Sweep-Line State Space Exploration for Coloured Petri Nets*

Guy Edward Gallasch[1], Lars Michael Kristensen[1], and Thomas Mailund[2]

[1] Computer Systems Engineering Centre
University of South Australia
Mawson Lakes Campus, SA 5095, AUSTRALIA
Email: guy.gallasch@postgrads.unisa.edu.au, lars.kristensen@unisa.edu.au
[2] Department of Computer Science, University of Aarhus
IT-parken, Aabogade 34, DK-8200 Aarhus N, DENMARK
Email: mailund@daimi.au.dk

Abstract. The basic idea of the sweep-line state space method is to exploit a formal notion of progress found in many concurrent and distributed systems. Exploiting progress makes it possible to sweep through the state space of a CP-net while storing only a small fragment of the states in memory at any time. Properties of the system can then be verified on-the-fly during the sweep of the state space. This can lead to significant savings in peak memory usage and computation time. Examples of systems possessing progress are transport protocols, transactions protocols, workflow models, and systems modelled with Timed CP-nets. We present SWEEP/CPN, a library extension to DESIGN/CPN supporting the sweep-line method, and demonstrate its use.

Keywords: State space methods, State explosion problem, extensions to DESIGN/CPN, Verification and validation.

1 Introduction

State space exploration and analysis is a powerful way to investigate the correctness of distributed and concurrent systems, and is one of the main analysis methods for Coloured Petri Nets (CP-nets or CPNs) [11, 12]. The basic idea behind state space exploration is to construct a directed graph (called the state space, or the reachability/occurrence graph) representing all reachable states of the system and the transitions between these states. From this graph a large number of dynamic properties of the system can be analysed algorithmically. The main drawback of using state spaces for analysis of systems is the *state explosion problem:* even for systems of moderate complexity the number of reachable states can be astronomical, and storing the state space in the available computer memory might not be feasible.

In an attempt to alleviate the state explosion problem a number of state space reduction methods and techniques have been developed. These techniques can be split into three main classes. The first are those methods that represent the state space in a compact or condensed form. Symmetry reduction [4, 5, 13] is an example of this, where a representative state is used to represent a set of symmetric states. The second class of methods represent only a subset of the full state space. Such methods include partial order reduction methods [17, 20, 21]

* Supported by an Australian Research Council (ARC) Discovery Grant (DP0210524).

where only some orderings of independent events and not all orderings are stored. The reduction is done in such a way that the answers to verification questions can still be determined from the reduced state space.

The third class of reduction techniques involve deleting states or state information during state space exploration. Such methods include hash-compaction [18,22] and bit-state hashing [8,9] in which a hash-value is calculated from the state and only the hash-value – not the entire state – is stored. The state space caching method [6,10], is another method based on deleting information during exploration. State space caching exploits the fact that during a depth-first exploration of the state space, only the states on the depth-first stack need to be stored to ensure termination. States not on the stack can be deleted once memory becomes scarce without compromising the termination of the state space exploration.

The *sweep-line method* [3, 15] is a state space method belonging to the third category. It is aimed at systems for which the notion of progress can be formalised as a *progress measure* mapping from markings to a set of ordered progress values. The sweep-line method uses the progress values on markings to determine which states can safely be deleted. Progress measures provide a conservative estimate of reachability: For a marking to be reachable from the set of unprocessed markings, it would have to have a progress value larger than at least one of the unprocessed markings. Markings with progress value smaller than all unprocessed markings can therefore safely be deleted while still guaranteeing termination of the state space exploration. Progress measures will often be specific to the system under consideration, such as sequence numbers and retransmission counters in communication protocols, and the control flow of the system. However in some instances a progress measure can be defined based on the modelling language itself, and such progress measures then apply to all models of systems constructed using this modelling language. The global clock representing time in Timed Coloured Petri Nets [12] is an example of one such progress measure. The sweep-line method was used in [7] for verification of transactions in the Wireless Application Protocol (WAP) with a reduction in peak memory usage to 20%. The use of the sweep-line method on Timed CP-nets was studied in [2]. The sweep-line method is not tied to CP-nets, but can be applied to all modelling languages where state space methods are applicable.

The contribution of this paper is to study the sweep-line method in the context of CP-nets, and to present SWEEP/CPN [16], a library extension to the DESIGN/CPN state space tool [1] implementing the basic sweep-line method from [3] for CP-nets. Consequently there is a shift in the focus of the paper, from a theoretical introduction to the sweep-line method to a description of the SWEEP/CPN implementation, which could be regarded as a high level manual for practitioners.

The rest of this paper is organised as follows. Section 2 introduces the stop-and-wait protocol which we will use as a running example throughout this paper. Section 3 gives the necessary background information on state spaces for CP-nets, while Sect. 4 and 5 present the sweep-line method and the application of SWEEP/CPN to the stop-and-wait protocol. Section 6 presents the set of generic state space exploration functions available for verification using

SWEEP/CPN. Section 7 shows how the generic state space exploration functions can be specialised for the verification of standard dynamic properties of CP-nets. Finally, in Sect. 8 we sum up with a conclusion and discuss future work.

2 The Stop-and-wait Communication Protocol

To introduce the sweep-line method and demonstrate the use of SWEEP/CPN, we will use a simple communication protocol as a running example. The protocol under consideration is a stop-and-wait communication protocol from the datalink control layer of the OSI (Open Systems Interconnection) network architecture. The CPN model of the stop-and-wait protocol is taken from [14]. The stop-and-wait protocol is not a sophisticated protocol, however it is interesting enough to deserve closer investigation, and complex enough to demonstrate the use of SWEEP/CPN. In this paper we only explain the part of the CPN model necessary to understand the application of SWEEP/CPN. A complete description of the CPN model can be found in [14].

Figure 1 shows the prime page of the CPN model. The system consists of a Sender (left) transmitting data packets to a Receiver (right) across a bidirectional Communication Channel (bottom). The packets to be transmitted are stored in a Send buffer at the sender side, and the packets received by the receiver are stored in a Received buffer. The Communication Channel is unreliable, which means that loss and overtaking is possible, however for simplicity only loss is considered in this example. The data packets in the Send buffer have to be delivered over the communication channel exactly once, and in the correct order, to the Received buffer. A stop-and-wait strategy is employed to achieve this, whereby the sender will send a data packet and continue to retransmit the same data packet until a matching acknowledgement is received, at which point the next data packet can be transmitted. The number of retransmissions of a data packet allowed by this stop-and-wait protocol is unbounded for simplicity. In order to make the CPN model tractable for state space analysis, the buffers between the sender, receiver, and communication channel (TransmitData, ReceiveAck, TransmitAck, ReceiveData) have been modelled with a maximum capacity of one packet each. This still allows for unbounded retransmissions, whilst ensuring the state space will be finite.

3 State Spaces of CP-nets

In this section we briefly recall the basic ideas of full state space exploration for CP-nets [12]. The state space of a CP-net can be characterised as a directed graph $G = (V, E)$, where $V = [M_0\rangle$ (the set of nodes or markings reachable from the initial marking M_0), and $E = \{(M, (t, b), M') \in V \times BE \times V \mid M[(t, b)\rangle M'\}$, i.e., there is an arc $(M, (t, b), M')$ in the state space if the binding element (t, b) (belonging to the set of all binding elements BE) is enabled in the marking M and its occurrence leads to the marking M'.

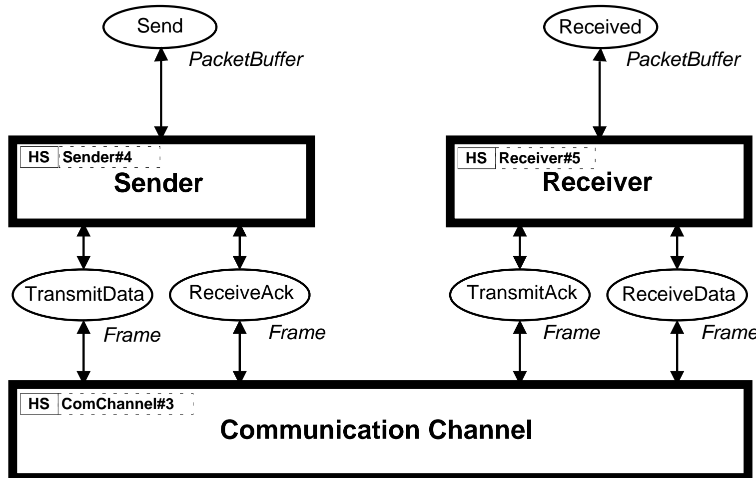

Fig. 1. Stop-and-wait Communication Protocol.

Figure 2 shows a variant of the classical algorithm for generating the state space of a CP-net. The algorithm operates on two sets: NODES and UNPROCESSED. The set NODES holds all markings generated so far, and the set UNPROCESSED holds the markings for which successor markings have not yet been calculated. Initially (line 1) UNPROCESSED contains only M_0. As long as there are unprocessed markings, the algorithm selects one marking among the unprocessed (line 4) and calculates all its successors (line 5). If a successor has not been processed and is not contained in NODES, it is added to NODES and UNPROCESSED (lines 6-9). When the algorithm terminates, NODES contains all reachable markings.

```
 1: UNPROCESSED ← {M0}
 2: NODES ← {M0}
 3: while ¬ UNPROCESSED.EMPTY() do
 4:   M ← UNPROCESSED.GETNEXTELEMENT()
 5:   for all ((t,b), M') such that M[(t,b)⟩M' do
 6:     if ¬(NODES.CONTAINS(M')) then
 7:       NODES.ADD(M')
 8:       UNPROCESSED.ADD(M')
 9:     end if
10:   end for
11: end while
```

Fig. 2. Full state space exploration algorithm.

Figure 3 shows a snapshot of the state space generation for the stop-and-wait protocol. Dashed nodes are *fully processed* markings (i.e. markings that are stored in memory and all their successor markings have been calculated). Nodes with a thick solid black border are *unprocessed* nodes (i.e. nodes that are stored in memory, but their successor markings have not yet been calculated). Nodes with a thin solid black border are markings that have not yet been calculated. Node 1 corresponds to the initial marking. In order to make the state space fragment comprehendable, the situations where packets are lost on the com-

munication channel have been ignored. The states in this fragment have been arranged into two *layers*: Layer 1, where no packets have yet been received by the Receiver, and Layer 2, where one packet has been received. Such an ordering is useful for explaining the concept of progress in the next section. Arcs have been labelled with their corresponding action in the CPN model. Accept Packet 0 indicates that the Sender has accepted the first packet for sending. Send Packet 0 indicates that the Sender has placed the packet in the Communication Channel. Transmit Packet 0 indicates that the Communication Channel has (successfully) transmitted the packet from the sender to the receiver. Receive Packet 0 shows the Receiver receiving the packet from the Communication Channel. Send Ack indicates that the Receiver has placed an acknowledgement into the Communication Channel. TimeOut shows an occurrence of the Sender timing out (i.e. expiration of a timer) because it has not yet received an acknowledgement.

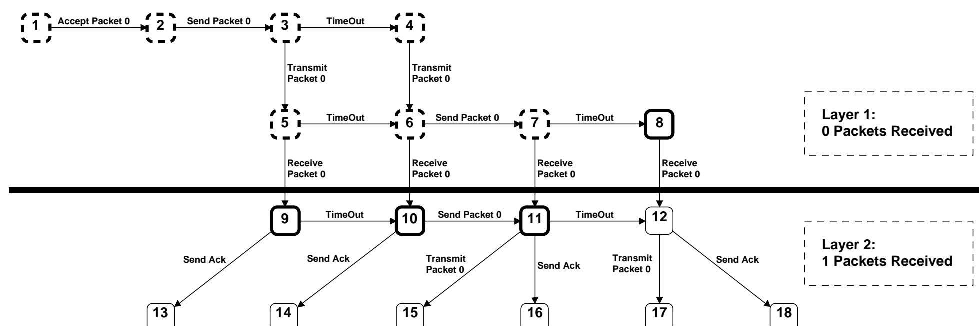

Fig. 3. Initial fragment of state space.

If we analyse states on-the-fly, the set NODES in Fig. 2 is used solely for determining whether successor states of a given state have already been examined, or whether they should be considered unprocessed. However, at any time some states in NODES are no longer reachable from the states in UNPROCESSED, and will not be used for this test in the further processing. These states could be removed from NODES, and the algorithm would still visit each reachable state. The problem with this approach is, of course, determining whether a state is reachable from one of the unprocessed states. The sweep-line method uses a notion of *progress* to obtain a conservative estimate of the reachability relation, and uses this estimate to reduce NODES.

4 The Sweep-Line Method

The concept of progress exploited by the sweep-line method can be thought of in terms of how far we have moved towards a desired end-state or a predefined goal, or towards the termination of execution of a system. To give an example of each of these, consider a farmer who has a number of fields to plow. If we consider the goal of the farmer to have all fields plowed, we can then consider the progress made by the farmer towards achieving this goal. Progress could be measured in terms of number of fields plowed. In this case, we have a predefined

goal, which is to have all fields plowed. The farmer would start with no fields plowed, and plow field after field until all fields had been plowed. Progress could also be measured in terms of total time spent plowing. We can see that at any given time during the plowing, the farmer would have plowed the same amount or more than at any given time previous to this. In this case we do not know how long the plowing will take to complete, so we cannot specify a predefined goal, but we can work towards the termination of the system whereby the farmer has completed plowing.

There is an intuitive presence of progress in the stop-and-wait protocol as more and more packets are being transferred from the sender to the receiver. This progress is also reflected in the state space of the CPN model, an initial fragment of which was shown in Fig. 3. The key observation to make is that progress in the stop-and-wait protocol manifests itself by the property that a marking in a given layer has successor markings either in the same layer or in some layer with a higher number, but never in a layer with a smaller number. The idea underlying the sweep-line method is to exploit such progress by deleting markings on-the-fly during state space exploration. The deletion is done such that the state space exploration will eventually terminate and upon termination all reachable markings will have been explored exactly once.

Examining the initial state space fragment shown in Fig. 3, if the state space exploration algorithm processes markings according to the progress of the protocol they correspond to, node 8 will be the marking among the unprocessed markings that will be selected for processing next. This will add node 12 to the set of stored markings and mark it as unprocessed. At this point it can be observed that it is not possible for any of the unprocessed markings to reach one of the markings 1-8. The reason is that nodes 1-8 represent markings where the protocol has not progressed as far as in any of the unprocessed markings, i.e. no packets have been received by the receiver. Hence, it is safe to delete nodes 1-8, as they cannot possibly be needed for comparison with newly generated markings when checking (during the state space exploration) whether a marking has already been visited. In a similar way, once all the markings in the second layer have been fully processed, these nodes can be deleted from the set of nodes stored in memory. Intuitively, one can think of a *sweep-line* being aligned with the layer currently being processed, i.e. the layer that contains unprocessed markings. During state space exploration, unprocessed markings are selected for processing in a least-progress-first order causing the sweep-line to move forwards. Markings will thereby be added in front of the sweep-line and deleted behind the sweep-line.

The progress exploited by the sweep-line method is formally captured by a *progress measure*. A progress measure consists of a *progress mapping* assigning a *progress value* to each marking, and a *partial order* on the progress values.[1] Moreover, the partial ordering of the progress values is required to preserve the reachability relation between markings of the CP-net. The definition of progress measure below is identical to Def. 1 in [3] except that we give the definition in

[1] A partial order $(O, \sqsubseteq)$ consists of a set O and a relation $\sqsubseteq \subseteq O \times O$ which is reflexive, transitive, and antisymmetric.

a CP-net formulation. In the definition $\mathbb{M}$ denotes the set of all markings. If a marking M' is reachable from a marking M via some occurrence sequence, we write this as $M' \in [M\rangle$. In particular $M \in [M\rangle$ for all markings $M \in \mathbb{M}$, since the empty occurrence sequence leads from M to M.

Definition 1. *(Def. 1 in [3]) A* ***progress measure*** *is a tuple $\mathcal{P} = (O, \sqsubseteq, \psi)$ such that $(O, \sqsubseteq)$ is a partial order and $\psi : \mathbb{M} \to O$ is a progress mapping from markings into O satisfying: $\forall M, M' \in [M_0\rangle : M' \in [M\rangle \Rightarrow \psi(M) \sqsubseteq \psi(M')$.*

Exploring the state space using the sweep-line method is based on the algorithm used for conventional state space construction. Figure 4 shows the state space exploration algorithm for the sweep-line method. It is derived from the standard algorithm by including deletion of states that can no longer be reached by states still to be explored, and by exploring the states in UNPROCESSED according to their progress values. In each iteration (lines 3-15) a new unprocessed marking is selected (line 4) such that this node has a minimal progress value among the states in UNPROCESSED. After a marking has been processed, states with a progress value strictly smaller than the minimal progress value among the markings in UNPROCESSED can be deleted (line 14). In the case of a total order, there will only be one minimal progress value. In the more general case of a partial order, it is possible for all unprocessed states to have different, incomparable progress measures. Hence, in the 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. The condition that the progress measure preserves the reachability relation is checked on-the-fly when successors of a marking are being calculated (lines 6-8). State space exploration is terminated if the progress measure is not valid, and a triple $(M, (t, b), M')$ is given to demonstrate why the progress measure was not valid.

```
1: UNPROCESSED ← {M_0}
2: NODES ← {M_0}
3: while ¬ UNPROCESSED.EMPTY() do
4:   M ← UNPROCESSED.GETMINELEMENT()
5:   for all ((t,b), M') such that M[(t,b)⟩M' do
6:     if ψ(M) ⋢ ψ(M') then
7:       STOP("Progress measure rejected:",(M, (t,b), M'))
8:     end if
9:     if ¬(NODES.CONTAINS(M')) then
10:      NODES.ADD(M')
11:      UNPROCESSED.ADD(M')
12:    end if
13:  end for
14:  NODES.DELETE(min{ψ(M) | M ∈ UNPROCESSED})
15: end while
```

Fig. 4. The sweep-line state space exploration algorithm.

In order to maximise the number of states that can be deleted in each *garbage collection* (or *memory reclaim*), the unprocessed state chosen by the

GetMinElement() method (line 4) should always return (as the name implies) a state with a minimal progress measure among the states in Unprocessed. 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. A breadth-first generation (with respect to progress measure) of the state space will result, ensuring that the number of states that can be deleted is maximised.

5 Sweep-Line Exploration of the Stop-and-Wait Protocol

The Sweep/CPN library supports progress measures based on total orderings, and consists of a set of Standard ML [19] files which can be loaded into the Design/CPN state space tool [1]. Currently there is no graphical user interface (GUI) support associated with Sweep/CPN, and all interaction with the library is via evaluation of auxiliary boxes containing SML code. The GUI of the Design/CPN state space tool can however be used to draw nodes and arcs of the state space.

The user specifies the progress measure to be used by writing an SML function mapping from the markings into unbounded integers, that is, from the type SLMark into IntInf.int. Verification of the user specified progress measure is done on-the-fly as described in the previous section, whereby validity is checked upon the calculation of the successors of a marking. The progress measure for the stop-and-wait protocol based on the number of packets received by the receiver can be implemented as follows. An explanation of the function is given below.

```
fun SWPM M = IntInf.fromInt
     (List.length (ms_to_col (SLMark.SWProtocol'Received 1 M)));
```

For a marking `M`, the function works by extracting the marking of place Received on instance 1 of the page SWProtocol. (Page SWProtocol is the name of the prime page of the CPN model shown in Fig. 1.) This is done by using the accordingly named function of the SLMark structure. The marking of place Received is a multi-set containing a single token which is a list of the packets received until now. The list of received packets is extracted from the multi-set using the function ms_to_col. Finally, the function List.length is used to obtain the length of the list, and the function IntInf.fromInt is used to convert the integer denoting the length of the list into an infinite (unbounded) integer.

A number of functions are available in Sweep/CPN for state space exploration with the sweep-line method. These will be discussed in more detail in the following section. However, in order to provide the complete picture for the stop-and-wait protocol, the simplest of these functions will be described. This is the SL.ExploreStateSpace function. For the stop-and-wait protocol example, the SL.ExploreStateSpace function is used as follows:

```
SL.ExploreStateSpace{PM=SWPM,
                     GC = 100,
```

```
Secs = 300,
Logfile = (SOME "/tmp/SW.dat")
};
```

The function takes a record with four fields, the *sweep options*, as an argument and conducts sweep-line state space exploration according to the values of these fields. The first field of the sweep options, PM, specifies the progress measure to use. For this stop-and-wait example, it is the progress measure defined by the SWPM function specified above. The GC field specifies the *garbage collection* threshold. This is the number of additional nodes that have to be explored before a deletion of markings is initiated in line 14 of the algorithm in Fig. 4. In this case, garbage collection of nodes will occur every time 100 new nodes are explored. The Secs field specifies an upper bound (in seconds) on the time that the state space exploration will run for. Once this time limit is reached the state space exploration will terminate regardless of whether the exploration is complete. The file specified in the Logfile field will contain a log generated by SWEEP/CPN during state space exploration. The log file will contain statistical information about the state space exploration.

Table 1 lists statistics for the application of the sweep-line method for different configurations of the stop-and-wait communication protocol. The results were obtained when garbage collecting after each 2000 new nodes (markings). The table consists of four columns. The Packets column gives the configuration under consideration, i.e. the number of packets that must be delivered in order and without loss to the receiver from the sender. The Full State Spaces column gives the number of markings in the full state space and the amount of CPU time needed to generate it. The third column, Sweep-Line Method, gives the maximum number of markings stored in memory at any given instance during the sweep-line state space exploration, and the time it took to sweep through the entire state space. The Reduction gives the reduction in the number of markings required to be stored (and thus the reduction in memory consumption) as well as the reduction in time taken to generate the full state space, when using the conventional DESIGN/CPN state space tool and when using the SWEEP/CPN library. All results were obtained using an Athlon XP 2000+ with 1Gb DDR memory.

Table 1. Experimental results – Stop-and-Wait Communication Protocol.

	Full State Spaces		Sweep-Line Method		Reduction	
Packets	Markings	Time	Markings	Time	Markings	Time
10	2,576	0:00:01	2,001	0:00:01	22.3%	0.0%
50	13,416	0:00:05	2,280	0:00:04	83.0%	20.0%
100	26,966	0:00:14	2,280	0:00:07	91.5%	50.0%
200	54,066	0:00:42	2,280	0:00:14	95.8%	66.7%
500	135,366	0:03:11	2,287	0:00:38	98.3%	80.1%
1000	270,866	0:11:21	2,287	0:01:21	99.2%	88.1%
2000	541,866	0:47:10	2,287	0:03:05	99.6%	93.5%

Table 1 shows that the use of Sweep/CPN saves both memory and time. For example, when there are 50 packets to be delivered from sender to receiver, the reduction in the number of markings that must be stored in memory at one time is 83%, while the reduction in time is 20%. In addition to this, the savings in memory consumption and time grow larger as the size of the full state space grows larger. This can be seen when examining the situation where 2000 packets must be delivered from the sender to the receiver. The saving in memory consumption is 99.6%, and the saving in time is 93.5%.

Savings in memory consumption were expected since markings are being deleted on-the-fly during state space exploration. The saving in runtime is perhaps more surprising. The explanation can be found in the test for whether successor states are new or have previously been processed (in line 9 of the algorithm in Fig. 4). By keeping the set Nodes small we avoid this runtime performance penalty. For this example at least, Table 1 indicates that using Sweep/CPN and the sweep-line method is faster than generating the full state space by conventional means. The time overhead required to perform garbage collection (delete states on-the-fly) is more than compensated for by having significantly fewer states to compare with when determining whether a newly generated marking has already been visited.

6 State Space Exploration

The general sweep-line algorithm was introduced in Section 4. This algorithm demonstrates how a state space can be explored using the sweep-line method. Using conventional state space exploration, analysis can be done by examining all states in the state space once the entire state space has been constructed. However, since states are deleted during the state space exploration when using the sweep-line method, properties must be verified *on-the-fly*. The general algorithm from Section 4 must be augmented so that it provides not just a sweep through the state space, but also the possibility for verification and analysis of properties of the system.

Figure 5 shows the sweep-line state space exploration algorithm augmented with hooks for analysing *state-related properties* (e.g. boundedness, other model-specific properties such as duplication of packets), *arc-related properties* (e.g dead transition instances) and *dead markings*. The analysis is done through three procedures: ExploreNode, ExploreArc, and ExploreDead, operating on the variables NodeResult, ArcResult, and DeadResult respectively. The three variables, NodeResult, ArcResult and DeadResult along with the set Nodes are assumed to have a scope that includes these procedures. The analysis procedures are shown in Fig. 6, Fig. 7, and Fig. 8. As seen, they are all more or less of the same form: A predicate, e.g. NodePred, selects whether a node or arc should be analysed. If it should, then an *evaluation function*, e.g. NodeEval, extracts a value from the node or arc. This value is then combined, via a *combine function* e.g. NodeComb, with the result so far. Predicate, evaluation functions, and combination functions are all user specified. The value so far is stored in the respective result variable, NodeResult, ArcResult, and DeadResult, which are all initialised with user provided values

NodeInit, ArcInit, and DeadInit. Additional predicates, NodeStore and DeadStore, once again user specified, are used to prevent garbage collection for selected nodes.

1: Unprocessed ← $\{M_0\}$
2: Nodes ← $\{M_0\}$
3: NodeResult ← NodeInit
4: ArcResult ← ArcInit
5: DeadResult ← DeadInit
6: **while** ¬ Unprocessed.Empty() **do**
7: M ← Unprocessed.GetMinElement()
8: ExploreNode(M)
9: **if** Dead(M) **then**
10: ExploreDead(M)
11: **end if**
12: **for all** $((t, b), M')$ such that $M[(t, b)\rangle M'$ **do**
13: ExploreArc($M, (t, b), M'$)
14: **if** $\psi(M) \not\sqsubseteq \psi(M')$ **then**
15: Stop("Progress measure rejected:",$(M, (t, b), M')$)
16: **end if**
17: **if** ¬(Nodes.Contains(M')) **then**
18: Nodes.Add(M')
19: Unprocessed.Add(M')
20: **end if**
21: **end for**
22: Nodes.GarbageCollect(min$\{\psi(M)$ $M \in$ Unprocessed$\}$)
23: **end while**

Fig. 5. The augmented sweep-line state space exploration algorithm.

The first procedure, ExploreNode, is invoked for every marking M encountered during the sweep. An example of how this can be used would be to inspect all markings M to determine the best upper integer bound on a particular place. The best upper bound of a place is the maximum number of tokens that may reside on a place in any reachable marking. In this case, NodeResult would be of type integer, NodePred would always return true, and the number of tokens for this particular place in M would be returned by NodeEval. NodeComb would then determine whether the number of tokens as returned by NodeEval is larger than the best upper integer bound computed until now (in NodeResult) and update NodeResult if necessary. The second procedure, ExploreArc, is called for every arc (edge) encountered during the sweep. It works in a very similar way to ExploreNodes, except that it works on arcs rather than nodes, and there is no provision for the storage of arcs. The third procedure, ExploreDead, is only evaluated for markings that have no sucessor markings, but otherwise works in a similar way to ExploreNodes.

The augmented sweep-line algorithm has been implemented in Sweep/CPN as the exploration function called SL.Explore. The function is parameterised with 15 arguments. To provide for more convenient use of Sweep/CPN, three specialisations to SL.Explore have been added to the library. These are SL.ExploreStates, SL.ExploreArcs, and SL.ExploreDead, and are specialisations to examine all nodes, all arcs, and all dead markings of the state space respectively. Dead

```
1: procedure ExploreNode(M) do
2:   if NodePred(M) then
3:     NodeResult ← NodeComb(NodeResult, NodeEval(M))
4:   end if
5:   if NodeStoreM) then
6:     Nodes.MarkPersistent(M)
7:   end if
8: end procedure
```

Fig. 6. The ExploreNode procedure.

```
1: procedure ExploreArc(M, (t, b), M') do
2:   if ArcPred(M, (t, b), M') then
3:     ArcResult ← ArcComb(ArcResult, ArcEval(M, (t, b), M'))
4:   end if
5: end procedure
```

Fig. 7. The ExploreArc procedure.

markings are treated as a separate case to ordinary nodes/markings, because of the underlying structure and implementation of the existing state space tool. As an example, the function SL.ExploreStates has the following type:

```
SL.ExploreStates : {PM      : SLMark -> IntInf.int,
                    GC      : int,
                    Secs    : int,
                    Logfile : string option,
                    Init    : 'a,
                    Comb    : 'a * 'b -> 'a,
                    Eval    : SLMark -> 'b,
                    Pred    : SLMark -> bool,
                    Store   : SLMark -> bool} -> 'a
```

The first four fields in the record (PM, GC, Secs, Logfile) correspond to the four sweep options discussed in Section 5 and are the progress measure, garbage collection threshold, exploration time-out and logfile location respectively. Init allows the user to provide the initial value for the Comb funtion. The Comb is the combination function on nodes, Eval is the evaluation function on nodes, and Pred is the predicate on nodes. Store is a function which determines if the nodes should be made persistent. The types of SL.ExploreArcs and SL.ExploreDead are

```
1: procedure ExploreDead(M) do
2:   if DeadPred(M) then
3:     DeadResult ← DeadComb(DeadResult, DeadEval(M))
4:   end if
5:   if DeadStore(M) then
6:     Nodes.MarkPersistent(M)
7:   end if
8: end procedure
```

Fig. 8. The ExploreDead procedure.

similar to SL.ExploreStates. We give examples of the use of these functions in the next section.

7 Standard Query Functions

We now present the set of standard query functions available in SWEEP/CPN for verifying a subset of the standard dynamic properties of CP-nets as defined in [12]. The standard dynamic properties of CP-nets are often the first set of properties investigated for the system under consideration. In addition to presenting the available standard query functions, we also show how some of these have been implemented by means of the generic state space exploration functions presented in Sect. 6. The structure of SWEEP/CPN (i.e. as a set of query functions) has been inspired by the structure of the existing state space exploration tool, which was in turn partially inspired by functional programming concepts such as mapping and folding [19]. The standard dynamic properties of CP-nets are informally introduced throughout this section. The reader interested in the full and formal definition of the standard dynamic properties of CP-nets is referred to [12].

7.1 Reachability Properties

A marking is said to be *reachable*, if it is reachable via some occurrence sequence starting in the initial marking. The function Reachable is available for determining whether a reachable marking exists satisfying a given predicate:

```
SLReachable : SweepSpec * (SLMark -> bool) * bool -> bool
```

The first parameter (of type SweepSpec) is a record which provides the parameters for the sweep-line exploration. This is the same record as shown in Section 6, where the four arguments (progress measure, garbage collection threshold, upper bound on the time for state space exploration, and the log file name and location) are given. The second argument is the predicate on markings. The third argument determines whether the encountered markings which satisfy the marking predicate should be available upon the completion of the sweep.

To illustrate the use of SLReachable, we will show how it can be used to investigate whether the stop-and-wait protocol may duplicate packets. If the stop-and-wait protocol can duplicate packets, a reachable marking will exist in which some packet occurs twice or more in the list on place Received. The following marking predicate DupPackets determines whether this is the case in a marking M. The function remdupl removes duplicates from a list.

```
fun DupPackets M =
    let
       val packets = (ms_to_col (SLMark.Receiver'Received 1 M))
    in
       List.length (remdupl packets) <> (List.length packets)
    end;
```

The DupPackets predicate can now be used in an invocation of the SLReachable query function as follows:

```
SLReachable ({PM=SWPM,GC=100,Secs=300,Logfile=NONE},
             DupPackets, true);
```

In this case, markings satisfying DupPackets will not be deleted during the sweep. Reachability of a specific marking M can be checked by using a predicate as argument to SLReachable which evaluates to true only on M.

The SLReachable function has been implemented using the SL.ExploreStates function as follows:

```
fun SLReachable ({PM=pm,GC=gc,Secs=secs,Logfile=logfile},
                 markpred,keep) =
    (SL.ExploreStates
        {PM=pm,GC=gc,Secs=secs,Logfile=logfile,
         Pred = markpred,
         Store = keep,
         Eval = (fn _ => 1),
         Comb = (fn (a,b) => a+b),
         Init = 0}) > 0;
```

The sweep specification is given by the sweep specification provided as the first argument to SLReachable. The evaluation (Eval) and combination (Comb) functions are used to count the number of times the marking predicate evaluates to true. If it evaluates to true at least once, a marking exists satisfying the predicate.

7.2 Boundedness Properties

The boundedness properties are concerned with the number of tokens on places and their possible colours. We first consider integer bounds. The *best upper integer bound* of a place is the maximum number of tokens present on the place in any reachable marking. Similarly, the *best lower integer bound* is the minimum number of tokens present on the place in any reachable marking. Two functions are available for determining the integer bounds of a place:

```
SLBestUpperInteger : SweepSpec * (SLMark -> int) -> int
SLBestLowerInteger : SweepSpec * (SLMark -> int) -> int
```

Both functions take the sweep specification as their first argument. The second argument is a function mapping from markings into integers. The following shows how the functions can be used to obtain the best upper and best lower integer bound of the place TransmitData (see Fig. 1). The function TransmitDataCount counts the number of tokens on the place TransmitData in a marking M. The function mssize returns the number of tokens in a multi-set.

```
fun TransmitDataCount M = mssize (SLMark.Sender'TransmitData 1 M);
```

```
SLBestUpperInteger ({PM=SWPN,GC=100,Secs=300,Logfile=NONE},
                     TransmitDataCount);
SLBestLowerInteger ({PM=SWPN,GC=100,Secs=300,Logfile=NONE},
                     TransmitDataCount);
```

The SLBestUpperInteger and SLBestLowerInteger functions are both implemented using the SL.ExploreStates function. Below we give the implementation of SLBestUpperInteger. The implementation of SLBestLowerInteger is similar.

```
fun SLBestUpperInteger ({PM=pm,GC=gc,Secs=secs,Logfile=logfile},
                         evalfun) =
        (SL.ExploreStates
            {PM=pm,GC=gc,Secs=secs,Logfile=logfile,
             Pred = (fn _ => true),
             Store = (fn _ => false),
             Eval = evalfun,
             Comb = (fn (a,b) => Int.max(a,b)),
             Init = 0})
```

The predicate function (Pred) ensures that the evaluation function is invoked on each marking encountered. The maximum function on integers (Int.max) is used to obtain the maximum value resulting from the invocation of the evaluation function.

Multi-set boundedness is similar to integer boundedness, except we are no longer dealing with the size of the multi-set of tokens on a particular place, but rather with the multi-set of tokens itself. For a particular place, it is possible to find the smallest multi-set of tokens that is larger than any of the multi-sets of tokens found on this place in any reachable marking. This multi-set is called the *best upper multi-set bound* for this particular place. Finding the upper multi-set bound can be done by examining the marking of this place for every reachable marking, and by using multi-set operations to determine the multi-set comprising the best upper multi-set bound. Similarly, the *best lower multi-set bound* can also be found. This is the largest multi-set that is smaller than all multi-sets found on the particular place, for every reachable marking.

Sweep/CPN provides two query functions to find multi-set bounds. The first is SLBestUpperMultiSet, to find the best upper multi-set bound, and the second is SLBestLowerMultiSet, to find the best lower multi-set bound:

```
SLBestUpperMultiSet : SweepSpec -> (SLMark -> ''a ms) -> ''a ms
SLBestLowerMultiSet : SweepSpec -> (SLMark -> ''a ms) -> ''a ms
```

These are similar to the functions for integer bounds except that they work on the level of multi-sets, not integers. The argument of type SweepSpec is again a record that gives the sweep-line parameters (progress measure, garbage collection threshold, exploration time-out, log file.) The second argument is again an evaluation function, but it returns the multi-set of tokens on the specified place upon evaluation, not an integer. Below we show how functions can be used to find the best upper and best lower multi-set bound of the place TransmitData.

```
SLBestUpperMultiSet ({PM=SWPM,GC=100,Secs=300,Logfile=NONE},
                     (SLMark.Sender'TransmitData 1));
SLBestLowerMultiSet ({PM=SWPM,GC=100,Secs=300,Logfile=NONE},
                     (SLMark.Sender'TransmitData 1));
```

The functions SLBestUpperMultiSet and SLBestLowerMultiSet have been implemented based on the SL.ExploreStates function in a similar way to the query functions for integer bounds.

7.3 Liveness Properties

Dead markings are the reachable markings of the CPN model without enabled binding elements. These correspond to the states of the system where it has terminated. Finding such dead markings is often an important step in the analysis of a system. SWEEP/CPN provides a function SLListDeadMarkings which performs a sweep of the state space and records all of the dead markings found in the process. Upon termination of the sweep, the list of dead markings/nodes is returned.

```
SLListDeadMarkings : SweepSpec -> Node list
```

The argument of type SweepSpec is the same as in previous functions, it is a record containing the sweep parameters. The return type of this function is Node list, a list of the dead markings found during the sweep. For the stop-and-wait communication protocol, the SLListDeadMarkings function would be invoked as follows:

```
SLListDeadMarkings {PM=RNPM,GC=100,Secs=300,Logfile=NONE};
```

The returned list of nodes can then be examined further in any way desired by the user, e.g., by drawing the nodes using the state space tool, and then by inspecting the descriptors of the nodes which give details about the marking of each place. The SLListDeadMarkings function has been implemented using the SL.ExploreDeadStates function as follows:

```
fun SLListDeadMarkings {PM=pm,GC=gc,Secs=secs,Logfile=logfile} =
    SL.ExploreDeadStates {PM=pm,GC=gc,Secs=secs,Logfile=logfile,
                          Pred = (fn _ => true),
                          Store = (fn _ => true),
                          Eval = GetNodeNo,
                          Comb = (fn (a,b) => b::a),
                          Init = []};
```

The function GetNodeNo maps from markings into node numbers. The store functions ensure that the dead markings are available after the sweep, and the combination function is used to accumulate the node numbers of the dead markings encountered.

In a manner similar to obtaining dead markings, SWEEP/CPN provides a function for obtaining *dead transition instances*. A dead transition instance is a

transition instance that is not enabled in any marking reachable from the initial marking. The function is called SLListDeadTIs and has the type shown below:

```
SLListDeadTIs : SweepSpec -> TI.TransInst list
```

As has been the case for all sweep-line state space exploration functions so far, this function performs a sweep of the state space and thus the sweep parameters are supplied by the argument of type SweepSpec. The return value is of type `TI.TransInst list`, a list of transition instances representing the dead transition instances. To find the dead transition instances of the stop-and-wait communication protocol example, SLListDeadTIs would be invoked as follows:

```
SLListDeadTIs {PM=RNPM,GC=100,Secs=300,Logfile=NONE};
```

The function has been implemented based on the SL.ExploreArcs function. The implementation is shown below. The function SLArcToTI maps an arc of the state space into the corresponding transition instance. The function RemoveTI removes a transition instance (t) from a list of transitions instances (ts). The constant TI.All is the list of all transition instances of the CPN model.

```
fun SLArcToTI (_,b,_) = BEToTI b;

fun RemoveTI (ts,t) = List.filter (fn t' => t' <> t) ts;

fun SLListDeadTIs {PM=pm,GC=gc,Secs=secs,Logfile=logfile}) =
      SL.ExploreArcs
        {PM=pm,GC=gc,Secs=secs,Logfile=logfile,
         Pred = (fn _ => true),
         Eval = SLArcToTI,
         Comb = RemoveTI,
         Init = TI.All};
```

8 Conclusions and Future Work

We have presented Sweep/CPN, an extension to the Design/CPN state space analysis tool which supports state space exploration with the basic sweep-line method as given in [3]. Moreover, we have demonstrated the use of Sweep/CPN on a small example of a communication protocol.

Future work on Sweep/CPN will consider the further development of Sweep/CPN to support also the generalised sweep-line method presented in [15]. The basic sweep-line method is only useful in situations where the strongly connected component graph is non-trivial (i.e. has multiple nodes) and cannot be used on fully reactive systems. The generalised sweep-line method in [15] supports a relaxed notion of progress compared to the monotone notion of progress considered in this paper. With the relaxed notion of progress, it is possible to have arcs in the state space leading from states with high progress values to states with lower progress values. This is achieved by detecting such

cases and conducting multiple sweeps of parts of the state space. The generalised sweep-line method can be used on fully reactive systems.

In this paper we have shown how the sweep-line method can be used to determine standard dynamic properties of CP-nets such as reachability and boundedness properties, and dead markings. Future work will be concerned with extending the set of properties that can be verified. It follows from the definition of progress measures that all markings in a strongly connected component of the state space will have the same progress value. This means that markings belonging to a given strongly connected component will be stored simultaneously in memory at some point during the sweep. This observation holds the key to extending the properties that can be verified to home markings, home spaces, and fairness properties.

Another topic of future work is also the generation of counter examples (error-traces). With the sweep-line method, part of the path leading from the initial marking to a marking satisfying, e.g., a given marking predicate might have been deleted, and needs to be reconstructed to obtain an error-trace. Combining the sweep-line method with disk-based storage seems a promising approach to obtain error-traces with the sweep-line method. With the sweep-line method, markings can be written to disk as they are deleted from main memory, and there is no need to search for markings on disk. In this way, the usual run-time penalty encountered in disk-based searching is avoided altogether. To obtain the error trace one can work backwards (on disk) from the marking satisfying the marking predicate to the initial marking. This can be done efficiently by storing information about predecessor markings on the disk instead of successor markings. This approach may not give the shortest error-trace in terms of occurrences of transitions, but it can be used to obtain a shortest error-trace in terms of how far the system has progressed according to the progress measure.

References

1. S. Christensen, K. Jensen, and L.M. Kristensen. *Design/CPN Occurrence Graph Manual*. Department of Computer Science, University of Aarhus, Denmark. On-line version: `http://www.daimi.au.dk/designCPN/`.
2. S. Christensen, K. Jensen, T. Mailund, and L. M. Kristensen. State Space Methods for Timed Coloured Petri Nets. In *Proceedings of 2nd International Colloquium on Petri Net Technologies for Modelling Communication Based Systems*, Berlin Germany, September 2001.
3. S. Christensen, L.M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proceedings of TACAS 2001*, volume 2031 of *LNCS*, pages 450–464. Springer-Verlag, 2001.
4. E.M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting Symmetries in Temporal Logic Model Checking. *Formal Methods in System Design*, 9(1/2):77–104, 1996.
5. E.A. Emerson and A.P. Sistla. Symmetry and Model Checking. *Formal Methods in System Design*, 9(1/2):105–131, 1996.
6. P. Godefroid, G.J. Holzmann, and D. Pirottin. State-Space Caching Revisited. *Formal Methods in System Design*, 7(3):227–241, 1995.
7. S. Gordon, L.M. Kristensen, and J. Billington. Verification of a Revised WAP Wireless Transaction Protocol. In *Proceedings of ICATPN'02*, volume 2360 of *LNCS*, pages 182–202. Springer-Verlag, 2002.

8. G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall International Editions, 1991.
9. G.J. Holzmann. An Analysis of Bitstate Hashing. *Formal Methods in System Design*, 13(3):287–305, 1998.
10. C. Jard and T. Jeron. Bounded-memory Algorithms for Verification On-the-fly. In *Proceedings of CAV'91*, volume 575 of *LNCS*, pages 192–202. Springer-Verlag, 1991.
11. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts.* Monographs in Theoretical Computer Science. Springer-Verlag, 1992.
12. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2, Analysis Methods.* Monographs in Theoretical Computer Science. Springer-Verlag, 1994.
13. K. Jensen. Condensed State Spaces for Symmetrical Coloured Petri Nets. *Formal Methods in System Design*, 9(1/2):7–40, 1996.
14. 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.
15. L.M. Kristensen and T. Mailund. A Generalised Sweep-Line Method for Safety Properties. In *Proceedings of FME'02*, volume 2391 of *LNCS*, pages 549–567. Springer-Verlag, 2002.
16. L.M. Kristensen, T. Mailund, and G. Gallasch. SWEEP/CPN. Department of Computer Science, University of Aarhus, 2002. www.daimi.au.dk/designCPN/libs/sweepcpn/.
17. D. Peled. All from One, One for All: On Model Checking Using Representatives. In *Proceedings of CAV'93*, volume 697 of *LNCS*, pages 409–423. Springer-Verlag, 1993.
18. U. Stern and D.L. Dill. Improved Probabilistic Verification by Hash Compaction. In P.E. Camurati and H. Eveking, editors, *Correct Hardware Design and Verification Methods*, volume 987, pages 206–224. Springer-Verlag, 1995.
19. J.D. Ullman. *Elements of ML Programming.* Prentice-Hall, 1998.
20. 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.
21. P. Wolper and P. Godefroid. Partial Order Methods for Temporal Verification. In *Proceedings of CONCUR'93*, volume 715 of *LNCS*, pages 233–246. Springer-Verlag, 1993.
22. P. Wolper and D. Leroy. Reliable Hashing withoug Collision Detection. In *Proceddings of CAV'93*, volume 697 of *LNCS*, pages 59–70. Springer-Verlag, 1993.