Obtaining Memory-Efficient Reachability Graph Representations Using the Sweep-Line Method

Thomas Mailund and Michael Westergaard

Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark,
{mailund,mw}@daimi.au.dk

Abstract. This paper is concerned with a memory-efficient representation of reachability graphs. We describe a technique that enables us to represent each reachable marking in a number of bits close to the theoretical minimum needed for explicit state enumeration. The technique maps each state vector onto a number between zero and the number of reachable states and uses the sweep-line method to delete the state vectors themselves. A prototype of the proposed technique has been implemented and experimental results are reported.

Keywords: Verification; state space methods; state space reduction; memory efficient state representation; the sweep-line method.

1 Introduction

A central problem in the application of reachability graph (also known as state-space) methods is the memory usage. Even relatively simple systems can have an astronomical number of reachable states, and when using basic exhaustive search [16], all states need to be represented in memory at the same time. Even methods that explore only parts of the reachability graph [13,33,29,2] or explore a reduced reachability graph [21,11,23], often need to store thousands or millions of states.

When storing states explicitly—as opposed to using a symbolic representation such as Binary Decision Diagrams [4,5]—the minimal number of bits needed to distinguish between N states is $\lceil \log_2 N \rceil$ bits per state. In a system with R reachable states we should therefore be able to store all reachable states using only in the order of $R \cdot \lceil \log_2 R \rceil$ bits. The number of reachable states, R, however, is usually unknown until after the reachability graph exploration; rather than knowing the number of reachable states we know the number of *syntactically possible* states S, where S is usually significantly larger than R. To distinguish between S possible states $\lceil \log_2 S \rceil$ bits are needed, so to store the R reachable states $R \cdot \lceil \log_2 S \rceil$ bits are needed. Aditional memory will be needed to store transitions.

In this paper we consider mapping the state vectors of size $\lceil \log_2 S \rceil$ bits (the full state vectors or markings) to representations of length $\lceil \log_2 R \rceil$ (the condensed representations), in such a way that full state vectors can be restored when the reachability graph is subsequently analysed. Our approach is the following: We conduct a reachability graph exploration and assign to each new unprocessed state a new number, starting from zero and incrementing with one after each assignment. The states are in this way represented by numbers in the interval $0, \ldots, R-1$. Since the state representation obtained in this way has no relation to the information stored in the full state vector, the condensed representation cannot be used to distinguish between previously processed states and new states. To get around this problem, we keep the original (full) state vectors in a table as long as needed to recognise previously seen states. The *sweep-line method* [7, 25] is used to remove the full state vectors when they are no longer needed, from memory.

In this paper we will use Place/Transition Petri nets [10] (P/T net) formalism as example to illustrate the different memory requirements needed to distinguish between the elements of the set of syntactically possible states and the set of reachable states. The use of P/T nets is only an example, the presented method applies to all formalisms where the sweep-line method can be used.

The paper is structured as follows: In Sect. 2 we summarise the notation and terminology for P/T nets and reachability graphs that we will use. In Sect. 3 we describe the condensed representation of a reachability graph, how this representation can be traversed, and how to restore enough information about the full state vectors to verify properties about the original system. In Sect. 4 we consider how the condensed representation can be calculated and in Sect. 5 we describe how the sweep-line method can be used to keep memory usage low during this construction. In Sect. 6 we report experimental results and in Sect. 7 we give our conclusions.

2 Petri Nets and Reachability Graphs

In this section we define *reachability graphs* of Place/Transition Petri nets.

Definition 1. *A* ***Place/Transition Petri net*** *is a tuple* $\mathcal{N} = (P, T, F, m_I)$ *where* P *is a set of* places*,* T *is a set of* transitions *such that* $P \cap T = \emptyset$*,* $F \subseteq P \times T \cup T \times P$ *is the* flow-relation*, and* $m_I : P \to \mathbb{N}$ *is the* initial marking*.*

We will use the usual notation for pre- and post-sets of nodes $x \in P \cup T$, i.e., $\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$ and $x\bullet = \{y \in P \cup T \mid (x, y) \in F\}$. The state of a P/T net is given by a *marking* of the places, which is formally a multi-set over the places $m : P \to \mathbb{N}$. Since sets are a special cases of multi-sets, we will use the notation $\bullet x$ to denote both the set $\bullet x$ as defined above, but also the multi-set given by $y \mapsto 1$ when $y \in \bullet x$ and $y \mapsto 0$ when $y \notin \bullet x$. We will assume that the relations $<$, $\leq$, $>$, and $\geq$, and operations $+$ and $-$, on multi-sets are defined as usual, i.e. for two multi-sets, $m_1, m_2 : P \to \mathbb{N}$, $m_1 \leq m_2 \iff \forall p \in P. m_1(p) \leq m_2(p)$, $m_1 < m_2 \iff m_1 \leq m_2 \wedge m_1 \neq m_2$, $(m_1 + m_2)(p) = m_1(p) + m_2(p)$, and $(m_1 - m_2)(p) = m_1(p) - m_2(p)$ when $m_1 \leq m_2$ and $m_1 - m_2$ is undefined when $m_1 \not\leq m_2$.

Definition 2. *A transition $t \in T$ is **enabled** in marking $m : P \to \mathbb{N}$ if $m \geq \bullet t$. If t is enabled in m, it can **occur** and lead to marking m'. This is written $m\,[t\rangle\, m'$, where m' is defined by $m' = (m - \bullet t) + t\bullet$.*

We will use the common notation $m\,[\sigma\rangle\, m'$ for $\sigma = t_1 t_2 \ldots t_n \in T^*$ to mean $\exists m_i : P \to \mathbb{N}$ for $i = 0, \ldots, n$ such that $m = m_0$, $\forall i = 0, \ldots, n-1. m_i\,[t_i\rangle\, m_{i+1}$, and $m' = m_n$. We will also write $m\,[*\rangle\, m'$ to mean $\exists \sigma \in T^*$ such that $m\,[\sigma\rangle\, m'$. We say that a marking m' is *reachable* from another marking m if $m\,[*\rangle\, m'$ and we let $[m\rangle = \{m' \mid m\,[*\rangle\, m'\}$ denote the set of markings reachable from m. When we talk about the set of *reachable markings* of a P/T net, we usually mean the set of markings reachable from the initial marking, i.e., $[m_I\rangle$. We will use R to denote the number of reachable markings, i.e., $R = |[m_I\rangle|$.

The reachability graph of a P/T net is a rooted graph that has a vertex for each reachable marking and an edge for each possible transition from one reachable marking to another.

Definition 3. *A **graph** is a tuple $(V, E, \mathsf{src}, \mathsf{trg})$ where V is a set of* vertices*, E is a set of* edges*, and $\mathsf{src}, \mathsf{trg} : E \to V$ are mappings assigning to each edge a* source *and a* target*, respectively. A **rooted graph** is a tuple $(V, E, \mathsf{src}, \mathsf{trg}, r)$ such that $(V, E, \mathsf{src}, \mathsf{trg})$ is a graph and $r \in V$ is the* root.

Definition 4. *Let $\mathcal{N} = (P, T, F, m_I)$ be a P/T net. The **reachability graph** of $\mathcal{N}$ is the rooted graph $(V, E, \mathsf{src}, \mathsf{trg}, r)$ defined by:*

- *$V = [m_I\rangle$—the set of nodes is the set of reachable markings.*
- *$E = \{(m, t, m') \in V \times T \times V \mid m\,[t\rangle\, m'\}$—the set of edges is the set of transitions from one reachable marking to another.*
- *src is given by $\mathsf{src}(m, t, m') = m$.*
- *trg is given by $\mathsf{trg}(m, t, m') = m'$.*
- *$r = m_I$—the root is the initial marking.*

We can only represent a finite reachability graph, but the reachability graph for a P/T net need not be finite, so we put some restrictions on the P/T net we consider to ensure a finite reachability graph. The first assumption we make is that the P/T net under consideration, $\mathcal{N} = (P, T, F, m_I)$, has a finite set of places, $|P| < \infty$, and a finite set of transitions, $|T| < \infty$. The second assumption is that the net is *k-bounded* for some $k \in \mathbb{N}, k > 0$, as defined below, and consider the set of possible markings to be $\mathbb{K}^P$ where $\mathbb{K} = \{0, 1, \ldots, k\}$.

Definition 5. *A P/T net (P, T, F, m_I) is k-**bounded** if and only if for all $m \in [m_I\rangle$ and for all $p \in P$: $m(p) \leq k$.*

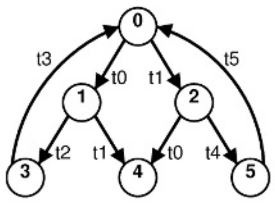
Although the assumptions above ensure that the reachability graph is finite, it is still necessary to distinguish between $|\mathbb{K}^P|$ different states when we calculate the reachability graph. If we let S denote the number of possible states, $S = |\mathbb{K}^P|$, at least $\lceil \log_2 S \rceil$ bits are needed per state. Most likely more bits will be used since the naive representation of a state vector assigns $\lceil \log_2 (k+1) \rceil$ bits per place using $|P| \cdot \lceil \log_2 (k+1) \rceil$ bits per state. Our goal is to reduce this to $\lceil \log_2 R \rceil$ bits per state.

3 Condensed Graph Representation

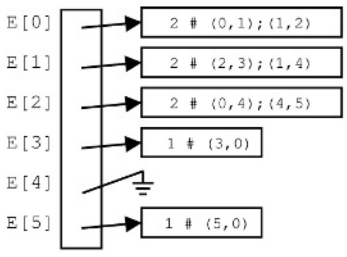
We now turn to the problem of mapping the full markings to the condensed representation. Our approach is to assign to each reachable marking a unique integer between 0 and $R-1$, which can be represented by $\lceil \log_2 R \rceil$ bits. In this section we describe the data structure used to represent the reachability graph $\mathcal{G} = (V, E, \mathsf{src}, \mathsf{trg}, m_I)$ in this condensed form, and how to construct it from the sets V and E as calculated by the reachability graph construction algorithm. Calculating the full reachability graph and then reducing it, defeats the purpose of using a condensed representation. We only describe the algorithm in this way to present the condensed representation in an uncomplicated setting, and we will later discuss how to construct the condensed representation on-the-fly.

3.1 Representing the Reachability Graph

We want to represent V by the numbers 0 to $R-1$. For a marking $m \in V$ we will let $\mathsf{idx}_M(m) \in \{0, 1, \ldots, R-1\}$ denote the (unique) index of m in this range. We will represent the initial marking m_I by index 0, $\mathsf{idx}_M(m_I) = 0$. With this representation of V, we can represent the set of edges as an array, `E`, with R entries, where each entry, `E[`i`]`, points to an array containing the edges out of the vertex v with index i. The array pointed to by `E[`i`]` consists of a header—a number, indicating the length of the array, so we can later decode the array—and the edges $\{(m, t, m') \in E \mid \mathsf{idx}_M(m) = i\}$. Each edge (m, t, m') is represented as a pair $(\mathsf{idx}_T(t), \mathsf{idx}_M(m'))$ where the first element is the index of the transition—we assume some statically defined mapping $\mathsf{idx}_T : T \to \{0, \ldots, |T|-1\}$ assigning a number to each transition—and the second element is the index of the target node of the edge. An example of this representation is shown in Fig. 1.



(a) Graph.



(b) Condensed representation.

Fig. 1. Representation of the reachability graph. The condensed representation of the graph in (a) is shown in (b). The edge array `E[`$\mathsf{idx}_M(v)$`]` for vertex v is written in the form n`#(`$\mathsf{idx}_T(t_0)$`,`$\mathsf{idx}_M(v_0)$`) ; ... ; (`$\mathsf{idx}_T(t_n)$`,`$\mathsf{idx}_M(v_n)$`)` where $n+1$ is the length of the array and the pairs represent the edges out of v. To save memory we represent a pointer to the empty array as a grounded pointer.

Each of the pairs in the edge arrays can be represented with $\lceil \log_2 |T| \rceil + \lceil \log_2 R \rceil$ bits. In addition there is an overhead of one pointer and one number for each state in V. We assume that all edge arrays can be represented in main memory and thus that we can represent both the pointer and the number in a computer word each.[1] With this encoding, we can represent the graph $\mathcal{G} = (V, E, \mathsf{src}, \mathsf{trg}, m_I)$ using just $2wR + |E| (\lceil \log_2 |T| \rceil + \lceil \log_2 R \rceil)$ bits, where w denotes the number of bits in a computer word. Notice that this efficient representation is only possible because of our mapping $\mathsf{idx}_M : V \to \{0, \ldots, R-1\}$, which saves us from storing any of the R markings explicitly.

From the sets V and E of $\mathcal{G}$, the translation of the reachability graph to the condensed representation is as one would expect: We build the mapping idx_M as a table mapping nodes to numbers, allocate the array `E` and the individual edge arrays, and insert the data in the arrays.

3.2 Exploring the Condensed Reachability Graph

The condensed representation for the reachability graph explicitly contains the transition structure but does not store any information about the markings. For some applications, such as protocol consistency using language equivalence [3], this suffices; for other applications, however, we are interested in both marking and transition information. For such applications we need a method of recreating the markings from the transition information, without significant blowup in the memory requirements. The property that we will exploit for this is the marking equation, $m' = m - \bullet t + t \bullet$, from Def. 2.

When we follow an edge (i, t, i') in the condensed representation, where we know the marking of i, we calculate the marking of i' using the marking equation. If we explore the reachability graph in a depth-first manner, we can even use the rewriting of the marking equation, $m = m' - t \bullet + \bullet t$, to obtain the marking of i from the marking of i' when we return along the edge. Exploiting this, it is possible to do a depth-first graph exploration, storing only one single marking explicitly at any one time, while still having the full state vector available at each visited state. An algorithm for this is shown in Fig. 2.

By extending the algorithm in Fig. 2 with a table of sub-expressions indexed by $1, \ldots, R-1$, it can be used to check Computation Tree Logic (CTL) as in [8, Sect. 4.1], and by extending the algorithm to use nested depth-first search [17], it can be adapted to check Linear Temporal Logic (LTL).

4 Creating the Condensed Representation On-the-Fly

To calculate the condensed representation on-the-fly we want to construct the idx_M mapping as new markings are calculated, and create the edge array at `E[`$\mathsf{idx}_M(m)$`]` as soon as the successors of m have been calculated.

[1] It is possible to represent both number and pointer in $\lceil \log_2 |E| \rceil$ bits, but representing both in a computer word of a fixed size independent of $|E|$ simplifies the constructions for creating the representation on-the-fly.

```
visited := $\emptyset$
$m$ := $m_I$
DFS(0)

where proc DFS($i$) is
    if $i \in$ visited return
    /* analyse $m$ here */
    visited := visited $\cup \{i\}$
    for each $(t, i')$ in E[$i$] do
        $m$ := $m - \bullet t + t \bullet$
        DFS($i'$)
        $m$ := $m + \bullet t - t \bullet$
    end for
end proc DFS
```

Fig. 2. Depth-first traversal of the reachability graph. A global variable m contains the current marking during the exploration. This marking is updated before and after each recursive call. The set `visited` keeps track of the visited nodes, can efficiently be implemented as a bit vector.

A few subtleties complicate the construction: we do not know the number R, and therefore we cannot immediately allocate the array `E`, nor can we allocate the individual edge arrays. There is also a problem with storing the numbers in the representation of the idx_M mapping, since we do not know how many bits are needed to store the numbers $\{0, \ldots, R-1\}$. We will assume, however, that $R < 2^w$, and we can therefore represent the numbers in the table using computer words. This is potentially a waste of memory, when $\log_2 R \ll w$, but it is not likely to be a bottleneck; the majority of the memory used by the idx_M mapping (represented as a table mapping full state vectors to numbers) will be for storing the full state vectors, which will end up using $R \cdot \lceil \log_2 S \rceil$ bits. Reduction of the memory needed for storing the full state vectors in the representation of the idx_M mapping is addressed in Sect. 5.

For managing the array `E` note that the entries in `E` are all of size w bits and do not depend on the total size of $[m_I\rangle$. We can work on the *entries* of `E` without knowing the full size of `E`. For handling `E` itself one possibility is using a dynamically extensible array [9], expanding and relocating as needed with an amortised constant time complexity. The dynamic array approach potentially allocates an array that is too large, but will not allocate more than twice the required storage, that is, the dynamic array will use between $R \cdot w + w$ and $2 \cdot R \cdot w + w$ bits of memory (where the $+w$ is a word needed to keep track of the size of the array). To be able to relocate the dynamic array, an additional $R \cdot w$ bits of memory might be needed.

After calculating all the successors of a marking m, we can construct the edge array for m. At this point we have added all successors of m to the representation of idx_M, and since we know the number of successors, we know the size of the edge array. In the edge array we can represent each successor, m', as $\mathsf{idx}_M(m')$, using w bits. Since we have added all successors of m to the representation of idx_M, we know the maximal index, M, used in the edge array for m, so we can actually represent each successor using only $\lceil \log_2 M \rceil$ bits. With this encoding, the bits allocated per marking will now vary between the different edge arrays. To decode the arrays we must store this number with the arrays. We therefore extend the header of the edge arrays, such that it now contains both the number of edges in the array and also the number of bits allocated per marking.

5 Reducing Peak Memory Usage

When creating the condensed representation of the reachability graph as described in Sect. 4, memory is wasted because, when the algorithm terminates, the memory holds both the graph, the set of reachable markings, and the idx_M mapping. In this section we use the sweep-line method [25,7] to keep peak memory usage small by deleting entries in the idx_M mapping.

5.1 The Sweep-Line Method

When constructing the reachability graph, it is neccesary to distinguish between new states and already visited states. For this we need to store the already visited states in memory. However, there is no need to store any states that are not reachable from the unprocessed states. Once a state is no longer reachable from the unprocessed states, it can be safely removed from memory.

The sweep-line method exploits this observation to delete states, using an approximation of the reachability relation, called a *progress measure*. The progress measure provides an ordering of the markings; states ordered less than the unprocessed states are assumed to be unreachable from the unprocessed states, and can therefore be deleted.

Definition 6 (Def. 3 in [25]). *For a P/T net* (P, T, F, m_I) *a* ***progress measure*** *is a tuple* $\mathcal{P} = (\mathcal{V}, \sqsubseteq, \psi)$ *where* $\mathcal{V}$ *is a set of* progress values*,* $\sqsubseteq$ *is a* partial order *of* $\mathcal{V}$*, and* $\psi : \mathbb{N}^P \rightarrow \mathcal{V}$ *is a mapping assigning a progress value to each marking. We say that* $\mathcal{P}$ *is* ***monotone*** *if* $m\,[*\rangle\, m'$ *implies* $\psi(m) \sqsubseteq \psi(m')$*.*

For monotone progress measures, the assumption that states with lower progress values are unreachable from the unprocessed states, is correct. For non-monotone progress measures, it is no longer safe just to delete states. To address this problem, we save the target nodes of edges that are not monotonic—so-called *regress edges:* (m, t, m') such that $\psi(m) \not\sqsubseteq \psi(m')$—as *persistent* markings and never delete persistent markings. The states saved as persistent in a sweep of the state space are either previously seen states or new states; there is no way for the algorithm to know which. When we see regress edges, we therefore perform another sweep, using the new persistent states as roots for the sweep. We repeat this until we no longer find new persistent states. For details of this algorithm, see [25]. A detailed example of the construction and optimisation of a progress measure can also be found in [25].

The observation used in the sweep-line method to delete states can also be used to clean up the idx_M mapping. When constructing the condensed graph representation, we only need to store the index mapping of markings we can reach from the currently unprocessed states. Using the sweep-line method for exploring the reachability graph, we can reduce the peak memory usage by deleting states in the set V and the idx_M mapping. Deleting states is only safe if the progress measure is monotone; otherwise, the condensed graph may be an unfolding of the full graph. This is treated in Sect. 5.2.

The algorithm combining the sweep-line method and the construction of the condensed graph representation is shown in Fig. 3. Like the sweep-line algorithm, this algorithm performs a number of sweeps until it no longer finds new persistent states (lines 7–9). Each sweep (lines 11–35) consists of processing unprocessed states in order of their progress measure (lines 15–18), assigning indices to their previously unseen successors (lines 21–22), and either adding the new successors to the set of unprocessed states (line 24) or to the set of persistent states and roots for the next sweep (lines 26–27). When all successors of a state are processed, the edge array is updated (line 31) using the method `CREATE_EDGE_ARRAY` (lines 37–43) as described in Sect. 4, and states behind the sweep-line are removed from the set V and the index mapping idx_M (lines 32–33).

By using this algorithm we only store a subset of the reachable markings explicitly while creating the condensed graph. This enables us to construct the reachability graph, in the condensed representation, in cases where storing all reachable markings in memory is impossible.

5.2 An Unfolding of the Reachability Graphs

When using a non-monotone progress measure, the reachability graph obtained from the algorithm in Fig. 3 is not the reachability graph from Def. 4; rather it is an *unfolding* of this graph [26, Chap. 13]. For poor choices of progress measures, this unfolded graph can be much larger than the original reachability graph, completely eliminating the benefits of reduction. For good choices of the progress measures, the blowup in size will be manageable and the condensed representation of nodes more than compensates for the graph unfolding. It is important to consider the relationship between the unfolded graph and the original reachability graph, to know which properties are preserved by the unfolding.

The unfolding is due to regress edges—edges along which the progress measure decreases. When following a regress edge we may reach a state which has previously been explored and since the actual marking has been deleted, we do not recognise it and explore its successor states again.

One can easily define the unfolded graph, $\mathcal{G}^u$, and show that it is bisimilar to the full reachability graph [26, Chap. 13]. This result is especially interesting in the context of model checking, since bisimulation is known to preserve CTL* in the sense of Theorem 1, which in turn implies that both CTL and LTL, the most commonly used temporal logics for model checking, are preserved.

Theorem 1 (From [8, Chap. 12]). *If $\mathcal{G}$ and $\mathcal{G}'$ are bisimilar then for every CTL* formula ϕ we have $\mathcal{G} \models \phi \Leftrightarrow \mathcal{G}' \models \phi$.*

```
$V$ := $\{m_I\}$
$Roots$ := $\{m_I\}$
$Persistent$ := $\emptyset$
$\mathsf{idx}_M$($m_I$) := $0$
$n$ := $1$

while $Roots \neq \emptyset$ do
    SWEEP($Roots, V, Persistent$,$\mathsf{idx}_M$,$n$)
end while

where proc SWEEP($Roots, V, Persistent, \mathsf{idx}_M, n$) is
    $U$ := $Roots$
    $Roots$ := $\emptyset$
    while $U \neq \emptyset$ do
        select $m \in U$ s.t. $\nexists m' \in U : \psi(m') \sqsubset \psi(m)$
        $U$ := $U - \{m\}$
        $X$ := $\{t, m') \,|\, m\,[t\rangle\, m'\}$
        for all $(t, m') \in X$ do
            if $m' \notin V$ then
                $V$ := $V \cup \{m'\}$
                $\mathsf{idx}_M$($m'$) := $n$
                $n$ := $n + 1$
                if $\psi(m) \sqsubseteq \psi(m')$ then
                    $U$ := $U \cup \{m'\}$
                else
                    $Persistent$ := $Persistent \cup \{m'\}$
                    $Roots$ := $Roots \cup \{m'\}$
                end if
            end if
        end for
        E[$\mathsf{idx}_M$($m$)] := CREATE_EDGE_ARRAY($X$,$\mathsf{idx}_M$)
        $V$ := $\{m \in V \mid \exists m' \in U : \psi(m') \sqsubseteq \psi(m)\} \cup Persistent$
        $\mathsf{idx}_M$ := $\{m \mapsto i \,|\, m \in V \wedge \mathsf{idx}_M(m) = i\}$
    end while
end proc SWEEP

where proc CREATE_EDGE_ARRAY($X, \mathsf{idx}_M$) is
    M := $\max\{\mathsf{idx}_M(m') \,|\, (t, m') \in X|\}$
    A := allocate $2 \cdot w + |X| \cdot (\lceil \log_2 |T| \rceil + \lceil \log_2 M \rceil)$ bits
    A.header := ($|X|$, $\lceil \log_2 M \rceil$)
    A.edges  := ($\mathsf{idx}_T(t)$, $\mathsf{idx}_M$($m'$)) for each $(t, m') \in X$
    return A
end proc CREATE_EDGE_ARRAY
```

Fig. 3. The sweep-line method for obtaining a condensed graph representation.

6 Experimental Results

In order to validate and evaluate the performance of the new algorithm a proof-of-concept implementation has been developed. For the theoretical presentation in the previous sections we used Place/Transition Petri nets; the techniques introduced, however, generalise to higher level net classes, such as *coloured Petri nets* (CPN) [22], in a straightforward manner. The prototype is build on top of the Design/CPN tool [1], a tool for the construction and analysis of CPNs. The prototype is implemented in the *Standard ML* (SML) programming language [32] and the progress measure is provided by the user as an SML function.

Since the Design/CPN tool is used for analysing CPN models the markings of the nets are not multi-sets over places but multi-sets over more complex data types. Consequently the markings are not integer vectors of length $|P|$, but variable-length encodings of the more complex markings. On the edges of the reachability graph it is no longer sufficient to store transitions, also the bindings are needed.

The prototype implementation of the new algorithm is slightly simpler than the algorithm described in this paper. We do not implement the variable-length numbers for node indices, but represent each index as a four byte computer word. This greatly simplifies the implementation but uses slightly more memory for smaller systems and limits the prototype to models with less than 2^{32} states, which is no serious limitation.

All experiments were conducted on a 500Mhz Pentium III Linux PC with 128 Mb of RAM.

Database Replication Protocol. The first example we consider is a database replication protocol [22, Sect. 1.3]. The protocol describes the communication between a set of database managers for maintaining consistent copies of a distributed database. When a database manager updates its local copy of the database it broadcasts an update request to all other database managers who then perform the update on their local copies and then acknowledge that the update has been performed. The progress measure for the protocol is based on the control flow of the database managers and an ordering on the database managers. See [25] for details.

Table 1 shows the performance of full reachability graph generation compared with the new algorithm. The $|D|$ column shows the number of database managers in the different configurations, the following four columns show the values for the full reachability graph, and the last four columns show the values for the new algorithm. In the full reachability graph columns the *States* column shows the number of states for each configuration, the *Avg* column shows the average number of bytes in the state vector in the different configurations, the *Memory* column shows the total memory usage in bytes for storing all states, and the *Time* column shows the time used for calculating the reachability graph in seconds. In the sweep-line columns the *States* column shows the number of states explored by the sweep-line algorithm, the *Peak* column shows the peak number of states stored during the exploration, the *Memory* column shows the number of bytes

Table 1. Database Replication Protocol.

	Full Reachability Graph				**Sweep-Line based Algorithm**					
$\|D\|$	States	Avg	Memory	Time	States	Peak	Memory	(%)	Time	(%)
4	110	122	13,420	0	219	14	2,584	(19)	0	(-)
5	407	146	59,422	0	813	33	8,070	(14)	0	(-)
6	1,460	169	246,740	0	2,919	88	26,548	(11)	2	(-)
7	5,105	191	975,055	4	10,209	251	88,777	(9)	11	(275)
8	17,498	214	3,744,572	23	34,995	738	297,912	(8)	51	(222)
9	59,051	237	13,995,087	105	118,101	2,197	993,093	(7)	229	(218)

used for storing the states in the condensed representation plus the states in *Peak*, the number in the parentheses indicates the memory consumption of the condensed representation as a percentage of the full representation, the *Time* column shows the time used for calculating the condensed graph, and the number in parentheses shows the amount of time used for calculating the condensed representation as a percentage of the amount of time used to generate the full representation.

In the database replication protocol all states but the initial state are explored twice by the sweep-line algorithm, and consequently the condensed graph has twice as many nodes as the full graph and the time for calculating the condensed graph is roughly twice as long as the time for calculating the full reachability graph. The *Memory* in the sweep-line columns is calculated as $4 \cdot States + Avg \cdot Peak$ since one computer word (4 bytes) is used for representing each condensed state and $Avg \cdot Peak$ bytes are used for representing the states on the sweep-line. We only compare the memory usage for storing the states, as the memory usage for storing the remaining graph structure would be comparable for the two methods. Although the unfolded graph generated by the sweep-line method contains twice as many nodes as the original reachability graph the memory usage—as seen in the two Memory columns—is significantly improved. For four database managers the reduction is down to around 20%, while for nine database managers the reduction is further improved, down to around 7% of the full representation.

Stop and Wait Communication Protocol. The second example is a stop-and-wait communication protocol [24]. The protocol is parameterised with the number of packets to be sent. We use the number of packets successfully received as a monotone progress measure [7]. The performance is shown in Table 2. Here the *# packets* column shows the number of packets to be transmitted in the different configurations; the remaining columns have the same meaning as in Table 1.

For this model the peak number of states fully stored in the sweep-line method does not increase for larger configurations. As the number of packets increases the total number of states increases, but the number of states with the same progress measure does not. As for the database replication protocol, the

Table 2. Stop and Wait Communication Protocol.

	Full Reachability Graph				Sweep-Line based Algorithm					
# packets	States	Avg	Memory	Time	States	Peak	Memory	(%)	Time	(%)
20	5,286	145	766,470	17	5,286	287	62,759	(8)	24	(141)
40	10,706	146	1,563,076	35	10,706	287	84,726	(5)	50	(143)
60	16,126	146	2,354,396	53	16,126	287	106,406	(5)	77	(145)
80	21,546	146	3,145,716	71	21,546	287	128,086	(4)	103	(145)
100	26,966	146	3,937,036	89	26,966	287	149,766	(4)	129	(145)

experiments shows significant memory reduction—from around 8% for 20 packets to around 4% for 100 packets—at the cost of a slight increase in runtime—an increase about 45%–50% of the runtime of the full reachability graph algorithm in all configurations.

7 Conclusion

In this paper we have presented a condensed representation of the reachability graph of P/T nets. The condensed graph represents each marking with a number in $\{0, 1, \ldots, R-1\}$, where $R = |[m_I\rangle|$, and avoids representing markings explicitly. We have developed an algorithm that constructs this representation exploiting local information about successor markings only to represent edges efficiently without knowing R, and dynamic arrays for storing edge information for each node. Using the sweep-line method we are able to reduce peak memory usage during the construction of the graph representation. When the progress measure used is monotone, the graph is isomorphic to the original reachability graph, and when the progress measure is non-monotone the graph is bi-similar to the original graph.

We have demostrated the performance of the new algorithm using two examples. The chosen examples have a quite clear notion of progress, so the sweep-line method performs well, and the amount memory used to store the reduced graphs is significaltly less than the amount of memory used to store the full graphs. The presented algorithm will not perform well on systems with little or no progress. An example of a system with little progress is the Dining Philosophers problem. If we use the number of eating philosophers as progress measure, we will at some time during the construction store nearly all states, and the memory used for storing the compact representation is overhead. Compared to the amount of memory used for storing the full state vectors, this amount is not significant, however, and the only real disadvantage is that we still use extra time for the construction. If the number of reachable states is close to the number of syntactically possible states, the amount of memory used for the condensed representation is comparable to the amount of memory used for the full representation, and little is gained from using the new algorithm.

By exploiting the marking equation of P/T nets, the ability to calculate the predecessor or successor of a state given a transition, we are able to reconstruct the markings of the reduced nodes while exploring the graph. In general, when the predecessors and successors can be deterministically determined, this approach can be used. If only successors can be calculated deterministically, the reachability graph can still be traversed and states reconstructed, by saving the current state on the depth-first stack before processing successors.

The algorithm presented here resembles the approach used in [14], where the basic sweep-line method (applicable to monotone progress measures only) was used to translate the reachability graph of a CPN model to a finite state automaton, which in turn was used to check language equivalence between a protocol specification and its service specification. In this approach the automaton is constructed by writing edge-information onto a disk before the sweep-line method garbage collects the edges, and this edge-information is the processed by another tool to translate it to an automaton. On the disk the states are represented as numbers, thus reducing memory consumption when the automaton is constructed from the file.

Using the graph construction algorithm presented in this paper, the potentially expensive step of going through a disk-representation can be avoided when constructing the language automaton. Furthermore, with the algorithm in Fig. 2 it is possible to traverse the graph reconstructing state information after the graph is constructed. The results from Sect. 5.2, relating the reachability graph to the unfolded graph, can also be used to generalise the method from [14] to non-monotone progress measures. In [14] the basic sweep-line method from [7] is used, guaranteeing that the automaton generated represents the language of the protocol being analysed. The results in Sect. 5.2 ensure that, when using non-monotone progress measures, the unfolded graph is language equivalent to the original reachability graph.

The new algorithm is designed for explicit state reachability graph analysis. For condensed state representation, such as finite automata [20], or for symbolic model checking [28, 5], where states are represented as e.g., Binary Decision Diagrams [4], the memory used for storing a set of states does not depend directly on the number of states in the set, but on regularity in the state information. Deleting states during the graph construction, as the sweep-line method does, will not necessarily reduce memory usage. On the contrary, deleting states can actually increase the memory needed to store the set of states. Combining the new algorithm with symbolic model checking, therefore, does not appear to be immediately possible.

The new technique reduces the memory usage using knowledge about the number of reachable states, and complements techniques that are aimed at efficiently representing arbitrary states from the set of syntactically possible states. The state representation in SPIN [18], Design/CPN [6], and MARIA [27], for example, exploit modularity of the system being analysed to share parts of the state vector between different states. LoLA [30] exploits invariants to avoid storing information that can be derfived from the invariant. Using one or more of these approaches one can represent sets of arbitrary states efficiently, though at least $\lceil \log_2 S \rceil$ bits are still needed per state to distinguish between S syntactically possible states. [12] considers storing sets of markings efficiently using very tight hash tables, which allows storing sets of states using less than $\lceil \log_2 S \rceil$ bits per state, but using the knowledge about the number of reachable states is not considered. Representing arbitrary states efficiently benefits the algorithm presented here as well, by reducing the memory needed for the table mapping states to indices. The reduction differs from probabilistic methods such as bit-state hashing [19,15] and hash-compaction [31,34], where all possible states are, in a sense, mapped onto a range $\{0, 1, \ldots, n\}$, for some n, but with a mapping that may not be injective on $[m_I\rangle$. The states are in this way also represented in a condensed form, but since hash collisions can occur, full coverage of the reachability graph cannot be guaranteed.

With the algorithm presented here, the sweep-line method can be used for checking more general properties than just state properties as in [25]. In particular, checking CTL* formulae, and thereby CTL and LTL formulae, now becomes possible. Future work includes using this in case studies.

References

1. Design/CPN. Online `http://www.daimi.au.dk/designCPN`.
2. G. Behrmann, K.G. Larsen, and R. Pelánek. To Store or Not to Store. In W.A. Hunt and F. Somenzi, editors, *Proc. of CAV 2003*, volume 2725 of *LNCS*, pages 433–445. Springer, 2003.
3. J. Billington, M.C. Wilbur-Ham, and M.Y. Bearman. Automated protocol Verification. In *Proc. of IFIP WG 6.1 5th International Workshop on Protocol Specification, Testing, and Verification*, pages 59–70. Elsevier, 1985.
4. R.E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
5. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.
6. S. Christensen, J.B. Jørgensen, and L.M. Kristensen. Design/CPN—A Computer Tool for Coloured Petri Nets. In *Proc. of TACAS'97*, volume 1217 of *LNCS*, pages 209–223. Springer-Verlag, 1997.
7. S. Christensen, L.M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proc. of TACAS'01*, volume 2031 of *LNCS*, pages 450–464. Springer-Verlag, 2001.
8. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.
9. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*, chapter 18.4, pages 367–375. The MIT Press, 1990.
10. J. Desel and W. Reisig. Place/Transition Petri Nets. In *Lecture on Petri nets I: Basic Models*, volume 1491 of *LNCS*, pages 122–173. Springer-Verlag, 1998.
11. E.A. Emerson and A.P. Sistla. Symmetry and Model Checking. *Formal Methods in System Design*, 9, 1996.
12. J. Geldenhuys and A. Valmari. A Nearly Memory-Optimal Data Structure for Sets and Mappings. In T. Ball and S.K. Rajamani, editors, *Proc. of SPIN 2003*, volume 2648 of *LNCS*, pages 136–150. Springer, 2003.

13. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems—An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer-Verlag, 1996.
14. S. Gordon, L.M. Kristensen, and J. Billington. Verification of a Revised WAP Wireless Transaction Protocol. In *Proc. of ICATPN'02*, volume 2360 of *LNCS*, pages 182–202. Springer-Verlag, 2002.
15. G.J. Holzmann. An Improved Protocol Reachability Analysis Technique. *Software, Practice and Experience*, 18(2):137–161, 1988.
16. G.J. Holzmann. Algorithms for Automated Protocol Validation. *AT&T Technical Journal*, 69(2):32–44, 1990.
17. G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International Editions, 1991.
18. G.J. Holzmann. State Compression in SPIN: Recursive Indexing and Compression Traning Runs. In *Proc. of 3rd SPIN Workshop*, 1997.
19. G.J. Holzmann. An Analysis of Bitstate Hashing. *Formal Methods in System Design*, 13:289–307, 1998.
20. G.J. Holzmann and A. Puri. A Minimized Automaton Representation of Reachable States. *Journal on Software Tools for Technology Transfer*, 2(3):270–278, 1999.
21. C.N. Ip and D.L. Dill. Better Verification Through Symmetry. *Formal Methods in System Design*, 9, 1996.
22. K. Jensen. *Coloured Petri Nets—Basic Concepts, Analysis Methods and Practical Use. Volume 1: Basic Concepts*. Springer-Verlag, 1992.
23. K. Jensen. *Coloured Petri Nets—Basic Concepts, Analysis Methods and Practical Use. Volume 2: Analysis Methods*. Springer-Verlag, 1994.
24. L.M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.
25. L.M. Kristensen and T. Mailund. A Generalised Sweep-Line Method for Safety Properties. In *Proc. of FME'02*, volume 2391 of *LNCS*, pages 549–567. Springer-Verlag, 2002.
26. T. Mailund. *Sweeping the State Space — A Sweep-Line State Space Exploration Method*. PhD thesis, Department of Computer Science, University of Aarhus, 2003.
27. M. Mäkelä. Condensed Storage of Multi-Set Sequences. In K. Jensen, editor, *Proc. of Workshop on Practical Use of High-level Petri Nets*, number DAIMI PB-547, pages 111–126. University of Aarhus, 2000.
28. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
29. D. Peled. All for One, One for All: On Model Checking Using Representatives. In *Proc. of CAV'93*, volume 697 of *LNCS*, pages 409–423. Springer-Verlag, 1993.
30. K. Schmidt. Using Petri Net Invariants in State Space Construction. In H. Garavel and J. Hatcliff, editors, *Proc. of TACAS 2003*, volume 2619 of *LNCS*, pages 473–488. Springer, 2003.
31. U. Stern and D.L. Dill. Improved Probabilistic Verification by Hash Compaction. In *Correct Hardware Design and Verification Methods*, volume 987 of *LNCS*, pages 206–224. Springer-Verlag, 1995.
32. J.D. Ullman. *Elements of ML Programming*. Prentice-Hall, 1998.
33. A. Valmari. Stubborn Sets for Reduced State Space Generation. In *Advances in Petri Nets '90*, volume 483 of *LNCS*, pages 491–515. Springer-Verlag, 1990.
34. P. Wolper and D. Leroy. Reliable Hashing without Collision Detection. In *Proc. of CAV'93*, volume 697 of *LNCS*, pages 59–70. Springer-Verlag, 1993.