Combining the Sweep-Line Method with the use of an External-memory Priority Queue*

Sami Evangelista and Lars Michael Kristensen

[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 is an explicit-state model checking technique that uses a notion of progress to delete states from internal memory during state space exploration and thereby reduce peak memory usage. The sweep-line algorithm relies on the use of a priority queue where the progress value assigned to a state determines the priority of the state. In earlier implementations of the sweep-line method the progress priority queue is kept in internal memory together with the current layer of states being explored. In this paper we investigate a scheme where the current layer is stored in internal memory while the priority queue is stored in external memory. From the perspective of the sweep-line method, we show that this combination can yield a significant reduction in peak memory usage compared to a pure internal memory implementation. On an average of 60 example instances, this combination reduced peak memory usage by more than 25% at the cost of an increase execution time by a factor 2.5. From the perspective of external memory state space exploration, we demonstrate experimentally that the state deletion performed by the sweep-line method may reduce the I/O overhead induced by duplicate detection compared to a pure external memory state space exploration method.

Keywords. Algorithms and storage methods for explicit-state model checking, Engineering and implementation of software verification tools, External-memory algorithms, Sweep-line method.

1 Introduction

A large collection of explicit state space-based methods for software verification has been developed relying on various paradigms to make the approach feasible in presence of the inherent state explosion problem. Of particular relevance in the context of this paper are methods that delete states from internal memory to free storage resources during state space exploration, and methods that use external memory to increase the storage resources available.

* This work has been supported by an Yggdrasil mobility grant from the Research Council of Norway.

Deleting states from memory during state space exploration to free storage resources is the paradigm underlying state caching [13], the to-store-or-not-store method [3], and the sweep-line method [5]. 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 fragments of the state space in internal memory at a time. This means that the peak memory usage is reduced. Progress in a system can originate from, e.g., phases in a transaction protocol, sequence numbers, control flow, and retransmission counters. The foundation of the sweep-line method has been developed in several papers [5,15,10] and the method has been implemented in the ASAP verification platform [19] and the LoLA tool [17]. The sweep-line method has been applied for the software verification, in particular in the domain of protocols [11,12,14,18].

Increasing the resources available for storing the set of visited states is the paradigm underlying external memory model checking algorithms. Checking whether a newly encountered state has already been explored (i.e., performing duplicate detection) then ultimately involves costly I/O operations. Most external memory algorithms are based on the idea of *delayed duplicate detection*: duplicate detections are not interleaved with state explorations but instead grouped together to reduce I/O overhead. From an I/O perspective this replaces multiple "random" accesses by a single file scan. Breadth-first search [6] is a typical exploration algorithm that can be efficiently coupled with that strategy. Another approach to reducing the I/O overhead of duplicate detection is to use partitioning [2] and store the set of currently visited states (and unprocessed states) in a set of files (e.g., one file for each partition). A single partition is then loaded into memory at a time. When no more processing is possible for the currently loaded partition, it is moved to disk and another partition is loaded into memory for processing. Both the breadth-first and partitioning approaches will be compared to our new algorithm in this paper.

The primary contribution of this paper is the idea of combining the sweep-line method with the use of external memory, and to conduct an extensive experimental evaluation based on an implementation in the ASAP platform [19]. Our experimental results show that our approach can be viewed as both an improvement of the sweep-line method (reducing peak memory usage) and of external memory algorithms (reducing I/O overhead). A secondary contribution is the identification and experimental evaluation of an external memory priority queue [4] in the context of explicit state model checking. Our algorithm can also be viewed as a state space partitioning algorithm that uses the progress notion of the sweep-line method to partition the state space, but this partitioning is used internally in the external memory priority queue in contrast to the explicit partitioning approach [2]. We thus do not have to deal with it explicit in contrast to other external memory model checking algorithms.

Outline. In Sect. 2, we introduce the required background on the sweep-line method and external memory algorithms, and we present some initial experimental results that made us pursue the combination of the sweep-line method

with the use of an external memory priority queue. In Sect. 3 we present the sweep-line algorithm that uses external memory, the data structure used to realise the priority queue in external memory, and we give a theoretical analysis of the I/O complexity of the algorithm. Section 4 presents the results from the experimental evaluation of our algorithm. Finally, in Sect. 5, we sum up the conclusion and discuss future work. The reader is assumed to be familiar with the basic ideas of explicit state space exploration methods.

2 The Sweep-line Method and Motivation

For the presentation, we assume a universe of system states $\mathcal{S}$, an initial state $s_0 \in \mathcal{S}$, and a successor function $succ : \mathcal{S} \rightarrow 2^{\mathcal{S}}$. We want to explore the state space implied by these parameters, i.e., the triple (R, E, s_0) such that $R \subseteq \mathcal{S}$ is the set of *reachable states* and $E \subseteq R \times R$ is the set of *edges* defined by:

$$\begin{aligned} R &= \{s_0\} \cup \{\, s \in \mathcal{S} \mid \exists s_1, \ldots, s_n \in \mathcal{S} \text{ with } s = s_n \wedge \\ &\qquad \forall i \in \{0, \ldots, n-1\} : s_{i+1} \in succ(s_i)\} \\ E &= \{(s, s') \in R \times R \mid s' \in succ(s)\} \end{aligned}$$

The progress exploited by the sweep-line method is formalised by providing a *progress measure* as defined below.

Definition 1 (Progress Measure). *A* ***progress measure*** *is a tuple* $\mathcal{P} = (O, \sqsubseteq, \psi)$ *such that* O *is a set of* ***progress values****,* $\sqsubseteq$ *is a total order on* O*, and* $\psi : \mathcal{S} \rightarrow O$ *is a* ***progress mapping****.* $\mathcal{P}$ *is* ***monotonic*** *if* $\forall (s, s') \in E : \psi(s) \sqsubseteq \psi(s')$*. Otherwise,* P *is* ***non-monotonic*** □

A progress mapping implies a partition of edges upon *progress edges* marking a system step that increase the progress value (i.e., edges (s, s') with $\psi(s) \sqsubset \psi(s')$); *stationary edges* connecting states having the same progress value; and *regress edges* that decrease the progress value (i.e., edges (s, s') with $\psi(s') \sqsubset \psi(s)$).

The progress measure used by the sweep-line method can either be obtained based on a structural analysis of the model or it can be provided by the user/analyst based on knowledge about the modelled system. It is important to note that the sweep-line method can use any mapping from states to progress values. In particular, there is no proof obligation associated with a provided progress measure for the sweep-line method to work.

The operation of the sweep-line method is illustrated in Fig. 1 which depicts a generic snapshot during state space exploration. The progress mapping partitions the state space into *layers* where all states in a given layer shares the same progress values. State space exploration starts from the initial state s_0 and states are *processed* (i.e., successor states calculated) in a least-progress first-order using a *priority queue* to store states that are still to be processed. At any given moment, the state space being explored is divided into three regions: *past layers* (where all states have been processed), *current layer*, and *future layers*. The heuristic assumption underlying the sweep-line method is that the system makes

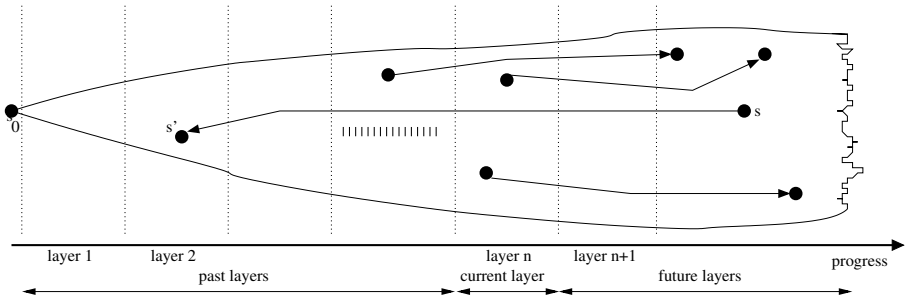

Fig. 1. Snapshot illustrating basic principle of sweep-line state space exploration.

progress which means that no states in future layers will have edges going back to states in the current or past layers. This means that once all states in the current layer n have been processed, then these can be deleted from memory, and the states in layer $n+1$ can now be processed. This heuristic assumption is indeed valid if the progress measure used in monotonic (which can be checked on-the-fly during the state space exploration). If the progress measure is non-monotonic, i.e., there exists regess edges leading from a state s to a state s' such that the progress value of s is larger than the progress value of s', then the sweep-line method will mark s' as *persistent* which means that it can never be deleted again. The sweep-line method then uses multiple explorations (called *sweeps*) of the state space where new persistent states added in the current sweep are used as roots in the subsequent sweep. In case of non-monotonic progress measures, the sweep-line method may therefore explore parts of the state space multiple times. As proved in [15], complete state space coverage and termination is guaranteed.

Peak memory usage is reduced with the sweep-line method compared to conventional state space exploration due to the fact that states in past layers are not stored in memory. The actual peak experienced with the use of the sweep-line method is influenced by the number of states in each layer, the number of states generated in future layers, and the number of persistent states. A heuristic for getting a low peak memory usage is to keep the layers small and also ensure locality so that not too many states are pushed into future layers as these also need to be stored in memory. The initial hypothesis underlying the work presented in this paper was that in many cases the individual layers contains few states, but a substantial number of the states stored in memory are states in future layers that will not be processed until much later in the state space exploration. It would therefore be potentially useful to store the states in future layers in external memory instead of internal memory.

To initially investigate this hypothesis, we report below on some statistics we collected using the ASAP [19] verification tool. We ran the sweep-line algorithm on a number of models from the BEEM database [16] and recorded for each run:

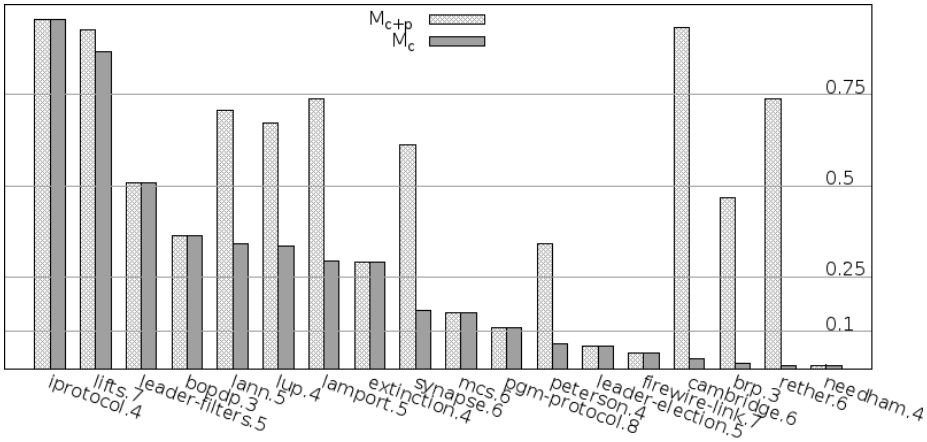

Fig. 2. Measuring memory usage of the sweep-line algorithm.

- M_{c+p+f} — The peak memory usage during the state space exploration[1].
- M_{c+p} — The peak memory usage when not counting states in future layers.
- M_c — The peak memory usage when not counting states in future layers and persistent states in past layers.

Hence, M_{c+p} is insensitive to the quantity of states in future layers while M_c is also insensitive to the quantity of persistent states found so far. Obviously it holds that $M_{c+p+f} > M_{c+p} > M_c$.

We have plotted on Fig. 2, the values M_{c+p} and M_c for a number of models. Both values are expressed relatively to M_{c+p+f} which is given a value of 1. For instance, for model pgm_protocol.8, we have $M_{c+p} \approx M_c \approx 0.11 \cdot M_{c+p+f}$. Table 1 also gives, for each of these models, the proportion of regress (Reg.), forward (Fwd.) and stationary (Sta.) edges. Figure 2 shows that there is indeed some room for improvement. For example, the state space of model needham.4 has $6,525,019$ states. The progress measure used is very successful in clustering the state space as attested by the fact that the largest layer only contains 420 states. However, the peak memory usage of the algorithm still reaches $1,339,178$ states and since the state space does not have any regress edge, this means that a huge proportion of states memorised by the algorithm are states belonging to future layers that will not be processed immediately. Moreover, the distribution of edges indicates that even in the extreme case where we only keep in memory the states of the current layer (i.e., with a peak memory usage of 420 states), only 30.5% of the edges will generate an I/O since 69.5% of edges connects states

[1] Throughout this paper, we will use as a memory usage measure the number of states present in memory at a given step rather than actual memory sizes. This is due to implementation issues that prevent us from easily measuring memory usage during a run. Moreover, this measure has the advantage of being implementation independent.

Table 1. Edge distribution for the models of Fig 2.

Model	Edges			Model	Edges		
	Reg.	Fwd.	Sta.		Reg.	Fwd.	Sta.
bopdp.3	0.0 %	27.6 %	72.4 %	leader_filters.5	0.0 %	9.1 %	90.9 %
brp.3	3.3 %	38.6 %	58.1 %	lifts.7	0.5 %	2.4 %	97.1 %
cambridge.6	3.5 %	39.8 %	56.7 %	lup.4	17.3 %	82.7 %	0.0 %
extinction.4	0.0 %	22.5 %	77.5 %	mcs.6	0.0 %	40.2 %	59.8 %
firewire_link.7	0.0 %	13.4 %	86.6 %	needham.4	0.0 %	30.5 %	69.5 %
iprotocol.4	0.0 %	2.4 %	97.6 %	peterson.4	1.8 %	46.7 %	51.5 %
lann.5	2.0 %	17.3 %	80.7 %	pgm_protocol.8	0.0 %	27.9 %	72.1 %
lamport.5	2.9 %	19.4 %	77.6 %	rether.6	5.5 %	47.4 %	47.1 %
leader_election.5	0.0 %	29.9 %	70.1 %	synapse.6	14.7 %	60.7 %	24.6 %

within the same layer and that should be simultaneously present in memory. Note that this distribution is somehow surprising since we would expect from the small layer sizes to instead have a large majority of forward edges.

The initial investigations reported above indicated that states stored in future layers can be a main determining factor in terms of peak memory usage and that storing future layers instead in external memory would be beneficial (Fig. 2). Furthermore, compared to a pure external memory state space exploration algorithm, the deletion of states in the past layers would potentially be able to reduce the I/O overhead as there would be fewer states for which duplicate detection needs to be performed (Table 1). The latter is certainly the case when the progress measure is monotonic, and in case of non-monotonic progress measure the I/O overhead may be reduced in cases where there are not too many states being re-explored.

3 Using an External Memory Priority Queue

We introduce in this section a new algorithm that combines the sweep-line method with the use of external memory. Details are also given on the external priority queue data structure we use, as this represents a central component of our algorithm. Moreover, the description of this data structure is required to have a better insight on the I/O complexity of our algorithm which is examined in the last part of this section.

3.1 Description of the Algorithm

Algorithm 1 presents a sweep-line algorithm that uses an external priority queue $\mathcal{Q}$ to store future layers and persistent states. The algorithm maintains the following data structures:

– The disk files $\mathcal{AP}$, $\mathcal{P}$, and $\mathcal{NP}$ contain, respectively, the set of all persistent states found so far; the set of persistent states discovered by the current sweep; and the set of new persistent states obtained by making the difference of the two first ones.

Algorithm 1 A sweep-line algorithm designed for external memory priority queues

```
1:  procedure externalSweep is
2:      AP := ∅
3:      Q := {s_0}
4:      while Q ≠ ∅ do
5:          sweep ()
6:          detectNewPersistent ()
7:  procedure sweep is
8:      P := ∅
9:      while Q ≠ ∅ do
10:         H := ∅
11:         U := ∅
12:         φ := minProgress (Q)
13:         while φ = minProgress (Q) do
14:             s := deleteMin (Q)
15:             if s ∉ H then
16:                 H := H ∪ {s'}
17:                 U := U ∪ {s'}
18:         expandLayer ()
19: procedure detectNewPersistent is
20:     NP := P \ AP
21:     AP := AP ∪ NP
22:     Q := ∅
23:     for s ∈ NP do
24:         Q := Q ∪ {s}
25: procedure expandLayer is
26:     while U ≠ ∅ do
27:         pick and delete s from U
28:         for s' ∈ succ(s) do
29:             if ψ(s') = ψ(s) then
30:                 if s' ∉ H then
31:                     H := H ∪ {s'}
32:                     U := U ∪ {s'}
33:             else if ψ(s') ⊏ ψ(s) then
34:                 P := P ∪ {s'}
35:             else
36:                 Q := Q ∪ {s'}
```

– An internal memory hash table $\mathcal{H}$ contains, during a sweep, states of the currently processed layer and a set $\mathcal{U} \subseteq \mathcal{H}$ contains states to be processed by the algorithm. Both are present in internal memory.
– An external memory priority queue $\mathcal{Q}$ stores, during a sweep, states in future layers. This structure also has some internal memory part as described later in this section.

The main procedure alternates between sweeps exploring the state space and detection of new persistent states. A sweep may indeed find new persistent states through the exploration of regress edges and the purpose of procedure *detectNewPersistent* is to determine whether these are actually new or have already been found during previous sweep(s). This set of new persistent states, $\mathcal{NP}$, is computed by removing all persistent states found during previous sweeps ($\mathcal{AP}$) from the set of persistent states discovered during the last sweep ($\mathcal{P}$). It is then inserted into $\mathcal{AP}$ and all its elements put in the priority queue $\mathcal{Q}$ to serve as root states during the next sweep. The difference of the two sets can be efficiently implemented by first loading $\mathcal{P}$ into $\mathcal{H}$, and then reading states from $\mathcal{AP}$ one by one to remove them from the table. If $\mathcal{P}$ is too large to fit in memory an alternative is to first sort the states in $\mathcal{P}$ which can be done efficiently in $\mathcal{O}(N \cdot log_2 N)$ I/O operations [1] and then merge the two files.

An iteration of procedure *sweep* first loads in an internal memory hash table $\mathcal{H}$ all states of $\mathcal{Q}$ sharing the same minimal progress value. These states are also added to $\mathcal{U}$ to be processed by procedure *expandLayer*. Once this procedure has finished, *sweep* can terminate if the priority queue has been emptied or otherwise move to the next layer.

Procedure *expandLayer* works as a basic exploration algorithm operating on a queue of unprocessed states $\mathcal{U}$ and storing visited in an hash table $\mathcal{H}$. The only difference is that when a state s' has a different progress value than the one of the state s it is generated from, s' is put in the priority queue $\mathcal{Q}$ if it belongs to a future layer or in the set of persistent set $\mathcal{P}$ if it belongs to a past layer.

3.2 External Priority Queue Data Structure

A priority queue is a data structure that must support two operations: an *insert* operation; and a $deleteMin$ operation that removes and returns from a queue the smallest state (i.e., with the smallest progress measure in our case).

We use an external data structure introduced in [4]. The structure is described in Section 3.1 of the referenced article. Our choice is mainly motivated by the simplicity of this data structure and the fact that it achieves a nearly optimal I/O complexity. This priority queue is a two level data structure with smallest states being kept in internal memory and others in external memory. Figure 3 is a graphical representation of its organisation. The internal part can be implemented with any data structure that efficiently supports $insert$, $deleteMin$ and $deleteMax$ operations. For instance, a balanced binary tree matches these requirements and we will assume this choice hereafter. The internal memory part is split in two parts: one balanced tree $\mathcal{T}_{ins}$ storing states put in the queue via the *insert* operation and one balanced tree $\mathcal{T}_{del}$ used as a buffer to store the last states read from external memory and filled in by the $deleteMin$ operation. It is an invariant property that a state stored in external memory cannot be smaller than a state stored in internal memory (in $\mathcal{T}_{ins}$ or $\mathcal{T}_{del}$).

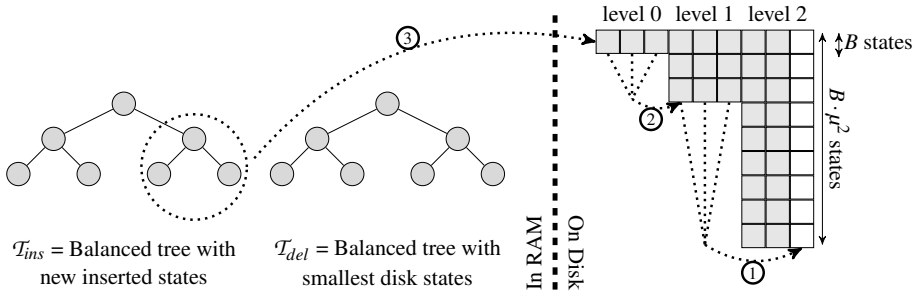

Fig. 3. Organisation of the external memory priority queue with $\mu = 3$. Dashed arrows indicate operations triggered when tree $\mathcal{T}_{ins}$ reaches the bound of T states: merging of the 3 files of level 1 in an available slot of level 2 (op. 1), same operation for level 0 merged into a newly available slot of level 1 (op. 2), and write of the B largest states of $\mathcal{T}_{ins}$ in the first slot of level 0 (op. 3).

The *insert* operation puts the new state in $\mathcal{T}_{ins}$. If the size of $\mathcal{T}_{ins}$ exceeds a specified threshold denoted T, we remove the B largest states from this tree and write them into a new disk file as described below.

The $deleteMin$ operation removes and returns the smallest state kept in internal memory in one of the two balanced trees $\mathcal{T}_{ins}$ or $\mathcal{T}_{del}$. If the state is taken from $\mathcal{T}_{del}$ the deletion can then trigger disk accesses as described below.

Let us now describe the organisation of the external memory part of this data structure. It consists of an array of disk files. Each disk file is a sorted list of states, the smallest state first. This array is organised in levels. Each level consists of μ disk files: level 0 contains files $0, \ldots, \mu - 1$, level 1 contains files $\mu, \ldots, 2 \cdot \mu - 1$ and so on. Following an *insert* operation, we may have to write B sorted states to a disk file of level 0. If there is no free slot at this level, all files of level 0 are merged resulting in a new sorted list that is written to a disk file of level 1. If level 1 is also full, we first have to merge level 1 into a new disk file of level 2 and so on. Hence, a disk file of level $l \geq 0$ always contains at most $B \cdot \mu^l$ states. The next unprocessed K states (i.e., to be removed from the queue via the $deleteMin$ operation) of every file are kept in internal memory in the binary tree $\mathcal{T}_{del}$. If, following a $deleteMin$ operation, $\mathcal{T}_{del}$ does not contain anymore any state of a specific disk file, the K next states of this file are read and inserted in $\mathcal{T}_{del}$. The correctness of $deleteMin$ stems from the fact that (1) files are sorted and (2) all the smallest unprocessed state stored in external memory are in $\mathcal{T}_{del}$.

The dashed arrow of Fig. 3 describes the operation sequence performed after a state insertion. If the memory tree become full, i.e., contains T states after the insertion, the largest B states must be sorted and written in a disk file associated with a slot of level 0. Since this level is full, we first have to merge its slots in a slot of level 1, which for the same reason implies to merge the slots of level 1 in the last slot of level 2 which is free (arrow 1). We can then merge level 0 in the first slot freed by the previous operation (arrow 2) and finally write the B largest states of $\mathcal{T}_{ins}$ in the first slot of level 0 (arrow 3).

This data structure occupies at most $T + K \cdot f$ states in internal memory where f is the maximal number of disk files simultaneously in use (T for $\mathcal{T}_{ins}$ and $K \cdot f$ for $\mathcal{T}_{del}$). Although f is theoretically unbounded, it grows logarithmically, hence it is not a problem in practise.

3.3 I/O complexity

We now examine the I/O complexity of our algorithm.

Theorem 1. *Let F be the number of forward edges, P be the number of persistent states computed by Alg. 1 (i.e., set $\mathcal{AP}$) and w be the number of sweeps performed by the algorithm (i.e., calls to procedure sweep). Algorithm 1 with the external priority queue data structure presented in Sect. 3.2 and parametrised by levels of size μ and an internal memory buffer of size B performs at most $P \cdot (w + 2) + w \cdot F \cdot 2 \cdot \log_\mu \left(\frac{F}{B} + 1\right)$ state I/Os.*

Proof. A persistent state is read and written once in $\mathcal{NP}$. It is then read from $\mathcal{AP}$ during each subsequent call to procedure *detectNewPersistent* (invoked once for every sweep). This procedure thus performs at most $P \cdot (w+2)$ state I/Os.

Since only forward edges generate states put in the priority queue $\mathcal{Q}$, the size of this structure is bounded by F. The largest level containing disk files is the smallest integer l^m satisfying: $\mu \cdot \sum_{i \in [0..l^m]} \mu^i \cdot B \geq F$ where $\mu^i \cdot B$ denotes the number of states stored in each disk file of level i. Since $\sum_{i \in [0..l^m]} \mu^i = \frac{\mu^{l^m+1}-1}{\mu-1}$, it follows that $\mu \cdot \frac{\mu^{l^m+1}-1}{\mu-1} \geq \frac{F}{B}$ and that $l^m \geq \log_\mu \left(\frac{F}{B} \cdot \frac{\mu-1}{\mu} + 1 \right) - 1$. Now let $l = \log_\mu \left(\frac{F}{B} + 1 \right) - 1 > l^m$. The destination state of a forward edge is written once in a file of level 0 and then due to level merging, it can be moved l times from any level $i \in \{0, \ldots, l-1\}$ to level $i+1$. This implies at most $2 \cdot (l+1)$ I/Os per state. Since each sweep performed by the algorithm can explore all the F forward edges, it follows that the overall number of state I/Os performed to maintain the priority queue is bounded by $w \cdot F \cdot 2 \cdot \log_\mu \left(\frac{F}{B} + 1 \right)$. □

For the sake of comparison, we give in Table 2 the I/O complexity of two other external memory algorithms: BFS$^{\text{ext}}$, the external memory breadth-first search from [6] and PART, a state space partitioning based algorithm from [2].

Table 2. Comparison of I/Os bound of three external memory algorithms.

Algorithm	Source	State I/Os bound
BFS$^{\text{ext}}$	[6]	$S \cdot (1+h)$
PART	[2]	$S \cdot (1+C^{max})$
SL$^{\text{ext}}$	This work	$P \cdot (w+2) + w \cdot F \cdot 2 \cdot \log_\mu \left(\frac{F}{B} + 1 \right)$

S = states in the graph
C^m = max. over all partitions p of cross edges with a destination in p
F = forward edges
P = persistent states computed by Alg. 1
h = height of the graph
w = sweeps performed by Alg. 1

Using BFS$^{\text{ext}}$, a state will be written once in external memory when first met, and then read once during each of the subsequent duplicate detections. Since the algorithm performs exactly one duplicate detection per BFS level, a state s will be read at most h times where h is the height of the graph, i.e., the number of BFS levels.

An important parameter for the I/O complexity of PART is the number of cross edges implied by the partitioning function mapping states to partitions. A cross edge is an edge linking to states belonging to different partitions. To give a better insight of its complexity we recall that the principle of algorithm PART is to cluster the state space using a partition function and to store each partition in a single disk file. During the exploration, only one partition is loaded in memory at a time. Any cross edge (s, s') visited by the algorithm is eventually followed by the reading of the disk file associated with the partition of s' to check whether s' is a new state or not. With this algorithm, a state will be written once in

external memory when first met and then read again each time the partition it is stored in is loaded in memory, hence at most C^m times where C^m is the maximum over all partitions p of cross edges with a destination in p.

In the case of $\mathsf{BFS}^{\mathsf{ext}}$, the bound given is usually close to the number of I/Os actually performed while the practical I/O complexity of PART is in general much smaller than the theoretical bound we give here. The proportion of cross edges has nevertheless a large impact on its performance. Similarly, the bound of Algorithm 1 may seem high at first sight since the number of sweeps performed by the algorithm is bounded by the number of regress edges. However, in practise, w is usually low, typically less than 10. This is precisely why the sweep-line method works well in practise for a wide range of models. First because the progress measure provided usually generates few regress edges. Second because it is very seldom that a single sweep identifies only one new persistent state which in turn means that the number of iterations performed is usually not correlated to the number of regress edges. Moreover, the upper bound we give here does not take into account caching effects that might further decrease the amount of disk accesses. This occurs when the destination state s of a forward edge is already present in T_{ins} when the edge is explored. Then the queue is unchanged.

4 Experimental Evaluation

We have implemented the external memory algorithm introduced in Sect. 3 in the ASAP verification platform [19] and experimented with models of the BEEM database [16]. We followed the procedure described in [8] to automatically derive a progress measure from the full state space. Our progress measures project the state vector of the system to a subset of its components. This subset is selected with a heuristic that aims to limit the proportion of regress edges.

We compared our external algorithm (denoted $\mathsf{SL}^{\mathsf{ext}}$) to the internal memory sweep-line algorithm of [15] (denoted SL) and to the external memory algorithms $\mathsf{BFS}^{\mathsf{ext}}$ [6] and PART [2] combined with the dynamic partitioning strategy we introduced in previous work [7]. For each model, we first ran SL and $\mathsf{SL}^{\mathsf{ext}}$, and then $\mathsf{BFS}^{\mathsf{ext}}$ and PART giving them the same memory amount that was used by $\mathsf{SL}^{\mathsf{ext}}$. The priority queue of $\mathsf{SL}^{\mathsf{ext}}$ has been parametrised as follows: $\mu = 10$, $T = 20,000$, $B = 10,000$ and $K = 1,000$. We recall that the memory usage of the priority queue data structure is bounded by $T + K \cdot f$ where f is the maximal number of files simultaneously in use. We also experimented with different comparable (wrt. memory usage) parameter configurations but since this had few consequences we selected the configuration above.

The 43 instances we selected all have between 1,000,000 and 50,000,000 states and each instance has from 1 to 6 progress measures. This selection resulted in 125 pairs (model instance, progress measure) analysed. Out of these 125 pairs we only kept those for which $\mathsf{SL}^{\mathsf{ext}}$ consumed significantly less memory than SL (selected to be 5 times less the consumption of SL) and leave out the other pairs. Keeping instances for which both algorithms consumed a comparable amount of memory would indeed not be relevant to study the performances of $\mathsf{SL}^{\mathsf{ext}}$ and

Table 3. Statistics on selected models with experimental results of SL^{ext}.

Model	States	Results of SL^{ext}					
		Peak memory	Revisit factor	Visited	Edges Reg.	Fwd.	Sta.
bakery.6	11,845,035	30,119	1.305	52,406,033	11.3 %	51.0%	37.6%
bopdp.7	15,236,725	32,244	2.566	100,779,572	1.0 %	37.6%	61.4%
brp.4	12,068,447	43,892	1.122	27,752,570	1.1 %	40.3%	58.6%
cambridge.6	3,354,295	39,392	12.360	106,771,249	3.5 %	39.8%	56.7%
elevator2.3	7,667,712	30,811	1.172	64,737,253	0.3 %	88.4%	11.2%
extinction.7	20,769,427	85,764	1.215	92,560,320	1.8 %	35.3%	62.9%
firewire_link.5	18,553,032	128,620	1.000	59,782,059	0.0 %	6.9%	93.1%
firewire_tree.5	3,807,023	22,837	1.000	18,226,963	0.2 %	80.8%	19.0%
iprotocol.6	41,387,484	89,480	1.724	239,771,446	3.8 %	22.8%	73.5%
leader_election.5	4,803,952	32,400	1.000	28,064,092	0.5 %	53.5%	46.0%
needham.4	6,525,019	30,420	1.000	22,203,081	0.0 %	30.5%	69.5%
peterson.4	1,119,560	38,338	5.286	19,974,479	1.8 %	46.7%	51.5%
pgm_protocol.10	29,679,589	25,142	1.324	81,985,078	2.5 %	55.7%	41.8%
plc.4	3,763,999	2,447	1.043	6,358,220	1.1 %	11.3%	87.5%
rether.6	5,919,694	31,599	1.496	11,902,295	5.5 %	47.4%	47.1%
synapse.7	10,198,141	203,448	1.333	24,201,729	11.2 %	66.9%	21.9%

may moreover lead to a biased analysis. This second filtering resulted in 60 pairs of which we picked out a representative set of 16 instances to be presented in this section. These models are listed in Table 3 together with some experimental results obtained with algorithm SL^{ext}. Peak memory corresponds to the maximal number of states stored in internal memory during the search (regardless of their location). Revisit factor is the number of states visited by algorithm SL^{ext} relative to the number of states in the state space (i.e., column States). Columns Visited gives the number of edges visited by algorithm SL^{ext}. It may then be larger than the number of edges in the state space if the revisit factor is larger than 1. The last three columns give the distribution of these visited edges upon regress, forward and stationary edges.

Comparison of memory usage and disk access. Figure 4 compares SL^{ext} to SL with respect to peak memory usage and to PART with respect to state I/Os. We deliberately left out data of algorithm BFS^{ext} in Fig. 4 and Fig. 5. As attested by Table 4, BFS^{ext} was not competitive on these models and including its results in the figures would have reduced their readability. We gave the data observed with SL^{ext} or PART a reference value of 1. For example, with instance firewire_tree.5, SL^{ext} used 5–6% of the internal memory used by SL and performed 10–12% of the state I/Os performed by PART.

The conclusions we can draw from Fig. 4 are rather positive as the general trend in that SL^{ext} consumes significantly less memory than SL while performing less disk accesses than PART. The comparison of SL and SL^{ext} confirm our initial intuition that SL is sometimes unable to significantly reduce the peak memory

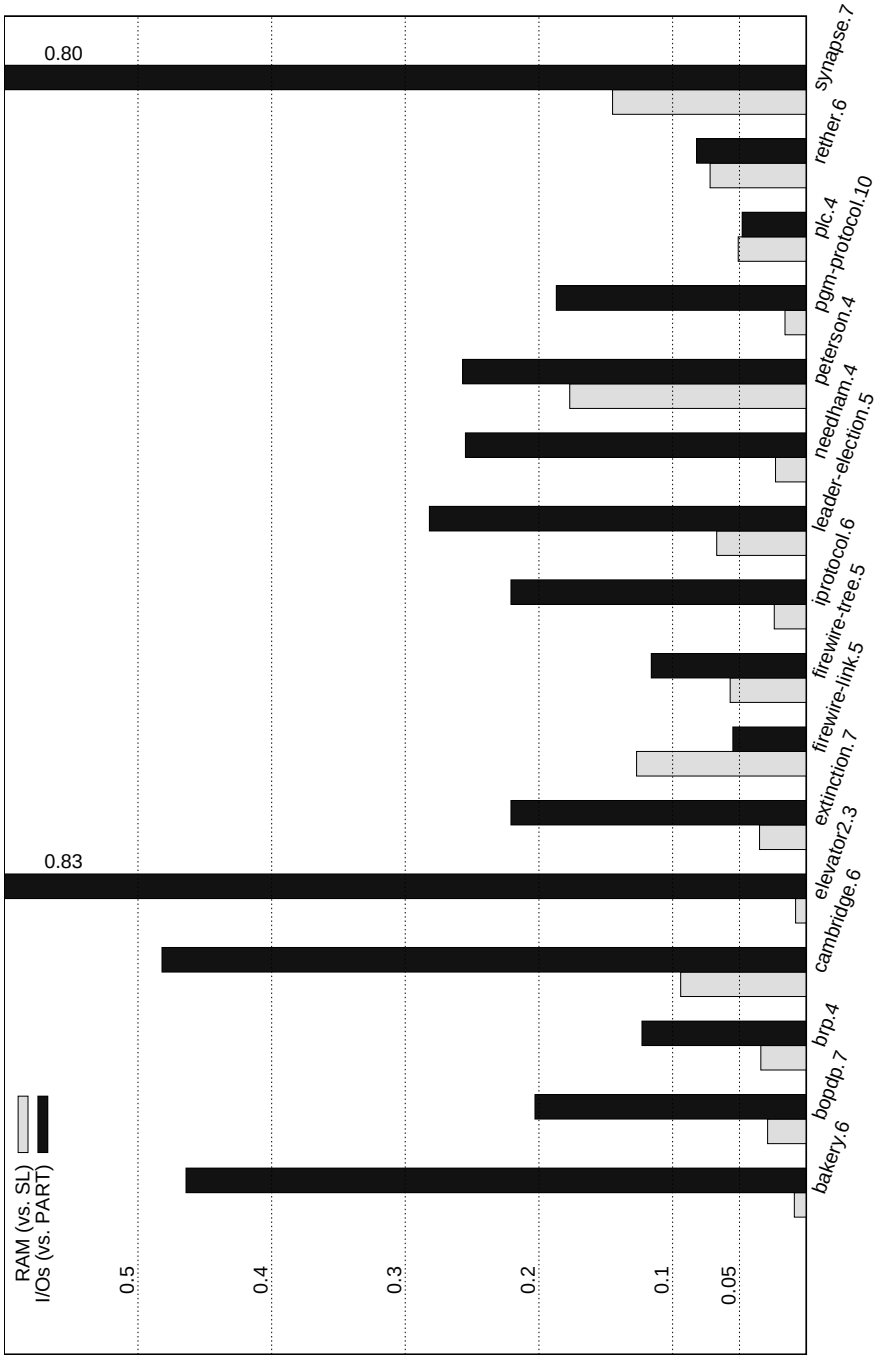

Fig. 4. Peak memory usage of $\mathsf{SL}^{\mathsf{ext}}$ (this work) compared to SL [15] and state I/Os performed by $\mathsf{SL}^{\mathsf{ext}}$ compared to PART [2] on the instances of Table 3.

usage although the progress measure used efficiently divides the state space upon multiple progress layers. Out of the 16 instances we selected, our algorithm performed a comparable number of I/Os with respect to PART on 2 instances: elevator2.3 and synapse.7. The high proportion of forward edges (that increases disk accesses performed by $\mathsf{SL}^{\mathsf{ext}}$) combined with the special shape of their state space (wide and short) which makes them especially suited for PART can explain this difference. The figure indeed attests that the shape of the graph has an impact on the disk accesses performed by the two algorithms. The advantage of $\mathsf{SL}^{\mathsf{ext}}$ over PART is more significant when the state space is long and narrow, e.g., for instances plc.4 or rether.6. If we only consider disk accesses, the performances of PART degrade for these models while $\mathsf{SL}^{\mathsf{ext}}$ is insensitive to that parameter. For some models like cambrige.6 or, to a lesser extent, bopdp.7, Table 3 shows that, it is likely that the relatively high amount of state I/Os performed stems from the revisit factor of algorithm $\mathsf{SL}^{\mathsf{ext}}$ (whereas PART does not revisit states).

Comparison of execution times. Figure 5 gives the execution times of algorithms SL, PART and $\mathsf{SL}^{\mathsf{ext}}$ on the same instances. For each model we gave to the slowest algorithm a reference value of 1, and the execution times of the two other algorithms are expressed relatively to this one. We see a correlation between the disk accesses we previously observed and the execution times of $\mathsf{SL}^{\mathsf{ext}}$ over PART. For most instances on which $\mathsf{SL}^{\mathsf{ext}}$ performed significantly less disk accesses it outperformed PART. This is the case for instances brp.4, plc.4 or rether.6. Models firewire_link.5 and firewire_tree.5 go against this trend for reasons explained below. Nevertheless, the conclusions are somewhat disappointing in that the clear advantage of $\mathsf{SL}^{\mathsf{ext}}$ over PART with respect to disk accesses is less significant when it comes to execution times even though we will see below that $\mathsf{SL}^{\mathsf{ext}}$ is, on the average, faster than PART. One reason is that $\mathsf{SL}^{\mathsf{ext}}$ visits more states that PART and the visit of a state implies some non-trivial operations like the computation of enabled events and the generation of successor states. We also profiled the code of some instances, and found out that comparing states according to their progress values — operations that are not performed by PART — contribute to degrade the relative performances of $\mathsf{SL}^{\mathsf{ext}}$. This operation is performed not only during state exploration but also during the merging of files operated for maintaining the priority queue. This explains the divergence between disk accesses and execution times for models firewire_link.5 and firewire_tree.5. On these two models, internal memory operations are the most time consuming operations and disk accesses play a lesser role.

Summary. We conclude this analysis with Table 4 that compares $\mathsf{SL}^{\mathsf{ext}}$ to the three other algorithms we experimented with. Each cell gives for a specific parameter (Time, I/Os per state, or Memory usage) the result of a specific algorithm averaged on our 60 problem instances and with respect to $\mathsf{SL}^{\mathsf{ext}}$. For instance, on the average of our 60 problem instances, $\mathsf{SL}^{\mathsf{ext}}$ has been 8.29 times faster than $\mathsf{BFS}^{\mathsf{ext}}$. As we previously mentioned, the table shows that $\mathsf{BFS}^{\mathsf{ext}}$ is clearly not competitive with other algorithms. PART generates significantly more disk ac-

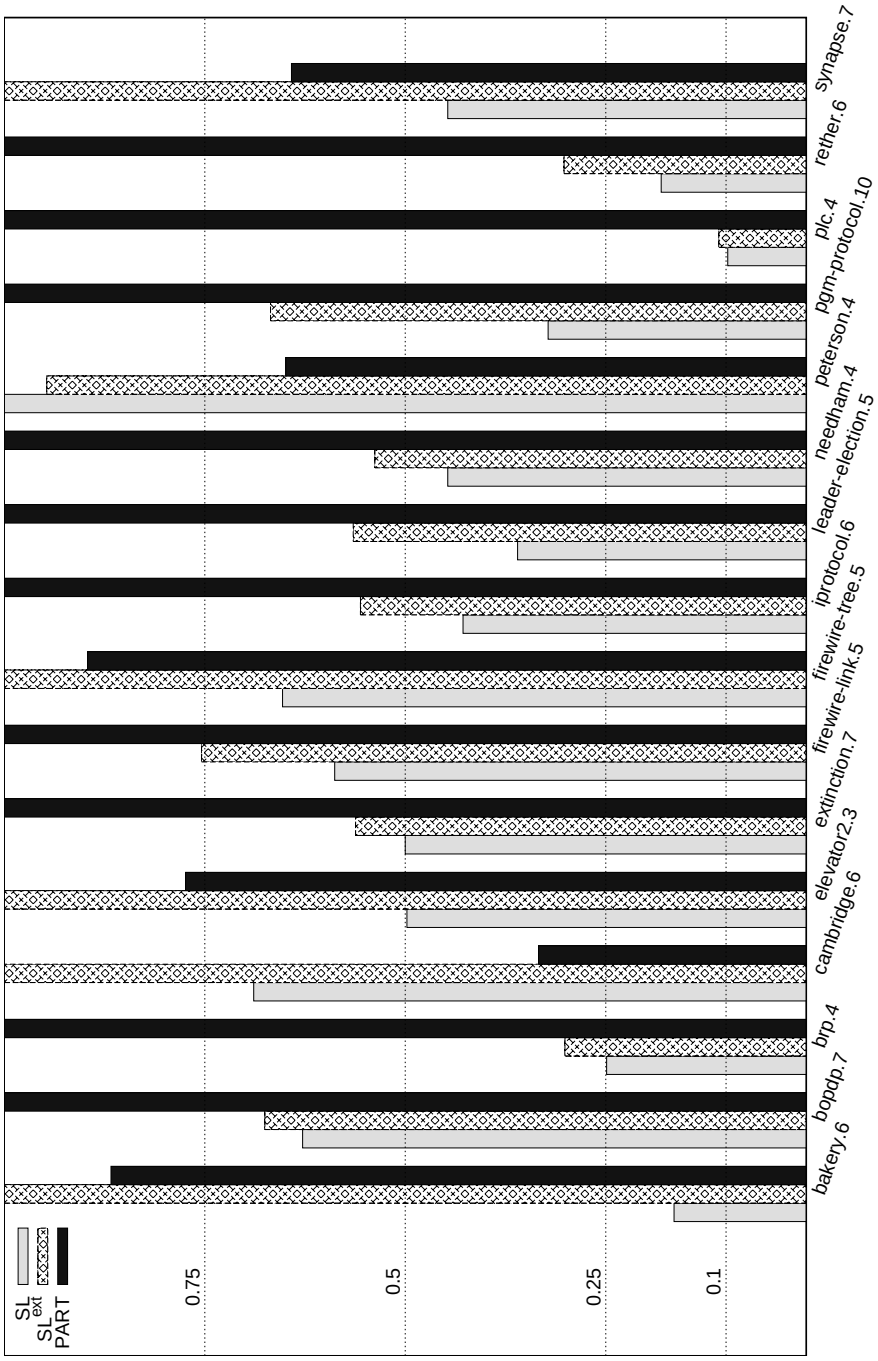

Fig. 5. Execution times of algorithms SL [15], SL$^{\text{ext}}$ (this work) and PART [2] on the instances of Table 3.

cesses than SL^{ext} but the latter is only approximately twice faster than PART for the reasons given above. The table also shows that keeping the priority queue and the persistent states in external memory does not bring an intolerable increase of the run time especially if we relate the increase of the execution time to the important memory usage reduction we observe. This suggests that SL^{ext} can be an interesting alternative when the I/O overhead of PART is too severe and SL fails to reduce peak memory usage even if the progress measure efficiently splits the state space upon multiple small layers. Figures 4 and 5 show that models like brp.4, plc.4 or rether.6 are good application examples for our algorithm.

Table 4. Summary of the performances on 60 problem instances.

	SL^{ext}	SL	PART	BFS^{ext}
Time	× 1.00	× 0.40	× 1.88	× 8.29
I/Os per state	× 1.00	× 0.00	× 9.27	× 217.50
Memory usage	× 1.00	× 27.74	× 1.00	× 1.00

5 Conclusions and Future Work

In this paper we have explored the combination of the sweep-line method and the use of external memory. The key to this combination is the use of an external memory priority queue for storing states that are not in the current layer of states being processed. Using the benchmark suite of examples from the BEEM database, we have experimentally demonstrated that the combined approach significantly reduced peak memory usage (compared to the conventional RAM-based sweep-line method) and that the combination in many cases also reduces I/O operation (compared to a conventional external memory state space exploration). Furthermore, the reduction in peak memory usage comes with an acceptable run-time penalty caused by the use of the external memory priority queue. An important property of the combined approach is that it is compatible with all existing sweep-line based model checking algorithms as none of these rely on a particular implementation of the progress priority queue. It can therefore be used in the context of safety model checking [15] or LTL model checking [9].

With the use of an external memory priority queue our experiments demonstrate that we are able to unleash more of the potential of the sweep-line method and that reduction in peak memory usage can be significant. A follow-up research question is therefore whether the sweep-line method has more potential memory reduction that can be leveraged, i.e., how close are the provided progress measures in terms of being optimal? For progress measures that are monotonic, the size of the largest strongly connected component is a lower bound on the reduction that can be obtained since all states belonging to a strongly connected component must have the same progress measure. For non-monotonic progress

measures it is less obvious how to compute a good lower bound since the presence of regress edges must also be taken into account, and since optimality now needs to take into account both space (peak memory usage) and time (due to re-exploration caused by regress edges). Computing optimal progress measure hence remains an open and highly relevant aspect to explore as part of future work.

We also observe that for preserving termination, it is only required that persistent states are identified during a sweep, put in a separate set, and ignored by the current sweep. The sweep-line can hence move backward and forward, as long as regress edges are taken care of, the sweep-line will eventually converge and the sweep will terminate. This means that any form of queue can theoretically be used although a "strict" priority queue ensures that a sweep cannot revisit states. Starting from this observation, another research direction is thus to evaluate whether relaxing the priority queue requirements can further help reduce disk accesses. This can naturally come at the cost of state revisits meaning that a tradeoff would thus have to be made.

References

1. A. Aggarwal and J.S. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM*, 31(9):1116–1127, 1988.
2. T. Bao and M. Jones. Time-Efficient Model Checking with Magnetic Disk. In *TACAS'2005*, volume 3440 of *LNCS*, pages 526–540. Springer, 2005.
3. G. Behrmann, K.G. Larsen, and R. Pelánek. To Store or Not to Store. In *CAV*, volume 2725 of *LNCS*, pages 433–445. Springer, 2003.
4. K. Brengel, A. Crauser, P. Ferragina, and U. Meyer. An Experimental Study of Priority Queues in External Memory. In *WAE'1999*, volume 1668 of *LNCS*, pages 346–360. Springer, 1999.
5. S. Christensen, L.M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *TACAS*, volume 2031 of *LNCS*, pages 450–464. Springer, 2001.
6. D.L. Dill and U. Stern. Using Magnetic Disk Instead of Main Memory in the Murϕ Verifier. In *CAV'1998*, volume 1427 of *LNCS*, pages 172–183. Springer, 1998.
7. S. Evangelista and L. M. Kristensen. Dynamic state space partitioning for external memory state space exploration. *Science of Computer Programming*, 0(0):–, 2011.
8. S. Evangelista and L.M. Kristensen. Search-Order Independent State Caching. *Transactions on Petri Nets and Other Models of Concurrency IV*, 6550:21–41, 2010.
9. S. Evangelista and L.M. Kristensen. Hybrid On-the-fly LTL Model Checking with the Sweep-Line Method. In *Petri Nets*, volume ??? of *LNCS*, page ??? Springer, 2012. To appear.
10. G. E. Gallasch, J. Billington, S. Vanit-Anunchai, and L.M. Kristensen. Checking Safety Properties On-the-fly with the Sweep-Line Method. *STTT*, 9(3-4):371–392, 2007.
11. G.E. Gallasch, B. Han, and J. Billington. Sweep-Line Analysis of TCP Connection Management. In *ICFEM*, volume 3785 of *LNCS*, pages 156–172. Springer, 2005.
12. G.E. Gallasch, C. Ouyang, J. Billington, and L.M. Kristensen. Experimenting with Progress Mappings for the Sweep-Line Analysis of the Internet Open Trading Protocol. In *CPN*, pages 19–38, 2004.

13. P. Godefroid, G.J. Holzmann, and D. Pirottin. State-Space Caching Revisited. In *CAV'1992*, volume 663 of *LNCS*, pages 178–191. Springer, 1992.
14. S. Gordon, L.M. Kristensen, and J. Billington. Verification of a Revised WAP Wireless Transaction Protocol. In *ICATPN*, volume 2360 of *LNCS*, pages 182–202. Springer, 2002.
15. L.M. Kristensen and T. Mailund. A Generalised Sweep-Line Method for Safety Properties. In *FME*, volume 2391 of *LNCS*, pages 549–567. Springer, 2002.
16. R. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *SPIN'2007*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007. `http://anna.fi.muni.cz/models/`.
17. K. Schmidt. LoLA: A Low Level Analyser. In *ICATPN*, volume 1825 of *LNCS*, pages 465–474. Springer, 2000.
18. S. Vanit-Anunchai, J. Billington, and G. E. Gallasch. Analysis of the Datagram Congestion Control Protocols Connection Management Procedures using the Sweep-line Method. *STTT*, 10(1):29–56, 2008.
19. M. Westergaard, S. Evangelista, and L.M. Kristensen. ASAP: An Extensible Platform for State Space Analysis. In *ICATPN*, volume 5606 of *LNCS*, pages 303–312. Springer, 2009.