Theoretical Computer Science 429 (2012) 169–179

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

The sweep-line state space exploration method

Kurt Jensen [a], Lars M. Kristensen [b,∗], Thomas Mailund [c]

[a] *Department of Computer Science, Aarhus University, Denmark*

[b] *Department of Computer Engineering, Bergen University College, Norway*

[c] *Bioinformatics Research Centre, Aarhus University, Denmark*

ARTICLE INFO

Keywords:
Explicit state space exploration
Model checking
Concurrent systems
Verification
State explosion
Sweep-line method
Coloured Petri Nets

ABSTRACT

The sweep-line method exploits intrinsic progress in concurrent systems to alleviate the state explosion problem in explicit state model checking. The concept of progress makes it possible to delete states from the memory during state space exploration and thereby reduce peak memory usage. The contribution of this paper is twofold. First, we provide a coherent presentation of the sweep-line theory and the many variants of the method that have been developed over the past 10 years since the basic idea of the method was conceived. Second, we survey a selection of case studies where the sweep-line method has been put into practical use for the verification of concurrent systems.

1. Introduction

Model checking [1,2] based upon the seminal work of Clarke and Emerson, and Queille and Sifakis has, over the past 30-years, matured into a technology that can be used for the formal verification of concurrent systems [3]. A main approach to model checking is exploration of the *state space* of the modelled system with the goal of determining whether the system satisfies formally stated behavioural properties. Temporal logics [4] such as Computation Tree Logic (CTL) and Linear Time Temporal Logic (LTL) are widely used to express behavioural properties of systems. The task of a model checking computer tool is to take a model (expressed in some formal modelling formalism based, e.g., on Petri nets [5], Communicating Sequential Processes (CSP) [6], or timed automata [7]) and determine whether the model of the system satisfies the stated property or not. Model checking algorithms are often specified at the level of transition systems which make them independent of a concrete formal modelling language. A main paradigm underlying many model checking tools such as SPIN [8], DiVinE [9], and CPN Tools [10] is that of *explicit state space exploration*. In its simplest form, this relies on an explicit enumeration of all reachable states of the system. Other prominent paradigms include symbolic [11], satisfiability (SAT)-based [12], and unfolding-based model checking [13].

The advantages of model checking is the high-degree of automation, the systematic exploration of all relevant executions, and that counter examples can be provided. The main disadvantage is the *state explosion problem* [14], and the majority of model checking research has been driven by the development of techniques for alleviating this inherent complexity problem. Within explicit state model checking, several families of *reduction methods* have emerged. Each family of methods typically exploits certain characteristics of systems, and/or limit the class of behavioural properties that can be verified. One example is *partial-order reduction methods* [15] which explores a reduced state space (a subset of the full state space), and where the reduced state space is sound and complete with respect to the property to be verified. Another family of methods provides a compact representation of the explored state space. This family includes state reconstruction-based methods [16] and approximative methods such as *bit-state hashing* [17] and *hash compaction* [18]. Symmetry-based methods [19,20]

∗ Corresponding author.
E-mail address: Lars.Michael.Kristensen@hib.no (L.M. Kristensen).

0304-3975/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2011.12.036

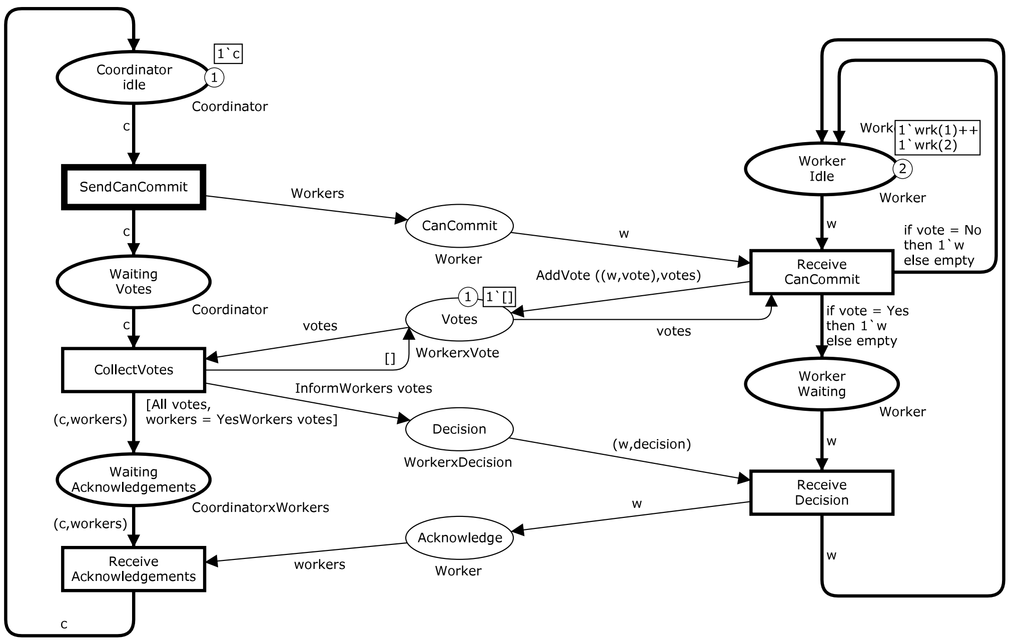

Fig. 1. CPN model of a two-phase commit protocol.

providing a condensed representation of states using equivalence classes also belong to this family. Finally, methods have been developed that rely on increasing the space resources using external storage [21] or time and space resources using parallel architectures [9].

The amount of memory is often the limiting factor in model checking. During exploration of the state space, the set of states encountered is kept in memory in order to recognise already visited states and thereby ensure that the state space exploration terminates. This has lead to a family of methods that combat state explosion by deleting states from memory *during* state space exploration. This family of methods includes the *state caching method* [22], the *to-store-or-not-to-store method* [23], and the *sweep-line method* [24,25]. 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 memory at a time. This means that the peak memory usage is reduced. The sweep-line method is in its basic form 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 developed in several papers [24–29] and the method has been implemented in the ASAP platform [30] and the LoLA tool [31]. The sweep-line method has been used [32–35] for the verification of several industrial-sized protocols specified using the Coloured Petri Nets (CPN) modelling language [36].

The paper is organised as follows: Section 2 introduces the basic concepts of the sweep-line method using a small example of a CPN model. Section 3 presents the range of extensions that have been developed to the basic method. Section 4 surveys several case studies where the sweep-line method and its implementation in computer tools have been put into practical use for verification of protocols. Finally, in Section 5 we sum up the conclusions and discuss further related work. We assume that the reader is familiar with the basic concepts of Petri nets and explicit state space exploration.

2. The basic sweep-line method

To introduce the concepts of the sweep-line method, we use the CPN model in Fig. 1 that models a two-phase commit protocol for distributed atomic transactions. The goal of the protocol is to coordinate whether all the processes participating in a transaction is to commit or abort the transaction. We do not have the necessary space to describe all details of the CPN model. In particular, we do not describe the types (colour sets) and the functions in guards and arc expressions. Readers with limited knowledge of high-level Petri Nets may use Fig. 1 as an informal drawing illustrating the operation of the protocol. Readers with more knowledge of high-level Petri nets will see that the CPN model constitutes a complete and unambiguous formal specification of the two-phase commit protocol.

The left-hand side of Fig. 1 models the loop executed by the *coordinator process* which is initially in an idle state as modelled by the one token with colour c initially marking the place CoordinatorIdle. The right-hand side models the loop executed by a set of *worker threads* that are all initially idle as modelled by the place WorkerIdle. The place WorkerIdle initially contains a token for each of the worker threads—in this case two workers wrk(1) and wrk(2). The four places CanCommit, Votes, Decision, and Acknowledge in the middle are used to model the messages exchanged between the coordinator and

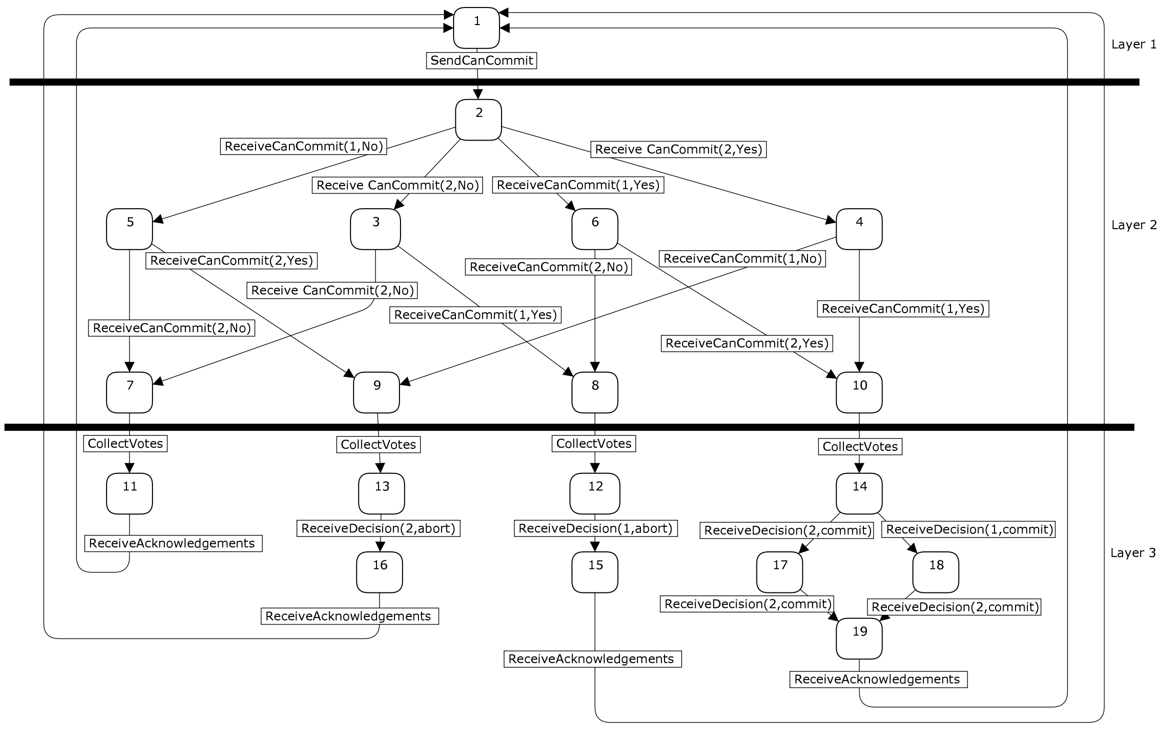

Fig. 2. State space of the two-phase commit protocol.

worker threads. Initially, the coordinator will Send a CanCommit message to each worker asking whether the transaction can be committed and then enter a state where it is Waiting for Votes. Each worker will then Receive the CanCommit message and send back a Vote message indicating whether the transaction may be committed (Yes) or not (No). The Worker threads that are ready to commit the transaction will enter a Waiting state whereas the other threads will reenter the Idle state. The coordinator will Collect the Votes once all workers have replied, and send a message to all waiting workers indicating a decision as to whether the transaction is to be committed or aborted. All workers receiving notifications will send back a message to the coordinator acknowledging receipt of the decision message. The coordinator will Receive all Acknowledgement and then reenter the Idle state.

The two-phase commit protocol makes *progress* from the state where the coordinator and all workers are in their idle state to the state where a commit or abort decision has been made and the coordinator has collected all acknowledgements. This progress is also reflected in the state space of the two-phase commit protocol. Fig. 2 shows the state space (ignore the thick horizontal lines for now) of the commit protocol for two workers, where node 1 represents the initial state of the protocol. To simplify the drawing, detailed information has been omitted about the states represented by each node. Each edge has an associated label specifying the name of the occurring transition to which it corresponds. For the transitions modelling the actions of the workers, the label also specifies in parentheses the identity (1 or 2) of the worker executing the corresponding action. In the case of the ReceiveCanCommit transition, the label specifies whether the worker voted Yes or No, and in the case of ReceiveDecision the label specifies the decision, i.e., whether the worker is to abort or commit its part of the distributed transaction.

Notation. To make the presentation of the sweep-line method independent of a particular modelling language, we formulate the method in the context of a *labelled transition system* $\mathcal{S} = (S, T, \Delta, s_I)$, where S is a finite set of *states*, T is a finite set of *transitions*, $\Delta \subseteq S \times T \times S$ is the *transition relation*, and $s_I \in S$ is the *initial state*. Let $s, s' \in S$ be two states and $t \in T$ a transition. If $(s, t, s') \in \Delta$, then we say that t is *enabled* in s, and that the *occurrence* of t in the state s leads to the state s'. This is also written $s \xrightarrow{t} s'$. A state s' is *reachable* from a state s iff there exists a sequence of states $s_1, s_2, s_3, \ldots, s_{n-1}, s_n$ and a (possible empty) sequence of transitions $t_1, t_2, \ldots t_{n-1}$ such that $s = s_1, s_n = s'$, and $(s_i, t_i, s_{i+1}) \in \Delta$ for $1 \leq i \leq n-1$. If state s' is reachable from state s we write $s \rightarrow^* s'$. For a state s, $\mathsf{reach}(s) = \{ s' \in S \mid s \rightarrow^* s' \}$ denotes the set of states reachable from s. The set of *reachable states* of $\mathcal{S}$ is then $\mathsf{reach}(s_I)$. The *state space* of a system is the directed graph (V, E) where $V = \mathsf{reach}(s_I)$ is the set of nodes and $E = \{(s, t, s') \in \Delta \mid s, s' \in V\}$ is the set of edges. In the rest of this paper we assume a transition system $\mathcal{S} = (S, T, \Delta, s_I)$.

In Fig. 2, we have organised the state space into *layers* (separated by thick horizontal lines) according to how far the coordinator has *progressed* in the protocol. Layer 1 contains the states in which the coordinator is in the idle state (place CoordinatorIdle), layer 2 contains the states where the coordinator is waiting (place WaitingVotes) to collect votes, and layer 3 contains the states where the coordinator is waiting (place WaitingAcknowledgements) to receive acknowledgements.

The progress exploited by the sweep-line method for a system 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 $\psi : S \to O$ is a **progress mapping**. $\mathcal{P}$ is **monotonic** if $\forall s, s' \in \mathsf{reach}(s_I) : s \to^* s' \Rightarrow \psi(s) \sqsubseteq \psi(s')$. Otherwise, P is **non-monotonic**. $\square$

The progress measure used by the sweep-line method can either be obtained based on an analysis of the model or it can be provided by the user 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. A progress measure $\mathcal{P}_{\mathrm{cp}}$ for the commit protocol measuring how far the coordinator has progressed is defined by $\mathcal{P}_{\mathrm{cp}} = (\mathbb{N}, \leq, \psi_{\mathrm{cp}})$ where:

$$\psi_{\mathrm{cp}}(s) = \begin{cases} 1 & \text{if } s(\mathsf{CoordinatorIdle}) = 1\text{`c} \\ 2 & \text{if } s(\mathsf{CollectVotes}) = 1\text{`c} \\ 3 & \text{if } s(\mathsf{WaitingAcknowledgments}) = 1\text{`c} \end{cases}$$

and $s(p)$ denotes the tokens present on the place p in the state s. As an example, the progress measure above maps all states where the coordinator is idle to progress value 1. A *monotonic progress measure* preserves the reachability relation, i.e., if a state s' is reachable from a state s, then $\psi(s) \sqsubseteq \psi(s')$. The $\mathcal{P}_{\mathrm{cp}}$ progress measure is not monotonic since it has *regress edges*, i.e., edges where the source state has a larger progress value than the destination state. One example of this is the edge leading from state 19 with progress value 3 to state 1 with progress value 1. In the commit protocol, regress is due to the coordinator and workers returning to their initial state after execution of the protocol.

The basic idea of the sweep-line method is to explore the state space in a *least-progress-first order*, one layer at a time. Once all states in a given layer has been *processed* (i.e., direct successor states has been calculated) the states belonging to this layer are deleted from memory and exploration continues with the next layer. As an example, consider layer 2 in Fig. 2: when we have calculated all the direct successors of states 2–10 in layer 2, then we have these states plus states 11–14 from layer 3 in memory. Before we continue calculating successor states of states in layer 3, the states in layer 2 are deleted. Intuitively, we can think of a *sweep-line* moving through the state space. At any given point during state space exploration, the sweep-line is aligned with a single layer – all of the states in the layer are 'on' the sweep-line – and all new states calculated are either on the sweep-line (in the same layer) or in front of the sweep-line (in a subsequent layer). An interesting point arises when the regress edges in layer 3 is encountered since they lead to a previously visited state (state 1) which has been deleted from memory. In general, we cannot determine whether the destination state of a regress edge is a new state or a state that has been visited before (as in the example). The sweep-line method therefore conservatively marks destination states of regress edges as *persistent* and uses newly encountered persistent states as roots for a subsequent sweep. When a state has been marked as persistent it means that it can no longer be deleted from memory. For the state space in Fig. 2 this means that in the first sweep through the state space, state 1 will be marked as persistent and used as a root for a subsequent sweep of the state space. This implies that with the sweep-line method, parts of the state space may be visited multiple times. For the commit protocol all states are visited twice, but the peak number of states stored in memory is 13 (layer 2 plus direct successor states) and not 19 states which is the total number of reachable states. This demonstrates for a simple example how the sweep-line method trades time in favour of memory. A refined progress measures for the commit protocol could take into account also the progress made by the worker threads. This would split the state space into more layers and reduce peak memory usage further.

The basic sweep-line algorithm is provided in Fig. 3. The algorithm starts with the initial state s_I as the only root (line 1). The algorithm then performs multiple sweeps (lines 8–29) and in each sweep regress edges are identified (line 21), and the destination states are marked as persistent and new ones are used as root states for the next sweep (lines 22–23). When all nodes in the current layer has been processed and the sweep-line moves (line 10), then all non-persistent states in the current layer are deleted from memory. Since all destinations of regress edges are marked as persistent, a state that has been used as a root in one sweep will never be root again in a subsequent sweep.

Soundness, completeness, and termination of the sweep-line method is stated in the following theorem.

Theorem 1. *The sweep-line algorithm in Fig. 3 terminates after having explored at most $(|B| + 1) \cdot |V|$ states, where B denotes the destination states of regress edges: $B = \{ s' \mid \exists s : s \to s' \wedge \psi(s') \sqsubset \psi(s) \}$ and V denotes the nodes in the state space: $V = \mathsf{reach}(s_I)$. Upon termination all states reachable from s_I have been explored at least once.* $\square$

Proof. The inner loop (lines 8–29) is a conventional graph traversal using a least-progress-first order. This graph traversal will explore the subgraph reachable from the nodes in Roots, exploring each node once giving an upper bound of $|V|$ on the number of states explored. The outer loop (lines 3–30) is executed initially and repeated whenever the inner loop recognises at least one regress edge leading to a state not previously marked as persistent. Since each node added to Roots is marked as persistent, and therefore will never be added to Roots again, $|B|$ is an upper bound on the number of nodes that will be added to Roots after the initial state. $|B| + 1$ is therefore an upper bound on the number of times the outer loop is repeated.

```
Roots.Insert($s_I$)
Nodes.Insert($s_I$)
while $\neg$ (Roots.Empty()) {start next sweep} do
  Unprocessed $\leftarrow$ Roots
  Roots $\leftarrow \emptyset$
  Layer $\leftarrow \emptyset$
  $\psi_c$ = Unprocessed.GetMin() {progress value for current layer}
  while $\neg$ (Unprocessed.Empty()) do
    s $\leftarrow$ Unprocessed.GetMinElement()
    if $\psi_c \sqsubset \psi(s)$ {sweep-line moves} then
      for all $s' \in$ Layer such that $\neg$ Nodes.Persistent($s'$) do
        Nodes.Delete($s'$) {delete non-persistent states in current layer}
      end for
      Layer $\leftarrow \emptyset$
      $\psi_c = \psi(s)$ {update progress value for current layer}
    end if
    Layer.Insert($s$)
    for all $(t, s')$ such that $s \xrightarrow{t} s'$ do
      if $\neg$(Nodes.Contains($s'$)) then
        Nodes.Insert($s'$)
        if $\psi(s) \sqsupset \psi(s')$ {regress edge} then
          Nodes.MarkPersistent($s'$)
          Roots.Insert($s'$)
        else
          Unprocessed.Insert($s'$)
        end if
      end if
    end for
  end while
end while
```

Fig. 3. The basic sweep-line state space exploration algorithm.

Now assume that there exists reachable states that are not explored, and choose among these a state s such that the length of a shortest path from s_I to s is minimal among the unexplored states. This implies that there exists a sequence $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} s_n$ where $s_0 = s_I$ and $s_n = s$, such that for all $i < n$, s_i is visited. Since s_I is trivially explored $s \neq s_I$ so $n > 0$ and we can consider the last edge in this sequence, $s_{n-1} \xrightarrow{t_n} s$. We split in two cases:

- If $\psi(s_{n-1}) \sqsubseteq \psi(s)$, then processing of s_{n-1} would discover s, store it in Unprocessed, and later visit it during the inner loop. Hence, if $\psi(s_{n-1}) \sqsubseteq \psi(s)$ then s_{n-1} cannot have been explored, which contradicts our choice of s, and we must conclude $\psi(s_{n-1}) \sqsupset \psi(s)$.
- If $\psi(s_{n-1}) \sqsupset \psi(s)$, then the processing of s_{n-1} would identify $s_{n-1} \xrightarrow{t_n} s$ as a regress edge and add s to the set of roots, and s would be visited in the next sweep. Again, this implies that s_{n-1} cannot have been explored which contradicts the choice of s.

Hence, no such s exists and all states are explored. □

It follows from Theorem 1 that for a monotonic progress measure, all reachable states are visited exactly once. At first sight the complexity of the algorithm may look high for non-monotonic progress measures, considering that $|B|$ could be of the order of the number of vertices $|V|$ of the state space. In practise, however, each sweep discovers more than a single regress edge, and the number of regress edges is not high for a good progress measure. To formalise this, we introduce the concept of *regress edge connectedness* for non-monotonic progress measures:

Definition 2. Let $\mathcal{P} = (O, \sqsubseteq, \psi)$ be a non-monotonic progress measure for $\mathcal{S}$. The state space of $\mathcal{S}$ is n**-regress edge connected** if and only if it is possible to reach all the reachable states by following at most n regress edges, i.e., for all states $s \in \mathsf{reach}(s_I)$ there exists a sequence $s_I = s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots \xrightarrow{t_m} s_m = s$ such that $|\{\, i \mid \psi(s_{i-1}) \sqsupset \psi(s_i) \,\}| \leq n$, where $i \in \{1, 2, \ldots, m\}$. □

A 0-regress-edge connected state space is said to be *monotonically connected*, since all states are reachable through occurrence sequences with monotonically increasing progress values (even if the state space has regress edges). As a refinement of the proof of Theorem 1, it can be observed that after m iterations of the outer loop (i.e., m sweeps), we have explored all states reachable through at most $m - 1$ regress edges. Hence, for an n-regress-edge connected state space, all states have been visited after $n + 1$ sweeps. Unfortunately, since the last persistent states are not recognised until the last sweep, they can still be added as roots for an additional sweep despite the fact that they have already been explored once. The number of sweeps for an n-regress-edge connected state space is therefore $n+2$, the $n+1$ to explore all states, and the last sweep to recognise this.

Theorem 2. *For an n-regress-edge connected state space, the sweep-line algorithm terminates after having explored at most* $((n + 2) \cdot |V|)$ *states.* $\square$

Both Theorems 1 and 2 are concerned with the number of states explored which determines the time it takes to complete the state space exploration. The peak memory usage, i.e., maximum number of states stored simultaneously in memory during the sweep, depends on how the progress measure partitions the set of reachable states, and on the graph structure of the state space. If no regress edges exists, a trivial lower bound for states simultaneously stored in memory is the maximum number of reachable states mapped to the same progress value plus immediate successor states.

3. Extensions to the basic sweep-line method

In this section we survey extensions to the basic method from the previous section. The extensions concern path finding and counter examples [27], compositional progress measures [37], combination with equivalence-based state space reduction, and memory-efficient state space representation [26].

3.1. Path finding and counter examples

An important advantage of state space methods is the ability to provide counter examples. The deletion of states performed by the sweep-line method prohibits the immediate generation of, e.g., a path in the state space leading from the initial state to a state satisfying a state predicate $\phi : S \mapsto \{true, false\}$. As an example, for the commit protocol (see Fig. 2), we might be interested in a path leading from state 1 to state 16. However, when state 16 is encountered all states in layers 1 and 2 have been deleted, and we do not have sufficient states stored in memory to obtain a path.

Path finding with the sweep-line method is possible [27] by writing a spanning tree for the state space to external storage (a disk file) during the exploration. The idea is to store (in external storage) with each state s', an *index* i_s specifying the file position of the predecessor state s from which s' was generated. Consider again Fig. 2 and state 16: when encountering state 16 the indexes stored on disk is traversed backwards to obtain a path back to state 1. Index i_{13} (stored with state 16) is used to obtain the file position of state 13, index i_9 (stored with state 13) is used to obtain the file position of state 9, and i_5 (stored with state 9) is used to obtain the file position of state 4 (assuming that state 9 was generated from state 4 and not state 5). This process continues until state 1 (the initial state) is encountered. Writing states and indexes to external storage can be done by augmenting the algorithm in Fig. 3 such that the initial state s_I is additionally written to disk in line 2. Furthermore, whenever a new state s' is encountered from state s in line 20, then s' is written to disk together with the index of state s (the predecessor state). An important property of the above scheme is that it requires no searches on disk *during* the exploration, and new states and indexes are only appended to the file on disk. Obtaining a path requires one seek on disk per state on the path.

A path obtained with the sweep-line method to a state satisfying a state predicate ϕ is (in general) not a shortest path due to the least-progress first search order being different from breadth-first search order. With the sweep-line method we terminate the exploration when the first encountered layer containing a state satisfying ϕ has been processed. In this case, the path obtained is shortest subject to progress as stated below and proved in [27].

Theorem 3. *Let* $\sigma_\phi = (s_I = s_0, t_0, s_1, t_1, \ldots, t_{n-1}, s_n)$ *be the path obtained with the sweep-line method leading to a state* s_n *satisfying a state predicate* ϕ*, and let* $\sigma'_\phi = (s_I = s'_0, t'_0, s'_1, t'_1, \ldots, t'_{n-1}, s'_n)$ *be any path leading to a state* s'_n *satisfying* ϕ*. For a path* σ*, let* $RE(\sigma)$ *denote the number of regress edges in* σ*, then:* $RE(\sigma_\phi) \leq RE(\sigma'_\phi)$*, and if* $RE(\sigma_\phi) = RE(\sigma'_\phi)$ *then* $\psi(s_n) \sqsubseteq \psi(s'_n)$*.* $\square$

Paths obtained with the sweep-line method is in practise often close to being a shortest path. The reason is that the progress value of a state (in many cases) is proportional to the number of transitions occurrences required to reach it.

3.2. Compositional sweep-line exploration

A concurrent system S is in many modelling formalisms specified as a parallel composition $S = S_1 \parallel S_2 \parallel \cdots S_n$ of subsystems $S_1, S_2, \ldots, S_n$. In this setting, the subsystems are often specified as a labelled transition system (LTS) directly or have a small state space that can be computed explicitly prior to constructing the parallel composition. A variant of the sweep-line method targeting a compositional setting was developed in [37] in which also the concept of a progress measure was generalised such that only a partial ordering $\sqsubseteq$ on progress value O is required. The basic observation is that given progress measures $\mathcal{P}_i = (O_i, \sqsubseteq_i, \psi_i)$ for $1 \leq i \leq n$ for the subsystems, a *product progress measure* $\mathcal{P} = (O, \sqsubseteq, \psi)$ for S can be defined where $O = \prod_{i=1}^{n} O_i$, $\psi(s_1, \ldots, s_n) = (\psi_1(s_1), \ldots, \psi_n(s_n))$, and $(o_1, \ldots, o_n) \sqsubseteq (o'_1, \ldots, o'_n)$ if and only if $o_i \sqsubseteq_i o'_i$ for all $1 \leq i \leq n$. The product progress measure associates a progress vector to each state $s = (s_1, s_2, \ldots, s_n)$ and positions the reachable states into an n-dimensional vector space. A nested algorithm was developed in [37] that explores subspaces corresponding to non-monotonic components of the product progress measure before considering subspaces where a monotonic component has a higher progress value. This allows persistent states within a subspace to be deleted before progressing to the next subspace and provides for further reduction of peak memory usage.

The compositional sweep-line method also provides the foundation for two approaches to automatic computation of progress measures. For a subsystem S_i specified explicitly as an LTS, a monotonic progress measure for S_i can be obtained by computing the strongly connected component (SCC) graph for S_i. The progress value for a state s is the identity of the SCC to which s belongs, and the partial order on progress value is determined by the reachability relation of the SCC-graph. If the LTS for S_i is strongly connected, then the result is the trivial progress measure since all states are mapped to the same SCC and no peak memory reduction can be obtained. In this case, a non-monotonic progress measure can be obtained by computing a spanning tree and using a topological sorting of the nodes as an ordering consistent with the reachability relation of the spanning tree. A product progress measure for the full system can then be obtained as explained above.

The compositional framework was developed further in [29] to obtain an on-the-fly automata-based approach for verifying safety properties [14] based on finite state automata (FSAa) and language comparison. In this approach, an FSA S_p is used to specify the legal finite executions of the system S. The parallel composition (synchronised product) $S \parallel \overline{S_p}$ between the system S and the complement $\overline{S_p}$ of the property automaton S_p is then computed in order to determine whether the system model exhibits any illegal behaviours. Since S_p is usually given explicitly, the two algorithms discussed above can be used to automatically compute a progress measure for $\overline{S_p}$.

3.3. Behavioural equivalence reduction

Equivalence relations are widely used in state space-based methods for reduction of state spaces. Bisimulation and trace equivalence are (classical) examples of behavioural equivalences. If an equivalence relation on states and on transitions consistent with system behaviour [36] is provided prior to exploration of a state space, then a *condensed state space* can be constructed on-the-fly in which the nodes represent equivalence classes of states and the arcs represent equivalence classes of transitions. The nodes and arcs in a condensed state space are often represented by computing a canonical representative for the corresponds equivalence class. As an example, the symmetry method [19,20] exploits inherent symmetries in the system specification model (e.g., symmetrically behaving processes) to define equivalence relations. In the commit protocol (see Fig. 1), the workers have symmetric behaviour which allows two states to be considered equivalent (symmetric) if one can be obtained from the other by a permutation of the identities of the workers. This implies, e.g., that states 3 and 5, and 4 and 6 (see Fig. 2) are mutually equivalent and that such equivalent states have equivalent sets of immediate successor states.

The combination of the sweep-line method with behavioural equivalence reduction was investigated in [28,38]. When the equivalence relation $\equiv_S$ on states and the progress measure are *compatible*, i.e., for all $s, s' \in S : s \equiv_s s' \Rightarrow \psi(s) = \psi(s')$, then the two methods combine immediately. The only modification required to the sweep-line algorithm in Fig. 3 is to modify lines 1 and 2 such that a canonical representative $\textsc{Canon}(s_I)$ for the equivalence class of the initial state s_I is inserted into Roots and Nodes. Furthermore, in lines $18 - 28$ the canonical representative $\textsc{Canon}(s')$ for the equivalence class of s' is used instead of s'. If the progress measure and the equivalence relation are not compatible, the sweep-line algorithm adapted as described above will still terminate and explore all equivalence classes, but some equivalence classes will be explored multiple times. An equivalence class may be explored multiple times since two states belonging to the same equivalence class may belong to different layers, and hence we may not recognise that a state in the equivalence class has already been explored. In [38] it was shown how the sweep-line method can be used in the case where equivalence reduction is applied to represent an infinite-state system as a finite number of state equivalence classes. Here the concept of *cut-off predicates* was introduced to ensure termination of the sweep-line exploration in cases where the progress measure and equivalences are incompatible, and where the system contains infinite executions with no upper limit on the progress values of the states in the infinite sequence.

3.4. Memory-efficient Kripke structures and model checking

The deletion of predecessor states performed by the sweep-line method means that the method can primarily be used for model checking of safety properties as discussed in Section 3.2. LTL model checking using, e.g., Büchi automata is not immediately possible because acceptance cycles may span multiple layers. Traditional CTL model checking relies on the use of the predecessor relation which is not compatible with the least-progress first exploration of the sweep-line method. An algorithm was developed in [26] that uses the sweep-line method to compute a memory-efficient representation of a state

Table 1
Experimental results — Wireless Transaction Protocol (WTP).

		Sweep-line method			Breadth-first disk		
Config	States	Peak (%)	Path (%)	L	Peak (%)	Time (%)	L
2-T	10,333	26.9	108.0	6	25.9	128.0	5
4-T	65,873	18.8	108.1	6	18.0	129.6	5
6-T	172,657	15.1	112.3	6	13.1	129.6	5
8-T	326,667	13.6	108.0	6	10.3	130.6	5
2-F	24,905	28.9	98.9	6	27.0	116.8	5
4-F	154,231	18.9	110.1	6	18.7	122.5	5
6-F	397,583	14.9	107.9	6	13.6	122.0	5
8-F	748,505	13.4	108.1	6	10.6	121.0	5

space graph. This algorithm can also be used to obtain a memory efficient representation of a Kripke structure [2] for the system which can then in turn be used as a basis for applying standard LTL and CTL model checking algorithms.

The idea is that states that are normally deleted when the sweep-line enters the next layer is instead of a complete deletion replaced by a state number (an integer) and a bit-vector specifying the truth value of each of the atomic propositions $\phi_1, \phi_2, \ldots, \phi_n$ in the temporal logic formula to be checked. The outgoing edges of a state s is represented by an array specifying for each edge a transition index and the state number of the successor state. The detailed state information (e.g., marking of places in case of Petri nets) is no longer stored in memory. With this approach, the Kripke structure corresponding to a state space (V, E) can be stored using $(2w+n)|V|+|E|(\lceil \log_2 |T| \rceil)+\lceil \log_2 |V| \rceil)$ bits [26] where w is the size of a machine word and n is the number of atomic propositions.

The Kripke structure obtained by making a sweep of the state space as outlined above contains more nodes than the state space if regress edges are present. This means that the resulting Kripke structure is an unfolding of the Kripke structure determined by the state space. The following theorem [26,39] shows that the truth value of temporal logic formulae are preserved by the unfolding. That both LTL and CTL are contained in CTL* provides the link to conducting CTL and LTL model checking.

Theorem 4. *Let $G = (V, E)$ be a state space and let ϕ be a CTL* formula with atomic propositions $\phi_1, \phi_2, \ldots, \phi_n$. Let K be the Kripke structure determined by G and let K_s be the Kripke structure obtained using the sweep-line method. Then K and K_s are bisimilar and $K \models \phi \Leftrightarrow K_s \models \phi$.*

The experimental results in [26] demonstrated that memory requirement can be reduced to between 5% and 10% compared to a conventional representation with full state information. The reduction in memory usage comes at an increase in exploration time of typically 150% to 200%.

4. Application examples

In this section we present experimental results from representative case studies on industrial-sized protocols where the sweep-line method has been put into practical use for verification.

The Wireless Application Protocol. Verification of the Wireless Transport Protocol (WTP) was performed in [27,32]. WTP constitutes the transaction layer of the Wireless Application Protocol (WAP) architecture. A monotonic progress measure was defined based on the transaction control flow of the two WTP protocol entities and their re-transmission counters. Furthermore, the path finding technique (see Section 3.1) was used to obtain a path leading to a state where both protocol entities are in their terminating state. The performance of the sweep-line method was compared to a breadth-first disk-based exploration [21].

Selected experimental results are provided in Table 1. Configurations are written in the form X–Y where X specifies the maximum value of the retransmission counters, and Y specifies whether *user acknowledgement* is on (T) or off (F). The WTP specification suggests 4 as the maximum retransmission value for GSM networks and 8 as the maximum value for IP networks, and the goal of the case study was to verify WTP for these parameters. The States column lists the number of reachable states. The Sweep-Line Method columns show the Peak number of states stored with the sweep-line method and the Path column gives time (relative to basic sweep-line state space exploration) used to explore the state space when using the path finding technique. The column L gives the length of the path obtained using the sweep-line path finding technique. The Breadth-first disk columns show the relative performance of a disk-based breadth-first exploration [21] compared to the basic sweep-line method. The Peak column gives in this case the widest breadth-first level encountered during state space exploration, the Time column gives the time used, and *L* is the length of the shortest path leading to the desired state.

It can be seen that the sweep-line method reduces peak memory usage to about 15%–30% with a time overhead of approximately 10%. The overhead represents the cost of computing progress measures for states and deleting states during exploration. The *Path* column shows that there is only a marginal overhead incurred by writing to external storage in conjunction with path finding. It can be seen that this overhead is indeed marginal. The sweep-line method performs better

Table 2
Experimental results — Datagram Congestion Control Protocol (DCCP).

		Sweep-line				Sweep-line	
Config	States	Peak (%)	Time (%)	Config	States	Peak (%)	Time (%)
A	370,721	35.8	105.7	F	537,867	33.9	89.0
B	32,456	30.1	106.6	G	283,516	33.8	91.6
C	194,890	38,2	117.9	H	151,025	38.0	112.3
D	23,657	32.1	115.4	I	251,921	33.6	105.1
E	22,360	33.3	108.3	J	181,952	35.8	106.0

in time than the disk-based breadth-first exploration which on the other hand achieves a slightly better memory reduction. This demonstrates how the two methods trades space and time. The path obtained with the sweep-line method is only one longer than a shortest path and hence in this case the length of a progress shortest path is close to the length of a shortest path.

Datagram Congestion Control Protocol. Application of the sweep-line method on the Datagram Congestion Control Protocol (DCCP) being developed by the Internet Engineering Task Force was investigated in [29]. The verification in [29] focused on the connection establishment procedures and used a monotonic progress measure based on the phases of the protocol entities when establishing a connection, and sequence number state variables. The main goal was to verify that the connection establishment protocol conformed to its service specification. This was done using the on-the-fly method for safety properties (see Section 3.2) using an FSA specifying the service language as the property automata. Table 2 provides selected experimental results for different configurations $A - J$. The States column lists the number of states in the product automaton of the model state space (specifying the protocol language) and the complement of the service FSA (specifying the service language). The Peak column lists the peak number of states in the product automaton when explored with the sweep-line method relative to the number of States. The Time column specifies the time for sweep-line exploration in percentage relative to an ordinary full state space exploration.

The application of the sweep-line method reduced the memory usage for verification of the DCCP protocol to about $1/3$ at an increase in time of 10%–20% – which is representative for many case studies conducted with the sweep-line method. In addition to the configurations listed in Table 2, the sweep-line method enabled verification of configurations that could not be handled with conventional state space exploration of the product automaton. Application of the sweep-line method to the DCCP protocol was further investigated in [33]. Here is was shown that increasing the size of the full state space by augmenting the model (states) with progress information made it possible to further lower the peak memory usage of the sweep-line method.

Audio/Video Lock Management Protocol. The lock management protocol (LMP) of a Bang & Olufsen BeoLink audio/video system was verified using the sweep-line and symmetry method in [28]. The LMP protocol is used to grant devices exclusive access to services, and is based on the notion of a *key* that a device must possess in order to access services in the system. Special devices in the system, called *audio* and *video masters*, are responsible for generating the key. In the case study, the sweep-line and symmetry methods were used in combination (see Section 3.3) to verify that when the BeoLink system is switched on, exactly one key is generated (as required) within 2.0 seconds. Application of the sweep-line method exploited that the LMP protocol was modelled using a timed CPN. The global clock in a timed CPN model increases as transitions are executed, and hence determines a monotonic progress measure. Application of the symmetry method exploited that the identify of non-master devices can be interchanged since these devices are behaviourally equivalent.

Table 3 lists results for the LMP protocol relative to full ordinary state space exploration. The Sweep-Line columns gives the performance of the sweep-line method, the Symmetry columns give the performance of the symmetry method, and the Combined columns indicate the performance when combining the two methods. The AM:n rows correspond to configurations with an audio master (AM) and n devices, and the VM:n rows correspond to configurations with a video master (VM) and n devices.

It can be seen that the combined method achieves significantly better memory reduction than both the sweep-line method and the symmetry method used alone, with a time usage comparable with that achieved by using the symmetry reduction and better than the time usage of the sweep-line method. In [40] the sweep-line method in combination with the symmetry method was applied to the full LMP protocol (not only the initialisation phase). This was based on a non-monotonic progress measure derived from the control loop executed by the devices in the system and by combining it with the use of time equivalence [36].

5. Conclusions and perspectives

The sweep-line method relies on a notion of progress being provided either by the modeller or computed automatically. Practical applications have shown that it is feasible to identify progress in a system, and formalise it in the form of a progress measure. Also, for some formalisms (e.g., timed CPNs) progress is inherent in the formalism. Furthermore, the compositional

Table 3
Experimental results — BeoLink Lock Management Protocol (LMP).

		Sweep-line		Symmetry		Combined	
Config	States	Peak (%)	Time (%)	States (%)	Time (%)	Peak (%)	Time (%)
AM:3	1,839	20.8	102.6	65.3	118.7	13.7	132.4
AM:4	22,675	21.4	99.5	27.4	68.1	5.5	73.3
AM:5	282,399	18.4	102.2	9.0	43.2	1.5	44.3
AM:6	3,417,719	17.7	22.0	2.6	7.6	0.4	8.4
VM:3	1,130	37.9	105.3	67.1	123.1	24.9	135.4
VM:4	13,421	41.2	103.0	28.3	71.4	10.6	77.6
VM:5	164,170	36.2	103.3	9.2	44.0	2.9	48.2
VM:6	1,967,159	35.2	68.5	2.6	24.0	0.7	26.9

framework [37] shows how progress measures can be computed automatically for subsystems provided in the form of an explicit transition system. In [41] an approach to automatically computing progress measures for low-level Petri nets based on invariants was developed and implemented in the LoLA tool [31]. In the context of LoLA, the sweep-line method has also been combined with the use of the stubborn set method [42] demonstrating that the two methods are orthogonal in terms of reduction. A highly relevant area of future work is to investigate how progress measures can be computed, e.g., from the control flow graph of process descriptions. This would be applicable, e.g., in the context of the Promela language and SPIN [8].

In addition to the examples considered in Section 4, the sweep-line method has also been used for the verification of transactions in the Internet Open Trading Protocol (IOTP) [34], and the connection establishment procedures of the Transmission Control Protocol (TCP) [35]. The progress exploited in these protocols is similar to that exploited for the WAP and DCCP protocols. Finally, the sweep-line method has been used for validation of business processes in [43] exploiting progress from the start of a business process towards the termination of the process. The experimental evaluation performed confirms the hypothesis that the sweep-line method typically reduces the memory requirement to 10%–30% (compared to the full state space) and that it seldom explores each state more than twice leading to an acceptable increase in run-time despite the poor theoretic worst-case complexity in terms of state re-explorations.

The sweep-line method is in its present form primarily suited for verification of safety properties either expressed as state predicates [25] or a finite state automaton [29]. The derivation of a Kripke structure using the approach in [26] opens up for more general CTL and LTL model checking, but a sweep-line method for general CTL and LTL model checking where states are truly deleted from memory is currently under investigation based upon approaches to LTL model checking developed in the context of external memory [44] and distributed state space exploration [45]. A main challenge is to handle cycles in the state space that span multiple layers.

Acknowledgements

The sweep-line method has been influenced by several co-authors of papers. In particular, we acknowledge the contributions of Jonathan Billington, Søren Christensen, Guy E. Gallasch, and Michael Westergaard.

References

[1] C. Baier, J.-P. Katoen, Principles of Model Checking, MIT Press, 2008.
[2] E. Clarke, O. Grumberg, D. Peled, Model Checking, The MIT Press, 1999.
[3] E. Clarke, E. Emerson, J. Sifakis, Turing lecture: model checking – algorithmic verification and debugging, Communications of the ACM 52 (2009) 74–84.
[4] E.A. Emerson, Temporal and Modal Logic, in: Handbook of Theoretical Computer Science, vol. B, Elsevier, 1990, pp. 995–1072. (chapter 16).
[5] W. Reisig, Petri Nets — An Introduction, in: EATCS Monographs on Theoretical Computer Science, vol. 4, Springer, 1985.
[6] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.
[7] R. Alur, D. Dill, A Theory of Timed Automata, TCS 126 (2) (1994) 183–235.
[8] G.J. Holzmann, The SPIN Model Checker, Addison-Wesley, 2003.
[9] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, P. Šimeček, DiVinE — A Tool for Distributed Verification, in: CAV, in: LNCS, vol. 4144, Springer, 2006, pp. 278–281.
[10] Ratzer, et al., Cpn tools for editing, simulating, and analysing coloured petri nets, in: ICATPN, in: LNCS, vol. 2679, Springer, 2003, pp. 450–462.
[11] K. McMillan, Symbolic Model Checking, Kluwer, 1993.
[12] A. Biere, A. Cimatti, E. Clarke, M. Fujita, Y. Zhu, Symbolic model checking using SAT procedures instead of BDDs, in: DAC, 1999, pp. 317–320.
[13] J. Esparza, S. Römer, W. Vogler, An improvement of McMillan's unfolding algorithm, Formal Methods in System Design 20 (3) (2002) 285–310.
[14] A. Valmari, The state explosion problem, in: Lectures on Petri Nets I: Basic Models, in: LNCS, vol. 1491, Springer, 1998, pp. 429–528.
[15] E.M. Clarke, O. Grumberg, M. Minea, D. Peled, State space reduction using partial order techniques, STTT 2 (3) (1999) 279–287.
[16] M. Westergaard, et al., The ComBack method – extending hash compaction with backtracking, in: ICATPN, in: LNCS, vol. 4546, Springer, 2007, pp. 445–464.
[17] G. Holzmann, An analysis of bitstate hashing, Formal Methods in System Design 13 (1998) 289–307.
[18] U. Stern, D. Dill, Improved probabilistic verification by hash compaction, in: CHARME, in: LNCS, vol. 987, Springer, 1995, pp. 206–224.
[19] E. Clarke, E. Emerson, S. Jha, A.P. Sistla, Symmetry reductions in model checking, in: CAV, in: LNCS, vol. 1427, Springer, 1998, pp. 147–158.
[20] K. Jensen, Condensed state spaces for symmetrical coloured petri nets, Formal Methods in System Design 9 (1/2) (1996) 7–40.

[21] U. Stern, D. Dill, Using magnetic disk instead of main memory in the Murphi verifier, in: CAV, in: LNCS, vol. 1427, Springer, 1998, pp. 172–183.
[22] P. Godefroid, G.J. Holzmann, D. Pirottin, State-space caching revisited, Formal Methods in System Design 7 (3) (1995) 227–241.
[23] G. Behrmann, K. Larsen, R. Pelnek, To store or not to store, in: CAV, in: LNCS, vol. 2725, Springer, 2003, pp. 433–445.
[24] S. Christensen, L. Kristensen, T. Mailund, A sweep-line method for state space exploration, in: TACAS, in: LNCS, Vol. 2031, Springer, 2001, pp. 450–464.
[25] L. Kristensen, T. Mailund, A generalised sweep-line method for safety properties, in: FME, in: LNCS, vol. 2391, Springer, 2002, pp. 549–567.
[26] T. Mailund, M. Westergaard, Obtaining memory-efficient reachability graph representations using the sweep-line method, in: TACAS, in: LNCS, vol. 2988, Springer, 2004, pp. 177–191.
[27] L. Kristensen, T. Mailund, Efficient path finding with the sweep-line method using external storage, in: ICFEM, in: LNCS, vol. 2885, Springer, 2003, pp. 319–337.
[28] J. Billington, G. Gallasch, L. Kristensen, T. Mailund, Exploiting equivalence reduction and the sweep-line method for detecting terminal states, IEEE Transactions on SMC — Part A 34 (1) (2004) 23–38.
[29] G.E. Gallasch, J. Billington, S. Vanit-Anunchai, L. Kristensen, Checking safety properties on-the-fly with the sweep-line method, STTT 9 (3–4) (2007) 371–392.
[30] M. Westergaard, S. Evangelista, L. Kristensen, ASAP: an extensible platform for state space analysis, in: ICATPN, in: LNCS, vol. 5606, Springer, 2009, pp. 303–312.
[31] K. Schmidt, LoLA: a low level analyser, in: ICATPN, in: LNCS, vol. 1825, Springer, 2000, pp. 465–474.
[32] S. Gordon, L. Kristensen, J. Billington, Verification of a revised WAP wireless transaction protocol, in: ICATPN, in: LNCS, vol. 2360, Springer, 2002, pp. 182–202.
[33] S. Vanit-Anunchai, J. Billington, G.E. Gallasch, Analysis of the datagram congestion control protocols connection management procedures using the sweep-line method, STTT 10 (1) (2008) 29–56.
[34] G. Gallasch, C. Ouyang, J. Billington, L. Kristensen, Experimenting with progress mappings for the sweep-line analysis of the internet open trading protocol, in: CPN, 2004, pp. 19–38.
[35] G.E. Gallasch, B. Han, J. Billington, Sweep-Line analysis of TCP connection management, in: ICFEM, in: LNCS, vol. 3785, Springer, 2005, pp. 156–172.
[36] K. Jensen, L. Kristensen, Coloured Petri Nets — Modelling and Validation of Concurrent Systems, Monograph, Springer, 2009.
[37] L. Kristensen, T. Mailund, A compositional sweep-line state space exploration method, in: FORTE, in: LNCS, vol. 2529, Springer, 2002, pp. 327–343.
[38] T. Mailund, Analysing infinite-state systems by combining equivalence reduction and the sweep-line method, in: ICATPN, in: LNCS, Vol. 2360, Springer, 2002, pp. 314–333.
[39] T. Mailund, Sweeping the state space a sweep-line state space exploration method, Ph.D. Thesis, Computer Science Department, University of Aarhus, 2003.
[40] L.M. Kristensen, J.B. Jørgensen, K. Jensen, Application of Coloured Petri Nets in System Development, in: Proc. of 4th Advanced Course on Petri Nets, in: LNCS, Vol. 3098, Springer, 2004, pp. 626–685.
[41] K. Schmidt, Automated generation of a progress measure for the sweep-line method, in: TACAS, in: LNCS, vol. 2988, Springer, 2004, pp. 192–204.
[42] A. Valmari, A stubborn attack on state explosion, in: Proc. of CAV'90, in: LNCS, vol. 531, Springer, 1990, pp. 156–165.
[43] B. Mitchell, L.M. Kristensen, L. Zhang, Formal specification and state space analysis of an operational planning process, STTT 9 (3–4) (2007) 255–267.
[44] J. Barnat, L. Brim, P. Simecek, M. Weber, Revisiting resistance speeds up i/o efficient LTL model checking, in: TACAS, in: LNCS, vol. 4963, Springer, 2008, pp. 48–62.
[45] L. Brim, I. Cerná, P. Moravec, J. Simsa, Accepting predecessors are better than back edges in distributed ltl model-checking, in: FMCAD, in: LNCS, vol. 3312, Springer, 2004, pp. 352–366.