A Sweep-Line Method for State Space Exploration

Søren Christensen[1], Lars Michael Kristensen[1,2*], and Thomas Mailund[1]

[1] Department of Computer Science, University of Aarhus
IT-parken, Aabogade 34, DK-8200 Aarhus N., DENMARK,
{schristensen,lmkristensen,mailund}@daimi.au.dk

[2] School of Electrical and Information Engineering, University of South Australia
Mawson Lakes Campus, SA 5095, AUSTRALIA
lars.kristensen@unisa.edu.au

Abstract. We present a state space exploration method for on-the-fly verification. The method is aimed at systems for which it is possible to define a measure of progress based on the states of the system. The measure of progress makes it possible to delete certain states on-the-fly during state space generation, since these states can never be reached again. This in turn reduces the memory used for state space storage during the task of verification. Examples of progress measures are sequence numbers in communication protocols and time in certain models with time. We illustrate the application of the method on a number of Coloured Petri Net models, and give a first evaluation of its practicality by means of an implementation based on the Design/CPN state space tool. Our experiments show significant reductions in both space and time used during state space exploration. The method is not specific to Coloured Petri Nets but applicable to a wide range of modelling languages.

1 Introduction

State space exploration has proven to be powerful for investigating the correctness of concurrent systems. The basic idea behind state space exploration is to construct a directed graph, called the *state space*, in which the nodes correspond to the set of reachable states of the system, and the arcs correspond to the state changes. Such a state space represents all possible executions of the system, and can be used to algorithmically verify and analyse an abundance of properties about the system under consideration.

The main disadvantage of using state spaces is the *state explosion problem*: even relatively small systems may have an astronomical number of reachable states, and this is a serious limitation to the use of state space methods in the analysis of real-life systems. This has led to the development of many different reduction methods for alleviating the state explosion problem. Examples of reduction methods are partial order reduction methods [19, 21, 22], the symmetry

* Supported by the Danish Natural Science Research Council.

method [5,6,14], and the unfolding method [7,17]. Reduction methods represent the full state space in a compact or condensed form, or represent only a subset of the full state space. The reduction is done such that the answer to the verification questions can still be determined from the reduced state space.

Reduction methods typically exploit certain characteristics of the system, and hence work well for systems possessing these characteristics, but fail to work well for systems which do not have these characteristics. An example of this is the symmetry method which exploits the symmetry present in many concurrent systems, but fails to work on systems which do not possess symmetry. This paper presents a *sweep-line method* for state space exploration. The method is aimed at systems for which it is possible to define a measure of progress based on the states of the system. Examples of such progress measures are sequence numbers in communication protocols and time in certain systems with time. The key property of a progress measure is that for a given state s, all states reachable from s have a progress measure which is greater than or equal to the progress measure of s. The progress measure will often be specific for the system under consideration. However, for some modelling languages a progress measure can be defined based on only the modelling language itself. The progress measure then applies to all models of systems constructed in that modelling language. Time in Coloured Petri Nets [13,16] is an example of such a progress measure.

A progress measure makes it possible to delete certain states on-the-fly during state space generation, since it ensures that these states can never be reached again. Since we only delete states that can never be reached again, we never risk processing the same state twice. The sweep-line method therefore ensures that the generation terminates for finite-state systems after having visited all reachable states. Intuitively, the idea is to drag a *sweep-line* through the full state space, calculate the reachable states in front of the sweep-line and delete states behind the sweep-line.

The sweep-line method makes it possible to investigate a number of interesting properties of systems, such as deadlocks, reachability, and safety properties. Practical experiments with a prototype implementation of the sweep-line method show significant savings in both time and space for finite state systems. For infinite-state systems the sweep-line method can be used to explore and analyse larger prefixes of the state space than can be done with ordinary state space exploration.

Deleting and/or throwing state information away on-the-fly during state space generation, is also the underlying idea of the *bit-state hashing method* [10,11] and the *state space caching method* [9,12]. The basic idea of state space caching is to make a depth-first generation of the state space, keep the states of the depth-first search stack in memory, and allow for deletion of states during the state space generation which are not on the depth-first search stack. An advantage of the sweep-line method compared to state space caching is that states are only generated and processed once. With the state space caching method, the same state may be regenerated and processed several times leading to an increase in run-time. As shown in [8] this run-time penalty can be fought against by combining state space caching and partial order methods. The bit-state hashing method always keeps the states of the depth-first search stack in memory, but reduces (in its simplest form) the information stored about a single state to a hash value (index value). This hash value is then stored using a single bit in a bit-vector. A main difference between bit-state hashing and the sweep-line method is that with the sweep-line method full coverage of the state space is guaranteed. This is not the case with bit-state hashing, since two different states may be mapped to the same hash value due to hash collisions.

The idea of using a system specific progress measure to improve state space analysis is due to Kurt Jensen. The detailed realisation presented in this paper is the responsibility and work of the authors alone. The idea of utilising a measure of progress to delete states during on-the-fly verification is intriguingly simple, but to the best of our knowledge it has never been documented before.

The rest of this paper is organised as follows. Section 2 gives an informal introduction to the sweep-line method using a small Coloured Petri Net model as an example. Section 3 formalises the concept of progress measure. Section 4 gives the generic state space construction algorithm for the sweep-line method. Section 5 shows how this generic algorithm can be tailored for on-the-fly verification. Section 6 contains additional examples of progress measures and gives some numerical data on the performance of the sweep-line method using these examples. Section 7 contains the conclusions and a further discussion of related work.

2 The Sweep-Line Method

In this section we informally introduce the sweep-line method using a small example of a sliding window communication protocol [1]. The protocol makes efficient use of the network by sending a certain number of packets, called the window size, before it waits for an acknowledgement from the receiver. The packets transmitted on the network include a cyclic sequence number used to order the packets received from the network correctly at the receiver. For efficiency reasons the number of bits reserved for this counter must be kept at a minimum, but reserving too few bits can result in erroneous acceptance of packets by the receiver. We assume that the network allows both overtaking and loss of packets. The purpose of the analysis is to investigate whether the design of the protocol is sufficiently robust to work correctly under these conditions.

We have modelled the sliding window protocol using Coloured Petri Nets (CP-nets or CPNs) [13, 16], but the sweep-line method is not specific to this formalism. Our aim here is not to explain the CPN model of the sliding window protocol in great detail, but just to use it for introducing the basic ideas of the sweep-line method.

Figure 1 gives an overview of the CPN model of the sliding window protocol. It consists of three modules corresponding to a Sender, a Network, and a Receiver. In the CPN model we have supplemented the cyclic sequence number with a non-cyclic counter specifying the *generation* of the cyclic counter. This allows us to detect if the receiver ever is in a situation where a faulty acceptance of a packet occurs.

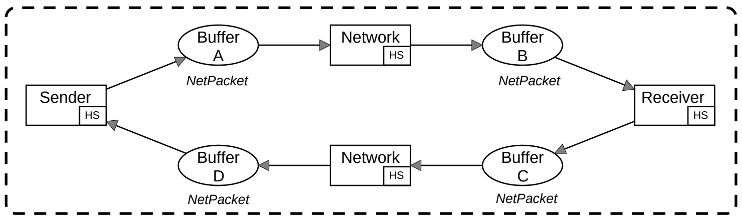

Fig. 1. Module overview of the sliding window protocol.

The left-hand side of Fig. 2 shows the sender module which has a single counter (NextSend) specifying the sequence number of the next packet to be sent. The idea of the window mechanism is that it allows the sender to send a number of packets without receiving an acknowledgement. Whenever the sender receives an acknowledgement from the receiver, the window is updated accordingly.

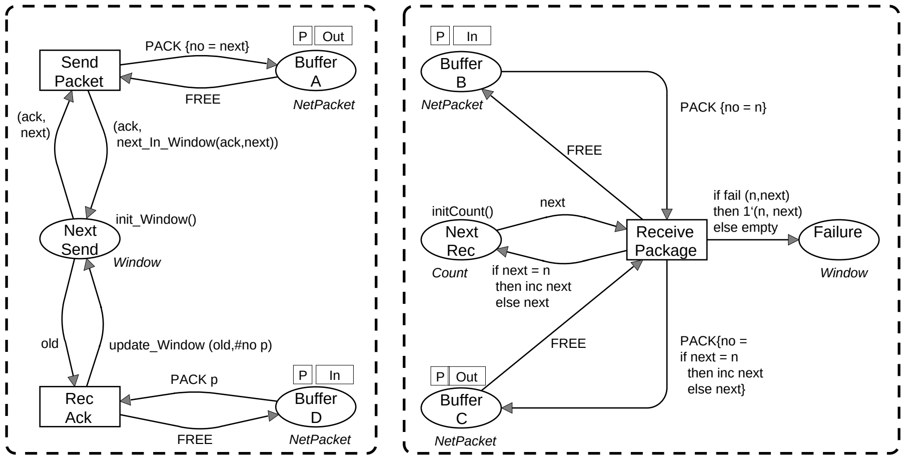

Fig. 2. Sender module (left) and Receiver module (right).

The right-hand side of Fig. 2 shows the receiver module, which has a counter (NextRec) specifying the next packet which can be accepted. NextRec is a counter on the form (gen, seq), where gen is an integer giving the current generation, and seq is the sequence number of the packet within the generation. If the expected packet arrives, the counter is increased and an acknowledgement with the sequence number of the next packet is sent back to the sender. If a packet arrives out of order, the receiver will respond with an acknowledgement containing the sequence number of the packet it was expecting to receive. In the CPN model a check has been inserted to inspect the validity of the packets being successfully received. This means that the Failure state will be marked if and only if the receiver ever accepts an invalid packet.

The aim of the analysis is to check whether the cyclic sequence numbers are too small, i.e., whether the Failure state of the receiver will ever be marked. Using conventional state spaces generation, a straightforward way to investigate this property would be to generate the state space and check for each newly created state whether the place Failure is marked. If a state is detected where the place Failure is marked, generation can be terminated and an error reported. For the sliding window protocol example it is, however, possible to exploit the NextRec counter in the receiver to reduce the memory used for state space storage during the search. The basic observation is that the NextRec counter has the property that as the protocol executes, generation numbers are increasing and so are the sequence numbers within each generation. This in turn makes it possible to define a *progress measure* on the states by means of which the progress of the sliding window protocol can be quantified, and which makes it possible to compare states wrt. their progress measure. Let for a state s, (gen_s, seq_s) denote the value of the receiver's NextRec counter in s. We can then talk about a state s representing a state where the protocol has progressed further than in another state s' (written $s' \leq s$) if and only if $(gen_{s'} < gen_s) \vee ((gen_{s'} = gen_s) \wedge (seq_{s'} \leq seq_s))$. Since generation numbers are increasing and so are the sequence numbers within each generation, the above progress measure has the property that for all successors s'' of a state s, $s \leq s''$. This means that from a state s, it is never possible to reach a state s'' with a progress measure less than the progress measure of s.

In conventional state space generation, the states are kept in memory to recognise already examined states. However, states which have a progress measure which is strictly less than the minimal progress measure of those states for which successors have not yet been calculated can never be reached again. It is therefore safe to delete such states. Saving memory by deleting such states is the basic idea underlying the sweep-line method. The role of the progress measure is to be able to recognise such states.

Figure 3 illustrates the intuition behind the sweep-line method. s_0 denotes the initial state of the system. The two gray areas show the states kept in memory. Some of these have been processed (light gray), i.e., their successor states have been calculated, and some have only been calculated (dark gray). There is a *sweep-line* through the stored states separating the states with a progress measure which is strictly less than the minimal progress measure among the unprocessed states, from the states which have progressed further than the minimal unprocessed states. States strictly to the left of the sweep-line can never be reached from the unprocessed states and can therefore safely be deleted. As the state space generation proceeds, the sweep-line will move from left to right. We drag the sweep-line through the state space, calculating the reachable states in front of the sweep-line and deleting states behind the sweep-line. All states on the sweep-line have the same progress measure.

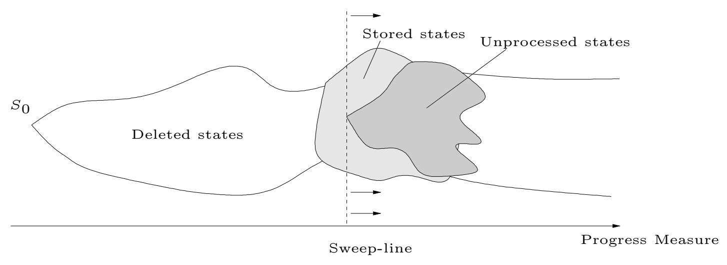

Fig. 3. The sweep-line method.

The full state space of the sliding window protocol with 5 packets has 28,438 nodes. If the sweep-line method is applied, using our prototype implementation, then at most 8,622 nodes are stored at any moment during state space generation. This means that the memory consumption measured in the number of nodes is reduced by almost 70%. Moreover, the time used for keeping track of progress measures and deleting states on-the-fly is more than compensated for by the time gained in faster insertion of new states in the state space. We will provide further statistics for the application of the sweep-line method on the sliding window protocol in Sect. 6, where we also give other examples of applications of the sweep-line method.

3 Progress Measures

In this section we formalise the notion of *progress measure* which is the fundamental concept underlying the sweep-line method. We assume that the systems we are considering can be characterised as a tuple $\mathcal{M} = (\mathbb{S}, T, \Delta, s_0)$, where $\mathbb{S}$ is the set of *states*, T is the set of *transitions*, $\Delta \subseteq \mathbb{S} \times T \times \mathbb{S}$ is the *transition relation*, and s_0 is the *initial state*. Most models of concurrent systems including CPN models, fall into this category of systems.

Let $s, s' \in \mathbb{S}$ be two states and $t \in T$ a transition. If $(s, t, s') \in \Delta$ 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_n is *reachable* from a state s_1 iff there exists states $s_2, s_3, \ldots, s_{n-1}$ and transitions $t_1, t_2, \ldots t_{n-1}$ such that $(s_i, t_i, s_{i+1}) \in \Delta$ for $1 \leq i \leq n-1$. If a state s' is reachable from a state s we write $s \to^* s'$. For a state s, $reach(s) = \{ s' \in \mathbb{S}, | s \to^* s' \}$ denotes the set of states reachable from s. The set of *reachable states* of $\mathcal{M}$ is then $reach(s_0)$. The *state space* of a system is the directed graph (V, E) where $V = reach(s_0)$ and $E = \{(s, t, s') \in \Delta \,|\, s, s' \in V\}$.

A *progress measure* specifies a *partial order* $(O, \sqsubseteq)$ on the states of the system. A partial order $(O, \sqsubseteq)$ consists of a set O and a relation $\sqsubseteq \subseteq O \times O$ which is reflexive, transitive, and antisymmetric. Moreover, the partial order is required to preserve the reachability relation $\to^*$ of the system:

Definition 1. *A **progress measure** is a tuple $\mathcal{P} = (O, \sqsubseteq, \psi)$ such that $(O, \sqsubseteq)$ is a partial order and $\psi : \mathbb{S} \to O$ is a mapping from states into O satisfying: $\forall s, s' \in reach(s_0) : s \to^* s' \Rightarrow \psi(s) \sqsubseteq \psi(s')$.* □

It is worth noting that the definition of progress measure implicitly states that for all $s \in reach(s_0) : \psi(s_0) \sqsubseteq \psi(s)$, i.e., the initial state is minimal among the reachable states with respect to the progress measure. We only require $s \to^* s' \Rightarrow \psi(s) \sqsubseteq \psi(s')$ for reachable states $s, s' \in reach(s_0)$. In general we cannot determine whether $s \in reach(s_0)$ for arbitrary states $s \in \mathbb{S}$ without calculating the full state space. Hence in practice, a conservative approach is to ensure the property for all states in $\mathbb{S}$. In general we cannot determine whether $s \to^* s'$ either, without calculating the state space, but for some systems it is possible to determine statically from the model that for all transitions t and all states $s, s' \in \mathbb{S}$ we have $s \xrightarrow{t} s' \Rightarrow \psi(s) \sqsubseteq \psi(s')$ and hence by transitivity that $s \to^* s' \Rightarrow \psi(s) \sqsubseteq \psi(s')$. Since progress measures in the general case will be user-specified, they can be erroneous. The mapping $\psi : \mathbb{S} \to O$ could violate $s \to^* s' \Rightarrow \psi(s) \sqsubseteq \psi(s')$. However, since $s \to^* s' \Rightarrow \psi(s) \sqsubseteq \psi(s')$ for all $s, s' \in reach(s_0)$ iff $s \xrightarrow{t} s' \Rightarrow \psi(s) \sqsubseteq \psi(s')$ for all $s, s' \in reach(s_0)$ and all $t \in T$, this property can easily be checked during the state space exploration. When we process the enabled transitions in a state s, we check that all successor states have a progress measure greater than or equal to s.

All models have progress measures. A trivial progress measure is $O = \{\bot\}$ (the one-point set), $\sqsubseteq = \{(\bot, \bot)\}$ (the order on that set), and $\psi(s) = \bot$ for all $s \in \mathbb{S}$. However, this progress measure offers no reduction of the state space. Another progress measure is $O = \mathbb{S}_{\text{SCC}}$, the set of *strongly connected components* of the state space, $\sqsubseteq = \to^*_{\text{SCC}}$, the reachability relation between the strongly connected components, and $\psi(s) = SCC(s)$, i.e., ψ maps a state into the strongly connected component to which it belongs. This progress measure offers maximal reduction for the sweep-line method, but since in general we cannot compute the strongly connected component to which a state belongs without calculating the state space, this progress measure is of little practical interest. What is needed is a non-trivial progress measure that can be computed based on the individual states alone, i.e., no knowledge of the state space is required. One example of this is the progress measure of the sliding window protocol defined in the previous section.

4 State Space Exploration

Exploring the state space with the sweep-line method is based on the algorithm used for conventional state space construction. Figure 4 shows the standard algorithm for constructing a state space. It works on three sets: Nodes, the set of nodes/states in the state space; Edges, the set of edges in the state space; and Unprocessed the set of states that has been reached so far, but which have not been further processed, i.e., their successor states have not been calculated.

The sweep-line state space generation algorithm is derived from the standard algorithm by adding *garbage collection*. At certain intervals we delete all states

```
1: Unprocessed.Add($s_0$)
2: Nodes.Add($s_0$)
3: while $\neg$ Unprocessed.Empty() do
4:   $s \leftarrow$ Unprocessed.GetNext()
5:   for all $(t, s')$ such that $s \xrightarrow{t} s'$ do
6:     Edges.Add($s, t, s'$)
7:     if $\neg$ Nodes.Contains($s'$) then
8:       Nodes.Add($s'$)
9:       Unprocessed.Add($s'$)
10:    end if
11:  end for
12: end while
```

Fig. 4. Generic algorithm for state space exploration.

that have a progress measure which is strictly less than all states in Unprocessed, and at the same time we delete all edges connecting deleted states. If the order is total, as was the case in Sect. 2, there will be one minimal progress measure among the unprocessed states, and it suffices to compare the progress measure of a state to that minimal progress measure. Since in general $\sqsubseteq$ is only required to be a partial order, it is possible for all unprocessed states to have different, incomparable progress measures. Hence, in worst case the progress measure of a state needs to be compared to each of them in order to determine whether it can be deleted or not.

When to garbage collect can be decided in different ways. Garbage collecting in each loop is likely to be too time consuming and the number of states that can be deleted in each iteration is likely to be small. Collecting at fixed intervals can suffer from the same problems, but it is less likely. If the intervals are chosen to be too large, however, there may not be sufficient memory to store all states calculated between two collections. This tradeoff can be adjusted dynamically, by determining when to collect based on available memory and/or the amount of memory reclaimed in previous collections. When memory is scarce, the interval should be decreased, when little memory is reclaimed per collection, the interval should be increased. In our prototype implementation (see Sect. 6), garbage collection is done whenever a fixed, user specified, number of nodes have been added to the state space. The results obtained with this simple strategy were quite satisfactory for our experiments, so we have not yet experimented with other strategies.

To maximise the number of states that can be deleted in each garbage collection, the minimal unprocessed states should have progressed as much as possible. Therefore, the method GetNext on Unprocessed (which return the next state to process) should preferably always return a state with a minimal progress measure. If the progress measure maps to a total order, as has been the case for all our applications, then Unprocessed can be implemented as a priority queue using the progress measure as the priority, and $\sqsubseteq$ as the ordering. In this case a breadth-first generation based on progress measure is obtained. With other progress measures, other data structures might be needed to implement UNPROCESSED efficiently.

It is worth observing that the progress measure ensures that all states of a strongly connected component will be garbage collected at the same time. The reason for this is that nodes in the same strongly connected component have the same progress measure as a consequence of Def. 1. This means that an efficient way to capture strongly connected components is to do this as an integrated part of the garbage collection algorithm. This is interesting since strongly connected components are used to check certain properties of systems in an efficient way.

5 Checking Properties

The sweep-line method garbage collects nodes shortly after having created them. Hence, to obtain verification results, properties must be checked on-the-fly. In this section we show how the sweep-line method can be used to verify a number of standard behavioural properties of systems. The properties considered do not represent an exhaustive list of properties which can be verified with the sweep-line method. Here, we have chosen a set of behavioural properties which in our experience constitute properties which are often of interest for the analysis of systems. Combining the sweep-line method with more general temporal logic model checking is beyond the scope of this paper.

A *deadlock* is a state in which no transitions are enabled. Using the algorithm in Fig. 4, the state s could be examined between line 4 and line 5, and if there are no enabled transitions in s it should be reported as a deadlock.

Checking if a state satisfying a given predicate is *reachable* is straightforward. The sweep-line method guarantees that each reachable state is visited at some point. If one of the visited states satisfies the predicate we answer "yes", if none satisfy the predicate we answer "no". Checking a *safety property* means checking that all reachable states satisfy a given predicate. This amounts to checking that the negation of the predicate is not reachable.

Checking that a given predicate is a *home predicate* means checking that from all reachable states, it is possible to reach a state where the predicate holds. A typical predicate could check if a given transition is enabled. Home predicates can be decided using the strongly connected components, i.e., a predicate is a home predicate iff each terminal strongly connected component contains at least one state satisfying the predicate. Since all states in a strongly connected component have the same progress measure, no part of a strongly connected component can be garbage collected before the entire strongly connected component has been computed. Checking this kind of property requires only that terminal strongly connected components are analysed before they are deleted. When we garbage collect, we calculate the strongly connected components of the states we are about to delete, isolate the terminal strongly connected components and check whether they contain a state satisfying the predicate.

When a deadlock or a violation of some property is detected, a trace leading to the deadlock or a state violating the property should be reported. With the sweep-line method, reporting such a trace is complicated by the fact that states between the initial state and the state at the end of the trace might have been garbage collected. These states need to be re-computed. In some cases it is possible to calculate transitions "backwards", i.e., from a state s it is possible to calculate all states s' and transitions t such that $s' \xrightarrow{t} s$. In such cases, a trace can be found by searching backwards for the initial state. Since all reachable states have a progress measure greater than or equal to the initial state, searching can be stopped along a given path if the progress measure drops below this value.

In many cases, however, it is not possible to search backwards. In such cases, the following scheme can be used. When garbage collecting, we make sure that for any unprocessed state s we have at least one predecessor s' with a progress measure strictly less than that of s among the states we do not garbage collect. In this way, when we need to calculate a trace leading to state s, the intersection between NODES and the set of predecessors of s is non-empty. In this intersection, there is a set of minimal states. We can start a search for one of these, using the sweep-line method. When one is found, we construct the last part of the trace as the path from this state to s. The state we find will have a set of predecessors stored in NODES, and we can start a search for one of these. This can then be iterated until we have built a path from s_0 to s. In each iteration, the distance to s_0 is shortened by at least one, so the algorithm is guaranteed to terminate.

6 Experimental Results

A prototype of the sweep-line method has been implemented based on the state space tool of DESIGN/CPN [4]. In this section we give a first evaluation of the practicality of the sweep-line method by applying this prototype on three examples. The first example is the sliding window protocol from Sect. 2. The second example is a stop-and-wait communication protocol taken from [16]. The third example is taken from the industrial case-study [3] in which state spaces of timed CP-nets and the DESIGN/CPN tool were used to validate vital parts of the B&O BeoLink system. All results in this section were obtained using a Pentium II 166 Mhz PC with 160 Mb of memory.

The prototype implementation uses a simple algorithm for initiating a garbage collection during the sweep: Whenever n new states have been added to the state space, a garbage collection is initiated. The garbage collection is implemented as a copying collector: when collecting, the states that should not be deleted are copied into a new state space. This new state space then becomes the current state space, and the old state space is deleted. This scheme was chosen for its simplicity, but it has the drawback that it requires space for two copies of the states that are not deleted. This problem can be avoided using other garbage collection techniques, but for our experiments the copy collection proved sufficient.

Sliding Window Protocol. Table 1 lists statistics for the application of the sweep-line method for different configurations of the sliding window protocol from

Table 1. Experimental results – Sliding Window Communication Protocol.

	Full State Spaces		Sweep-Line Method		Reduction	
Packets	States	Time	States	Time	States	Time
5	28,438	0:02:39	8,622	0:01:26	69.7 %	45.9 %
10	60,013	0:08:48	8,622	0:03:26	85.6 %	61.0 %
15	91,588	0:21:15	8,622	0:05:26	90.6 %	74.4 %
20	123,163	0:33:16	8,622	0:07:36	93.0 %	77.7 %
25	154,738	0:51:10	8,622	0:09:22	94.4 %	77.9 %

Sect. 2. The results were obtained by garbage collecting for each 2000 new states. The table consists of four main columns. The Packets column gives the configuration under consideration, i.e., the number of packets that the sender is requested to send to the receiver. The Full State Spaces column gives the number of states in the full state space and the CPU seconds it took to generate it. The Sweep-Line Method column gives the maximal number of states stored during the sweep, and the time it took to make the sweep through the state space. The Reduction column compares the sweep-line method to full state spaces by giving the reduction obtained in terms of states which had to be stored, and the reduction in run-time. E.g., for 5 packets a reduction of 69.7 % is obtained in the number of states, and a reduction of 45.9 % in run-time.

Table 1 shows that the use of the sweep-line method saves both memory and time. Moreover, the savings obtained grew with the system configuration. It was to be expected that the sweep-line method would reduce the memory consumption since states are being deleted during generation of the state space. However, from the results listed in Table 1 it follows that for the sliding-window protocol the sweep-line method is also faster than full state spaces. Hence, in this case the overhead added by having to delete states is accounted for by having significantly fewer states to compare with in the hash collision list when a new state has been generated and is to be inserted in the state space.

Stop-and-wait Communication Protocol. The stop-and-wait protocol contains a single sender and a single receiver. Data packets are to be transmitted from sender to receiver such that each packet is received exactly once, and in the right order. Each packet contains a sequence number. This number is increased for each new packet and the sequence number is never reset. Thus the sequence number of the packet currently being transmitted by the sender can be used as a progress measure.

Table 2 lists statistics for the application of the sweep-line method for different configurations of the stop-and-wait protocol. The results were obtained when garbage collecting after each 2000 new states. The table consists of the same four main columns as the table for the sliding window protocol.

BeoLink System. The Bang & Olufsen BeoLink system makes it possible to distribute audio and video throughout a home via a network. The state space

Table 2. Experimental results – Stop-and-Wait Communication Protocol.

	Full State Spaces		Sweep-Line Method		Reduction	
Packets	States	Time	States	Time	States	Time
10	2,576	0:00:04	2,002	0:00:04	22.3%	0.0%
50	13,416	0:00:29	2,278	0:00:21	83.0%	27.6%
100	26,966	0:01:20	2,278	0:00:43	91.6%	46,2%
200	54,066	0:04:04	2,281	0:01:29	95.8%	63.5%
300	81,166	0:08:36	2,282	0:02:17	97.2%	73.4%
400	108,266	0:14:23	2,282	0:03:05	97.9%	78.6%
500	135,366	0:21:57	2,282	0:03:58	98.4%	81.9%
1000	270,866	1:21:10	2,284	0:08:51	99.2%	89.1%

analysis in [3] focused on the *lock management protocol* of the BeoLink system. This protocol is used to grant devices exclusive access to various services in the system. The exclusive access is implemented based on the notion of a *key*. A device is required to possess the key in order to access services. When the system boots no key exists, and the lock management protocol is (among other things) responsible for ensuring that a key is generated when the system starts. It is the obligation of the so-called *video* or *audio master* device to ensure that new keys are generated when needed. Timed CP-nets were applied in [3] since timing is crucial for the correctness of the lock management protocol. Here we used the sweep-line method to verify that when the BeoLink system starts eventually a key is generated by the lock management protocol.

The progress measure used for the CPN model of the BeoLink system is based on the time concept of CP-nets. The progress measure is based on the fact that timed CP-nets have a global clock giving the current model time, and that time cannot go backwards in a timed CP-net. That time always progresses is a general property of the time concept of CP-nets, and as a consequence, time can be used as a progress measure on timed CPN models of systems. This is an example of a progress measure being given by the formalism, rather than being specific to individual models.

Table 3 lists statistics for the verification of the initialization phase of the BeoLink system for different configurations of the BeoLink system. The Config column specifies the configuration in question. Configurations with one video master are written on the form VM: n, where n is the total number of devices in the system. Configurations with one audio master are written on the form AM: n. The results were obtained when garbage collecting after each 3000 new states.

7 Conclusion

In this paper we have presented a sweep-line method for alleviating the state explosion problem. The method relies on the notion of progress measure combined with breadth-first state space generation. The experimental results ob-

Table 3. Experimental results – BeoLink System.

	Full State Spaces		Sweep-Line Method		Reduction	
Config	States	Time	States	Time	States	Time
VM: 3	1,130	0:00:06	1,130	0:00:06	0.0 %	0.0 %
AM: 3	1,839	0:00:11	1,839	0:00:11	0.0 %	0.0 %
VM: 4	13,421	0:02:40	5,170	0:02:50	61.5 %	-6.0 %
AM: 4	22,675	0:05:32	5,170	0:04:39	77.2 %	16.0 %
VM: 5	164,170	2:30:27	35,048	1:08:28	78.7 %	54.5 %
AM: 5	282,399	5:03:53	35,048	1:59:39	87.6 %	60.6 %

tained with a prototype implementation of the method have been encouraging, and demonstrated significant savings in memory as well as in time.

The aim of this paper has been to develop and specify a basic version of the sweep-line method, and we have shown how the basic sweep-line generation algorithm can be adapted to allow on-the-fly verification. Investigating how other model checking algorithms can be combined with the sweep-line method is a topic of future work. The constraint is that the sweep-line method relies inherently on breadth-first generation of the state space. The model checking procedure therefore needs to be breadth-first based in order to be compatible with the sweep-line method. Future work also includes investigating the combination of the sweep-line method and other state space reduction methods such as partial order reduction methods [21, 19, 22] and the symmetry method [6, 14, 5].

The sweep-line method is geared towards systems for which it is possible to quantify its progress based on the states. The method does not work well for fully or almost fully reactive systems, where most of the state space is strongly connected, i.e., the state space has very few strongly connected components. In fact, the number of nodes in the largest strongly connected component gives a lower bound on the memory consumption of the sweep-line method. The examples contained in this paper demonstrate, however, that there exists many interesting and non-trivial systems for which a progress measure can be specified, and where the sweep-line method is a very efficient way of analysing these systems. A measure of progress seems likely to be present also in other systems.

A disadvantage of the sweep-line method is that when a violation of a property has been detected, a run-time expensive backwards search is required in order to provide an execution/counter example showing why the property does not hold. With other similar methods such as state space caching [9,12] and bit-state hashing [10,11], the counter example is immediately available on the depth-first search stack. With state space caching, state space generation is expensive and counter example generation is inexpensive, whereas with the sweep-line method the opposite is true. On the other hand, since the sweep-line method relies on breadth-first generation it is more geared towards finding short counter examples which is not the case for the state space caching method.

Two other methods for deleting states during state space generation have been given in [18] and [15]. The observation behind [18] is that a state can be deleted once all its predecessor states have been explored. The method in [18] relies on being able to compute the number of predecessor states. This is for instance possible when the transition relation of the system can be represented using BDDs [2]. The method in [15] combines partial-order reduction and bit-state hashing with the idea of deleting states which are invisible to the LTL-X temporal logic property to be verified. Invisible states are deleted on-the-fly during state space generation, and to alleviate the problem of revisiting states, a preprocessing phase based on bit-state hashing is used to compute approximative information about the number of predecessors of states. This makes it possible to avoid blindly deleting states which could possibly be visited again. The sweep-line method and the methods in [18] and [15] all exploit different information to obtain the criteria for deleting states. The sweep-line method exploits a progress measure which is typically a property of the system to be verified, [18] exploits semantic information about the predecessors of each state, whereas [15] exploits the temporal logic property to be verified.

We have seen that for timed CP-nets it is possible to define a progress measure based on the modelling formalism and which, as a consequence, is applicable to all timed CPN models of systems. In other cases the user is required to provide the progress measure as input to the sweep-line generation algorithm. It is therefore relevant to ask how difficult it is to come up with a progress measure. We claim that if there is progress present in the system, then the modeller has in most cases an intuition about this which can be formalised into a progress measure and provided to the tool. In the prototype implementation of the sweep-line method in the Design/CPN state space tool, the full Standard ML [20] programming language is available to the user for specifying progress measures, and hence offers great flexibility for specifying progress measures. The provided progress measure is also required to fulfill the property that all successors of a state s have a progress measure which is greater than or equal to the progress measure of s. If the user specifies a progress measure as input to the tool which does not have the required property, i.e., the progress is not actually present in the system, then this is not disastrous. Violations against the required property can be detected fully automatically by the tool during state space generation. In case a violation is detected, the state space generation can be stopped and a state and one of its successor states reported back to the user demonstrating why the provided progress measure does not fulfill the required property.

References

1. D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, Inc., 1992.
2. R.E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
3. S. Christensen and J.B. Jørgensen. Analysis of Bang and Olufsen's BeoLink Audio/Video System Using Coloured Petri Nets. In P. Azéma and G. Balbo, editors,

Proceedings of ICATPN'97, volume 1248 of *Lecture Notes in Computer Science*, pages 387–406. Springer-Verlag, 1997.

4. S. Christensen, J.B. Jørgensen, and L.M. Kristensen. Design/CPN - A Computer Tool for Coloured Petri Nets. In E. Brinksma, editor, *Proceedings of TACAS'97*, volume 1217 of *Lecture Notes in Computer Science*, pages 209–223. Springer-Verlag, 1997.
5. E.M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting Symmetries in Temporal Logic Model Checking. *Formal Methods in System Design*, 9, 1996.
6. E.A. Emerson and A.P. Sistla. Symmetry and Model Checking. *Formal Methods in System Design*, 9, 1996.
7. J. Esparza. Model Checking using Net Unfoldings. *Science of Computer Programming*, 23:151–195, 1994.
8. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems, An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
9. G.J. Holzmann. Tracing protocols. *AT&T Technical Journal*, 64(10):2413–2433, December 1985.
10. G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International Editions, 1991.
11. G.J. Holzmann. An Analysis of Bitstate Hashing. *Formal Methods in System Design*, 13(3):287–305, November 1998. Extended and revised version of Proc. PSTV95, pp. 301-314.
12. C. Jard and T. Jeron. Bounded-memory Algorithms for Verification On-the-fly. In *Proceedings of CAV'91*, volume 575 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.
13. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts*. Monographs in Theoretical Computer Science. Springer-Verlag, 1992.
14. K. Jensen. Condensed State Spaces for Symmetrical Coloured Petri Nets. *Formal Methods in System Design*, 9, 1996.
15. S. Katz and H. Miller. Saving Space by Fully Exploiting Invisible Transitions. *Formal Methods in System Design*, 14:311–332, 1999.
16. L.M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.
17. K. L. McMillan. A Technique of State Space Search Based on Unfolding. *Formal Methods in System Design*, 6(1):45–65, 1995.
18. A. N. Parashkevov and J. Yantchev. Space Efficient Reachability Analysis Through Use of Pseudo-Root States. In *Proceedings of TACAS'97*, volume 1217 of *Lecture Notes in Computer Science*, pages 50–64. Springer-Verlag, 1997.
19. D. Peled. All from One, One for All: On Model Checking Using Representatives. In *Proceedings of CAV'93*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1993.
20. J.D. Ullman. *Elements of ML Programming*. Prentice-Hall, 1998.
21. A. Valmari. A Stubborn Attack on State Explosion. In *Proceedings of CAV'90*, volume 531 of *Lecture Notes in Computer Scienc*, pages 156–165. Springer-Verlag, 1990.
22. P. Wolper and P. Godefroid. Partial Order Methods for Temporal Verification. In *Proceedings of CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.