Chapter 2

Behavioural Verification by Means of Reachability Graphs

This chapter considers the implementation of the verifier box from Fig. 1.4. The job of the verifier is to check whether a model, denoted by $\mathcal{M}$, satisfies a given property, denoted φ. If $\mathcal{M}$ satisfies φ, we say that $\mathcal{M}$ is a model of φ, and we write $\mathcal{M} \models \varphi$. The task of checking whether $\mathcal{M} \models \varphi$ is called *model checking*.

In this chapter we will introduce the basic idea behind reachability graph analysis (also known as state space analysis) and a number of reduction techniques, i.e., variations of the basic reachability graph algorithm that make analysis possible for larger systems of certain classes of models (which class depends on the reduction technique). We start by introducing the basic algorithm for reachability graph construction in Sect. 2.1. We then turn to describing reduction techniques in general in Sect. 2.2, and the sweep-line method [25, 104] in Sect. 2.2.1 and the hash compaction reduction technique [155, 172] in Sect. 2.2.2. We give a summary of the papers [T1] and [T2] co-authored by the author of this thesis in Sects. 2.3 and 2.4. Full versions of the papers [T1] and [T2] can be found in Chapters 5 and 6, respectively. The paper [T1] extends the sweep-line method to allow checking more complex properties and [T2] makes the incomplete hash compaction reduction technique complete. We sum up the chapter by discussing the contribution of the papers [T1] and [T2] and provide directions for future work.

2.1 Basic Reachability Graph Analysis

To make our discussion of behavioural verification independent of the concrete modelling formalism, we will use an abstract definition of the behaviour of a formal model, namely a labelled transition system. A labelled transition system captures the intuition that a formal model starts in a certain state and progresses according to a transition relation:

Definition 2.1 (Labelled Transition System) *A **labelled transition system (LTS)** is a tuple, $\mathcal{LTS} = (S, T, \Delta, s_I)$, where*

- *$S \neq \emptyset$ is a set of **states**,*
- *T is a set of **transitions**,*
- *$\Delta \subseteq S \times T \times S$ is the **transition relation** indicating **successor states**,*
- *$s_I \in S$ is the **initial state**.*

Let $s, s' \in S$ be two states and $t \in T$ a transition. If $(s, t, s') \in \Delta$, then t is said to be *enabled* in s and the *occurrence* (execution) of t in s leads to the state s'. This is also written $s \xrightarrow{t} s'$. An *occurrence sequence* is an alternating sequence of states s_i and transitions t_i written $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \cdots s_n \xrightarrow{t_n} s_{n+1}$ and satisfying $s_i \xrightarrow{t_i} s_{i+1}$ for $1 \leq i \leq n$. We use $\rightarrow^*$ to denote the transitive and reflexive closure of Δ, i.e., $s \rightarrow^* s'$ if and only if there exists an occurrence sequence $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \cdots s_n \xrightarrow{t_n} s_{n+1}$, $n \geq 1$, with $s = s_1$ and $s' = s_{n+1}$. A state s' is *reachable* from s if and only if $s \rightarrow^* s'$, and $\mathsf{reach}(s) = \{ s' \in S \mid s \rightarrow^* s' \}$ denotes the set of states reachable from s.

We will often like to verify some *invariant property*, $\mathcal{I} : S \rightarrow \{\mathsf{tt}, \mathsf{ff}\}$, of all states reachable from the initial state, i.e., check whether $\forall s \in \mathsf{reach}(s_I).\mathcal{I}(s)$ holds. The most naive way to do that is by checking if it holds for the initial state. If it does not, we know the property does not hold for all states. If the property does hold for the initial state, we recursively check the property for all successor states. It is evident that this algorithm does not terminate if the invariant holds and we can reach an infinite number of states, i.e., if $|\mathsf{reach}(s_I)| = \infty$ and $\forall s \in \mathsf{reach}(s_I).\mathcal{I}(s)$. Even if the number of reachable states is finite, the algorithm will not terminate if the invariant holds and it is possible to reach some state state from itself by a non-empty transition sequence, i.e., when $|\mathsf{reach}(s_I)| < \infty$ but $\forall s \in \mathsf{reach}(s_I).\mathcal{I}(s)$ and $\exists s_b, s \in \mathsf{reach}(s_I).s_b \rightarrow^* s' \rightarrow s_b$, as the recursive check of successor states will eventually encounter the state s_b, and, as $s_b \rightarrow^* s_b$, loop when trying the validate the invariant for s_b.

In order to overcome this problem, we build a reachability graph. A reachability graph is a directed labelled graph, where the nodes correspond to states of the model and a labelled arc from one node to another, signify that it is possible to go from the first state to the second using the transition corresponding to the label. Formally, the reachability graph is the directed graph (V, E) where $V = \mathsf{reach}(s_I)$ is the set of nodes and $E = \{(s, t, s') \in \Delta \mid s, s' \in V\}$ is the set of edges. An edge (s, t, s') has s as source and s' as destination and the label is t. The reachability graph can be constructed using Algorithm 1, which makes the recursion stack explicit as the data-structure W. The intuition is that W contains states for which we have not yet calculated successor states whereas V and E contains the nodes, respectively edges, of the reachability graph for the states for which we have already calculated successor states. This algorithm not only terminates as long as $|\mathsf{reach}(s_I)| < \infty$, it is also more efficient than the previous algorithm, as successors are only calculated once for each state. Using the reachability graph, we can traverse all states of V and check the invariant property $\mathcal{I}$, even if S is infinite as long as V is finite. Algorithm 1 terminates iff $|\mathsf{reach}(s_I)| < \infty$ and $\forall s \in \mathsf{reach}(s_I).|\{(s, t, s') \mid (s, t, s') \in \Delta\}| < \infty$. This is a dynamic property, however, and can only be decided by generating the reachability graph (or something equivalent). To obtain a syntactic way to decide if the reachability graph is finite, we observe that $\mathsf{reach}(s_I) \subseteq S$ and $\forall s \in S.\{(s, t, s') \mid (s, t, s') \in \Delta\} \subseteq \Delta \subseteq S \times T \times S$, so it is a sufficient but not necessary condition that S and T are finite for Algorithm 1 to terminate. If a Place-transition Petri net is *bounded*, i.e., if the number of tokens on all places in all reachable states is less than some constant, the set of possible states, S, is finite (or can be picked to be finite). If we furthermore assume that the PT-net only has a finite number of transitions, Algorithm 1 always terminates. Some PT-net models are bounded by design. As an example, 1-safe PT-nets only allow transitions to be executed if it does not lead to more than one token on any place. Initially all places contain at most one token, so the number of tokens never exceed 1. Some PT-net models can be shown to be bounded, e.g., using place invariants or coverability graphs as described later. We can imple-

ment this algorithm by representing V and E as hash tables and W using, e.g., a queue or a stack.

Algorithm 1 Basic reachability graph algorithm.

Require: $\mathcal{LTS} = (S, T, \Delta, s_I)$ a labelled transition system
Ensure: (V, E) the corresponding reachability graph

```
1:  V := {s_I}
2:  W := {s_I}
3:  E := ∅
4:
5:  while W ≠ ∅ do
6:     Select an s ∈ W
7:     W := W \ {s}
8:     for all t, s' such that s --t--> s' do
9:        E := E ∪ {(s, t, s')}
10:       if s' ∉ V then
11:          V := V ∪ {s'}
12:          W := W ∪ {s'}
13:
14: return (V, E)
```

If $|\mathsf{reach}(s_I)| = \infty$, Algorithm 1 will not terminate. If $|\mathsf{reach}(s_I)|$ is finite but very large, the algorithm may not terminate successfully. The problem that the reachability graph can be very large or infinite for even simple models is known as the *state explosion problem* [161]. The state explosion problem can be the cause for unsuccessful execution of Algorithm 1 for several reasons. As an example, the available memory can be exhausted causing the algorithm to terminate prematurely or causing the operating system to start swapping internal memory to disk, leading to vastly decreased performance of the algorithm as it it ill-suited for external memory. The problem is basically line 10 of Algorithm 1, as the check whether $s' \in V$ will require access to external memory almost every time. Another problem is that the execution may simply take too long for the result to be interesting, e.g., if the calculation takes two months but the space robot we verify has to launched in one month. If the reachability graph is infinite we must use a method to represent it using only a finite amount of memory, e.g., by representing the graph using graph grammars [140], representing equivalence classes of states of the real reachability graph [26,92], or by using a coverability graph [52,97] instead of a reachability graph. The first two ways of representing infinite reachability graphs can be used for any formalism, whereas the coverability graph can only be constructed for Petri nets.

The idea of the coverability graph is based on the observation that transitions of Place-transition Petri nets are monotone, i.e., that adding more tokens do not inhibit the execution of transitions or alter the effect of executing transitions[1]. Thus, if we reach a state, s', which has at least the same number of tokens on all places as a previously visited state, s, written $s' \geq s$, all transitions enabled in s will also be enabled in s' (this is the definition of monotonicity). Thus it is possible to execute the transition sequence leading from s to s' an arbitrary number of times, each time producing more tokens. We can replace the number of tokens on places in s', which contains strictly more tokens than

[1]Certain extensions of PT-nets do break monotonicity, however, e.g., inhibitor arcs [21] that checks for absence of tokens, or bounds on places that prevent adding more than a fixed number of tokens to places.

in s, with infinity (∞), representing that it is possible to generate an arbitrary number of tokens on these places. As an example, in the case of the PT-net model of a network protocol in Fig. 1.7, we see that by executing Send Data we reach a state where the number of tokens on all places except Network 1 are the same as in the initial state. On Network 1 we have one more token after executing Send Data, so we replace the number of tokens on Network 1 with ∞, signifying that by executing Send Data an arbitrary number of times, we can produce an arbitrary number of tokens on Network 1. The coverability graph can be used for, e.g., determining upper bounds on the number of tokens on each place and thus whether the PT-net is bounded, so it is possible to use the coverability graph to determine if the reachability graph of a PT-net is finite. The coverability graph method cannot immediately be used for CP-nets as it is not possible to define a canonical ordering of states which makes transitions monotone and still guarantees that the coverability graph is finite as the types of places can be infinite.

In the rest of this chapter we will only consider finite but large reachability graphs. We are thus interested in reduction techniques for storing reachability graphs efficiently.

2.2 Reduction Techniques

Reduction techniques for finite reachability graphs basically fall into two categories: algorithms for explicit representation of the reachability graph and algorithms for symbolic representation of the reachability graph. Reduction techniques for explicit reachability graph analysis, basically fall into four categories: methods that explore only a subset of the reachability graph directed by the verification question [42, 134, 160]; methods that use external storage to store the set of visited states [153, 156]; methods that delete states from memory during reachability graph exploration [25, 60]; and methods that store states in a compact manner in memory [T1, T2, 57, 76, 93]. Symbolic reachability graph analysis typically use binary decision diagrams (BDD) [12] or multi-valued decision diagrams (MDD) [96] to store states, or represent each state of the system as a propositional formula and rely on a SAT-solver [148], e.g., MiniSAT [43] or HyperSAT [83], to do bounded model checking [8]. We will first take a look at some symbolic techniques and then turn to examples of reduction techniques from each of the four categories.

Symbolic reachability graph analysis using BDDs relies on representing each state of the system as a bit-vector. A set of bit-vectors can be efficiently represented using a finite automaton accepting exactly the bit-vectors in the set, and BDDs are one way to represent such automatons efficiently. BDDs reduce the memory needed to store each state by sharing the representation of common parts of the bit-vectors. If we cannot represent the state as a bit-vector, e.g., in the case of PT-nets where we have no a priori bound for all places, we can use MDDs, which are able to represent strings of integers using similar techniques as BDDs.

The basic idea of bounded model checking is to encode all executions of the system after executing k transitions as a boolean formula M_k. We conjunct this with a formula, $\neg\varphi_k$, which states that a property φ does not hold after executing k transitions. If (and only if) $M_k \wedge \neg\varphi_k$ is satisfiable, an execution of length k which does not satisfy φ is found. We assume that a state of the system can be encoded as a vector, s, consisting of n boolean variables, $s[0], \ldots, s[n-1]$. We let s_0 be the initial state and let $I(s_0)$ be a propositional formula encoding the initial state. We let $T(s_{i-1}, s_i)$ be a formula that is satisfiable if there is a

transition leading from the state s_{i-1} to the state s_i. The formula M_k is then expressed as $M_k = I(s_0) \wedge \bigwedge_{i=1}^{k} T(s_{i-1}, s_i)$. If we take $k = 2^n$, we are guaranteed to reach all possible states, as at most 2^n different states can be encoded using n boolean variables, and after executing 2^n steps we are thus guaranteed to have reached either states with no successors or previously seen states and therefore we have discovered a loop in the reachability graph. If we want to model check an invariant property, $\mathcal{I}$, we can take $\varphi_k = \bigvee_{i=0}^{k} \neg\mathcal{I}(s_i)$. Bounded model checking relies on solving the problem of whether a propositional formula is satisfiable. This is referred to as the SAT problem and is a well-known NP-hard problem [55]. This means that no known algorithm can solve the problem in time polynomial in the number of variables. There exist, however, very efficient heuristics, and the advantage of bounded model checking is that we check all states reachable in k steps in a single iteration. Bounded model checking is only applicable if we can encode the state of the system as a vector of boolean variables, which is, e.g., the case for bounded Place-transition Petri nets when the bound is known in advance.

The main problem with symbolic model checking is that it works best if all reachable states can be represented using a bit-vector or vector of boolean variables of the same size. This is not the case for CP-nets, which is why we are mainly interested in explicit reachability graph analysis. For example the CPN model of a network protocol in Fig. 1.8 contains integers, which in principle can grow unbounded as well as a list of packets (on In Buffer) which is also in principle able to grow arbitrarily long. Even if a bound can be found, it will often be very large as CPN states are often hundreds of bytes, so symbolic model checking would need to deal with thousands of bits/boolean variables, rendering the methods virtually useless for CP-nets.

Now let us instead consider explicit reachability graph analysis. Depending on the property we want to check, we may not need to explore all reachable states to be able to provide correct answers. One way to only explore a subset of all states is to use a partial order reduction [28, 136]. Partial order reduction exploit the fact that some transitions can be executed in any order yielding the same result. As an example, consider a state, s. If we can execute the transitions a and b *concurrently*, i.e., if $s \xrightarrow{a} s' \xrightarrow{b} s'''$ and $s \xrightarrow{b} s'' \xrightarrow{a} s'''$. If we are only interested in the behaviour after executing both transitions, we only need to consider one of the two execution sequences. This allows us to check, e.g., whether a system has any *dead-locks*, i.e., states with no enabled transitions. As the number of possible execution sequences grow exponentially as a function of the number of concurrently enabled transitions, only exploring one (or a few) of them yields a huge optimisation. The problem is, of course, identifying such sets of transitions. For PT-nets, we can check if all tokens required by a set of transitions are available by adding the number of tokens consumed for all transitions of the set. If all tokens are available the transitions can be executed concurrently. For non-monotone formalisms the analysis is more complex, as we also have to check whether the execution of one of the transitions in the set can inhibit the enabling of some of the others. There are numerous variants of partial order reductions, such as ample sets [134, 135], persistent sets and sleep sets [58, 59, 171], and stubborn sets [160]. Another way to only explore parts of the reachability graph is to use a weight function, which assigns higher weight to transitions that are likely to lead to states violating the property we wish to check. This is known as directed model checking [42], and such weight functions can either be provided manually by the user or, in some cases, computed automatically.

External memory algorithms [153, 156] for reachability graph analysis basically store states on disk sorted according to some ordering of the states. Storing states and checking whether states are already stored are batched, minimising the number of disk accesses required. In addition, an in-memory cache is used to further minimise the number of disk accesses required. While such algorithms are interesting, computers today often have enough memory available, counting in gibi-bytes on laptop computers to hundreds of gibi-bytes on large servers, that by just representing states more efficiently we can analyse systems for which filling internal memory would take weeks or months, in particular when analysing CP-nets, where calculating enabled transitions can be very time-consuming.

Among methods which delete states from memory during exploration are the state caching [60] and sweep-line methods [25, 104]. State caching basically performs a depth-first traversal of the reachability graph. Rather than storing all states of the reachability graph, only the states on the depth-first stack are guaranteed to be stored. If enabled transitions can be processed in a deterministic way, this will terminate whenever the reachability graph is finite. It is possible that the successors of certain states are explored more than once, however. This happens for states with more than one transition leading to them, i.e., if $s' \xrightarrow{a} s$ and $s'' \xrightarrow{b} s$ for $s', s'' \in \mathsf{reach}(s_I)$ and $(s', a) \neq (s'', b)$. In that case s will be explored more than once. In order to minimise the number of re-explorations, some states are cached in memory, even if they are not on the depth-first stack. Several methods [19, 40, 56] aims at finding clever ways to decide which states should be kept in memory and which should be discarded. Another method for intelligently removing states from memory during analysis is the sweep-line method, which uses a specification to detect when a state will not be encountered again. We go into more detail about the sweep-line method in Sect. 2.2.1, as it is needed to understand the summary of the paper [T1] in Sect. 2.3.

Several algorithms for storing states more efficiently exist, some are dependent on the formalism used and some are independent of the formalism used. For CP-nets, we can use an approach similar to BDDs for storing states, namely storing states in a tree sharing common parts of the state [24]. The idea is to observe that CP-nets are split into places and that the tokens on one place can also be encountered on other places or on the same place in other states. Furthermore the effects of transitions on CP-nets are usually local, meaning that only a few places are modified when an enabled transition is executed. By storing identical marking of places (multi-sets of values) only once, we can thus obtain a reduction in the memory required to store the state of a CPN model. The representation of a state just refer to the correct marking of each place. This can be used with the network protocol to share the markings of Send ID and Receive ID as well as the empty markings of the two network places. Furthermore, CP-nets are extended with a simple module concept, and the locality of transitions means that often only markings of places in one or a few modules are changed. By furthermore representing the state of each module separately, it is possible to re-use the representation of all unchanged modules. This method is implemented in CPN Tools [C1, 33]. Bit-state hashing [76] is a formalism-independent approach to storing states efficiently. Bit-state hashing uses a hash function to compute a hash value for each state. This hash value is then used as index in a bit-array (modulo the size of the array) to set a bit indicating a state with that hash value has been encountered. If multiple states have same hash value this will lead to a *hash collision*, i.e., two different states are considered the same because they have the same hash value. To reduce this

problem one can use more than one hash function or a linear combination of two or more independent hash functions using double hashing [38]. Hash compaction [155, 172], like bit-state hashing, applies a hash-function to each state. Instead of using the hash value as index in an array, the hash value itself is stored. Hash-compaction will be discussed in further detail in Sect. 2.2.2, as the ComBack method described in the summary of the paper [T2] in Sect. 2.4 builds on hash compaction.

2.2.1 The Sweep-Line Method

The sweep-line method [25, 104] introduced by Christensen, Kristensen, and Mailund is an example of a method that deletes states during the analysis. The idea is to introduce a *progress measure* assigning to each state a *progress value*, $\psi : S \rightarrow \mathrm{N}$. In fact, the progress measure can assign progress values from any partially ordered set, but for simplicity we will here assume that we use integers as progress values. In the basic sweep-line method from [25] the idea is to require that if $s \rightarrow s'$ then $\psi(s) \leq \psi(s')$. The progress measure is thus a syntactical way to recognise whether a state s' is reachable from s (if $\psi(s') < \psi(s)$ it is not). In the network protocol example from Fig. 1.8, we can let the progress value of each state be the sum of the Send ID and Receive ID counters. In Fig. 2.1 we have written the progress value of each state as a large number to the upper left of each state. Each state is represented by the value of the token on the Send ID place, the sequence numbers of the packets in Network 1, the number of tokens available on Limit, the sequence numbers of the packets on In Buffer, the value of the token on Receive ID, and the sequence numbers of the packets on Network 2. The initial state is marked by a red background. The progress value of each state is thus the sum of the numbers next to Send ID and Receive ID. Transitions are represented by an abbreviated version of their name and the sequence number of the packet being processed. We note that in this case $s \rightarrow s'$ implies $\psi(s) \leq \psi(s')$ for all reachable states. We explore the reachability graph by always picking states with the lowest progress values first. This means that we can safely delete states with lower progress values because of the contraposition of the requirement for a progress measure, namely that if $\psi(s) < \psi(s')$ then $\neg s \rightarrow s'$, which can be extended to that if $\psi(s) < \psi(s')$ then $\neg s \rightarrow^* s'$. Conceptually, the progress measure defines a sweep-line, so that states behind the sweep-line have all been processed and we know that none of the currently unexplored states will have transitions leading to states behind the sweep-line, so they can safely be removed from memory. In Fig. 2.1 the thick arrow below the states shows the direction we explore the reachability graph. If we draw a vertical line, like the one between the states with progress values 3 and 4, we notice that at no point do transitions cross the sweep-line from right to left (except that states with progress measure 4 use 2 columns for easier display). The basic sweep-line algorithm is given in Algorithm 2. The changes from Algorithm 1 is that we in line 6 select one of the states with the smallest progress value rather than an arbitrary state, we remove states from V with lower progress measure than any state in W in line 13, and we remove any edges that are connected to states that have been removed in line 14. The algorithm can be implemented by representing W using a priority queue with ψ as the priority function. Garbage collection can either be done each time we select a state s in line 6 with a higher progress value than in the previous iteration or every, say, 1000^{th} iteration depending on how V is implemented. If we create a double representation of V using a hash table and a priority queue with φ as priority, we can perform garbage collection each time we increase the progress value without an unreasonable

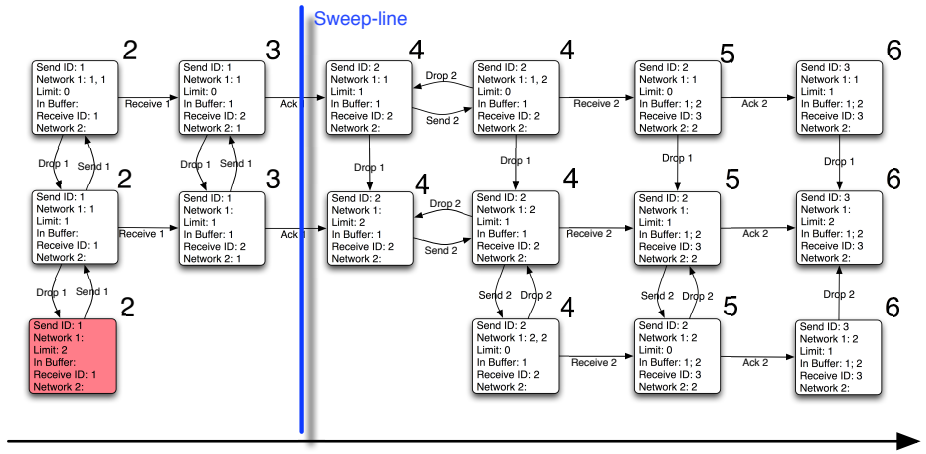


Figure 2.1: The reachability graph of the network protocol with progress values assigned to each state and a sweep-line drawn between states with progress values 3 and 4.

penalty in time. If V is just represented as a has table, how often we do garbage collection needs to be balanced between the cost of traversing all of V against the additional memory required to store additional states that can safely be garbage collected.

Algorithm 2 The basic sweep-line method for reachability graph traversal.

Require:
$\mathcal{LTS} = (S, T, \Delta, s_I)$ a labelled transition system,
$\psi : S \to \mathbf{N}$ a progress measure

$V := \{s_I\}$
$W := \{s_I\}$
$E := \emptyset$

while $W \neq \emptyset$ **do**
  Select an $s \in W$ s.t. $\forall s' \in W.\psi(s') \geq \psi(s)$
  $W := W \setminus \{s\}$
  for all t, s' such that $s \xrightarrow{t} s'$ **do**
    $E := E \cup \{(s, t, s')\}$
    if $s' \notin V$ **then**
      $V := V \cup \{s'\}$
      $W := W \cup \{s'\}$
  $V := \{s \in V \mid \exists s' \in W.\psi(s') \leq \psi(s)\}$
  $E := \{(s, t, s') \in E \mid s, s' \in V\}$

return (V, E)

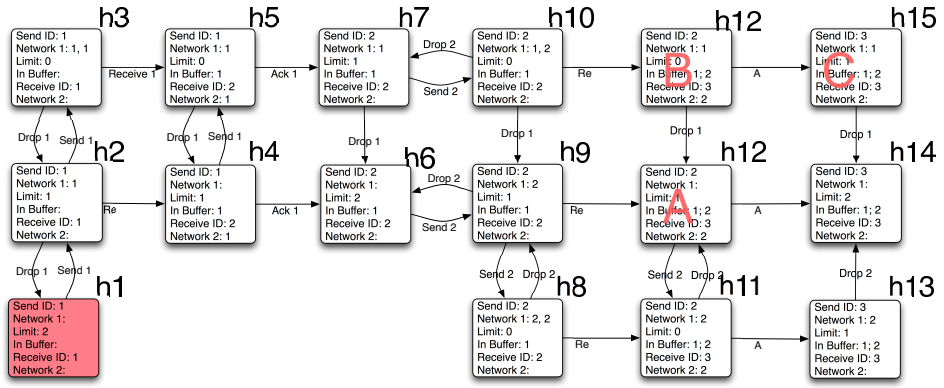
Unfortunately the property that $s \to s' \implies \psi(s) \leq \psi(s')$, i.e., that the progress measure is *monotone* does not hold for many interesting systems, such as reactive systems, unless we choose a trivial progress measure assigning the same progress value to all reachable states. The trivial progress measure does not yield any reduction in the number of states stored. To overcome this, the sweep-line method has been extended by Kristensen and Mailund in [104] to also handle systems where we may have $s \to s' \wedge \psi(s) > \psi(s')$. Edges satisfying

this property are called *regress edges*. By traversing the reachability graph multiple times, each traversal called a *sweep*, the sweep-line method is able to cope with regress edges. The idea is to start from the initial state in the first sweep and in all the following sweeps use the destinations of regress edges found in the previous sweep as starting points. Furthermore, we never remove destinations of regress edges from memory. The major advantage of the sweep-line method is that if the progress measure is good, i.e., if it separates the reachable states into many equivalence classes and yields few regress edges, only a fraction of the reachable states are kept in memory at any time. How to find good progress measures are the topic of much research. As an example Schmidt [151] use transition invariants of PT-nets to automatically synthesise efficient progress measures, and Vanit-Anunchai, Billington, and Gallash try to assist the user in manually obtaining good progress measures by counting the number of states in each class of states with the same progress value [165].

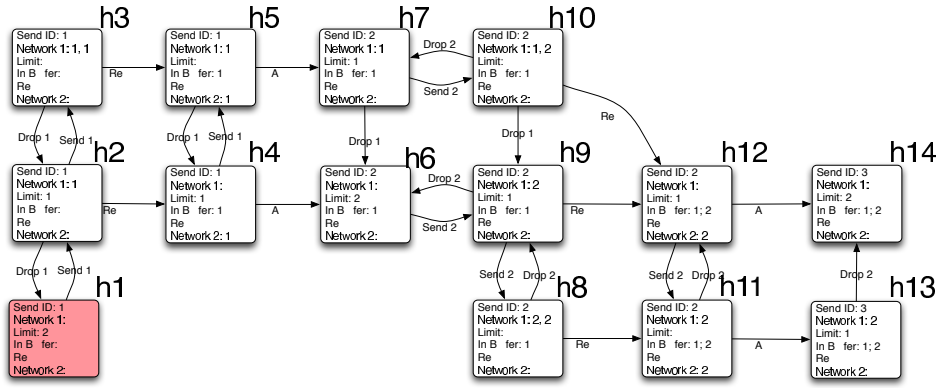
2.2.2 Hash Compaction

Hash compaction [155, 172] introduced by Wolper and Leroy uses a hash function, $H : S \to \{0,1\}^w$, to compress states to w bits before they are stored. As an example, in the network protocol in Fig. 1.8, to represent the state of the system, we would need 12 integers to store each state of the system (one for Send ID, Receive ID, and Limit, one to indicate how many packets and two to specify which packets are on either of Network 1, Network 2, and In Buffer), using 48 bytes assuming that 32 bits are used to represent each integer. By using a hash function generating 32 bit hash values, we would only use 32 bits or 4 bytes to store each state. The algorithm for reachability graph analysis using hash compaction is the same as the algorithm for basic reachability graph analysis, namely Algorithm 1. The only difference is that the checks in line 10 and the adding of nodes in line 11 are implemented in a different way—this actually holds for any algorithm which implements a more efficient state representation. For hash compaction we would check whether $H(s) \notin V$ (in line 10) and replace line 11 by $V := V \cup \{H(s')\}$.

The major caveat of hash compaction is that hash collisions may lead to not exploring all reachable states, as we may incorrectly conclude that a state s' has already been visited if we have visited a state s, whose compressed state descriptor $H(s)$ is equal to the compressed state descriptor of s', $H(s')$. Say we have a hash function assigning hash values $h1$–$h15$ to the states of the network protocol. In Fig. 2.2(a) we have written the hash values assigned to each state to the upper left of the states. If we assume that the state marked with a big A inside is discovered before the state marked B, we will believe we have already seen state B, and not process it further, so the state C will never be discovered. In fact, the reachability graph as explored using hash compaction will look like the one in Fig. 2.2(b). If we consider it an error to have received and acknowledged all packets successfully, yet still have an outstanding copy of the first packet, analysis using hash compaction would (in this case) not discover the error as state C is not explored. The hash compaction method can be improved by using more than one hash function, but the basic problem persists, namely that the method is incomplete in general. In [T2] we introduce the ComBack method, which improves hash compaction by adding a means to discover hash collisions on-the-fly during the traversal, making the method complete as well as sound.



(a)



(b)

Figure 2.2: Reachability graphs for the network protocol as seen when using hash compaction.

2.3 Obtaining Memory-Efficient Reachability Graph Representations Using the Sweep-Line Method [T1]

The major disadvantage of the sweep-line method is that at no time during exploration do we have a complete representation of the entire reachability graph in memory (unless we use the trivial progress measure assigning the same progress value to all reachable states, in which case the method yields no optimisation), so it is only possible to decide invariant properties. If we want to check more complex properties, such as liveness properties using Linear Temporal Logic (LTL), we will need a representation of the reachability graph in memory or the ability to perform depth-first traversal of the reachability graph as LTL can be checked by calculating strongly connected components of the reachability graph using, e.g., Tarjan's algorithm [157], or on-the-fly using nested depth-first traversal [74] of the reachability graph as described by Holzmann. Neither of these methods are immediately possible in combination with the sweep-line method. Tarjan's algorithm cannot be used as it requires that we have a representation of the reachability graph in memory (or that we are able to generate the graph in a depth-first manner), and nested depth-first

traversal of the graph is not usable as the sweep-line imposes a certain order of traversal depending on the reachability graph in order to perform well. If we use the basic sweep-line method it is possible to check LTL as all states in a strongly connected component will need to have the same progress value. The basic sweep-line method yields no optimisation for reactive systems, however, and it may often be possible to devise a better progress measure if we allow a few regress edges. Our paper [T1] uses the sweep-line method to construct a near memory-optimal representation of the reachability graph, so we can use either Tarjan's algorithm or nested depth-first traversal to check, e.g., liveness properties.

The most efficient representation of $|S|$ states use $\lceil \log_2 |S| \rceil$ bits to store each state[2]. Often the encoding actually used is not even this efficient, so even more than the required $\lceil \log_2 |S| \rceil$ bits are used to store each state. In the network protocol example, we would use 48 bytes (to represent 12 integers) or 384 bits to store each state. Only $\lceil \log_2 |\mathsf{reach}(s_I)| \rceil$ bits are actually needed to distinguish between the $|\mathsf{reach}(s_I)|$ reachable states, however. The idea of [T1] is that the number of reachable states is often much smaller than the number of syntactically possible states, $|\mathsf{reach}(s_I)| << |S|$, so we map representations of states from S (the full state descriptors) into bit-vectors of size $\lceil \log_2 |\mathsf{reach}(s_I)| \rceil$ (the condensed representation) in a way so that we can later analyse the reachability graph. In the network protocol we only need to use $\lceil \log_2 16 \rceil = 4$ bits for each state, using only around 1% of the memory used for our naive representation of each state. This representation is realised by representing each reachable state as a number $0, \ldots, |\mathsf{reach}(s_I)| - 1$, and using a standard successor-list representation of the reachability graph. Such numbers have no relation to the full state descriptor, so we need to keep the full state descriptors as long as needed to recognise previously seen states. In Fig. 2.3(a) we see the reachability graph of the network protocol in Fig. 1.8. We have assigned to all states a state number, written to the upper right of the state. A successor-list representation of the reachability graph can be seen in Fig. 2.3(b). For each node we store a pointer to a list of all successors. The list is preceded by the number of successors, and contains a list of pairs with the transition and the number of the state it leads to. As an example, we can see that the state with number 1 has 3 successors. One successor is reached by executing Drop 1 and leads to state number 0, and the other successors are reached by executing Send 1 leading to state number 2 respectively executing Receive 1 to state number 3.

If we assume that the transition relation is *deterministic*, i.e., if $s \xrightarrow{t} s'$ and $s \xrightarrow{t} s''$ then $s' = s''$, this structure can be traversed using Algorithm 3. The idea is to traverse the graph according to the condensed representation (the numbers), and calculate the full state descriptors during traversal using the transition information. We use the fact that the transition relation is deterministic to calculate the successors in line 10 of the algorithm. It is easy to change Algorithm 3 to check Computation Tree Logic (CTL) as in [27, Sect. 4.1] by adding a table of sub-expressions of the formula to check, indexed by $0, \ldots, \lceil \log_2 |\mathsf{reach}(s_I)| \rceil - 1$ so it is possible to calculate a fix-point of satisfied formulae in each state. We can also extend the algorithm to use nested depth-first search [74], so it can be adapted to check Linear Temporal Logic (LTL).

One problem is, of course, to recognise when a full state descriptor is no longer needed, i.e., when we will never encounter it again. We use the sweep-line to delete full state descriptors from memory when they are no longer needed. Another problem is that when we start the generation we do not know

[2]Assuming that $|S| < \infty$; if $|S| = \infty$ we would use a variable-length encoding.

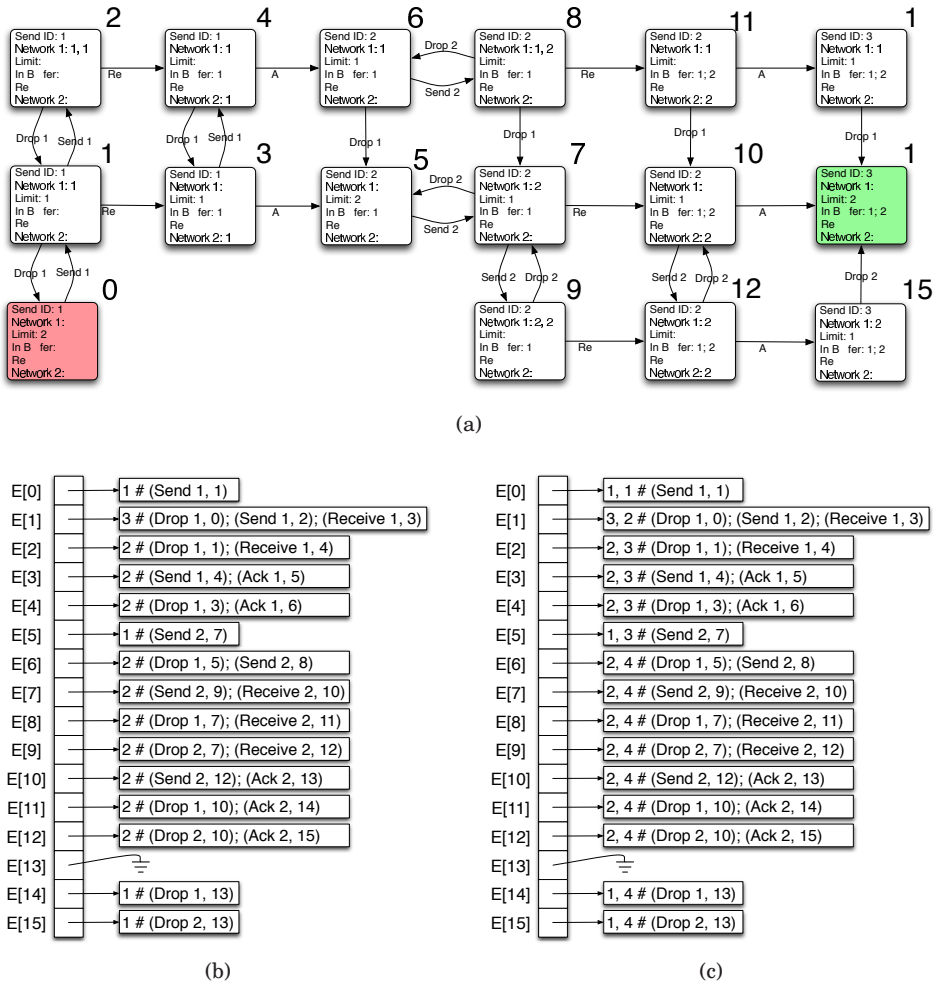


Figure 2.3: The reachability graph of the network protocol with progress values assigned (a) and two successor-list representations of the graph (b) and (c).

$|\mathsf{reach}(s_I)|$, so we do not know how many bits to use for each state. To circumvent this problem, we simply use as many bits as required to store all successors (which is known at the time we store successors) and store this length as well.

The algorithm works by defining a function $\mathsf{idx}_M : S \to \{0, \ldots, |\mathsf{reach}(s_I)| - 1\}$, mapping full state descriptors to state numbers. This function can, e.g., be implemented as a hash table. Whenever we encounter a new state we assign it a new state number and add it to idx_M. Consider, e.g., Fig. 2.3(a). If we are exploring state 1 and the state 0 is already in idx_M, we will encounter two new states, which we assign numbers 2 and 3. As we assign numbers to states when they are first discovered, we will always know the maximum state number of all successors when processing a state. In the case of state 1, this number is 3, so by using $\lceil \log_2 3 \rceil = 2$ bits, we can store all successors of state 1. The structure in Fig. 2.3(c) shows how we can represent the reachability graph by extending the header to also include how many bits are used to store each successor state. As an example we see that state number 1 has 3 successors, each represented using 2 bits. One of the states is reached by the transition Drop 0 and has number 0. When we have processed state 2 and move on to state 3, we notice

Algorithm 3 Depth-first traversal of the condensed reachability graph

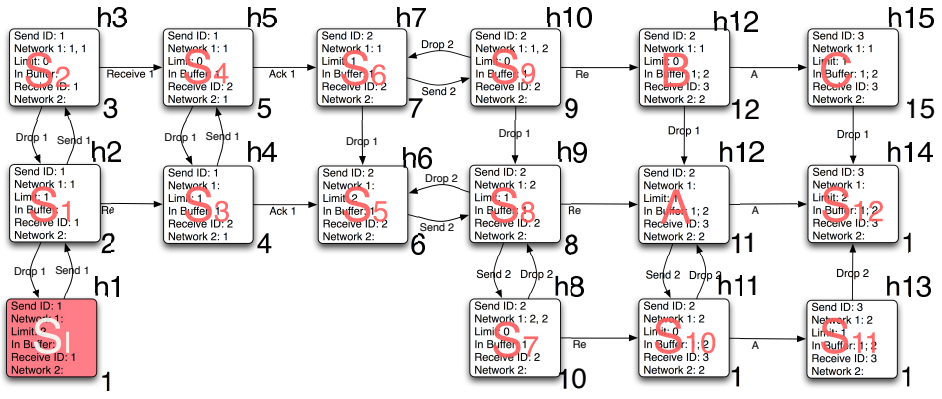
Require: E a successor-list representation of a reachability graph

```
 1: V := ∅
 2: DepthFirstTraversal(0, s_I)
 3:
 4: proc DepthFirstTraversal(i, s) is
 5:   if i ∈ V then
 6:     return
 7:   {analyse s here}
 8:   V := V ∪ {i}
 9:   for all (t, i') in E[i] do
10:     Let s' be such that s -t-> s'
11:     DepthFirstTraversal(i', s')
```

that there is no need to store the idx_M mapping for states 0 to 2 as we will never encounter them again. We know that because we can look at the reachability graph in Fig. 2.3(a), but using the sweep-line method, the algorithm is also able to realise that, as state 3 has progress value 4 (using the same progress measure as in Fig. 2.1, namely the sum of Send ID and Receive ID) and the states 0 to 2 have progress value 3. We therefore remove the mapping of the full state descriptors of states 0–2 from idx_M and proceed calculating successors of state 3 and assigning them state numbers. If the progress measure is not monotone, it is possible that we delete a full state descriptor from idx_M before we are done using it. This will lead to the state being assigned a new number, so the reachability graph constructed using this algorithm may actually be an *unfolding* of the original reachability graph. The unfolding is shown to be bisimilar [124] to the original reachability graph by Mailund in [118, Chap. 13], so CTL* and in particular LTL and CTL is preserved as shown by Clarke, Grumberg, and Peled in [27, Chap. 12]. The full algorithm can be seen in Fig. 5.3 on page 86.

2.4 The ComBack Method—Extending Hash Compaction with Backtracking [T2]

As can be seen in Fig. 2.2, the hash compaction reduction method may lead to not exploring all reachable states if there exist two states, $s \neq s'$, with the same compressed state descriptor, $H(s) = H(s')$, as is the case with states A and B in Fig. 2.2(a). The ComBack method circumvents this by storing enough information that we are able to realise that the states s and s' are actually different even though $H(s) = H(s')$. Like the method in the previously discussed paper [T1], we will represent each state as integers, in this case $1, \ldots, |\mathsf{reach}(s_I)|$. Furthermore, we use a hash function H to generate compressed state descriptors for each state. In Fig. 2.4(a) we have shown the reachability graph of the network protocol from Fig. 1.8 and assigned each state a number $1, \ldots, |\mathsf{reach}(s_I)|$ and a compressed state descriptor $h1, \ldots, h15$. Additionally, we have assigned each state a name, s1—s11, A, B, and C, to have a brief way of referring to the full state descriptor of each state. As an example, we see that the states A and B have the same compressed state descriptor, $h12$. For the sake of the description of the method, we will assume that the transition relation given is deterministic, i.e., that if $s \xrightarrow{t} s'$ and $s \xrightarrow{t} s''$ then $s' = s''$.



(a)

h1	1
h2	2
h3	3
h4	4
h5	5
h6	6
h7	7
h8	10
h9	8
h10	9
h11	13
h12	11, 12
h13	16
h14	14
h15	15

(b) State table

1	
2	1, Send 1
3	2, Send 1
4	2, Receive 1
5	3, Receive 1
6	4, Ack 1
7	5, Ack 1
8	6, Send 2
9	7, Send 2
10	8, Send 2
11	8, Receive 2
12	9, Receive 2
13	10, Receive 2
14	11, Ack 2
15	12, Ack 2
16	13, Ack 2

(c) Backedge table

Figure 2.4: The reachability graph of the network protocol with compressed state descriptors and state numbers (a), the state table (b) and the backedge table (c) used to represent the reachability graph using the ComBack method.

The method can be extended to also deal with non-deterministic transition relations; for more details refer to Sect. 6.5 on page 103.

The ComBack method works by storing a mapping from compressed state descriptors to state numbers, called a *state table*, which maps a compressed state descriptor into all encountered state numbers with the corresponding compressed state descriptor. The state mapping for the reachability graph of the network protocol can be seen in Fig. 2.4(b). Furthermore we maintain a spanning tree from the initial state to all explored states by storing, for each state number n' corresponding to a state s', the number, n of a predecessor state s and a transition t such that $s \xrightarrow{t} s'$. This information is stored in a data-structure called the *backedge table*. A possible backedge table for the reachability graph of the simple network protocol can be seen in Fig. 2.4(c). Here we see, e.g., that state number 2 can be reached from state number 1 via the transition Send 1.

Using the backedge table it is possible to reconstruct the full state descriptor for each state number. Say we want to reconstruct the full state descriptor for the state with number 11. We look up state number 11 in the backedge

table and obtain 8, Receive 2, meaning we must take a Receive 2 transition from the state with number 8 to reach the state with number 11. We now look up state number 8, and get 6, Send 2. We continue and obtain (4, Ack 1), (2, Receive 1), (1, Send 1). The state with number 1 is the initial state. We can see this as it has no backedges. Now we must execute the transition sequence we have obtained from the initial state in the reverse order, and get $s_I \xrightarrow{Send\ 1} s_1 \xrightarrow{Receive\ 1} s_3 \xrightarrow{Ack\ 1} s_5 \xrightarrow{Send\ 2} s_8 \xrightarrow{Receive\ 2} A$, which is indeed the state with state number 11. We can thus backtrack from any given state number to the initial state and execute all the transitions stored in the backedge table to obtain the full state descriptor corresponding to the state number.

The state table and the backedge table are created as we explore the reachability graph. Whenever we encounter a state, s', using the transition $s \xrightarrow{t} s'$, we calculate its compressed state descriptor, $H(s')$, and look up all state numbers corresponding to that compressed state descriptor in the state table. If no such state numbers exist, we just assign the state a new number and add it to the state table and the backedge table. If there are any such state numbers in the state table, we use the aforementioned technique to re-generate the full state descriptors for each of the numbers. These can then be compared with the full state descriptor of s'. If any of the full state descriptors are equal to s', we have already encountered s', and do not need to proceed. Otherwise we just assign s' a new state number and add it to the state table and the backedge table. All we need to know to add a state to the state table and the backedge table is the number of a predecessor and a transition from the predecessor as well as the highest used state number. The entire algorithm can be seen in Algorithm 5 on page 100.

In addition to the basic algorithm, a number of variants are given in [T2]. One variant is able to also handle non-deterministic transition relations. This variation also makes it possible to only store the number of a predecessor state in the backedge table and omitting the transition information, yielding a memory optimisation at a cost in time. Time-saving variants include shortening of how long we need to backtrack for each state. The lengths of backtracks depend on the traversal policy—in the example we have used a breadth-first traversal yielding optimal backtracks, but if we had used, e.g., depth-first traversal, the backtracks might not be optimal, so shortening backtracks would reduce the time required to reconstruct full state descriptors. Another time-saving variant simply caches some full state descriptors, thus spending a little more memory for increased performance.

2.5 Contribution and Future Work

In this chapter we have taken a look at behavioural verification of formal models including symbolic methods and four categories of explicit reachability graph analysis. We have in particular looked at the existing sweep-line and hash compaction reduction techniques and how our papers have improved upon these methods. In this section, we sum up the contributions made in this field and provide directions for future research.

Our first presented paper, [T1], improves upon the sweep-line reduction technique by using the sweep-line method to construct a near-optimal representation of the reachability graph. This representation makes it possible to use the sweep-line method to check properties that are more complex than invariant properties, e.g., liveness properties in LTL. The algorithm is evaluated on a number of examples in [T1]. Unsurprisingly, the algorithm works

best when the sweep-line method does, i.e., on reachability graphs with a clear notion of progress. One such example is an extended version of the network protocol in Fig. 1.8, where 5–10% of the memory required to construct the full reachability graph is used, and 25–130% of the time is spent. If we consider a model of the dining philosophers problem, it is possible to define a progress measure which separates the reachable states into a lot of classes and only yields few regress edges. Using the number of eating philosophers as progress value, we will store almost all states during the construction, so the memory used for the compact representation is overhead. Compared to the amount of memory used for the full state descriptors, this is negligible, however, and the only real disadvantage is the extra time is spent during construction.

Our second presented paper, [T2], makes the hash compaction reduction technique complete by storing, in addition to a compressed state descriptor, a spanning tree rooted in the initial state. This extra information makes it is possible to resolve hash collisions on-the-fly by reconstructing the full state descriptor when we encounter states with a compressed state descriptor already stored. The paper compares the ComBack algorithm to other algorithms that minimise the amount of memory used to store states, and find that the ComBack method naturally uses more memory than hash compaction, but on the other hand guarantees that all reachable states are explored. The method uses less memory than storing the full state descriptors, and uses around twice as long time. On the other hand, thanks to the lower memory consumption, the method is able to explore reachability graphs that are impossible to explore using the basic algorithm.

In [49] Evangelista and Pradat-Peyre introduce an approach similar to the ComBack method. The method also stores states as pairs of a predecessor and a transition but, compared to the ComBack method, the compressed state descriptor is not stored, and state numbers are merely inserted into a hash table, which can lead to many more reconstructions. Furthermore, [49] has stratified caching [56] built into the algorithm, which makes it less flexible unless we first factor out the caching mechanism, as is done by the ComBack method. In [49] stratified caching with a maximum parameter of 50 is used. This means that a backtrack has length at most 50 and corresponds to caching 2% of the full state descriptors, makes the algorithm use $200\% - 400\%$ of the time used for a basic exploration of the reachability graph. The ComBack method use a simpler caching strategy, namely inserting the mapping from compressed state descriptors to full descriptors into a hash table that does not handle collisions. Using this caching strategy we are able to obtain comparable time results using a cache of only 0.1%–1% of all the states, or approximately 20 times smaller than the cache used in [49]. Furthermore, [49] presents the algorithm solely as a depth-first traversal, whereas the ComBack method is presented in a traversal-independent manner, making the ComBack method easier to combine with other methods. Also, our experimental results show that breadth-first traversal of the reachability graph may be much faster for highly reactive systems where most of the reachability graph end up on the recursion stack.

2.5.1 Future Work

While prototype implementations of the two methods described in this chapter have shown promising performance, neither method has been used extensively in practise. The reason is that one of the methods, the ComBack method [T2], has only recently been published at the time of writing. The other method, the extended version of the sweep-line method [T1], is mainly useful for checking more complex properties, such as liveness using Linear Temporal Logic, and

this does not have easy accessible tool support in tools supporting the algorithm, making real-life applications difficult. In this section we will provide some directions for interesting future work, including some ideas on how to alleviate these problems.

Use the ComBack method in conjunction with other reduction techniques

As mentioned, a strong point of the ComBack method is that it is independent of the traversal type, making it easily adaptable to tasks such as on-the-fly verification of LTL using nested depth-first traversal or CTL model-checking using backwards fix-point calculation. This makes the algorithm well-suited for analysis, and it also makes it easy to combine the algorithm with other reduction techniques, which may impose a certain traversal order. Here it is in particular interesting to combine the method with partial order reduction techniques, which reduce the in-degree of nodes (as high in-degrees often occur when executing concurrent transitions), thereby reducing the number of reconstructions required.

It is also interesting to combine the ComBack method with the sweep-line method. As stated earlier, the sweep-line method and the method for obtaining an efficient reachability graph representation described in Sect. 2.3 work well for reachability graph with a clear notion of progress. We can call such reachability graphs "long", because they often consists of a few long execution traces only. The ComBack method described in Sect. 2.4 works well for "wide" reachability graphs where the graph consists of many short execution traces with little interaction. The sweep-line based method only conserves memory if the progress measure makes it possible to often remove full state descriptors, but the method uses extra time whenever a regress edge leads to an already discovered state because of rediscovery. The ComBack method uses long time reconstructing already visited states, but benefits greatly from a cache mapping compressed state descriptors to full state descriptors. We have only experimented with very simple caching strategies, and our own research as well as that of Evangelista and Pradat-Peyre [49] indicate that the method is very sensitive to caching strategy in terms of how much time is spent. One way to obtain a caching strategy that intuitively should perform well is to use the sweep-line method to define the caching strategy, and cache full state descriptors in front of the sweep-line. Another way to view this combination is that we utilise the ComBack method to test whether destinations of regress edges lead to new or to already discovered states during a run of the sweep-line method. We observe that the successor-list representation of the reachability graph created in our paper [T1] looks very similar to the backedge table of the ComBack method (compare Fig. 2.3(c) with Fig. 2.4(c)). The successor-list representation stores successors and the backedge table stores a predecessor for each state. Both of the tables rely on state numbers. If we extend the successor-list of the sweep-line based method with a predecessor like in the backedge table and introduce a state table like in the ComBack method, we are able to cope with regress edges by, rather than just concluding that they are regress edges and processing them in the next sweep, checking whether we have already encountered the state (by checking the state table and reconstructing states as necessary), and, if the state is new, either schedule it for later processing or process it immediately. As we use the sweep-line method to represent all full state descriptors in front of the sweep-line, we only need to reconstruct destinations of regress edges during the first sweep. As we are able to check whether the destination of a regress edge leads back to a previously unvisited state or

a completely new state, we never need to reconstruct parts of the reachability graph, which was the major caveat of the sweep-line based algorithm. The combined method should therefore be able to analyse systems which only exhibit limited progress, as regress edges no longer lead to a blowup in spent time.

Devise more usable specification language for properties of coloured Petri nets

As mentioned, the new method described in Sect. 2.3 has not been tested extensively due to the lack of reasonable tool support. The current tool, CPN Tools, provides provisional support for checking CTL and support for checking LTL on-the-fly has been experimentally implemented for coloured Petri nets in DESIGN/CPN [37] by Mikkelsen [122] and by the author of this thesis in a model-checker implemented in the BRITNeY Suite. All of these implementations use a textual syntax for describing the temporal formulae and use Standard ML predicates applied to a representation of the state of the model as atomic propositions. This has the disadvantage of requiring that the user is familiar with the complexities of temporal logics and the difficulty of writing often complex predicate functions.

Rather than inventing a new language for specifying properties, one can also just use the modelling formalism itself to specify properties. This has been done in SPIN [77], where properties are stated as so-called never-claims, which is a standard process in the native PROMELA language of SPIN. If the never-claim reaches a final state, it is considered an error. Something similar has been proposed by Petri [139] for Petri nets. Here we add special transitions, called facts, which must never be enabled. We could do something similar for CP-nets by introducing a module containing a place which must never be marked and/or a transition which must never be executed. CP-nets currently have tool support for synchronisation of modules by means of shared places, but support can dually be added for synchronisation by means of shared transitions. Using this it would be possible to create a module representing a scenario which must never happen.

It is also possible to try to define atomic propositions in a simpler way. This idea is partially inspired by Cardelli and Gordon's ambient logic [16] where atomic propositions are stated in a language closely resembling the language of the ambient calculus [17]. For coloured Petri nets something similar could, e.g., be achieved by showing the user a copy of the net/a module of the net. Tokens can then be assigned to places of interest to signify that these tokens must be present on the place in a state for the atomic proposition to hold. For example, in the case of the network protocol in Fig. 1.8, we may want to check if the token [(2, " model"), (1, "Formal")] can ever be present on In Buffer, signifying that the packets have arrived out of order.

Finally, we may want to specify temporal properties in simpler ways than by using a logic. We can, e.g., specify temporal properties using message sequence charts [67] as message sequence charts basically define a partial ordering of events. By annotating message sequence charts with atomic propositions, which must hold between events, users would be able to easily specify even complex temporal properties.

Create test-suite and tools for improvement of reachability graph analysis methods

As can be seen in Table 6.1 on page 107, re-printed from [T2], a lot of experiments have been run in order to validate the usefulness of the ComBack algo-

rithm. The results shown only comprise a small fraction of the total number of experiments, and it is not desirable to have to run these experiments manually.

In [133], Pelánek state that reachability graphs basically come in three variants: random graphs, reachability graphs generated by small academic examples, and reachability graphs generated by realistic/real-life models. Graph-theoretic properties varies for each kind of graph. Often the performance of reduction techniques varies hugely dependent on the structural properties of the reachability graph. For example, the ComBack method works best if the in-degree is small so only few states are reconstructed. This demonstrates that it is important to test methods on several different kinds of models, preferably both academic examples, like the dining philosophers, as well as real-life models.

Furthermore, we would also like to be able to compare results of tests with known good results and compare the execution time and memory consumption as time progresses and the implementation is improved. Both to compare different implementations of the same reduction technique and to compare different reduction techniques.

This implies that we would like a test-suite and supporting tools, which provide a means to automatically run several reachability graph analysis tasks and which preferably provide a means to easily specify such tasks. The test-suite should consist of several formal models, both simple academic examples and real-life models. It should be possible to store results of executions, compare each result to known-good values, and enable exploration of the execution time/memory consumption for different reduction techniques and different implementations.

One such test-suite is being developed within the ASCoVeCo (Advanced State Space Methods and Computer tools for Verification of Communication Protocols) project [3] at the University of Aarhus. The author of this thesis participates in this project and has contributed to the development of the test-suite and tools for running tests.

Improve memory handling

As can be seen in Table 6.1 on page 107, while the ComBack method conserves memory, it uses significantly more than the expected limit of 5 words (20 bytes when a machine word is 32 bits) per state (obtained from Theorem 6.1 on page 103 by using a machine word for each of the mentioned numbers). The theorem does not account for memory used for the state table and backedge table, which will in fact use 2 extra machine words for each state (if implemented as an dynamically extensible array). The state table will also use 2 extra machine words for each state. This yields a total of 9 machine words or 36 bytes for each state, assuming a 32 bit architecture. Yet the best result obtained in Table 6.1 is that 82 bytes is used per state, or more than twice the expected amount of memory required. This is primarily due to the fact that the algorithm is implemented in Standard ML. While Standard ML is a very nice language for specifying algorithms, it is not well-tailored to fine-grained control of memory use. In particular it usually stores a pointer in addition to the data we are interested in, doubling memory used when we primarily store machine words. By implementing the data-structures in C++ and keeping the algorithm implementation in Standard ML, it would be possible to maintain a nice declarative way to describe algorithms, while using a low-level language to use fine-grained control of memory consumption.

Implementing data-structures in a low-level language would also make it possible to implement the condensed representation of [T1] more efficiently.

The current implementation uses a machine word to store each state number, even though the algorithm facilitates using only the number of bits required. A similar trick could be done with the ComBack method from [T2] by, e.g., observing that we know that the number of bits required to store all predecessors of state number 2^n is only n as the predecessor will have a lower number (unless we make path optimisations). This allows us to drastically reduce the amount of memory needed to store the backedge table. We can also use Geldenhuys and Valmari's very tight hashing [57] to represent the state table of the ComBack algorithm more efficiently. All of these optimisations are not feasible when the data structures are implemented in Standard ML, as it is very expensive to pack and unpack data in order to store data that is not word aligned in Standard ML.

Visualisation of error traces for property violations

The BRITNeY Suite [T3, C2], which is described in the next chapter, supports visualisation of traces to violations of invariant properties and liveness properties formulated using LTL by means of simple message sequence charts. Such violations are quite easy to visualise, as a violation of an invariant property can be proved by providing a trace from the initial state to a state not satisfying the invariant. Violations of LTL properties are rather simple to visualise as well, as they can be proved by providing a trace to a loop not satisfying the property. The latter can be visualised by two message sequence charts, one showing the trace to the loop, and one showing the loop.

Violations of properties formulated using CTL are more complex, however. The reason is that a proof of a violation is an annotated version of the reachability graph. Such a proof can of course visualised as a huge graphical graph, but as soon as the graph contains more than a few dozen nodes, this becomes impractical. Instead, we propose another way to convince users that properties hold/do not hold. The idea is that if the users need convincing that the formula does not hold, it is because he thinks it does hold. In Sect. 3.5 we provide more details of how this could be done by letting a user try to prove sub-formulae of the system by choosing some transitions and letting the computer choose other transitions in a way such that it is impossible for the user to ever arrive at a proof of the property.