\Delta \subseteq S \times T \times S$ *is a* **transition relation***, and* $\iota \in S$ *is the* **initial state***.*

By Def. 1, we consider state spaces to be finite. Furthermore, we assume transitions to be deterministic to simplify the presentation of the stubborn set method. Our algorithm can, however, be adapted to accomodate also non-deterministic transitions.

A transition $t \in T$ is said to be *enabled* in a state $s \in S$ if there exists a state $s' \in S$ such that $(s, t, s') \in \Delta$. We denote by $\mathsf{enab} : S \to 2^T$ the mapping that associates with each state s the set of enabled transitions in s, i.e. for a state $s \in S$, $\mathsf{enab}(s) = \{t \in T \mid \exists s' \in S \wedge (s, t, s') \in \Delta\}$. Similarly, we denote by $\mathsf{succ} : S \times T \to S$ the mapping that associates with each state s and enabled transition t in s, the successor state s' resulting from executing t in s. Formally, for $(s, t, s') \in \Delta$, $\mathsf{succ}(s, t) = s'$. If $t \in \mathsf{enab}(s)$ we write $s - t \rightarrow$, and when $s' = \mathsf{succ}(s, t)$ we write $s - t \rightarrow s'$. When the transition is not important, we may omit the label and write $s \rightarrow s'$. A *transition sequence* $s_0 - t_1 \rightarrow s_1 - t_2 \rightarrow s_2 \cdots s_{n-1} - t_n \rightarrow s_n$ is a sequence of states s_i and transitions t_i such that $s_{i-1} - t_i \rightarrow s_i$ for all $1 \leq i \leq n$. If a state s' can be obtained from a state s by the execution of a (possibly empty) transition sequence, we say that s' is *reachable* from s and write this as $s - t_1 t_2 \cdots t_n \rightarrow s'$ Basic state space exploration starts from the initial state and successively *processes* states by exploring the enabled transitions and computing successor states. A set of *visited states* is stored in memory to ensure that each encountered state is only processed once.

The state space of a system can be viewed as a directed graph, and Fig. 1 depicts the state space that we will use for illustration purposes. Each node (state) is identified by an integer written inside the state. Node 0 corresponds to the initial state ι. Cycles and *strongly connected components (SCCs)* play a key role in our approach to enforcing the conditions on stubborn sets to preserve LTL_{-X} properties. The SCCs of the state space in Fig. 1 containing more than one node are indicated using light gray boxes and are maximal subgraphs such that two states s_1 and s_2 are in the same SCC if and only if s_1 is reachable from s_2 and vice versa. A *terminal SCC* S is an SCC such that no state contained in S has outgoing edges to states contained in other SCCs. The SCCs marked with B, D, and E are terminal SCCs. The dashed vertical lines and the numbered black squares will be explained in the next section.

An LTL property to be verified is composed of atomic propositions and temporal operators, where an atomic proposition $\phi : S \mapsto \{T, F\}$ maps each state into a Boolean value specifying whether the atomic proposition is true (T) or false (F) in the state. Given an LTL property Φ to be verified, a transition t is *visible* if there exists states s and s' with $(s, t, s') \in \Delta$ and an atomic proposition ϕ of Φ such that $\phi(s) \neq \phi(s')$, i.e., the execution of t in s changes the truth value of the atomic proposition. A transition that is not visible is *invisible*.

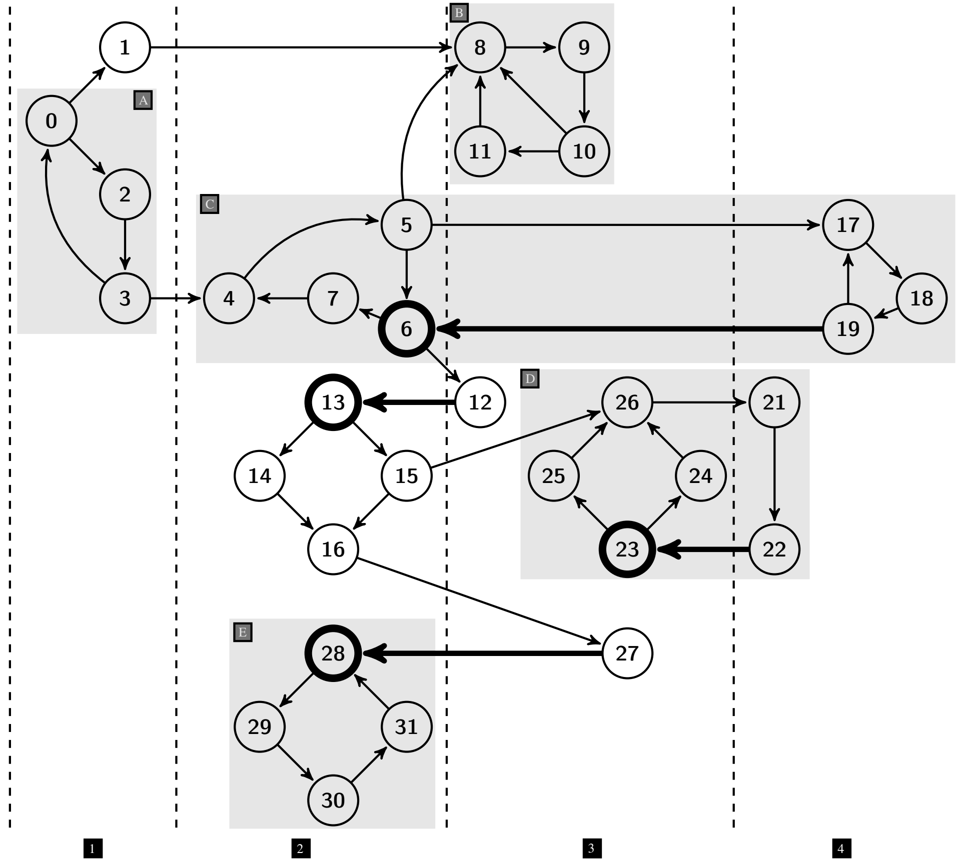

Fig. 1. Example state space and strongly connected components (light gray boxes).

3 Sweep-Line State Space Exploration

The sweep-line method [3,10] relies on a *progress measure* intended to quantify the system progression that a state represents. The basic intuition of a progress measure is that it must capture some inherent progress in the system such that progress generally increases as the system is being executed. The progress measure partitions the state space into *layers* where each layer consists of states mapped to the same progress value.

Definition 2. *Let* (S, T, Δ, ι) *be a state space. A* **progress measure** *is a mapping* $\psi : S \mapsto O$*, where O is a non-empty set of* **progress values** *equipped with a total order* $\sqsubseteq$*. A progress measure is* **monotonic** *if for all* $(s, t, s') \in \Delta : \psi(s) \sqsubseteq \psi(s')$*. A progress measure that is not monotonic is* **non-monotonic***. A* **forward edge** *is a triple* $(s, t, s') \in \Delta$ *such that* $\psi(s) \sqsubseteq \psi(s')$*, while a* **regress edge** *is a triple* $(s, t, s') \in \Delta$ *such that* $\psi(s') \sqsubset \psi(s)$*.*

The dashed vertical lines and the numbered black squares in Fig. 1 illustrate an example progress measure that partitions the state space into four layers based on progress values 1, 2, 3, and 4. Layer 1 consists of states 0-3. Layer 2 consists of states 4-7, states 13-16, and states 28-31. Layer 3 consists of states 8-12, and states 23-27. Layer 4 consists of states 17-19 and 21-22.

The sweep-line algorithm starts with the layer containing the initial state and then explores the state space one layer at a time. When all states in a given layer have been processed, the states in the current layer are deleted since the assumption is that they are not anymore needed for comparison with newly encountered states, i.e., determining whether a newly generated successor state is already contained in the set of visited states. For the state space in Fig. 1, the sweep-line method would start from state 0 and then compute the successor states 1 and 2 which will then be processed. Once all states in layer 1 have been visited, the states will be deleted from memory and the exploration will continue with states 4-7 in layer 2. States in layer 2 will then be deleted, and states 8-11 and state 12 in layer 3 will be processed. When the algorithm processes state 12, it will discover the *regress edge* to state 13 and mark state 13 as *persistent* which means that it can no longer be deleted from memory. Non-persistent states in layer 2 will then be deleted, and the algorithm will explore states 17-19 where it will discover the regress edge from state 19 to state 6 making state 6 persistent. States 6 and 13 will now be used as *roots* for a second iteration in the sweep-line algorithm. From state 6, states 4-7 will be (re)explored followed by state 12, states 8-11, and states 17-19. When the regress edge from state 12 to state 13 and the regress edge from state 19 to state 6 are explored for the second time, states 6 and 13 will be present in memory (as they became persistent in the previous iteration). From state 13, states 13-16 will be explored before the exploration will continue from state 26 and state 27 in layer 3 which will eventually lead to the full state space having been explored.

Algorithms 1 and 2 specify the generalised sweep-line algorithm [10]. The sweep-line algorithm starts with the initial state ι as the root state for the first sweep (l. 6-8). The algorithm iterates as long as new roots exists (l. 9-13) using the current roots $\mathcal{R}$ as unprocessed states Q. The procedure SWEEPITERATION (l. 16-18) explores the states layer by layer in a least progress first order as Q is a priority queue on progress values. The procedures EXPLORELAYER and EXPLORESTATE work basically as a standard state space exploration, where states are expanded and their successors that have not been visited so far are put in Q to be later processed. Unlike standard state space exploration, the algorithm maintains states of the current layer in $\mathcal{L}$ (l. 3 and 21 in Alg. 2) such that we can efficiently delete non-persistent states in the current layer when we have explored the layer (l. 8 in Alg. 2). Furthermore, persistent states, regress edges, and new roots are detected l. 14-16 in Alg. 2.

It can be seen from Fig. 1 that SCCs may span multiple layers. A (global) SCC is called an *inter-layer SCC* if it contains states from multiple layers, while a (global) SCC is said to be an *intra-layer SCC* if all its states are confined within a single layer. SCC C is an example of an inter-layer SCC, while SCC B is an example of an intra-layer SCC. Similarly, an *inter-layer cycle* is a cycle in the state space containing states from at least two different layers, and an *intra-layer cycle* is a cycle containing states only

Algorithm 1 The sweep-line algorithm of [10]

```
 1: Set of states 𝒮                          ▷ Visited states currently stored in memory
 2: Set of root states ℛ ⊆ 𝒮    ▷ States that will serve as roots in next sweep iteration
 3: Progress measure ψ
 4: Priority queue 𝒬 ⊆ 𝒮   ▷ States to be processed, priority based on progress value

 5: procedure SWEEP()
 6:     ι.pers ← false
 7:     ℛ ← {ι}
 8:     𝒮 ← {ι}

 9:     while ℛ ≠ ∅ do                     ▷ Explore starting from current set of roots
10:         𝒬 ← ℛ
11:         ℛ ← ∅
12:         SWEEPITERATION()
13:     end while

14: end procedure

15: procedure SWEEPITERATION()

16:     while 𝒬 ≠ ∅ do           ▷ Explore as long as there are unprocessed states
17:         ψ_l ← 𝒬.MINPROGRESS()
18:         EXPLORELAYER(ψ_l)
19:     end while

20: end procedure
```

from a single layer. Throughout this paper, we use the following fundamental properties of sweep-line state space exploration.

Theorem 1. *Let (S, T, Δ, ι) be a state space and let $\psi : S \mapsto O$ be a progress measure. Then the following holds:*

1. *The sweep-line state space exploration algorithm in Alg. 1-2 terminates after having explored all states reachable from ι at least once [10].*
2. *An inter-layer cycle and an inter-layer SCC contain at least one regress edge and at least one persistent state.*
3. *If ψ is a monotonic progress measure, then all cycles are intra-layer cycles and all SCCs are intra-layer SCCs.*

Proof. Item 1 on completeness and termination of the sweep-line algorithm was already proved in the original paper [10] that introduced the generalized sweep-line method. For item 2, let $s = s_0 {-} t_1 \rightarrow s_1 {-} t_2 \rightarrow \cdots s_{n-1} t_n {-} s_n = s \rightarrow$ be a cycle. Since it is an inter-layer cycle, there must be at least two states s_i and s_j with $i \neq j$ such that $\psi(s_i) \neq \psi(s_j)$. Hence, there must be at least one regress edge and associated persistent state on the cycle. The second part of item 2 follows as an inter-layer SCC spans multiple layers and

Algorithm 2 The sweep-line ExploreLayer and ExploreState procedures

```
 1: States 𝓛 ⊆ S                                          ▷ states in the current layer

 2: procedure ExploreLayer(ψ_l)
 3:     𝓛 ← ∅

 4:     while Q.minProgress() = ψ_l do        ▷ Explore as long as we are in current layer
 5:         s ← Q.dequeue()
 6:         ExploreState(s)
 7:     end while

 8:     S ← S \ { s ∈ S | s ∈ 𝓛 ∧ ¬s.pers }    ▷ Delete non-persistent states in current layer
 9: end procedure

10: procedure ExploreState(s)

11:     for (s, t, s′) ∈ Δ do

12:         if s′ ∉ S then
13:             S ← S ∪ {s′}
14:             s′.pers ← ψ(s′) ⊏ ψ(s)

15:             if ψ(s′) ⊏ ψ(s) then                      ▷ Regress edge discovered
16:                 𝓡 ← 𝓡 ∪ {s′}                   ▷ s′ will be root for a subsequent sweep
17:             else
18:                 Q.Enqueue(s′)
19:             end if

20:             if ψ(s) = ψ(s′) then                    ▷ new state within current layer
21:                 𝓛 ← 𝓛 ∪ { s′ }
22:             end if
23:         end if
24:     end for
25: end procedure
```

hence must contain at least one inter-layer cycle. Item 3 follows from the fact that when the progress measure is monotonic, regress edges cannot exist. Hence it is impossible to have inter-layer cycles since states belonging to two different layers cannot be mutually reachable without the existence of a regress edge. From this it also follows that it is impossible to have inter-layer SCCs with monotonic progress measures.

□

4 Stubborn Set State Space Exploration

The stubborn set method is based on computing in each state s encountered during state space exploration, a set of transitions $\mathsf{stub}(s) \subseteq T$ called the stubborn set. The actual

computation of stubborn sets [16] is based on a structural (static) analysis of enabling and disabling dependencies between transitions for the model under consideration. The method then only explores the enabled transitions in the stubborn set. This results in the exploration of a reduced state space which is a subset of the full state space. Hence, the very basic integration with the sweep-line algorithm is to replace the loop over all enabled transitions in l. 11-24 of Alg. 2 with a loop over the enabled transitions in the stubborn set of s. In the following we use *stubborn set transitions* to refer to transitions contained in the stubborn set and *outside transitions* to refer to transitions that are not stubborn set transitions. In its basic form, the stubborn sets computed must imply the following semantic requirements reformulated from [16] in each encountered state s.

Condition D0 (At least one enabled stubborn set transition, if it exists)
If $\mathsf{enab}(s) \neq \emptyset$, then $\mathsf{stub}(s) \cap \mathsf{enab}(s) \neq \emptyset$.

Condition D1 (Outside transitions cannot enable stubborn set transitions)
If $t \in \mathsf{stub}(s)$, $t_1, t_2, \ldots, t_n \notin \mathsf{stub}(s)$, $s = s_0 - t_1 t_2 \cdots t_n \rightarrow s_n$ and $s_n - t \rightarrow s'_n$ then there is an s'_0 such that $s_0 - t \rightarrow s'_0$ and $s'_0 - t_1 t_2 \cdots t_n \rightarrow s'_n$.

Condition D2 (Outside transitions cannot disable stubborn set transitions)
If $t \in \mathsf{stub}(s) \cap \mathsf{enab}(s)$, $t_1, t_2, \ldots, t_n \notin \mathsf{stub}(s)$ and $s = s_0 - t_1 t_2 \cdots t_n \rightarrow s_n$, then $t \in \mathsf{enab}(s_n)$.

Stubborn sets satisfying conditions **D0**, **D1**, and **D2** preserve in the reduced state space all terminal states and the existence of an infinite execution [16]. In order to preserve additional properties in the reduced state space, further dynamic requirements must be enforced on the stubborn sets used. Specifically, for a property in LTL_{-X}, the following conditions reformulated from [16] must be fulfilled by the stubborn sets used in states s encountered during exploration of the reduced state space:

Condition S (no enabled transition is ignored) If $t \in \mathsf{enab}(s)$, then there is an execution $s = s_0 - t_1 t_2 \cdots t_n \rightarrow s_n$ such that $t \in stub(s_n)$ and $t_i \in \mathsf{stub}(s_{i-1})$ for $1 \leq i \leq n$.

Condition V (no enabled visible transitions or all visible transitions)
If $t \in \mathsf{stub}(s) \cap \mathsf{enab}(s)$ and t is visible, then all visible transitions t' are in $\mathsf{stub}(s)$.

Condition L1 (at least one enabled invisible stubborn set transition, if it exists)
If there exists a transition $t \in \mathsf{enab}(s)$ and t is invisible, then there is at least one invisible transition $t' \in \mathsf{stub}(s) \cap \mathsf{enab}(s)$.

Condition L2 (cycles considers all visible transitions) If $s = s_0 - t_1 t_2 \cdots t_n \rightarrow s_n = s$ is a cycle of the stubborn set reduced state space, then for each visible transition t there is some state s_i on the cycle such that $t \in \mathsf{stub}(s_i)$.

Condition **S** can be implemented by ensuring that no transitions enabled in the states of terminal SCCs are ignored [15]. This requirement is compatible with the sweep-line method when using a monotonic progress measure since any SCC is confined within a single layer cf. Thm. 1(item 3). Hence, the states of the SCCs are in memory at the same time and SCCs can be computed layer by layer. It is an open problem how to enforce condition **S** when using non-monotonic progress measures. The reason is that non-monotonic progress measures do not guarantee that states belonging to an SCC of the (reduced) state space are confined within a layer. This is for instance the case with the terminal SCC D with states 21-26 in Fig. 1. It implies that the states of the SCC,

including terminal SCCs may not be in memory at the same time.

The conditions **V** and **L1** do not induce particular complications for the sweep-line method as these are local properties that can be checked directly from the current state being processed. Condition **L2** can be checked using nested-depth first search (or similar) within each layer in the case of monotonic progress measures since states on a cycle are confined within a single layer and hence in memory at the same time by Thm. 1 (item 3). In this paper we address the open problem of how to enforce **L2** in the case of non-monotonic progress measures where cycles may span multiple layers such as the cycle 5, 17, 18, 19, 6, 7, 4, 5 in Fig. 1. Table 1 summarises the presentation above in terms of where there are open problems to be explored in the combination of partial-order reduction and the sweep-line method. For the sweep-line method, the cases of monotonic and non-monotonic progress measures are distinguished.

Table 1. Open problems related to implementation of partial-order conditions for LTL_{-X} when combined with sweep-line state space exploration.

Stubborn Set	Progress Measure	
Condition	Monotonic	Non-monotonic
S	✓ (single layer property)	Open
V	✓ (single state property)	✓ (single state property)
L1	✓ (single state property)	✓ (single state property)
L2	✓ (single layer property)	Open

5 Enforcing LTL_{-X} Conditions

Preservation of LTL_{-X} requires that conditions **D0**, **D1**, **D2**, **S**, **V**, **L1**, and **L2** are enforced in the stubborn set reduced state space explored with the sweep-line method. It follows from the previous section as summarised in Table 1 that the two central research questions concern how to enforce and implement conditions **S** and **L2** for the generalised sweep-line method, i.e., in the presence of non-monotonic progress measures. We address these questions in the following two subsections.

5.1 Enforcing Condition S

Condition **S** has traditionally been enforced and implemented by on-the-fly computation of the SCCs during state space exploration, and ensuring that no transition is ignored in any of the terminal SCCs. It is sufficient to consider terminal SCCs since from any reachable state it is possible to reach the states contained in at least one terminal SCC.

If a transition is detected as being ignored in a terminal SCC, then the stubborn set in one of the states in the SCC is augmented with an ignored transition, the stubborn set recomputed, and the exploration of the reduced state space is resumed, including the on-the-fly computation of the SCCs. This approach can be carried forward in the case of monotonic progress measures as SCCs are confined within a single layer (Thm. 1, item 3) and as a consequence in memory at the same time. In the case of non-monotonic progress measures, some terminal SCCs may span multiple layers. SCC D in Fig. 1 illustrates this.

Our approach to resolving ignoring with non-monotonic progress measures is based on computing SCCs within each layer as they are being explored combined with additional checks. If we encounter a SCC in which no state has outgoing edges to states in other layers or to another SCC in the same layer, then this is intra-layer SCC is also a terminal SCC. We therefore check whether any transition is ignored in the SCC. If there are ignored transitions, then we augment the stubborn set in one of the states of the SCC with an ignored transition, recompute the stubborn set, and resume the state space exploration, including the computations of the SCCs. If we encounter a locally computed SCC with outgoing edges to other layers such as the local SCC comprised of states 4,5, 6, and 7 in Fig. 1, then we cannot at that stage know whether the states contained in this SCC are part of an inter-layer terminal SCC. The key observation is, however, that an inter-layer SCC must contain at least one regress edge and hence a persistent state (Thm. 1, item 2). This means that if the states of the SCC are part of an inter-layer terminal SCC, then a persistent state of the SCC will eventually be encountered again, and we can then ensure that no transition is ignored in such a state. Hence, the idea is to use persistent states (destination states of regress edges) as *anchor states* for ensuring that no transition is ignored in a terminal SCC. Clearly, this becomes a safe over-approximation as not all persistent states may belong to an inter-layer terminal SCC.

An *eager* approach to resolving ignoring in inter-layer (terminal) SCCs is therefore to include all enabled transitions in the stubborn sets of persistent states the first time that they are discovered via a regress edge. That would, however, potentially be wasteful as we may not know at the time that it becomes persistent (l. 14 of Alg. 2), whether the persistent state is part of an inter-layer SCCs. The persistent state 13 in Fig. 1 for instance is not part of an inter-layer terminal SCC and it may be sufficient in terms of the other stubborn set conditions to only explore one of its enabled transitions. It is, however, possible to apply a *lazy* approach that relaxes the eager resolving of ignoring by exploiting the following proposition.

Proposition 1. *Let (S, T, Δ, ι) be the state space explored by the sweep-line algorithm using $\psi : S \mapsto O$ as a progress measure. Let be $s_0 - t_1 \rightarrow s_1 - t_2 \rightarrow \cdots s_{i-1} - t_{i-1} \rightarrow s_i - t_i \rightarrow \cdots s_{n-1} - t_n \rightarrow s_n = s_0$ be an inter-layer cycle of the explored state space. There exists a persistent state s_i on the cycle that has been encountered once via a regress edge and after that at least once again either via a regress edge or a forward edge.*

Proof. The existence of a persistent state s_i follows from Thm. 1 (item 2). Since s_i has been made persistent it must have been encountered via a regress edge and used as root in a subsequent sweep iteration. Since s_i is on a cycle it is reachable from itself. Hence if we do not encounter s_i again in the sweep iteration that uses s_i as root (or in a subsequent

sweep iteration), it must be because the cycle contains another state s_j that was made persistent in an earlier sweep iteration. This means that s_j (which is also on the cycle) has been encountered twice: once when it was made persistent and now again. Hence, we can choose s_j as the persistent state on the cycle that is encountered twice. □

An inter-layer SCC will contain at least one inter-layer cycle and hence we can postpone the resolving of ignoring in a persistent state until the second time the persistent state is encountered. Formulated differently, it is only persistent states that are encountered twice that we need to fully expand to guarantee that ignoring is resolved. For the state space in Fig. 1 this implies that state 6 (explored twice via a regress and a forward edge), state 23 (explored twice via a regress edge) and state 28 (explored twice via a regress and a forward edge) will be fully expanded, but not state 13. The potential advantage of the lazy approach is that it may result in a smaller stubborn set reduced state space. The disadvantage is that it may cause additional re-exploration of states. It is the opposite for the eager approach.

5.2 Enforcing Condition L2

For the implementation of condition **L2** it can be observed that the exploration with the sweep-line method induces two categories of cycles. Intra-layer cycles which are confined within one layer and inter-layer cycles which span multiple layers. In the case of intra-layer cycles, all states of the cycle will be present in memory simultaneously and conventional methods, e.g. based on depth-first search can be used to ensure that condition **L2** is enforced. In the case of inter-layer cycles, then each such cycle contains at least one persistent state as per Thm. 1(item 2). Hence, we can use a similar approach as in the case of condition **S** on inter-layer SCCs using persistent states as anchor states for enforcing **L2**. We may again consider a lazy and eager approach, where the lazy approach exploits that at least one of the persistent states on the cycle will be encountered a second time after becoming persistent.

6 Stubborn Set Sweep-line Algorithm for LTL_{-X}

We now present our new algorithm for enforcing conditions **S**, **V**, **L1**, and **L2** when combining sweep-line state space exploration with partial-order reduction in the form of stubborn sets. As conditions **V** and **L1** can be enforced based upon the state for which a stubborn set is being computed, we denote by $\mathtt{stub}_{\{V,L1\}} : S \mapsto T$ a function which provides for a state s, a stubborn set $\mathtt{stub}_{\{V,L1\}}(s)$ satisfying **V** and **L1** (in addition to **D0**, **D1**, and **D2**). By $\mathtt{stub}^{E}_{\{V,L1\}} : S \mapsto T$ we denote a function which returns the enabled transitions contained in the stubborn set provided by $\mathtt{stub}_{\{V,L1\}}$. We assume similar functions $\mathtt{stub}_{\{S,V,L\}}$ and $\mathtt{stub}^{E}_{\{S,V,L\}}$ for stubborn sets satisfying **S**, **V**, **L1**, and **L2** by including all enabled transitions (for condition S) and visible transitions (for condition L2) in the stubborn set. We also assume that the anchor attribute of a state is set to 0 when this state is created.

Algorithm 3 shows the adapted SWEEP and SWEEPITERATION procedures. The priority queue Q in l. 4 controlling the exploration order now contains pairs of states and

enabled transitions to be explored. This is done so that we can dynamically augment the enabled transitions to be explored in a state when possibly augmenting the stubborn sets in order to enforce conditions **V** and **L1**. Priority is still based on the progress measure of the state. The main difference is in ll. 10–16 where we now ensure that if a state has been marked as an anchor state by having been discovered a second time via a regress edge, then we ensure that the stubborn set in such a state enforces conditions **S** and **L2**. There are no changes to the SWEEPITERATION procedure.

Algorithm 4 provides the EXPLORELAYER procedure. As we now need to compute SCCs and cycles in the current layer, we store in $\mathcal{L}$ all edges in the current layer. ll. 7–13 now make sure that when we are about to move into the next layer, we perform a check to ensure that conditions **S** and **L2** are enforced by checking the local terminal SCCs and the local cycle of the layer. If needed, we augment the stubborn set used in the current layer and add additional state transition pairs to be processed. Any new state and transition pairs added will effectively postpone moving into the next layer until we again are about to leave the current layer at which point the same checks will be performed again by ll. 7–13. We assume that procedure TERMINALLOCALSCCS in l. 8 computes the terminal local SCCs in the current layer, and that ENFORCECONDITIONS_S given a terminal SCC computes stubborn sets in the states of the SCC required to ensure condition **S** and return the corresponding pairs of states and enabled transitions. We use $stub_S^E$ to collect the additional state and enabled transitions pairs that may have to be added to ensure that the stubborn sets satisfies condition **S**. Similarly, we assume that procedure LOCALCYCLE in l. 10 computes the local cycles of the current layer, and that ENFORCECONDITION_L2 given a cycle computes stubborn sets in the states of the cycle required to ensure condition **L2** and returns the corresponding pairs of states and enabled transitions. We use $stub_{L2}^E$ to collect any additional states and enabled transitions pairs required by the stubborn sets to satisfy condition **L2**. We provide details on how these can be implemented in practice in the next section.

Algorithm 5 specifies the EXPLORETRANSITION procedure which is a transition-oriented variant of the EXPLORESTATE procedure in Alg. 2. The algorithm implements the lazy approach to ensure conditions **S** and **L2**. The first time we discover a persistent state s' via a regress edge, we therefore set the *anchor* attribute of s' to 1 in ll. 8–11.

If we discover a persistent state for the second time via a regress edge ll. 16–19, then we mark it as an anchor state and add it to the set of roots for the iteration such that it will be expanded in ll. 11–12 of Alg. 3 such that conditions **S**, **L1** and **L2** are enforced. In case of discovering a persistent state the second time via a forward edge (ll. 20–23), we insert the s' and the enabled transitions in the stubborn set into the priority queue for processing as they need to be handled in the current sweep iteration.

The correctness of our new algorithm in terms of preserving LTL_{-X} properties is stated in the theorem below. It should be noted that the theorem only concerns preservation of the LTL_{-X} properties and not how the property can be checked. To model check an LTL_{-X} property based upon the reduced state space explored by our new algorithm, one would have to employ for instance the MAP-based algorithm from [5] to perform the actual checking of the property under a sweep-line state space exploration.

Theorem 2. *Let $\mathcal{S} = (S, T, \Delta, \iota)$ be a full state space and let $\psi : S \mapsto O$ be a progress measure. The sweep-line algorithm in specified in Algorithms 3-5 terminates after hav-*

Algorithm 3 The stubborn set sweep-line SWEEP and SWEEPITERATION procedures

```
 1: Set of states 𝒮                          ▷ States currently stored in memory
 2: Set of states ℛ ⊆ 𝒮        ▷ States that will serve as roots in next sweep iteration
 3: Progress measure ψ
 4: Priority queue 𝒬 ⊆ 𝒮 × T   ▷ States to be processed, priority based on progress value

 5: procedure SWEEP()
 6:     ι.pers ← false
 7:     ℛ ← {ι}
 8:     𝒮 ← {ι}

 9:     while ℛ ≠ ∅ do

10:         for s ∈ ℛ do

11:             if s.anchor = 2 then
12:                 stub^E ← stub^E_{S,V,L}(s)   ▷ Expand anchor discovered twice via regress edge
13:             else
14:                 stub^E ← stub^E_{V,L1}(s)
15:             end if

16:             𝒬.ENQUEUE({ (s,t) | t ∈ stub^E })

17:         end for

18:         ℛ ← ∅
19:         SWEEPITERATION()

20:     end while

21: end procedure

22: procedure SWEEPITERATION()

23:     while 𝒬 ≠ ∅ do                ▷ Explore as long as there are unprocessed states
24:         ψ_l ← 𝒬.MINPROGRESS()            ▷ Progress value for the layer to be explored
25:         EXPLORELAYER(ψ_l)
26:     end while

27: end procedure
```

ing explored a reduced state space $\mathcal{S}' = (S', T', \Delta', \iota)$ *with* $S' \subseteq S$, $T' \subseteq T$, $\Delta' \subseteq \Delta$, *and an* LTL_{-X} *property* ϕ *holds in* $\mathcal{S}$ *if and only if* ϕ *holds in* $\mathcal{S}'$.

Proof. We assume termination of the procedures TERMINALLOCALSCCS, ENFORCE-CONDITION_S, LOCALCYCLES and ENFORCECONDITION_L2, and the computation of stubborn sets by the functions $\mathsf{stub}_{\{V,L1\}}$ and $\mathsf{stub}_{\{S,V,L\}}$ for which we have not provided a detailed implementation. Termination of the algorithm then follows from termi-

Algorithm 4 The stubborn set sweep-line ExploreLayer procedure

1: States and edges $\mathcal{L} \subseteq S \times T \times S$ ▷ current layer
2: **procedure** ExploreLayer(ψ_l)
3: $\mathcal{L} \leftarrow \emptyset$
4: **while** $\mathcal{Q}$.minProgress() $= \psi_l$ **do** ▷ Explore as long as we are in the current layer
5: $(s, t) \leftarrow \mathcal{Q}$.dequeue()
6: ExploreTransition(s, t)
7: **if** $\psi_l \sqsubset \mathcal{Q}$.minProgress() $\vee\ \mathcal{Q}$.empty() **then** ▷ About to complete current layer
8: $tscc \leftarrow$ TerminalLocalSccs($\mathcal{L}$) ▷ Check S on Terminal SCCs
9: $stub^E_S \leftarrow$ EnforceCondition_S($tccs$)
10: $cycs \leftarrow$ LocalCycles($\mathcal{L}$) ▷ Check L2 on cycles
11: $stub^E_{L2} \leftarrow$ EnforceCondition_L2($cycs$)
12: $\mathcal{Q}$.enqueue($stub^E_S \cup stub^E_{L2}$) ▷ Explore additional transitions
13: **end if**
14: **end while**
15: $S \leftarrow S \setminus \{ s \in S \mid (s, t, s') \in \mathcal{L} \wedge \neg s.pers \}$ ▷ Delete non-persistent states in layer
16: **end procedure**

nation of the generalised sweep-line algorithm (cf. Thm. 1(1)) and the fact that a persistent state can only be added as an anchor state into roots in ll. 16–19 or be expanded via ll. 20–23 of Alg. 5 once. A reduced state space is explored since the enabled transitions in the stubborn set in a given state is always a subset of the enabled transition in the state. For preservation of LTL_{-X} we need to argue that conditions **D0-D2**, **S**, **V**, **L1**, and **L2** are satisfied by the reduced state space. The computation of stubborn sets in l. 12 and l. 14 of Alg. 3, and l. 13 and ll. 21–22 of Alg. 5 ensures that conditions **D0-D2**, **V**, and **L1** are guaranteed. Condition **S** is satisfied for intra-layer terminal SCCs via ll. 8–9 in Alg. 4, and for inter-layer terminal SCCs via detection of anchor states in ll. 16–19 of Alg. 5 and the computation of stubborn set for anchor states in l. 12 of Alg. 3. Condition **L2** is satisfied for intra-layer cycles via ll. 10–11 in Alg. 4, and for inter-layer cycles via detection of anchor states in ll. 16–19 of Alg. 5 and the computation of stubborn set for anchor states in l. 12 of Alg. 3.

□

Algorithm 5 The stubborn set sweep-line EXPLORETRANSITION procedure

```
 1: procedure ExploreTransition(s, t)
 2:     s' ← succ(s, t)
 3:     if ψ(s) = ψ(s') then                                          ▷ edge within this layer
 4:         L ← L ∪ { (s, t, s') }
 5:     end if
 6:     if s' ∉ S then
 7:         S ← S ∪ {s'}
 8:         if ψ(s') ⊏ ψ(s) then                                   ▷ Regress edge discovered
 9:             s'.pers ← true                                           ▷ Make s' persistent
10:             s'.anchor ← 1           ▷ Potential anchor s' discovered once via a regress edge
11:             R ← R ∪ {s'}                           ▷ s' will be root for a subsequent sweep
12:         else
13:             Q.Enqueue({ (s', t) | t ∈ stub^E_{V,L1}(s') })
14:         end if
15:     else
16:         if ψ(s') ⊏ ψ(s) ∧ s'.pers ∧ s'.anchor = 1 ∧ s' ∉ R then
17:             s'.anchor ← 2             ▷ Persistent state s' discovered twice via regress edges
18:             R ← R ∪ {s'}              ▷ s' anchor state will be root for a subsequent sweep
19:         end if
20:         if ψ(s) ⊑ ψ(s') ∧ s'.pers ∧ s'.anchor = 1 then
21:             stub^E ← stub^E_{S,V,L}(s)   ▷ Persistent state s' discovered twice via forward edges
22:             Q.Enqueue({ (s, t) | t ∈ stub^E })
23:         end if
24:     end if
25: end procedure
```

The variants of Alg. 3-5 implementing the suggested eager approach can be obtained by eliminating the use of the anchor attribute on states, and removing the if-then-else statement in ll. 11–15 of Alg. 3 and keep only the statement in the if-branch such that persistent states are always expanded according to conditions **S**,**V**, **L1**, and **L2**.

7 Implementation and Initial Evaluation

We report in this section on results from preliminary experiments undertaken with the Helena verification tool [6] where we have implemented our new algorithm.

Input models and progress measures. We experimented with 14 Petri net instances from the Model Checking Contest [9] database. We focused on instances modeling real life protocols (e.g., coming from the industry) or algorithms (e.g., mutual exclusion algorithms). Progress measures were automatically generated by the Lola tool [17]. These are based on transition invariants following the method described in [14].

Partial order reduction implementation. We focused on the evaluation of the **L2** condition (required for the verification of LTL_{-X} properties). We did not check actual properties, but implemented our partial-order reduction algorithms in such a way that, in the reduced state space, it is guaranteed that at least one fully expanded state lies on each cycle – hence enforcing conditions **D0**, **D1**, **D2** and **L2**. Such a reduction mechanism is denoted by $\mathsf{POR}^{\mathsf{L2}}$ hereafter. For comparison purposes (i.e., in order to evaluate the cost of cycle detection and transition ignoring prevention) we also evaluated a reduction mechanism, denoted by POR, that only preserves conditions **D0**, **D1**, **D2** (i.e., conditions that can be checked on individual states) for terminal states detection.

It should be noted that guaranteeing that any cycle in the reduced graph contains a fully expanded state also trivially ensures that the **S** condition (i.e., any enabled transition is explored is some future) is enforced. Therefore, our implementation could also be used for the verification of safety properties, although it is clearly not optimal in that context as it could perform undesired full state expansions. Moreover, having each cycle contain a fully expanded state also implicitly assumes all transitions to be visible. Therefore our experiments evaluate our algorithm independently of any particular property and as such can be considered as a worst-case evaluation from the perspective of the **S** condition and the **V** condition.

Our implementation of the algorithm described in this paper combines the priority-first search induced by the sweep-line algorithm (SL) with local depth-first searches (DFS) within the same layer to detect intra-layer cycles. To implement $\mathsf{POR}^{\mathsf{L2}}$, we used a method described in [4] for intra-layer cycle detection — more precisely, the Colored-Dest algorithm based on the full expansion of destination states of cycle-closing edges.

Moreover, we experimented with two implementations of the algorithm described in this paper. In the first one, $\mathsf{POR}_{\mathsf{e}}^{\mathsf{L2}}$, a persistent state is systematicaly fully expanded (the eager approach), while the second, $\mathsf{POR}_{\mathsf{l}}^{\mathsf{L2}}$, makes use of the anchor attribute to fully expand persistent states at their second visit only (the lazy approach).

Overall, this means that we have 6 different algorithms implemented in our evaluation: depth-first exploration (DFS) equipped with POR or $\mathsf{POR}^{\mathsf{L2}}$ reduction; and sweep-line exploration without partial-order reduction or equipped with POR, $\mathsf{POR}_{\mathsf{e}}^{\mathsf{L2}}$ (eager approach) or $\mathsf{POR}_{\mathsf{l}}^{\mathsf{L2}}$ (lazy approach) reduction.

Experimental results. Table 2 summarises our results. For the two runs using DFS, the table gives the number of explored states which coincides with the number of stored states. For each of the four runs using the sweep-line exploration, two numbers are provided: the peak number of stored states and the number of explored states. *OOT* means that the execution used more than the 30 minutes we used as a timeout.

We first observe that the two methods (partial-order and sweep-line) combine nicely. A comparison of algorithms DFS + POR and Sweep-Line + POR reveals that the sweep-

Instance	DFS		Sweep-line exploration							
	POR	POR^{L2}	No POR		POR		POR_e^{L2}		POR_l^{L2}	
	explored	explored	peak	explored	peak	explored	peak	explored	peak	explored
aslink(1,a)	999,852	1,023,574	OOT		37,482	2,041,736	40,366	2,216,812	43,877	3,350,000
anderson(4)	12,621	12,621	7,541	72,652	2,355	33,402	2,355	33,568	2,399	46,188
deploy(3,a)	50,593	72,608	127,322	288,471	17,866	75,446	33,261	157,658	31,466	185,449
des(2,a)	71,666	81,238	OOT		2,803	128,601	3,221	183,868	4,275	253,184
discovery(6,a)	68	72,298	1,445,467	2,740,646	22	106	27,902	286,490	22,848	225,564
eisenmcguire(4)	304,178	305,183	998,480	3,048,866	69,836	547,990	92,022	782,193	88,223	1,086,742
firewire(7)	1,662,462	1,715,905	7,185,255	12,559,256	690,547	2,674,127	885,644	3,068,237	837,410	3,946,358
gpufp(4,b)	3,432	4,312	1,422,350	23,026,207	353	5,234	511	8,538	532	13,382
peterson(3)	109,360	133,311	1,724,805	7,852,439	20,567	313,428	32,892	467,491	33,086	636,291
shield(t,iip,1,b)	1,784	1,997	1,423,708	10,136,661	294	5,342	859	18,250	633	12,438
stigcomm(2,b)	377,605	2,781,703	OOT		14,003	548,850	671,234	5,924,879	641,220	7,323,018
stigelec(3,b)	52,513	407,183	2,988,578	51,968,497	4,087	118,857	84,959	1,030,966	84,679	1,374,618
szymanski(2)	28,779	59,203	39,263	162,383	12,113	73,758	21,477	160,284	21,420	254,722
tcp(5)	635,357	806,500	1,478,519	6,930,857	133,580	1,430,837	164,019	1,493,184	157,576	2,734,743

Table 2. Experimental results for 14 Petri net instances.

line method could further reduce the peak number of stored states produced by partial-order reduction by a factor of 9.88 on the average of the 14 model instances while increasing the number of explored states by a factor of 2.06 on the average.

In order to have a more precise understanding of the cost of transition ignoring prevention required for the verification of LTL_{-X} properties, we give in Table 3 a comparison of reduction POR with respect to POR^{L2} for DFS, and with respect to POR_e^{L2} and POR_l^{L2} for the sweep-line algorithm. Each number is obtained by dividing the number of states (from Table 2) of a run with POR^{L2}, POR_e^{L2} or POR_l^{L2} by the corresponding number with the POR setting.

Generally speaking, we see that the cost of transition ignoring prevention with the sweep-line method is comparable to this cost when DFS (which is the standard algorithm when checking LTL_{-X} properties) is used. However, we see three instances for which this assertion does not hold: stigcomm(2,b), stigelec(3,b) and, to a lesser extent, shield(t,iip,1,b). It would be interesting to study the structure of these nets and try to determine what could cause this behaviour. Nevertheless, the trend in these results is generally that although not as efficient as in depth-first search (that revealed to be especially efficient as reported in [7]), the ignoring prevention mechanism proposed in this paper performs quite well and does not cancel the benefits of combining these two methods. A comparison of algorithms DFS + POR^{L2} and Sweep-Line + POR_e^{L2} reveals that, even when transition ignoring is prevented, the sweep-line method can further reduce the memory consumption (i.e., peak number of stored states) produced by partial-order reduction by a factor 6.95 on the average at the cost of an increase of the number of explored states by a factor 2.95 on the average (compared respectively to 9.88 and 2.06 when ignoring is not taken care of).

Instance	DFS	Sweep-Line			
	POR^{L2} vs POR	POR_e^{L2} vs POR		POR_l^{L2} vs POR	
	explored	peak	explored	peak	explored
aslink(1,a)	× 1.024	× 1.077	× 1.086	× 1.171	× 1.641
anderson(4)	× 1.000	× 1.000	× 1.005	× 1.019	× 1.383
deploy(3,a)	× 1.435	× 1.862	× 2.090	× 1.761	× 2.458
des(2,a)	× 1.134	× 1.149	× 1.430	× 1.525	× 1.969
discovery(6,a)	× 1063	× 1268	× 2703	× 1039	× 2128
eisenmcguire(4)	× 1.003	× 1.318	× 1.427	× 1.263	× 1.983
firewire(7)	× 1.032	× 1.283	× 1.147	× 1.213	× 1.476
gpufp(4,b)	× 1.256	× 1.448	× 1.631	× 1.507	× 2.557
peterson(3)	× 1.219	× 1.599	× 1.492	× 1.609	× 2.030
shield(t,iip,1,b)	× 1.119	× 2.922	× 3.416	× 2.153	× 2.328
stigcomm(2,b)	× 7.367	× 47.94	× 10.80	× 45.79	× 13.34
stigelec(3,b)	× 7.754	× 20.79	× 8.674	× 20.72	× 11.57
szymanski(2)	× 2.057	× 1.773	× 2.173	× 1.768	× 3.453
tcp(5)	× 1.269	× 1.228	× 1.044	× 1.180	× 1.911

Table 3. Measuring the cost of transition ignoring prevention

We conclude this section by some observations regarding the benefits of the lazy approach (i.e., relying on the use of the anchor attribute). A comparison of columns POR_e^{L2} and POR_l^{L2} in Table 2 shows that this attribute does unfortunately not bring a significant improvement. It can even, for some instances (e.g., aslink(1,a)), increase the memory consumption while, in general, it tends to increase to number of explored states as it was expected (see Sec 5.1). We conjecture that this is due to the design of the progress measures we experimented with. The principle of the lazy approache is to avoid the full expansion of persistent states that are not part of a cycle. Since our progress measures are based on transition invariants it is likely that a destination state of a regress edge indeed lies on a cycle meaning that it must anyway be fully expanded (which is done by the eager approach, i.e., POR_e^{L2}, at the first visit). It is therefore worthwhile investigating our algorithm with other progress measures or with models specified in other formalisms that do not have structural analysis tools such as invariants computation.

8 Conclusions and Future Work

We have developed a variant of the sweep-line method such that it can be combined with the LTL_{-X} preserving stubborn set method. The core idea was the local check of terminal SCCs and cycles with each layer combined with using persistent states as anchor states for ensuring the conditions for terminal SCCs and cycles that span multiple layers. We have considered partial-order reduction in stubborn set framework, but our results apply also to other partial-order frameworks such as ample sets [12]. Our algorithm does not address how LTL_{-X} would be model checked during the combined stubborn set and sweep-line state space exploration. To perform model checking, one

would have to run a sweep-line compatible LTL model checking algorithm such as the MAP-based algorithm of [5] on top of our proposed algorithm.

A direct perspective is to perform a more thorough experimentation of our algorithm. First, although we have seen that our implementation could be used for the verification of safety properties (as it ensures both the **L2** and **S** conditions), it is not optimal in that context. We thus plan to experiment with a less conservative approach specifically tailored for the **S** condition. Second, we have seen that our transition ignoring mechanism is not suited for some model instances. We plan to further investigate these to identify where this inadequacy comes from. Along these lines, we plan to evaluate our algorithm with other progress measures or modelling languages to identify whether our conjecture regarding the anchor attribute is valid and identify the benefit of using this attribute in that context.

We have not considered the use of persistent predicates in our approach. This is an addition to the sweep-line method where a predicate is used to mark states as persistent the first time they are discovered. A persistence predicate can be used to reduce the re-exploration of states. To use persistent predicates in the context of our algorithm, we would have to make it an anchor state the first time that a persistent state is rediscovered via a regress edge. It can also be observed that some cycles in the state space may contain multiple persistent states and that it is not required to fully expand all in order to enforce conditions **S** and **L2**. As part of future work, it may be investigated whether the concept of maximal persistent predecessor inspired from [5] can be used to address this aspect.

The proposed LTL_{-X} stubborn set sweep-line method has addressed an open problem within the sweep-line state space exploration framework. While we are now able to perform LTL model checking, it remains an open problem how to perform full CTL and CTL_{-X} with the sweep-line method. The challenge here is that the CTL model checking would have to be done following the least-progress first exploration order of the sweep-line method while current algorithms for explicit-state CTL model checking are based on backwards traversal. Some initial algorithms for a fragment of CTL were investigated in [13], but they do not cover full CTL model checking.

References

1. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
2. Brim, L., Cerná, I., Moravec, P., Simsa, J.: Accepting predecessors are better than back edges in distributed LTL model-checking. In: Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD. Lecture Notes in Computer Science, vol. 3312, pp. 352–366. Springer (2004). `https://doi.org/10.1007/978-3-540-30494-4_25`, `https://doi.org/10.1007/978-3-540-30494-4_25`
3. Christensen, S., Kristensen, L.M., Mailund, T.: A sweep-line method for state space exploration. In: Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS. Lecture Notes in Computer Science, vol. 2031, pp. 450–464. Springer (2001). `https://doi.org/10.1007/3-540-45319-9_31`, `https://doi.org/10.1007/3-540-45319-9_31`
4. Duret-Lutz, A., Kordon, F., Poitrenaud, D., Renault, E.: Heuristics for checking liveness properties with partial order reductions. In: Artho, C., Legay, A., Peled, D. (eds.) Automated Technology for Verification and Analysis - 14th International Symposium, ATVA

2016, Chiba, Japan, October 17-20, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9938, pp. 340–356 (2016). https://doi.org/10.1007/978-3-319-46520-3_22, https://doi.org/10.1007/978-3-319-46520-3_22

5. Evangelista, S., Kristensen, L.M.: A sweep-line method for büchi automata-based model checking. Fundam. Informaticae **131**(1), 27–53 (2014). https://doi.org/10.3233/FI-2014-1003, https://doi.org/10.3233/FI-2014-1003
6. Evangelista, S.: High level petri nets analysis with helena. In: Ciardo, G., Darondeau, P. (eds.) Applications and Theory of Petri Nets 2005, 26th International Conference, ICATPN 2005, Miami, USA, June 20-25, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3536, pp. 455–464. Springer (2005). https://doi.org/10.1007/11494744_26, https://doi.org/10.1007/11494744_26
7. Evangelista, S.: Experimenting with stubborn sets on petri nets. In: Gomes, L., Lorenz, R. (eds.) Application and Theory of Petri Nets and Concurrency - 44th International Conference, PETRI NETS 2023, Lisbon, Portugal, June 25-30, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13929, pp. 346–365. Springer (2023). https://doi.org/10.1007/978-3-031-33620-1_19, https://doi.org/10.1007/978-3-031-33620-1_19
8. Holzmann, G.J., Peled, D.A., Yannakakis, M.: On nested depth first search. In: The Spin Verification System, Proceedings of a DIMACS Workshop. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 32, pp. 23–31. DIMACS/AMS (1996). https://doi.org/10.1090/DIMACS/032/03, https://doi.org/10.1090/dimacs/032/03
9. Kordon, F., Hulin-Hubard, F., Jezequel, L., Paviot-Adet, E., Nivon, Q., , Amat., N., Berthomieu, B., Dal Zilio, S., , Ding, Z., He, Y., Li, S., Jiang, C., Jensen, P., Srba, J., Thierry-Mieg, Y.: Complete Results for the 2025 Edition of the Model Checking Contest. https://mcc.lip6.fr/2025/results.php (June 2025)
10. Kristensen, L.M., Mailund, T.: A generalised sweep-line method for safety properties. In: FME 2002: Formal Methods - Getting IT Right, International Symposium of Formal Methods Europe. Lecture Notes in Computer Science, vol. 2391, pp. 549–567. Springer (2002). https://doi.org/10.1007/3-540-45614-7_31, https://doi.org/10.1007/3-540-45614-7_31
11. Peled, D.A.: Ten years of partial order reduction. In: Computer Aided Verification, 10th International Conference, CAV. Lecture Notes in Computer Science, vol. 1427, pp. 17–28. Springer (1998). https://doi.org/10.1007/BFB0028727, https://doi.org/10.1007/BFb0028727
12. Peled, D.A.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings. Lecture Notes in Computer Science, vol. 697, pp. 409–423. Springer (1993). https://doi.org/10.1007/3-540-56922-7_34, https://doi.org/10.1007/3-540-56922-7_34
13. Rodríguez, A., Kristensen, L.M., Rutle, A.: Verification of the MQTT iot protocol using property-specific CTL sweep-line algorithms. Trans. Petri Nets Other Model. Concurr. **15**, 165–183 (2021). https://doi.org/10.1007/978-3-662-63079-2_8, https://doi.org/10.1007/978-3-662-63079-2_8
14. Schmidt, K.: Automated generation of a progress measure for the sweep-line method. Int. J. Softw. Tools Technol. Transf. **8**(3), 195–203 (2006). https://doi.org/10.1007/S10009-005-0201-1, https://doi.org/10.1007/s10009-005-0201-1
15. Valmari, A.: Stubborn sets for reduced state space generation. In: Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets]. Lecture Notes in Computer Science, vol. 483, pp. 491–515. Springer (1989). https://doi.org/10.1007/3-540-53863-1_36, https://doi.org/10.1007/3-540-53863-1_36

16. Valmari, A.: The state explosion problem. In: Lectures on Petri Nets I: Basic Models, Advances in Petri Nets. Lecture Notes in Computer Science, vol. 1491, pp. 429–528. Springer (1996). https://doi.org/10.1007/3-540-65306-6_21, https://doi.org/10.1007/3-540-65306-6_21
17. Wolf, K.: Petri net model checking with lola 2. In: Khomenko, V., Roux, O.H. (eds.) Application and Theory of Petri Nets and Concurrency - 39th International Conference, PETRI NETS 2018, Bratislava, Slovakia, June 24-29, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10877, pp. 351–362. Springer (2018). https://doi.org/10.1007/978-3-319-91268-4_18, https://doi.org/10.1007/978-3-319-91268-4_18