# Hybrid On-the-Fly LTL Model Checking with the Sweep-Line Method*

Sami Evangelista[1] and Lars Michael Kristensen[2]

[1] LIPN — Laboratoire d'Informatique de l'Université Paris Nord
99, av. J-B Clément, 93430 Villetaneuse, France
sami.evangelista@lipn.univ-paris13.fr
[2] Department of Computer Engineering, Bergen University College, Norway
Lars.Michael.Kristensen@hib.no

**Abstract.** The sweep-line method allows explicit state model checkers to delete states from memory on-the-fly during state space exploration thereby lowering the memory demands of the verification procedure. The sweep-line method is based on a least progress-first search order that prohibits the immediate use of standard on-the-fly LTL model checking algorithms that rely on a depth-first search order. This paper proposes and experimentally evaluates an algorithm for LTL model checking compatible with the search order prescribed by the sweep-line method.

## 1 Introduction

A main paradigm in explicit state model checking is to limit memory requirements by storing only a subset of the visited states in memory at a time. This means that the peak memory usage is reduced. The subsets of the state space stored in memory during state space exploration are chosen in such a way that termination of the exploration is still guaranteed. State caching [14,17] was one of the first methods based on this paradigm and relies on storing only the states on the depth-first search stack in memory. The *sweep-line method* [5,18] and the *to-store-or-not-to-store method* [2] represent more recent methods based on the paradigm of on-the-fly state deletion, and use other conditions for determining the subsets of states that are to be stored in memory.

The basic idea of the sweep-line method is to exploit a notion of *progress* exhibited by many systems. Exploiting progress makes it possible to explore all reachable states while storing only small subsets of the state space in memory at a time. The subsets of states stored are determined via a progress value assigned to each state, and the method explores the states in a progress-first order. The sweep-line method explore all states with a given progress value before progressing to the states with a higher progress value. When the method proceeds to the consider states with a higher progress value, it deletes the subset of states with a lower progress value. The assumption is that the system does not make

* Supported by a Norwegian Research Council (NRC) Yggdrasil grant and NRC project 194521 (FORMGRID).

S. Haddad and L. Pomello (Eds.): PETRI NETS 2012, LNCS 7347, pp. 248–267, 2012.

regress, and hence states with a lower progress value will not be visited again and do not need to be kept in memory. If the system does make regress, then the method will mark states at the end of *regress edges* as persistent (i.e., make them permanently stored in memory) in order to ensure termination. In presence of regress, the sweep-line method may visit some states multiple times. The sweep-line method is in its simplest form [5,18] aimed at on-the-fly verification of *safety properties*, such as determining whether a reachable state exists that satisfies a given state predicate. The theoretical foundation of the sweep-line method has been further developed in several papers [3,11,18,20,19] and the method has been implemented in the ASAP platform [25] and in the LoLA tool [22]. The sweep-line method has been used [12,13,15,23] for the verification of several industrial-sized protocols specified using the CPN modelling language.

An open research question that has not been addressed in the earlier papers on the sweep-line method is how to combine the sweep-line method with on-the-fly model checking of Linear Time Temporal Logic (LTL) properties. The conventional approach to on-the-fly LTL model checking is based on the exploration of a product Büchi automaton: the negation of the LTL formula to be checked is represented as a Büchi automata [24] and the product of this property automaton and the state space (viewed as a Büchi automata) is explored using a nested depth-first traversal [7] in search for an *acceptance cycle*, i.e., a cycle containing an acceptance state. The challenge in the context of the sweep-line method is that the nested depth-first search of LTL model checking is incompatible with the progress-based search order of the sweep-line method as the latter cannot guarantee that states in an acceptance cycle will be present in memory simultaneously. The basic idea of the hybrid approach developed in this paper is to use nested depth-first search to detect acceptance cycles where the states on the cycle all have the same progress value, and use a variation of the MAP algorithm [1,4] to detect acceptance cycles that span multiple progress values.

The rest of this paper is organised as follows. Section 2 introduces the basic concepts underlying LTL model checking and the sweep-line state space exploration algorithm. Section 3 then presents our algorithm for conducting LTL model checking with the sweep-line method. The correctness of this algorithm is proved in Sect. 4 along with its complexity. In Sect. 5, we discuss some possible extensions and variations of our algorithm. Section 6 presents the results from the experimental evaluation that we have performed based on an implementation of the proposed algorithm. Finally, in Sect. 7 we sum up our conclusions and discuss directions for future work. The reader is assumed to be familiar with the basic idea of explicit state space exploration.

## 2 Background

### 2.1 LTL Model Checking

LTL model checking is usually performed following the automata-based approach originating from [24] that proceeds in two steps, the first being the translation of the negation of the LTL formula to be checked into a Büchi automata.

In this paper we focus only on the second step of the process that can be reduced to a graph problem [6]: given a graph representing the synchronised product of the Büchi property automaton and the state space of the system, find a cycle containing an accepting state. A state of the synchronised product is an acceptance state if the Büchi property automaton component of the state is an acceptance state. Any such identified cycle determines an infinite execution of the system violating the LTL formula. Acceptance cycles can be detected using nested depth-first search [7] or a variation of Tarjan's algorithm for strongly connected component (SCC) detection [8]. Hence, we will only reason on *automaton graphs* that result from the product of a Büchi property automaton and a state space graph describing the behaviour of the modelled system.

**Definition 1.** *An* ***automaton graph*** $G$ *is a 4-tuple* $(\mathcal{S}, \mathcal{E}, s_0, \mathcal{A})$ *where* $\mathcal{S}$ *is a finite and non-empty set of states;* $\mathcal{E} \subseteq \mathcal{S} \times \mathcal{S}$ *is a finite set of edges;* $s_0 \in \mathcal{S}$ *is an initial state; and* $\mathcal{A} \subseteq \mathcal{S}$ *is a set of accepting states.*

**Notation.** For an automaton graph $G = (\mathcal{S}, \mathcal{E}, s_0, \mathcal{A})$ we write $s \to s'$ if $(s, s') \in \mathcal{E}$; and $s \to^* s'$ if there exists $s_1, \ldots, s_n$ with $s_1 = s$, $s_n = s'$ and $s_i \to s_{i+1}$ for $1 \leq i \leq n-1$. Acceptance states are graphically represented using double circles.

### 2.2 The Sweep-Line Method

The sweep-line method [5,18] deletes states on-the-fly by exploiting a particular notion of progress in the system. Progress is formally captured by a *progress measure* that quantifies the progression of a state:

**Definition 2.** *Let* $G = (\mathcal{S}, \mathcal{E}, s_0, \mathcal{A})$ *be an automaton graph. A* ***progress measure*** *(or progress mapping) for* $G$ *is a mapping from* $\mathcal{S}$ *to* $\Phi$*, where* $\Phi$ *is a non-empty set of progress values equipped with a total order* $\sqsubseteq$*.*

A progress mapping determines a partition of edges into *progress edges* corresponding to steps that increase the progress value (i.e., edges $(s, s')$ with $\phi(s) \sqsubset \phi(s')$); *stationary edges* connecting two states with a same progress value; and *regress edges* corresponding to steps that decreases the progress value (i.e., edges $(s, s')$ with $\phi(s') \sqsubset \phi(s)$). Algorithm 1 is the generalised sweep-line algorithm [18] that maintains four data structures:

$\mathcal{H}$ is a hash table used to store states currently in memory;
$\mathcal{G} \subseteq \mathcal{H}$ contains states that will be garbage collected (deleted) when possible;
$\mathcal{R} \subseteq \mathcal{H}$ contains states that will serve as roots during the next sweep (i.e., an iteration of the algorithm at ll. 4–10); and
$\mathcal{Q} \subseteq \mathcal{H}$ contains states that have not yet been processed by the algorithm.

A sweep works basically as a standard state space exploration initiated from a set of root states $\mathcal{R}$ initialised, for the first sweep, with the initial state $s_0$. States are expanded and their successors that have not been visited so far are put in $\mathcal{Q}$ to be later expanded. This process ends when $\mathcal{Q}$ becomes empty.

**Algorithm 1.** Sweep, the generalised sweep-line algorithm of [18]

```
1  algorithm Sweep is
2      s0.pers := false ; R := {s0} ; H := {s0}
3      while R ≠ ∅ do
4          Q := R ; R := ∅
5          while Q ≠ ∅ do
6              G := ∅ ; φm := Q.minProgress ()
7              while Q.minProgress () = φm do
8                  s := Q.dequeue ()
9                  visit (s)
10             H := H \ G
11 procedure visit(s) is
12     for (s, s') ∈ E do
13         if s' ∉ H then
14             s'.pers := φ(s') ⊏ φ(s)
15             H := H ∪ {s'}
16             if s'.pers then
17                 R := R ∪ {s'}
18             else
19                 G := G ∪ {s'}
20                 Q := Q ∪ {s'}
```

The search progresses using a least progress-first policy determined by mapping $\phi$: the algorithm proceeds layer by layer, a *layer* being defined as a set of states sharing the same progress value, i.e., connected by stationary edges. Two differences are introduced by the sweep-line method compared to standard state space exploration. First, we perform garbage collection at l. 10 by removing a whole layer of states sharing progress value $\phi_m$ (ll. 7–9), and before processing states with a higher progress value. Any state can be deleted this way except for *persistent* states for which the detection is described below. Conceptually, there exists a *sweep-line* that separates already visited (and deleted) states from states to be processed present in $\mathcal{Q}$. This line advances to include new states after a whole layer of states with the same progress value has been processed. The only situation the sweep-line can move back is before the start of a new sweep. A second difference is that to guarantee termination, the algorithm identifies regress edges (l. 14), and marks their destination as persistent, indicating that these may not be deleted from memory. Indeed, for any cycle it holds that either all its states have the same progress value or at least two of its states are connected by a regress edge. In the first case, no state of the cycle will be garbage collected as long as at least one of its states remains in $\mathcal{H}$ and in the second case, the destination of the regress edge will always remain in $\mathcal{H}$ after having been detected. Hence, it is guaranteed that the algorithm will not visit the same states over and over since at least one state per cycle will be present in $\mathcal{H}$. Note that the destination of a regress edge is not put in $\mathcal{Q}$ but in $\mathcal{R}$ to serve as a root for the next sweep (ll. 16–17).

The snapshot of the Sweep algorithm at three different stages is presented in Fig. 1 for a simple example. Information on accepting states have not been drawn on this figure as they are not relevant to illustrate the principle of the algorithm. States are ordered left to right according to their progress value. The conceptual sweep-line is represented as a vertical dotted line. After the visit of states 0 and 1 (Stage 1) their successor states 2, 3 and 4 are put the queue. State 0 and 1 can then be deleted since they have been processed and their progress value is strictly smaller than the minimal value found in $\mathcal{Q}$, i.e., $\phi(2)$. Hence, the sweep-line method makes the assumption that they cannot be reached from the set of unprocessed states. At Stage 2, states 7 and 8 have been processed. All states from

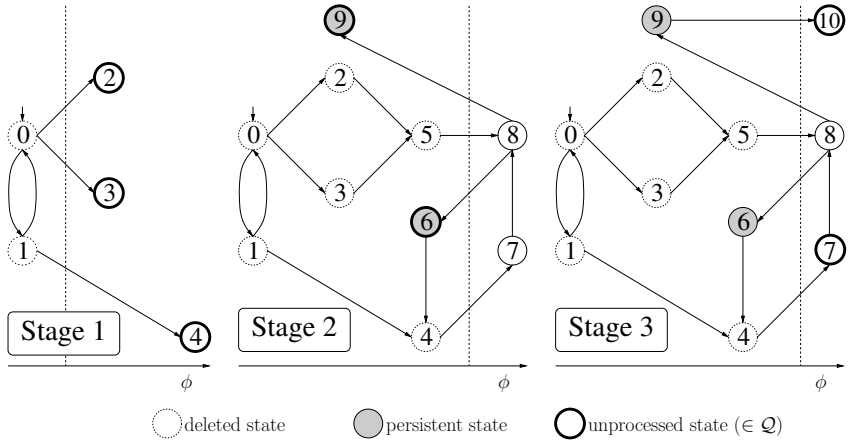



**Fig. 1.** Snapshot of algorithm Sweep at different stages

0 to 6 have been previously deleted from $\mathcal{H}$. The visit of state 8 then generated state 6, already explored but deleted from memory, and state 9, seen for the first time. Since edges (8,6) and (8,9) are both regress edges, states 6 and 9 are marked as persistent and put in set $\mathcal{R}$ to serve as roots for the next sweep. Once the expansion of states 7 and 8 is finished they are deleted and this sweep terminates. A new sweep starts with states 6 and 9 as roots. The algorithm then visits states in the following order given by $\phi$: 9, 6, 4, 10, 7, 8. After the visit of states 9, 6 and 4 (Stage 3), 9 and 6 will not be deleted since they were marked as persistent during the previous sweep. Hence, the cycle 8→6→4→7→8 is detected during the visit of state 8 that generates 6 already in $\mathcal{H}$.

## 3 A Sweep-Line Algorithm for LTL Model Checking

In this section we introduce our new LTL model checking algorithm. This algorithm consists of two distinct components each dedicated to the detection of specific kinds of accepting cycles. Before introducing these two components, we describe a property of accepting cycles.

Let us suppose that state 1 in Fig 1 is an accepting state. The accepting cycle 1→0→1 could be easily discovered by algorithm Sweep previously introduced since all its states share the same progress value and will therefore be simultaneously present in memory at some stage. If state 6 is an accepting state, then the accepting cycle 6→4→7→8→6 will not be discovered by the sweep-line method since its states are distributed upon several layers. This property of cycles is formalised through the following definition.

**Definition 3.** *Let $G = (\mathcal{S}, \mathcal{E}, s_0, \mathcal{A})$ be an automaton graph, and $\phi$ be a progress mapping for $G$. An accepting cycle $c = s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n \rightarrow s_1$ is a* ***single layer accepting cycle*** *(SLAC) if and only if $\phi(s_i) = \phi(s_j), \forall i, j \in \{1, \ldots, n\}$. Otherwise $c$ is a* ***multiple layers accepting cycle*** *(MLAC).*

Our algorithm separates the detection of SLACs from the detection of MLACs. In order to discover SLACs, the algorithm builds upon the classical NDFS algorithm while MLACs are taken care of by a combination of the sweep-line algorithm and the MAP algorithm [1,4]. Before introducing the LTL model checking algorithm, we specify the property it should have in terms of being compatible with the search order of the sweep-line method. The definition below specify that a sweep-line compliant algorithm may not keep in memory a state behind the sweep-line that would be deleted by algorithm Sweep.

**Definition 4.** *Let $G = (\mathcal{S}, \mathcal{E}, s_0, \mathcal{A})$ be an automaton graph and $\phi$ be a progress mapping for $G$. An LTL model checking algorithm storing states to process in a queue $\mathcal{Q}$ is* ***sweep-line compliant*** *if and only if, at any step $t$ of the algorithm:*

$$\forall s \in \mathcal{S}^t : (\exists s' \in \mathcal{S} \text{ with } (s', s) \in \mathcal{E}^t \wedge \phi(s) \sqsubset \phi(s')) \ \vee \ (min_{s' \in \mathcal{Q}}(\phi(s')) \sqsubseteq \phi(s))$$

*where $\mathcal{S}^t \subseteq \mathcal{S}$ denotes all states kept in memory by the algorithm at step $t$, and $\mathcal{E}^t \subseteq \mathcal{E}$ denotes all edges connecting states in $\mathcal{S}^t$.*

### 3.1 Combining NDFS and Sweep to Discover SLACs

Algorithm 2 contains the pseudo-code of procedure LTL-Sweep that is a variation of the sweep-line algorithm equipped with a mechanism that allows the detection of SLACs by performing local nested depth-first searches on layers of states sharing the same progress value. A state has three associated boolean flags, all initialised to *false* (ll. 12–14): *pers* indicating if the state is persistent and must not be garbage collected; *blue* and *red* that indicate if the state has been visited by the first level DFS (the blue DFS) or the second level DFS (the red DFS).

The principle of NDFS is to interleave a blue DFS looking for accepting states, and red DFSs looking for cycles containing accepting states reached by the blue DFS. When the blue DFS backtracks from an accepting state $s$, it initialises a red DFS rooted in $s$ (called *seed* in the first presentation of the algorithm [7]) to find whether $s$ is reachable from itself. The algorithm works in linear time since the result of a red DFS (i.e., marking visited states as red) can be reused in subsequent red DFSs. In addition to this linear complexity, NDFS also has other appreciable characteristics, among which: its low memory requirements (only 2 bits required per state, the blue and red bits), its ability to report accepting cycles on-the-fly, and its easy combination with partial-order reduction [16].

Starting from the initial state, algorithm LTL-Sweep repeatedly perform sweeps using procedure *findSLAC*, until no new persistent state is found. An iteration of procedure *findSLAC* (ll. 17–22) consists of removing from the priority queue $\mathcal{Q}$ all states with the lowest progress value and performing local NDFSs on theses states. All non-persistent states visited by these NDFSs are then present in $\mathcal{G}$ and can be removed from $\mathcal{H}$ (l. 22). If the main loop fails to find an SLAC, the *findMLAC* is invoked to look for an MLAC. Procedures *dfsBlue* and *dfsRed* follow the same principle as algorithm NDFS with the following modifications to the DFSs:

**Algorithm 2.** LTL-Sweep, a sweep-line algorithm for LTL model checking

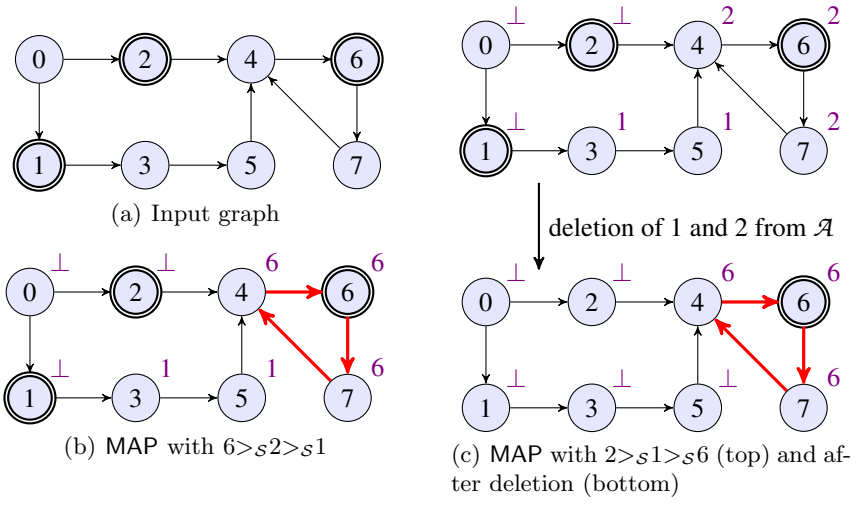
```
 1 algorithm LTL-Sweep is
 2     insert (s_0)
 3     R := {s_0}
 4     while R ≠ ∅ do
 5         Q := R
 6         R := ∅
 7         findSLAC ()
 8     findMLAC (H)
 9     report "no cycle found"
10 procedure insert(s) is
11     H := H ∪ {s}
12     s.pers := false
13     s.blue := false
14     s.red := false
15 procedure findSLAC() is
16     while Q ≠ ∅ do
17         G := ∅
18         φ_m := Q.minProgress ()
19         while Q.minProgress ()=φ_m do
20             s := Q.dequeue ()
21             dfsBlue (s)
22         H := H \ G
23 procedure dfsBlue(s) is
24     if ¬s.blue then
25         s.blue := true
26         if ¬s.pers then G := G ∪ {s}
27         for (s, s') ∈ E do
28             if s' ∉ H then
29                 insert (s')
30             if φ(s) = φ(s') then
31                 dfsBlue (s')
32             else if φ(s') ⊏ φ(s) then
33                 s'.pers := true
34                 R := R ∪ {s}
35             else
36                 Q := Q ∪ {s}
37         if s ∈ A then dfsRed (s, s)
38 procedure dfsRed(s, seed) is
39     s.red := true
40     for (s, s') ∈ E with φ(s) = φ(s') do
41         if s' = seed then
42             report "SLAC found"
43         else if ¬s'.red then
44             dfsRed (s', seed)
```

- Any visited state is put in the garbage set $\mathcal{G}$ if not persistent (l. 26).
- DFSs are limited to states sharing the same progress value (ll. 30–31, l. 40).
- Finally, two alternatives arise for any new state $s'$ reached by the first level DFS (*dfsBlue*) from a state $s$ belonging to a different layer: it is put in the root set $\mathcal{R}$ and marked as persistent if it is behind the sweep-line (ll. 32–34); or, if it is in front of the sweep-line (ll. 35–36), it is put in the priority queue $\mathcal{Q}$ to be later visited by a next NDFS during a subsequent iteration of procedure *findSLAC*.

### 3.2 Combining MAP and Sweep to Discover MLACs

Local nested DFSs are guaranteed to find any SLAC, but the algorithm relies on another procedure to find accepting cycles split upon several layers. This one is based on the principle that any MLAC always contains at least one regress edge and, hence, one persistent state. Thus, if the algorithm has failed to find an SLAC it will launch procedure *findMLAC* to possibly discover an MLAC containing a persistent state (l. 8 of Alg. 2). This procedure is invoked with the hash table $\mathcal{H}$ that contains, at l. 8 of Algorithm 2, all persistent states discovered.

The algorithm we propose to find MLACs is an adaptation of the MAP algorithm initially designed in the context of distributed memory model checking [4], and later adapted for external memory storage [1].




(a) Input graph

(b) MAP with $6 >_{\mathcal{S}} 2 >_{\mathcal{S}} 1$

(c) MAP with $2 >_{\mathcal{S}} 1 >_{\mathcal{S}} 6$ (top) and after deletion (bottom)

**Fig. 2.** Illustration of the MAP algorithm

**Principle of the MAP Algorithm.** For an automaton graph $G = (\mathcal{S}, \mathcal{E}, s_0, \mathcal{A})$, MAP assumes a total order relation $>_{\mathcal{S}} \subseteq \mathcal{S} \times \mathcal{S}$ that is used to determine a *maximal accepting predecessor* function $map_G : \mathcal{S} \to \mathcal{A} \cup \{\bot\}$. Intuitively $map_G(s)$ is the largest accepting state that is backward reachable from $s$ (or $\bot$ if there is no such state). The mapping $map_G$ can be computed using a breadth-first search (MBFS) that propagates forward information on the maximal accepting predecessor in $\mathcal{O}(|\mathcal{S}| \cdot |\mathcal{A}|)$. Trivially, $map_G(s) = s$ implies the existence of an accepting cycle looping on $s$. Unfortunately the converse does not necessarily hold (see Figure 2(c) for an example) as an accepting state $a$ that is outside a cycle containing an accepting state $b$ will prevent from discovering this cycle if $a \to^* b \wedge a >_{\mathcal{S}} b$ (which implies that $\Rightarrow map_G(b) = a$). We will then say that $b$ (or the cycle) is *hidden* by $a$. Hence, MAP alternates between MBFSs used to compute $map_G$ and delete transformations used to remove states from $\mathcal{A}$ that may hide accepting cycles. Deleted states are those that have been propagated along the graph, i.e., the set $\{s \in \mathcal{A} \mid \exists s' \in \mathcal{S}, map_G(s') = s\}$. If there is no such state to delete, then there is no accepting cycle. Otherwise, it is guaranteed that any accepting cycle will be discovered within a finite number of iterations.

The behaviour of MAP is illustrated on the graph of Fig. 2(a) with two different orders. With the first order (see Fig. 2(b)), MAP finds the accepting cycle around 6 during the first iteration (since $map_G(6) = \max_{\{1,2,6\}} = 6$). With the second order (see Fig. 2(c)), the accepting cycle around 6 is not discovered after the first computation of $map_G$ (since $map_G(6) = \max_{\{1,2,6\}} = 2$). States 1 and 2 that have been propagated during this first MBFS are both deleted from $\mathcal{A}$ before a second MBFS is initiated. Since 6 is now the only accepting state, the cycle is discovered. In the absence of edge $(6, 7)$, MAP would have stopped after this second iteration and reported the absence of accepting cycle.

**Adaptation of the MAP Algorithm for MLAC Detection.** Unlike state-of-the-art algorithms for LTL model checking, MAP is not based on a depth-first search and is as such a good candidate for a combination with the sweep-line method. However, we would like to have an algorithm that is sweep-line compliant (Def. 4) and a straightforward adaptation of the MAP algorithm would not have this property as it has to remember somehow accepting states removed from $\mathcal{A}$ by the delete transformation. We instead exploit the fact that any MLAC we are looking for contains at least one persistent state. This is a direct consequence of the fact that the cycle is distributed upon several layers and, hence, contains at least one regress edge. We therefore switch from the idea of maximal accepting predecessor to the one of maximal persistent predecessor formalised below. Note that the $mpp_G^{\mathcal{P}}$ function defined below associates a pair to each state, the second boolean component giving information on the backwards reachability of an accepting state as explained below.

**Definition 5.** *Let $G = (\mathcal{S}, \mathcal{E}, s_0, \mathcal{A})$ be a automaton graph, $\mathcal{P} \subseteq \mathcal{S}$ be a set of persistent states, and $>_{\mathcal{S}}$ be a total order relation on $\mathcal{S}$. For a state $s \in \mathcal{S}$, $R(s) = \{p \in \mathcal{P} \mid p \rightarrow^* s\}$ denotes the set of persistent states backwards reachable from $s$. The* ***maximal persistent predecessor*** *function $mpp_G^{\mathcal{P}} : \mathcal{S} \rightarrow \{\bot\} \cup (\mathcal{P} \times \{false, true\})$ is defined by:*

$$mpp_G^{\mathcal{P}}(s) = \begin{cases} (p, true) & \text{if } R(s) \neq \emptyset,\ \forall p' \in R(s) \setminus \{p\} : p >_{\mathcal{S}} p' \\ & \text{and } \exists a \in \mathcal{A} \text{ such that } p \rightarrow^* a \rightarrow^* s \\ (p, false) & \text{if } R(s) \neq \emptyset,\ \forall p' \in R(s) \setminus \{p\} : p >_{\mathcal{S}} p' \\ & \text{and } \nexists a \in \mathcal{A} \text{ such that } p \rightarrow^* a \rightarrow^* s \\ \bot & \text{otherwise} \end{cases}$$

Intuitively, if $mpp_G^{\mathcal{P}}(s) = (p, b)$, then $p$ is the largest persistent state that is backward reachable from $s$ and $b = true$ if and only if there is a path from $p$ to $s$ containing an accepting state. Hence, the following proposition is a direct consequence of Def. 5.

**Proposition 1.** *Let $G = (\mathcal{S}, \mathcal{E}, s_0, \mathcal{A})$ be an automaton graph and $\mathcal{P} \subseteq \mathcal{S}$. If $mpp_G^{\mathcal{P}}(s) = (s, true)$ then $G$ has an accepting cycle containing state $s$.*

Procedure *findMLAC* (see Algorithm 3) is an adaptation of the MAP algorithm for MLACs detection. Each iteration of the algorithm (ll. 4–7) consists of computing the $mpp_G^{\mathcal{P}}$ function; and then removing set $\mathcal{D}$ from $\mathcal{P}$. This set $\mathcal{D}$ contains states that have been propagated during the computation of $mpp_G^{\mathcal{P}}$ and that may hide some accepting cycle(s). It is initialised to $\mathcal{P}$ before each computation even though procedure *mpp* will discard from it hidden persistent states. In order to optimise the search, we also remove from $\mathcal{P}$ the states $s$ such that $mpp_G^{\mathcal{P}}(s) = (\_, false)$ (l. 7). It follows from Def. 5 that these cannot be part of an accepting cycle. The procedure terminates when set $\mathcal{P}$ has been emptied meaning that all persistent states have been propagated.

Procedure *mpp* is a sweep-line compliant algorithm computing the maximal persistent predecessor function with states behind the sweep-line being deleted

**Algorithm 3.** Procedure *findMLAC* to discover multiple layers accepting cycles

```
1  procedure findMLAC(H) is
2      P := H
3      while P ≠ ∅ do
4          D := P
5          mpp ()
6          P := P \ D
7          P := P \ {s | s.mpp = (_, false)}
8  procedure visit(s, prop) is
9      for (s, s') ∈ E do
10         if prop = (s', true) then
11             report "MLAC found"
12         if s' ∉ H then
13             insert (s') /* see Alg. 2 */
14             s'.mpp := ⊥
15         if prop >_mpp s'.mpp then
16             Q := Q ∪ {s'}
17             s'.mpp := prop
18 procedure mpp() is
19     for s ∈ P do
20         s.mpp := (s, s ∈ A)
21     Q := P
22     while Q ≠ ∅ do
23         G := ∅
24         φ_m := Q.minProgress ()
25         while Q.minProgress () = φ_m
           do
26             s := Q.dequeue ()
27             (p, acc) := s.mpp
28             prop := (p, acc ∨ s ∈ A)
29             if s ∈ P and p >_S s then
30                 D := D \ {s}
31             visit (s, prop)
32             if ¬s.pers then
33                 G := G ∪ {s'}
34         H := H \ G
```

at l. 34. Before visiting a state $s$, we first determine the maximal persistent predecessor *prop* it will propagate to its successors (ll. 27–28). It is $s.mpp$ with the second component set to *true* if $s$ is accepting. Moreover, if $s$ is persistent and hidden by $s.mpp$ (ll. 29–30) it must be removed from set $\mathcal{D}$ as it must not be touched by the deletion transformation operated at l. 6.

The *visit* procedure evaluates whether a maximal predecessor value *prop* computed as explained above should be propagated to the successors $s'$ of a state $s$. This decision is made according to the result of the comparison of *prop* and $s'.mpp$ using the order relation defined below.

**Definition 6.** *Let $G = (\mathcal{S}, \mathcal{E}, s_0, \mathcal{A})$ be an automaton graph and $>_{\mathcal{S}}$ be a total order relation on $\mathcal{S}$. We define the total order relation $>_{mpp}$ on $\{\bot\} \cup (\mathcal{S} \times \{false, true\})$ as follows:*

$$m >_{mpp} m' \Leftrightarrow \begin{cases} m = (s,b) \wedge m' = (s',b') \wedge (s >_{\mathcal{S}} s' \vee s = s' \wedge b \wedge \neg b') \\ \vee\ m = (s,b) \wedge m' = \bot \end{cases}$$

The definition states that propagation takes place if we have found for $s'$ a larger persistent predecessor than the previous one, or, starting from the same persistent predecessor, an alternative path containing an accepting state has now been found. State $s'$ then has to be put in the priority queue $\mathcal{Q}$ to be later visited according to the same process. Note that from Def. 6 and ll. 12–14, *prop* is always propagated if $s'$ is a new state. Finally, as stated by Prop. 1 an MLAC is found at ll. 10–11 if one reaches $s'$ with $s'.mpp = (s', true)$.

An example is illustrated by Fig. 3. At the first iteration, we have $\mathcal{P} = \{1, 2, 3\}$ (states in dark gray on the figure). We assume that the order relation on $\mathcal{S}$ is such that $3 >_{\mathcal{S}} 2 >_{\mathcal{S}} 1$. The computation of the maximal persistent

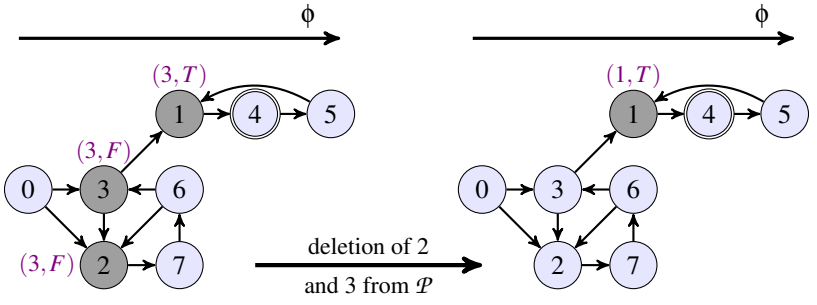



**Fig. 3.** Detection of an MLAC

predecessor function gives the following results: $mpp^{\mathcal{P}}_{\phi}(2) = mpp^{\mathcal{P}}_{\phi}(3) = (3, \mathit{false})$ and $mpp^{\mathcal{P}}_{\phi}(1) = (3, \mathit{true})$. The set $\mathcal{D}$ of states to remove from $\mathcal{P}$ is the singleton $\{3\}$ because 1 and 2 have been hidden by 3 and, hence, discarded from $\mathcal{D}$. However, we can also delete 2 from $\mathcal{P}$ since, as $mpp^{\mathcal{P}}_{\phi}(2) = (3, \mathit{false})$, no path starting from 1, 2, or 3 and leading to 3 can contain an accepting state. Hence, $\mathcal{P} = \{1\}$ after the removal, and after the second iteration, $mpp^{\mathcal{P}}_{\phi}(1) = (1, \mathit{true})$ and the MLAC 1→4→5→1 is discovered.

## 4 Correctness Proof and Complexity of LTL-Sweep

We first prove the correctness of our algorithm. The proof of Theorem 1 is inspired by the proof of the MAP algorithm [4].

**Theorem 1.** *Algorithm LTL-Sweep reports an accepting cycle if and only if the automaton graph has an accepting cycle.*

*Proof.* We prove that if the graph has an accepting cycle then a cycle is necessarily reported by the algorithm. The other direction follows immediately from the computation of $s.mpp$ and Prop. 1. Let $C = \{s_1, \ldots, s_n\}$ be an accepting cycle with $s_1 \rightarrow \ldots \rightarrow s_n \rightarrow s_1$. We consider two cases.

1. $C$ is an SLAC. Since algorithm Sweep visits all states, and $\forall s_i, s_j \in C, \phi(s_i) = \phi(s_j)$, the first NDFS initiated on one of the states $s_i \in C$ will obviously report the cycle.
2. $C$ is an MLAC. We denote by $\mathcal{P}_i$ the content of set $\mathcal{P}$ of states used as roots during the $i^{\text{th}}$ call to procedure *mpp*. Let $s^{\max}$ be the largest persistent state of the cycle (i.e. $s^{\max} \in C \cap \mathcal{P}_0$) such that $\forall s_i \in (C \cap \mathcal{P}_0) \setminus \{s^{\max}\}, s^{\max} >_{\mathcal{S}} s_i$. This state exists since otherwise, we would have $\mathcal{P}_0 = \emptyset$ which would in turn mean that $\forall s_i, s_j \in C, \phi(s_i) = \phi(s_j)$ and, hence, that $C$ is an SLAC. If the cycle $C$ is not reported by the $i^{\text{th}}$ call to procedure *mpp* then it necessarily holds that once the procedure has terminated, $s^{\max}.mpp = (m, \mathit{true})$ with $m >_{\mathcal{S}} s^{\max}$ which implies that $s^{\max} \notin \mathcal{D}$ and $m \in \mathcal{D}$. Now, once *mpp* has

terminated, if $\exists s \in \mathcal{S} \mid s.mpp = (s', b)$ with $s' \neq s$ then $s' \in \mathcal{D}$ and $s \notin \mathcal{D}$. This, combined with the fact that $s^{\max}.mpp = (m, true)$, implies that $s^{\max}$ is not touched by the deletion transformation (ll. 6–7 of Alg. 3) and therefore belongs to $\mathcal{P}_{i+1}$ while $m$ does not. Since $\mathcal{P}$ is finite, $C$ (or another MLAC) is necessarily reported by a $j^{\text{th}}$ (with $j > i$) call to *mpp*. □

**Theorem 2.** *Algorithm LTL-Sweep terminates after having explored at most* $2 \cdot |\mathcal{P}| \cdot |\mathcal{S}| + 2 \cdot |\mathcal{P}|^3 \cdot |\mathcal{S}|$ *states where* $\mathcal{P}$ *denotes the set of persistent states computed by Algorithm Sweep.*

*Proof.* Algorithm Sweep explores at most $|\mathcal{P}| \cdot |\mathcal{S}|$ states [18]. Therefore the same algorithm combined with NDFS to detect SLACs explores at most $2 \cdot |\mathcal{P}| \cdot |\mathcal{S}|$ states. Procedure *mpp* of Alg. 3 visits each state $s \in \mathcal{P}$ at most $2 \cdot |\mathcal{P}|$ times: for any $s' \in \mathcal{P}$ with $s' >_{\mathcal{S}} s$, it can be visited a first time with $s.mpp = (s', false)$ and a second time with $s.mpp = (s', true)$. Each visit by procedure *mpp* of $s \in \mathcal{P}$ generates at most $|\mathcal{S}|$ visits. Hence, procedure *mpp* terminates after visiting at most $2 \cdot |\mathcal{P}|^2 \cdot |\mathcal{S}|$. Therefore, since procedure *findMLAC* performs at most $|\mathcal{P}|$ iterations and calls to *mpp*, it explores at most $2 \cdot |\mathcal{P}|^3 \cdot |\mathcal{S}|$ states. □

## 5 Extensions

We propose in this section extensions to the algorithm we introduced in the previous section. The first extension has been implemented in our tool and the experimental section discusses its benefits. The second and third extensions are opportunities for future research directions and have not been implemented yet.

### 5.1 On-the-Fly MLAC Detection

The algorithm we introduced can detect SLACs on-the-fly, i.e., without the need of exploring the entire graph. Indeed, each time a sweep will encounter a layer containing an SLAC, the use of NDFS guarantees an early termination of the algorithm. However, an MLAC will be discovered only when no SLAC has been discovered and, hence, after a complete visit of the automaton graph. One could however prioritise the discovery of MLACs by interleaving both searches. The modification we propose is to launch procedure *findMLAC* each time a sweep of the first level algorithm (i.e., procedure *findSLAC* of Algorithm 2) looking for SLACs has finished. The search of *findMLAC* is then initiated from states that were used as roots by the first level sweep and is bound to persistent states that have already been visited (i.e., not discovered by the last sweep performed). Let us denote by $\mathcal{R}_i$ the set of root states during the $i^{\text{th}}$ sweep of the first level algorithm. After sweep $i$ has terminated, procedure *findMLAC* will be launched to look for an MLAC including at least one persistent state of $\mathcal{R}_i$ and possibly some states of $\cup_{j \in \{0..i-1\}} \mathcal{R}_j$. Procedure *findMLAC* of Alg. 3 has to be modified in such a way that, after sweep $i$ has terminated, $\mathcal{P}$ is initialised at l. 2 with $\mathcal{R}_i$. With this modification, it is guaranteed that an MLAC containing some persistent states $p_1 \in \mathcal{R}_{\hat{p}_1}, \ldots, p_n \in \mathcal{R}_{\hat{p}_n}$ will be reported once the $m^{\text{th}}$ sweep

has finished where $m = \max_{\{\hat{p}_1,\ldots,\hat{p}_n\}}$. In the following, we refer to LTL-Sweep$^{\text{off}}$ as the algorithm where *findMLAC* is followed by *findMLAC* and LTL-Sweep$^{\text{on}}$ as the version that interleaves the discovery of the two types of cycles.

### 5.2 Informed Search for MLACs

For now, there is no interaction between the two search procedures whereas procedure *findMLAC* could benefit from the experience of the previous search for SLACs. For example, in case the automaton graph does not have any accepting state, the search for MLACs could be avoided by just noticing this information during the previous step. More generally, we propose to build, as the search progresses, the progress graph as defined below.

**Definition 7.** *Let $G = (\mathcal{S}, \mathcal{E}, s_0, \mathcal{A})$ be an automaton graph with a progress mapping $\phi : \mathcal{S} \to \Phi$. The* progress graph *of $G$ with $\phi$ is an automaton graph $G^\phi = (\mathcal{S}^\phi, \mathcal{E}^\phi, s_0^\phi, \mathcal{A}^\phi)$ defined by:*

- $\mathcal{S}^\phi = \{s_\alpha \mid \exists s \in \mathcal{S}$ *with* $\phi(s) = \alpha\}$ ;
- $(s_\alpha, s_\beta) \in \mathcal{E}^\phi \Leftrightarrow \exists (a, b) \in \mathcal{E}$ *with* $\phi(a) = \alpha$ *and* $\phi(b) = \beta$ ;
- $s_0^\phi = s_{\phi(s_0)}$ ; *and*
- $s_\alpha \in \mathcal{A} \Leftrightarrow \exists s \in \mathcal{A}$ *with* $\phi(s) = \alpha$.

This graph provides information on the connectivity between progression layers of the state space graph and can be used to prune the search for MLACs. We can for instance avoid the visit of any state $s$ such that, in the progress graph, $s_{\phi(s)}$ is not in a strongly connected component with an accepting state. It is a direct consequence of Def. 7 that this state can not be part of an accepting cycle.

### 5.3 Using the Progress Measure to Order States

Another direction we would like to pursue is to study whether the progress mapping could be useful in the definition of the order relation used to calculate the maximal persistent predecessor function. We arbitrarily chose in the current implementation to order states according to the bit vectors they are encoded in before insertion in the hash table. The sweep-line method works well for systems for which it is possible to derive a progress measure clustering the state space into multiple layers with few regress edges. On the basis of this assumption, we would like to experiment with an order relation that considers the progress value of states. Let $s_1$ and $s_2$ be two states such that $\phi(s_1) \sqsubset \phi(s_2)$ with $s_2$ being part of an accepting cycle. If the progress measure has the desired properties, then it is more likely that $s_1 \to^* s_2$ than $s_2 \to^* s_1$. In this situation it would then make sense that $s_2 >_{\mathcal{S}} s_1$ so that if it is indeed the case that $s_1 \to^* s_2$ and if both states are used as roots during the computation of the maximal persistent predecessor function, then state $s_1$ would not hide $s_2$ and the accepting cycle containing that state. It would then not be necessary to perform an iteration of the algorithm in order to delete $s_1$ to be able at the next iteration to detect the

accepting cycle. For instance, with the graph of Fig. 3, this heuristic would imply to order states in such a way that $1 >_{\mathcal{S}} 2$ and $1 >_{\mathcal{S}} 3$ (since $\phi(2) \sqsubset \phi(1)$ and $\phi(3) \sqsubset \phi(1)$). We would then have $mpp(1) = (1, true)$ after the first iteration and the accepting cycle $1 \rightarrow 4 \rightarrow 5 \rightarrow 1$ would be detected without the need of deleting 2 and 3 from $\mathcal{P}$ and reiterating the search.

## 6 Experiments

We have implemented our algorithm in its two variants LTL-Sweep$^{\text{off}}$ and LTL-Sweep$^{\text{on}}$ on the ASAP verification platform [25], and experimented with it using DVE models from the BEEM database [21]. The 85 instances we selected have a number of states ranging from 100,000 to 10,000,000 states. Out of these 85 instances, 49 had an accepting cycles and the 36 remaining ones did not. We compared our algorithm to two LTL model checking algorithms: the classical NDFS algorithm [7] and the MAP algorithm [4]. We also compared it to the sweep-line algorithm Sweep from [18] designed for checking safety properties. As the full graph must be explored in the absence of an accepting cycle, the performance of Sweep served, in that context, as a baseline to assess the performance of LTL-Sweep: the latter cannot visit (or store) fewer states than Sweep. We used automatically generated progress measures for each model according to the generation process described in [9]. Each measure projects a state vector to some of its components (e.g., local variables, program counters) chosen after a preliminary exploration of the system. For the sake of clarity, we have selected a set of representative instances from our experiments with respect to several parameters (complexity of the model, size of the graph, and performance). However, for completeness, the reader may find all our experimental data in [10].

Our experimental data are reported in Table 1. We have separated instances for which the property analysed holds (top part of the table) from those containing an accepting cycle (bottom part). The first column provides information on each graph we analysed: the instance name, the number (in the BEEM database) of the analysed property and the number of states (st.) and edges (ed.) in the automaton graph[1]. Each entry in the table provides data for a single run, i.e., a triple (instance, property, algorithm): the peak number of states stored (first row) and the number of states visited (second row). For sweep-line based algorithms an entry also reports the number of persistent states once the search has terminated (third row). All these numbers are expressed as fractions of the number of states of the automaton graph. Finally, for instances exhibiting an accepting cycle, a small letter to the left of stored states indicates, for our algorithm, the type of cycle detected (S for an SLAC and M for an MLAC).

Our interpretation of the data first deals with graphs without an accepting cycle. We then discuss the models with falsified properties. Throughout this section all our comments dealing with LTL-Sweep apply to both versions of the algorithm: LTL-Sweep$^{\text{off}}$ and LTL-Sweep$^{\text{on}}$.

[1] We will not detail the models and their properties in this article but we invite the reader to consult this database online at `http://anna.fi.muni.cz/models/`.

**Table 1.** Experimental data

| | NDFS | MAP | Sweep | LTL-Sweep$^{on}$ | | LTL-Sweep$^{off}$ | |
|---|---|---|---|---|---|---|---|
| Verified properties | | | | | | | |
| bopdp.3, prop. 4 | 1.000 | 1.000 | 0.074 | 0.074 | | 0.106 | |
| 1,703,192 st. | 1.000 | 1.000 | 2.021 | 15.739 | | 8.349 | |
| 4,619,673 ed. | – | – | 0.009 | 0.009 | | 0.009 | |
| leader_filters.5, prop. 2 | 1.000 | 1.000 | 0.086 | 0.086 | | 0.086 | |
| 1,572,886 st. | 2.000 | 14.462 | 1.000 | 2.000 | | 2.000 | |
| 4,319,565 ed. | – | – | 0.000 | 0.000 | | 0.000 | |
| lifts.6, prop. 2 | 1.000 | 1.000 | 0.012 | 0.012 | | 0.012 | |
| 998,570 st. | 1.332 | 16.564 | 1.552 | 4.076 | | 3.387 | |
| 2,864,768 ed. | – | – | 0.006 | 0.006 | | 0.006 | |
| lup.3, prop. 2 | 1.000 | 1.000 | 0.330 | 0.336 | | 0.336 | |
| 2,346,373 st. | 1.170 | 10.954 | 3.909 | 50.486 | | 8.511 | |
| 4,965,501 ed. | – | – | 0.111 | 0.111 | | 0.111 | |
| peterson.4, prop. 4 | 1.000 | 1.000 | 0.184 | 0.205 | | 0.224 | |
| 2,239,039 st. | 1.500 | 11.546 | 5.322 | 103.883 | | 25.443 | |
| 11,449,204 ed. | – | – | 0.046 | 0.046 | | 0.046 | |
| pgm_protocol.8, prop. 4 | 1.000 | 1.000 | 0.043 | 0.045 | | 0.143 | |
| 3,069,399 st. | 1.000 | 5.000 | 1.127 | 2.995 | | 5.029 | |
| 7,125,130 ed. | – | – | 0.025 | 0.025 | | 0.025 | |
| rether.6, prop. 2 | 1.000 | 1.000 | 0.069 | 0.121 | | 0.175 | |
| 6,046,531 st. | 1.001 | 2.000 | 1.463 | 45.663 | | 9.681 | |
| 7,980,886 ed. | – | – | 0.045 | 0.045 | | 0.045 | |
| Falsified properties | | | | | | | |
| extinction.4, prop. 2 | 0.408 | 1.000 | 0.068 | S | 0.022 | S | 0.022 |
| 2,001,372 st. | 0.817 | 2.585 | 1.000 | | 0.115 | | 0.115 |
| 6,856,693 ed. | – | – | 0.000 | | 0.000 | | 0.000 |
| iprotocol.4, prop. 4 | 0.006 | 0.614 | 0.109 | M | 0.032 | M | 0.123 |
| 8,214,324 st. | 0.006 | 1.077 | 1.803 | | 0.938 | | 7.565 |
| 30,357,177 ed. | – | – | 0.108 | | 0.029 | | 0.108 |
| mcs.6, prop. 4 | 0.140 | 1.000 | 0.339 | S | 0.045 | S | 0.045 |
| 665,007 st. | 0.279 | 8.289 | 3.414 | | 0.183 | | 0.183 |
| 3,283,155 ed. | – | – | 0.003 | | 0.000 | | 0.000 |
| plc.2, prop. 3 | 0.004 | 0.005 | 0.013 | M | 0.000 | S | 0.001 |
| 130,220 st. | 0.004 | 0.006 | 1.051 | | 0.009 | | 0.053 |
| 210,710 ed. | – | – | 0.013 | | 0.000 | | 0.000 |
| rether.3, prop. 6 | 0.001 | 0.129 | 0.255 | M | 0.056 | M | 0.476 |
| 607,382 st. | 0.001 | 0.193 | 1.612 | | 0.413 | | 15.660 |
| 991,098 ed. | – | – | 0.040 | | 0.007 | | 0.040 |
| synapse.1, prop. 3 | 0.059 | 0.103 | 0.393 | S | 0.219 | S | 0.219 |
| 159,888 st. | 0.059 | 0.081 | 2.099 | | 1.050 | | 0.270 |
| 721,531 ed. | – | – | 0.183 | | 0.033 | | 0.033 |

**Instances without Accepting Cycle.** On the criterion of explored states, MAP and LTL-Sweep, are in general incomparable although we found that, on the average, MAP have more stable performance. However, this observation is not surprising if we recall that the time complexity of MAP is $\mathcal{O}(|\mathcal{A}|^2 \cdot |\mathcal{S}|)$ while our algorithm works in $\mathcal{O}(|\mathcal{P}|^3 \cdot |\mathcal{S}|)$. The relative performance of both algorithms then depends on the number of accepting states and the quality of the progress measure that has an impact on the number of persistent states. Still, even in the presence of very few persistent states (e.g., instances bopdp.3, rether.6) our algorithm can explore a large number of states: it also depends on how the deletion transformation (ll. 6–7 of Alg. 3) succeeds in removing states from the set of persistent states $\mathcal{P}$ that procedure *findMLAC* will search for cycles on.

If we compare the two variants of our algorithm on the same criterion (explored states) we observe that LTL-Sweep$^{\text{off}}$ is generally faster than LTL-Sweep$^{\text{on}}$. With algorithm LTL-Sweep$^{\text{off}}$, procedure *findMLAC* is invoked only once with all persistent states discovered by the algorithm whereas with algorithm LTL-Sweep$^{\text{on}}$, the procedure is invoked with $\mathcal{R}_0, \mathcal{R}_1, \ldots$ ($\mathcal{R}_i$ being the set of root states during the $i^{\text{th}}$ sweep of the top level algorithm looking for SLACs). Hence, the delete transformation is usually more successful in the off-line variant as it can potentially remove more states from $\mathcal{P}$ and perform fewer computations of the maximal persistent predecessor function. Stated in a different manner, it is better to perform a single invocation of *findMLAC* on a set $S$ rather than partitioning $S$ and then performing several invocations on each class of this partition. Averaged over all instances with no accepting cycle, LTL-Sweep$^{\text{on}}$ was 12.6 slower than NDFS while this number goes down to 5.3 with algorithm LTL-Sweep$^{\text{off}}$. If we now compare both algorithms to Sweep these ratios become 7.6 and 3.9.

From a different perspective, calling procedure *findMLAC* once with a large set naturally causes the algorithm to consume more memory (with respect to several calls with smaller sets). This is why the peak number of states stored observed with LTL-Sweep$^{\text{off}}$ is generally larger than with LTL-Sweep$^{\text{on}}$ (e.g., instances bopdp.3, pgm_protocol.8 or rether.6). For the on-line variant, the difference observed in stored states between Sweep and LTL-Sweep$^{\text{on}}$ is due to the way the *mpp* procedure processes: it does not really perform sweeps as algorithm Sweep does (i.e., visiting states layer-by-layer by increasing the progress value and then starting again a sweep from some persistent states) but each time it meets a persistent state $s$ after executing a regress edge, it puts $s$ in the priority queue, and then continues the search normally. Hence, the sweep-line moves back each time a persistent state is met. We plan to implement and experiment with both versions in a future version of our prototype. The negative observations we made on our algorithm regarding visited states must, however, be related to its lower memory usage. In most cases, Table 1 shows that the number of states stored of LTL-Sweep$^{\text{on}}$ and LTL-Sweep$^{\text{off}}$ equalled or at least approached the consumption of Sweep.

In order to compare algorithms on both visited and stored states we measured for each algorithm on a specific instance a score defined as the product of stored and visited states. This score indicates to which extent state revisits

**Table 2.** Instances for which an algorithm got the best score (i.e., minimised |Visited|·|Stored|)

| | MAP | NDFS | LTL-Sweep$^{on}$ | LTL-Sweep$^{off}$ |
|---|---|---|---|---|
| 36 instances without an accepting cycle | 11.1% | 44.4% | 61.1% | 36.1% |
| 49 instances with an accepting cycle | 34.6% | 85.7% | 53.06% | 53.06% |

are compensated by memory reduction, the lower score the better. We computed for each algorithm $A$ the percentage of instances for which algorithm $A$ got the smallest score. This data can be found in Table 2. Note that the sum of these percentages exceeds 100% since several algorithms can obtain the same best score. This for example occurs if the graph does not have any accepting state. The results indicate that the run time increase of LTL-Sweep is usually acceptable in that it is counterbalanced by an effective memory usage. Moreover, it appears that our algorithm obtained bad scores mainly on instances for which the sweep-line method is anyway not adapted. These include models such as peterson.4 or lup.3. Their graphs are composed of a single connected component and do not really exhibit progress. The only exception is model rether.6. As Table 1 shows, algorithm Sweep performs quite well on that model but the performance of LTL-Sweep$^{on}$ and LTL-Sweep$^{off}$ is poor.

**Instances with Accepting Cycle(s).** On most instances, NDFS is the algorithm performing the best, reporting an accepting cycle faster than its competitors. Even if we consider the number of states stored NDFS, Table 2 shows that NDFS is the clear winner. However, we found out some instances, for which LTL-Sweep outperformed both NDFS and MAP. Two such examples are extinction.4 and mcs.6. In both cases it happened that the SLAC reported by LTL-Sweep contained states close to the initial state (from the progress measure perspective) which possibly explains why the algorithm could terminate relatively early. In contrast, using NDFS, it is likely that this cycle would be discovered later since, by proceeding depth-first, the first states the algorithm backtracks from (launching the search for accepting cycles) are deeper in the graph and usually with a higher progress value. If we compare MAP and LTL-Sweep we again observe very different performances and there is no clear winner between the two. Relying in these two algorithms on an arbitrary order relation (comparison of bit state vectors) can also explain their unpredictable performances.

A comparison of the two variants of our algorithm reveals the impact of the type of accepting cycles found in the graph. If the graph only contains SLACs (or if contains MLACs including persistent states met after an SLAC is reported), then the number of states visited by LTL-Sweep$^{off}$ is guaranteed to be fewer or equal than the one with LTL-Sweep$^{on}$. Indeed, in that case, LTL-Sweep$^{on}$ will interleave between searches for SLACs and (useless) searches for MLACs whereas

LTL-Sweep$^{off}$ will postpone the latter and discover SLACs sooner. This, for example, explains the difference observed in visited states for instance synapse.1. If the graph has both types of cycles, looking for MLACs as soon as possible can be fruitful. This explains why LTL-Sweep$^{on}$ terminated faster on instance plc.2. Algorithm LTL-Sweep$^{off}$ could report an SLAC only during the last iterations. Finally, if the graph only has MLACs (e.g., for instances iprotocol.4 and rether.3), then we observe that LTL-Sweep$^{on}$ is usually much faster showing again the benefit of searching for MLACs as soon as possible.

## 7 Conclusion and Perspectives

We have introduced in this article an LTL model checking algorithm that can be used with the on-the-fly deletion of states performed by the sweep-line method. The major difficulty of designing such a combination stems from the algorithm the sweep-line method relies on. For reachability properties, the search uses a progress-based policy whereas state-of-the-art algorithms for LTL model checking rely on a depth-first search that is best suited for cycle detection.

Our algorithm LTL-Sweep is made of two distinct building blocks each one being dedicated to a specific kind of accepting cycle. For accepting cycles containing states with the same progress value (i.e., SLACs), we simply adapt the basic Sweep algorithm to perform nested depth-first searches on layers of states. If the accepting cycle spans several layers (i.e., MLACs) we use a variation of the MAP algorithm in order to look for accepting cycles containing persistent states. The choice of MAP originates from its independence from any search order policy which makes it more easily compatible with the sweep-line method. Since the two searches are independent, we propose two versions of our algorithm. The off-line version first tries to look for SLACs and then for MLACs if the first search did not detect an acceptable cycle. The on-line variation, interleaves both searches and is thus able to report existing MLACs faster. We have implemented th algorithms in the ASAP verification platform and compared it with other LTL model checking algorithms. The conclusions we drew from these experiments are:

- LTL-Sweep uses roughly the same amount of memory as Sweep while being 4 times slower than Sweep in its off-line version and 8 times slower in its on-line version;
- When the run time increases it is usually compensated by a low memory consumption that keeps LTL-Sweep competitive with other algorithms ;
- MAP and LTL-Sweep are in general incomparable: their performance can considerably vary according to the model analysed ;
- LTL-Sweep could in general not compete with NDFS for fast accepting cycles discovery but could terminate earlier for some specific models ;
- Both the on-line and off-line versions have their pros and cons: the former can usually report accepting cycles earlier while the latter usually visits fewer states in the absence of accepting cycle. This suggests that each could be useful at different stages of the verification process.

We identified some possible rooms for improvement. First, data could be exchanged between the two procedures to optimise the search for MLACs. We propose to maintain as the search progresses, a progression graph that summarises the connections between the layers and that could be useful to prune the search for MLACs. Second, we plan to investigate to which extent the progress measure could be used to order states efficiently when looking for MLACs.

Our experiments showed that our algorithm achieves a good memory reduction if we take algorithm Sweep as a reference. This can, however, be penalised by an increase of the execution time. One direction for future research would be to design a parallel version of LTL-Sweep to address this issue. As algorithm MAP, and unlike NDFS, LTL-Sweep does not use an inherently sequential nested depth-first search this motivates this research direction. Moreover, the two components LTL-Sweep is made of are relatively independent: the search for SLACs and the search for MLACs can be performed in parallel. Several issues still have to be tackled. For instance, we have to take care that such a combination does not cancel the sweep-line reduction by letting processes explore different layers of states (while the sequential algorithm always keep a single layer in memory at a time). On the other hand, using a global "clock" to determine how processes explore the system would not necessarily yield a good time reduction.

## References

1. Barnat, J., Brim, L., Šimeček, P., Weber, M.: Revisiting Resistance Speeds Up I/O-Efficient LTL Model Checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 48–62. Springer, Heidelberg (2008)
2. Behrmann, G., Larsen, K.G., Pelánek, R.: To Store or Not to Store. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 433–445. Springer, Heidelberg (2003)
3. Billington, J., Gallasch, G., Kristensen, L.M., Mailund, T.: Exploiting Equivalence Reduction and the Sweep-Line Method for Detecting Terminal States. IEEE Transactions on SMC - Part A 34(1), 23–38 (2004)
4. Brim, L., Černá, I., Moravec, P., Šimša, J.: Accepting Predecessors Are Better than Back Edges in Distributed LTL Model-Checking. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 352–366. Springer, Heidelberg (2004)
5. Christensen, S., Kristensen, L.M., Mailund, T.: A Sweep-Line Method for State Space Exploration. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 450–464. Springer, Heidelberg (2001)
6. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. The MIT Press (1999)
7. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory Efficient Algorithms for the Verification of Temporal Properties. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 233–242. Springer, Heidelberg (1991)
8. Couvreur, J.-M.: On-the-Fly Verification of Linear Temporal Logic. In: Wing, J.M., Woodcock, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 253–271. Springer, Heidelberg (1999)
9. Evangelista, S., Kristensen, L.M.: Search-Order Independent State Caching. In: Jensen, K., Donatelli, S., Koutny, M. (eds.) Transactions on Petri Nets and Other Models of Concurrency IV. LNCS, vol. 6550, pp. 21–41. Springer, Heidelberg (2010)

10. Evangelista, S., Kristensen, L.M.: Hybrid On-the-fly LTL Model Checking with the Sweep-Line Method. Technical report, Université Paris 13 (2012), http://www-lipn.univ-paris13.fr/ evangelista/biblio-sami/doc/ sweep-ltl.pdf
11. Gallasch, G.E., Billington, J., Vanit-Anunchai, S., Kristensen, L.M.: Checking Safety Properties On-the-fly with the Sweep-Line Method. STTT 9(3-4), 371–392 (2007)
12. Gallasch, G.E., Han, B., Billington, J.: Sweep-Line Analysis of TCP Connection Management. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 156–172. Springer, Heidelberg (2005)
13. Gallasch, G.E., Ouyang, C., Billington, J., Kristensen, L.M.: Experimenting with Progress Mappings for the Sweep-Line Analysis of the Internet Open Trading Protocol. In: CPN, pp. 19–38 (2004)
14. Godefroid, P., Holzmann, G.J., Pirottin, D.: State-Space Caching Revisited. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 178–191. Springer, Heidelberg (1993)
15. Gordon, S., Kristensen, L.M., Billington, J.: Verification of a Revised WAP Wireless Transaction Protocol. In: Esparza, J., Lakos, C.A. (eds.) ICATPN 2002. LNCS, vol. 2360, pp. 182–202. Springer, Heidelberg (2002)
16. Holzmann, G., Peled, D., Yannakakis, M.: On Nested Depth First Search. In: SPIN 1996 (1996)
17. Holzmann, G.J.: Tracing Protocols. AT&T Technical J. 64(10), 2413–2434 (1985)
18. Kristensen, L.M., Mailund, T.: A Generalised Sweep-Line Method for Safety Properties. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 549–567. Springer, Heidelberg (2002)
19. Kristensen, L.M., Mailund, T.: Efficient Path Finding with the Sweep-Line Method using External Storage. In: Dong, J.S., Woodcock, J. (eds.) ICFEM 2003. LNCS, vol. 2885, pp. 319–337. Springer, Heidelberg (2003)
20. Mailund, T., Westergaard, M.: Obtaining Memory-Efficient Reachability Graph Representations Using the Sweep-Line Method. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 177–191. Springer, Heidelberg (2004)
21. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
22. Schmidt, K.: LoLA A Low Level Analyser. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 465–474. Springer, Heidelberg (2000)
23. Vanit-Anunchai, S., Billington, J., Gallasch, G.E.: Analysis of the Datagram Congestion Control Protocols Connection Management Procedures using the Sweepline Method. STTT 10(1), 29–56 (2008)
24. Vardi, M.Y., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: LICS 1986, pp. 332–344 (1986)
25. Westergaard, M., Evangelista, S., Kristensen, L.M.: ASAP: An Extensible Platform for State Space Analysis. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 303–312. Springer, Heidelberg (2009)