CTL Model Checking with the Sweep-line State Space Exploration Method

Andreas Lilleskare, Lars M. Kristensen, Sven-Olai Høyland
Department of Computing, Mathematics, and Physics
Western Norway University of Applied Sciences

Abstract

Model checking is a powerful approach to verification of distributed systems. The sweep-line method alleviates the inherent state explosion problem in model checking by exploiting progress in the system being verified. Verification with the sweep-line method has until now been restricted to verification of safety and linear-time properties. The contribution of this paper is a new model checking algorithm that enables verification of two common branching time properties. The basic idea is to combine the sweep-line method with on-the-fly computation and inspection of strongly connected components. We experimentally evaluate our algorithm on a communication protocol.

1 Introduction

Designing and implementing distributed systems correctly is a challenging task. Model checking [1] of software is a main approach to automated verification of system properties and have been implemented in both industrial-strength tools and in academic prototypes. The basic idea underlying model checking is to explore all reachable states of a system model in order to algoritmically verify whether the system has a formally specified property or not. Typical examples of properties include absence of deadlocks, that a request from the user is always eventually followed by a response from the system, and that it is always possible to reach a stable state. Properties in model checking are typically specified in computation tree logic (CTL) or in linear-time temporal logic (LTL).

The main disadvantage of model checking is that the size of the state space (number of reachable states) often grows exponentially in the number of processes of the system. This is known as the *state space explosion problem* and is caused by the computational complexity of the model checking problem. In practice, it is mostly the amount of space (memory) which is the limiting factor. A wide range of methods have been proposed [12] that exploits intrinsic properties of the system to reduce space consumption. One example is the sweep-line method [7] which exploits progress in a system. Examples of progress include phases that the system goes through, control-flow of the processes, and sequence numbers in protocols. Progress is used to conduct a progress-first exploration of the state

This paper was presented at the NIK-2017 conference; see http://www.nik.no/.

space and delete states from memory during the state space exploration such that peak memory usage is reduced.

There exists sweep-line model checking algorithms for verification of safety properties [8] and LTL properties [5]. The challenge of performing CTL model checking with the sweep-line method is that conventional algorithms for CTL model checking [3] propagates information backwards from a state to its predecessors. This is incompatible with the forward progress-first exploration of the sweep-line method. In the context of symbolic model checking using binary-decision diagrams (BDDs), forward CTL model checking algorithms have been developed [6]. However, the sweep-line method is not compatible with the use of BDDs. The reason is that deleting states from a BDD (as required by the sweep-line method) may cause the memory usage for storing the BDD to increase. This counteracts the idea of how the sweep-line method alleviates the state explosion problem.

As a first step towards CTL model checking with the sweep-line method, we consider in this paper two commonly used properties in CTL:

P-AGEF: the possibility of always being able to reach a state satisfying a state predicate ϕ. This property can be used to verify for instance that the system can always return to its initial state after having processed a request.

P-AGAF: that a state satisfying a state predicate ϕ will always eventually be reached. This property can be used to verify for instance that the system always produces a response to the user.

The idea underlying our new algorithm is to consider systems with monotonic progress and compute the strongly connected components (SCCs) of the state space on-the-fly during the sweep-line state space exploration. The property P-AGEF holds if each so-called terminal SCC contains a state satisfying ϕ, while P-AGAF holds if a state satisfying ϕ is present on every cycle contained in an SCC.

The following sections are organised as follows: Section 2 introduces the formal foundation of this paper, and Sect. 3 introduces the sweep-line method and our new algorithm. In Sect. 4 we formalise the algorithm and prove its correctness. Section 5 presents some first experimental results, and finally in Sect. 6 we sum up the conclusions and discuss future work. The reader is assumed to be familiar with directed graphs and graph traversal algorithms.

2 Kripke Structures and Computation Tree Logic

As is common in model checking research, we use Kripke structures as the underlying formalism for formulating our algorithm. This makes our presentation independent of any particular modelling language used to model the system under verification. A Kripke structure is essentially a state-transition graph with nodes describing the reachable states of the system, a transition relation describing state changes (edges), and a state labelling specifying which atomic state propositions that are true in each state. An atomic state proposition may for instance be $x > 0$ where x is some variable of the system. A path in the Kripke structure then corresponds to an execution of the system.

Definition 1 (Kripke Structure [4]) *Let AP be a set of atomic state propositions. A Kripke structure M is a four-tuple $M = (S, s_0, R, L)$ where S is a finite set of states, $s_0 \in S$ is an initial state, $R \subseteq S \times S$ is a transition relation which is left-total, and $L : S \mapsto 2^{AP}$ is a function labelling each state with the atomic propositions that are true in that state.*

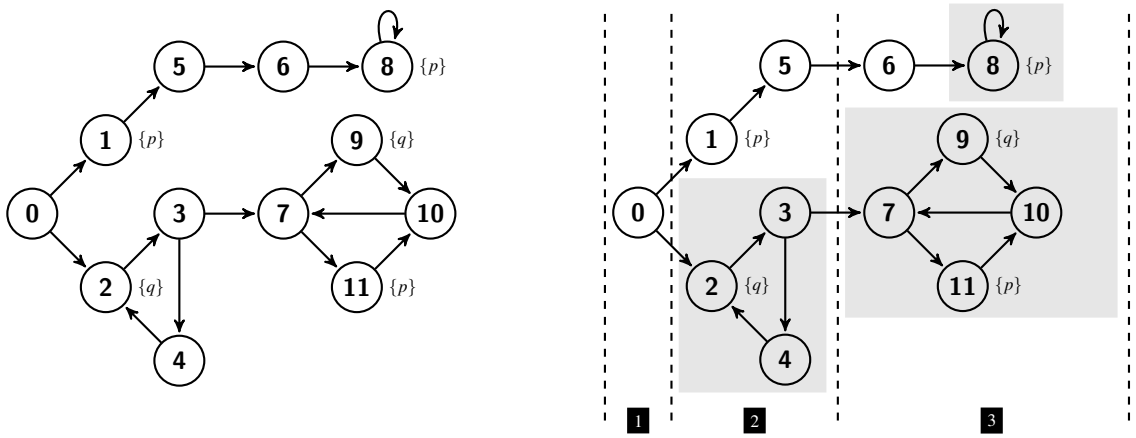

Figure 1: Example Kripke structure (left) and indication (right) of progress values (black squares) and non-trivial strongly connected components (grey boxes).

That the transition relation is left-total means that for every $s \in S$ there exists an $s' \in S$ such that $(s, s') \in R$. This ensures that we can define paths as infinite sequences of states in relation to the semantics of CTL.

Figure 1 (left) shows an example of a Kripke structure with two atomic propositions p and q that we use as a running example. Each state is identified by an integer written inside the state and next to each state we have written the atomic propositions that are true in that state. State 0 is the initial state. We explain Figure 1 (right) in the next section.

Temporal logics such as CTL are used to specify behavioural properties. In this paper, we do not consider the full CTL, but only formulas of the $AG\{EF, AF\}$-fragment that can be obtained from the following grammar, where $p \in AP$ and ϕ is called a *state predicate*:

$$\begin{aligned} \Phi &::= \mathbf{AG}\,\psi \\ \psi &::= \mathbf{EF}\,\phi \mid \mathbf{AF}\,\phi \\ \phi &::= p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \end{aligned}$$

The formulas expressing behavioural properties of the system under verification are interpreted over the paths of the Kripke structure. A *path* is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that for every $i \geq 0, (s_i, s_{i+1}) \in R$. The set of *reachable states* are those states that are present on some path starting in the initial state. The temporal operator **AG** states that ψ must hold in all reachable states, **EF** states that there must exists a future state in which ϕ holds, and **AF** states that eventually ϕ must hold. In particular, this means that $\mathbf{AG}\,\mathbf{EF}\,\phi$ holds if and only if from any reachable state it is always possible to reach a state in which ϕ holds. The property $\mathbf{AG}\,\mathbf{AF}\,\phi$ holds if and only if from any reachable state we eventually reach a state in which ϕ holds.

Consider the example in Figure 1 (left). The property $\mathbf{AG}\,\mathbf{EF}\,p$ holds since a state satisfying p can always be reached. Similarly, $\mathbf{AG}\,\mathbf{AF}\,(p \vee q)$ also holds since we always eventually reach a state in which either p or q holds. In contrast, the property $\mathbf{AG}\,\mathbf{EF}\,q$ does not hold since if we enter state 1, then we can no longer reach a state in which q holds. The property $\mathbf{AG}\,\mathbf{AF}\,p$ does not hold either since if the system starts in the initial state, then enters state 2, and keeps executing in the cycle comprised of states 2, 3 and 4, then we will never reach a state in which p holds.

For a path $\pi = s_0 s_1, s_2 \ldots$, we write $s \in \pi$ if s is one of the states on the path, i.e., $s = s_i$ for some $i \geq 0$. Based on this, we inductively define the semantics following [4] of the

CTL formulas considered in this paper, i.e., what it means for a formula f to hold in a state s of the Kripke structure M, which is written $M,s \models f$.

$M,s \models p \Leftrightarrow p \in L(s)$
$M,s \models \neg\phi \Leftrightarrow M,s \not\models \phi$
$M,s \models \phi_1 \wedge \phi_2 \Leftrightarrow M,s \models \phi_1 \wedge M,s \models \phi_2$
$M,s \models \phi_1 \vee \phi_2 \Leftrightarrow M,s \models \phi_1 \vee M,s \models \phi_2$
$M,s \models \mathbf{AG}\,\psi \Leftrightarrow$ for every path π starting from s and state $s_i \in \pi : M,s \models \psi$
$M,s \models \mathbf{EF}\,\phi \Leftrightarrow$ there exists a path π starting from s and a state $s_i \in \pi : M,s_i \models \phi$
$M,s \models \mathbf{AF}\,\phi \Leftrightarrow$ for every path π starting from s there exists a state $s_i \in \pi : M,s_i \models \phi$

The purpose of a model checking algorithm is given a formula f and a Kripke structure M to determine whether f holds in the initial state s_0, i.e., whether $M,s_0 \models f$.

3 Sweep-line CTL Model Checking

Conventional algorithms for explicit state model checking start from the initial state s_0 and then explores the complete state space (Kripke structure) using the transition relation R to compute for each encountered state, the set of successor states. The set of encountered states is stored in a set of *visited states* in order to ensure termination, and states for which successor states are still to be computed are stored as a set of *unprocessed states*. When processing a state, only successor states that have not already been visited are inserted into the set of unprocessed states. This means that the size of visited states grows as more of the state space is explored and eventually contain all reachable states of the system.

The idea of the sweep-line method is to exploit a notion of progress to delete states from the visited states when it is known that they are no longer needed for comparison with newly generated states. This means that the size of the set of visited states is reduced on-the-fly during state space exploration which in turn reduces memory usage.

Progress is formally quantified by means of a *progress measure* which maps each state into a progress value according to the following definition.

Definition 2 (Monotonic Progress Measure) *A monotonic progress measure on a Kripke structure $M = (S,s_0,R,L)$ is a tuple $\mathcal{P} = (O,\sqsubseteq,\psi)$ such that O is a set of progress values, $\sqsubseteq$ is a total order on O, and $\psi : S \to O$ is a progress mapping such that $(s,s') \in R \Rightarrow \psi(s) \sqsubseteq \psi(s')$.*

The progress mapping divides the state space into layers where a layer contains all states mapped to a given progress value. As an example, consider Fig. 1(right) where we have used dashed vertical lines to indicate the layers. Furthermore, we have used a black rectangle with a number indicating the progress value of the states in a layer. Layer 1 contains the initial state, layer 2 contains states 1-5, and layer 2 contains states 6-11.

The basic sweep-line algorithm starts with the layer containing the initial state and then explores the state space one layer at a time. When all states in a given layer have been processed, the states in the current layer are deleted since they are not needed for comparison with newly encountered states, i.e., determining whether a newly generated successor state is already contained in the set of visited states. The reason for this is that due to the monotonicity of the progress measure, any newly generated state cannot have

a successor in a previous layer. This means that with a monotonic progress measure, the system always makes progress. For the state space in Fig. 1(right), the sweep-line method would start from state 0 and then compute the successor states 1 and 2. Since there are then no longer any unprocessed states in layer 1, state 0 will be deleted. The algorithm now proceeds into layer 2 starting with the processing of states 1 and 2. This will cause states 3-7 to be encountered at which stage there are no longer any unprocessed states in layer 2. The states in layer 2 can then be deleted and the algorithm proceeds with the states in layer 3. Altogether this means that at most 7 states (the five states in layer 2, and state 6 and 7 in layer 3) at a time are stored in the set of visited states. This is in contrast to storing all 12 states in the set of visited states. It should be noted that a generalised variant of the sweep-line exists [8] that can handle non-monotonic progress measures, but in this paper we consider only monotonic progress measures.

Our new algorithm to model check the P-AGEF and P-AGAF properties with the sweep-line method exploits *strongly connected components (SCCs)*. An SCC of a directed graph is a maximal subset of nodes that are mutually reachable. By viewing the Kripke-structure as a directed graph we can compute the SCCs for the Kripke structure. In Fig. 1(right) we have indicated the SCCs consisting of more than just a single node (state) using a grey box. These are called *non-trivial SCCs*. In addition to these, each of the nodes 0, 1, 5, and 6 belongs to an SCC containing only the node itself. We use the notation SCC$\{x_1, x_2, \ldots, x_n\}$ to denote the SCC consisting of nodes $x_1, x_2, \ldots, x_n$. A *terminal SCC* is an SCC where no node has outgoing arcs to nodes in another SCC. As an example, SCC$\{8\}$ and SCC$\{7, 9, 10, 11\}$ in Fig. 1(right) are terminal SCCs.

Because of the monotonicity of the progress measure, an SCC can only contain nodes belonging to the same layer and hence each SCC is always contained in a single layer. In particular this means that we can compute the SCCs for a given layer immediately before the sweep-line deletes the states in the current layer and moves on to the next layer.

The key observation is that the property $\mathbf{AG}\,\mathbf{EF}\,\phi$ holds if and only if each terminal SCC contains at least one state satisfying the state predicate ϕ. As an example consider Fig. 1. Because states 8 and 11 both satisfy p, then each of the two terminal SCCs contains a state satisfying p, and hence $\mathbf{AG}\,\mathbf{EF}\,p$ holds. $\mathbf{AG}\,\mathbf{EF}\,q$ does not hold since SCC$\{8\}$ does not contain a state satisfying q. Furthermore, the property $\mathbf{AG}\,\mathbf{AF}\,q$ holds if and only if all cycles within each SCCs contains a state satisfying q. As can be seen in Fig. 1, all cycles contains a state satisfying either p or q, and hence $\mathbf{AG}\,\mathbf{EF}\,(p \vee q)$ holds. In contrast, $\mathbf{AG}\,\mathbf{AF}\,p$ does not hold since the cycle consisting of states 2, 3 and 4 contained in SCC$\{2, 3, 4\}$ does not include a state satisfying p. The cycle consisting of states 7, 9, 10 in SCC$\{7, 9, 10, 11\}$ is another cycle demonstrating why $\mathbf{AG}\,\mathbf{AF}\,p$ does not hold.

4 Algorithm and Correctness

Algorithm 1 specifies our sweep-line model checking algorithm. It is identical to the standard sweep-line algorithm for monotonic progress measures [2] with the exception of line 19 in which we invoke the procedure CHECKSCC to be presented later in this section.

The sweep-line algorithm starts by initialising the set of visited nodes $\mathcal{N}$ and the set of unprocessed nodes $\mathcal{U}$ to be the initial state s_0. In addition to the sets of visited nodes and unprocessed nodes, the algorithm also maintains a set $\mathcal{L}$ consisting of the nodes that have been processed in the current layer and a value *pcl* storing the progress value for the current layer. The algorithm executes a loop (lines 14-37) until we have no longer any unprocessed nodes (states). In each iteration of the loop, we pick one of the nodes s with the lowest progress measure among the unprocessed nodes (line 15). We then

Algorithm 1 The sweep-line $AG\{EF,AF\}$ model checking algorithm

```
 1: Set of nodes (states) 𝒩                          ▷ Visited nodes currently stored
 2: Set of nodes (states) 𝒰                          ▷ Unprocessed nodes
 3: Set of nodes (states) ℒ                          ▷ Nodes processed in the current layer
 4: pcl                                              ▷ Progress value for the current layer
 5: (s_0, R)          ▷ Initial state and transition relation for the Kripke structure
 6: (O, ⊑, ψ)                                        ▷ Progress measure
 7: Φ                                                ▷ CTL AG{EF,AF} formula to be checked

 8: procedure SWEEP()
 9:     𝒩 ← {s_0}
10:     𝒰 ← {s_0}
11:     plc ← ψ(s_0)
12:     L ← ∅
13:
14:     while 𝒰 ≠ ∅ do
15:         let s be such that s ∈ 𝒰 and ∀ s' ∈ 𝒰 : ψ(s) ⊑ ψ(s')
16:         𝒰 ← 𝒰 \ {s}
17:
18:         if plc ≠ ψ(s) then                      ▷ move into the next layer
19:             CHECKSCC(ℒ, R, Φ)
20:             𝒩 ← 𝒩 \ ℒ
21:             ℒ ← ∅
22:             pcl = ψ(s)
23:         end if
24:
25:         ℒ ← ℒ ∪ {s}
26:
27:         for all (s, s') such that (s, s') ∈ R do      ▷ explore successor nodes
28:             if s' ∉ 𝒩 then
29:                 if ψ(s) ⋤ ψ(s') then
30:                     EXIT("Error: progress measure not monotonic")
31:                 else
32:                     𝒰 ← 𝒰 ∪ {s'}
33:                     𝒩 ← 𝒩 ∪ {s'}
34:                 end if
35:             end if
36:         end for
37:     end while
38:
39:     return true
40:
41: end procedure
```

first check (line 18) if this node is in a subsequent layer, i.e., we are now about to move into the next layer. In that case we invoke the CHECKSCC procedure with the set of nodes $\mathcal{L}$ in the current layer before deleting the nodes in the current layer from the set

of visited nodes. The next step is then to add s to the current layer (line 25) and then compute the successors of node s. If the node has not yet been visited, then it is added to the set of unprocessed nodes. The algorithm also checks on-the-fly that the progress measure is indeed monotonic (line 29) and if not, then the algorithm terminates with an error message.

The correctness of the algorithm rests on the fact that the sweep-line method visits all reachable states, that each SCC is contained in a single layer, and that the CTL properties can be checked by inspection of the SCCs individually. That the basic sweep-line algorithm (without the invocation of the CHECKSCC procedure) terminates after having visited all reachable states (completeness) is stated in the following theorem which was already proved in [2]:

Theorem 1 (Completeness and Termination) *[2] Let $M = (S, s_0, R, L)$ be a Kripke structure and $\mathcal{P} = (O, \sqsubseteq, \psi)$ a monotonic progress measure on M. The sweep-line algorithm terminates after having visited all states reachable from the initial state s_0.*

To establish the correctness of the new sweep-line algorithm augmented with the CHECKSCC procedure, we first establish a proposition stating that all nodes of an SCC of the Kripke structure (state space) will be contained in a single layer, i.e., an SCC cannot span multiple layers.

Proposition 1 *Let $M = (S, s_0, R, L)$ be a Kripke structure and $\mathcal{P} = (O, \sqsubseteq, \psi)$ a monotonic progress measure. Let SCC be the set of strong connected components of M, and let $scc \in SCC$ be a strongly connected component. Then: $\forall s, s' \in scc : \psi(s) = \psi(s')$.*

Proof. Assume that there exists an $scc \in SCC$ and states $s, s' \in scc$ such that $\psi(s) \neq \psi(s')$. Hence either $\psi(s) \not\sqsubseteq \psi(s')$ or $\psi(s') \not\sqsubseteq \psi(s)$. Since s and s' are in the same SCC, then they are mutually reachable and there must therefore exists $(s_i, s_j) \in R$ on the path from either s to s' or s' to s such that $\psi(s_i) \not\sqsubseteq \psi(s_j)$. This contradicts that the progress measure is monotonic.

A consequence of Prop. 1 is that SCCs can be computed by considering one layer at a time. Furthermore, Thm. 1 ensures that the sweep-line method covers all reachable states which means that we will encounter all SCCs at some stage. The remaining step is therefore to link the inspection of SCCs to the model checking of the P-AGEF and P-AGAF properties. This is done in the proposition below which formalises the requirements informally introduced at the end of Sect. 3:

Proposition 2 *Let $M = (S, s_0, R, L)$ be a Kripke structure. Let SCC be the set of strongly connected components of M, $SCC_T \subseteq SCC$ the set of terminal strongly connected components, and let ϕ be a state predicate. Then:*

1. $M, s_0 \models \mathbf{AG}\,\mathbf{EF}\,\phi \Leftrightarrow \forall scc \in SCC_T \exists s \in scc : \phi(s)$
2. $M, s_0 \models \mathbf{AG}\,\mathbf{AF}\,\phi \Leftrightarrow \forall scc \in SCC : (scc \setminus \{s \in scc : \phi(s)\}, R)$ *is acyclic*

Proof. First we prove 1. Assume that $\mathbf{AG}\,\mathbf{EF}\,\phi$ holds and there exists a terminal SCC named *scc* such that no states in *scc* satisfy ϕ. Since all states belong to some SCC, then we can find a path from the initial state to a state s in *scc*. Since *scc* is terminal and do not contain states satisfying ϕ, then we can no longer reach states that satisfies ϕ from s. Hence $\mathbf{AG}\,\mathbf{EF}\,\phi$ cannot hold. Assume that each terminal SCC contains a state satisfying ϕ and let s be any reachable state. Since we cannot have cycles that spans multiple SCCs and all states belong to some SCC, there must exists a path from the SCC to which s belongs to a state s' in some terminal SCC. Within this terminal SCC, all states are mutually reachable and by our assumption at least one state in there satisfy ϕ. Hence $\mathbf{AG}\,\mathbf{EF}\,\phi$ holds.

Next we prove 2. Assume that $\mathbf{AG}\,\mathbf{AF}\,\phi$ holds and there exists an *scc* such that when all states satisfying ϕ are removed from *scc* we still have a cycle consisting of states in *scc*. In that case we can find a path $s_0, s_1 \ldots s$ leading to a state s on this cycle, and we can then extend this to an infinite path by repeating the states on the cycle to which s belong. Since no state on the cycle satisfy ϕ, then $\mathbf{AG}\,\mathbf{AF}\,\phi$ cannot hold. Hence we cannot have such cycles. Assume now that each strongly connected component becomes acyclic when removing states satisfying ϕ. Since all cycles belongs to some strongly connected component, then we cannot have cycles where no states satisfy ϕ. Hence from any states on an infinite path we must eventually encounter a state satisfying ϕ which means that $\mathbf{AG}\,\mathbf{AF}\,\phi$ holds.

Based on Prop. 2 we can now specify the CHECKSCC procedure which is given in Algorithm 2. The procedure first computes the SCCs of the given layer $\mathcal{L}$. Here any algorithm for computing SCCs can be used, and we do not specify this further. Based on the SCCs and Prop. 2, the procedure then checks whether the property being investigated is violated in which case false is returned and the entire algorithm terminates.

Algorithm 2 Checking strongly connected components of the current layer

```
1:  SCC                                          ▷ set of strongly connected components
2:
3:  procedure CHECKSCC(L, R, Φ)
4:      SCC ← COMPUTESCCS(L, R)
5:      if Φ ≡ AG EF φ then
6:          for all scc ∈ SCC do
7:              if ISTERMINAL(scc, R) ∧ ∀s ∈ scc : ¬φ(s) then
8:                  return false                 ▷ Terminal SCC without node satisfying φ
9:              end if
10:         end for
11:     end if
12:     if Φ ≡ AG AF φ then
13:         for all scc ∈ SCC do
14:             V ← scc \ {s ∈ scc | φ(s)}
15:             if HASCYCLE(V, R) then
16:                 return false     ▷ SCC with cycle not containing a node satisfying φ
17:             end if
18:         end for
19:     end if
20: end procedure
```

We have not specified the details of the ISTERMINAL and HASCYCLE procedures. The

ISTERMINAL procedure can be implemented by checking that all successors of nodes in the SCC are contained in the SCC. The HASCYCLE procedure can be implemented by, e.g., a depth-first search of the nodes in V.

The completeness of the basic sweep-line algorithm and Prop. 1 ensures that all strongly connected components will eventually have been computed and inspected in Algorithm 2. Furthermore, Algorithm 2 is a direct implementation of the two properties stated in Prop. 2. We therefore have the following theorem concerning the correctness of our algorithm:

Theorem 2 *Let $M = (S, s_0, R, L)$ be a Kripke structure, let $\mathcal{P} = (O, \sqsubseteq, \psi)$ be a monotonic progress measure on M, and let $\Phi \equiv \mathbf{AGEF}\phi$ or $\Phi \equiv \mathbf{AGAF}\phi$. Then Algorithm 1 terminates and $M, s_0 \models \Phi$ if and only if the algorithm returns true.*

In Algorithm 2 we have separated the computation of SCCs from the checking of the SCCs. As an optimisation it is possible to integrate the checking of the properties of the SCC into the SCC computation algorithm. This could make it possible to check the SCCs as they are encountered by the SCC-algorithm. As a further optimisation it is also possible to compute the SCCs as the layer is being explored and not at the end of exploring a layer. However, for reason of clarity, we have decided to separate the two steps in the formulation of the algorithm.

An important property of model checking algorithms is the ability of providing error-traces (counter examples) in the cases where a property cannot be verified. Obtaining error-traces (paths) that demonstrate why a property does not hold requires special techniques in the context of the sweep-line method. The reason is that parts of the error-traces will in most cases have been deleted from memory during the state space exploration. A technique based on writing an inverse spanning tree to external storage during the state space exploration was presented in [9] and is compatible with the algorithm described above.

5 Implementation and Experimental Results

We have made an implementation of our new sweep-line algorithm using the C++ programming language [10]. The implementation uses a hash table to store the set of visited nodes currently in memory, and a priority queue for storing the set of unprocessed states. The progress value of a state is used as priority in the priority queue in order to ensure a least-progress-first exploration of the state space (smallest progress value has highest priority). We have used Tarjan's algorithm [11] to compute the SCCs in each layer during sweep-line exploration. The implementation also includes a simple implementation of the spanning tree algorithm [9] in order to provide error-traces.

We have evaluated the implementation of our algorithm on a communication protocol system in which a sender is to send a number of data packets to a receiver across a network where packets may be lost and overtake each other. The protocol employs sequence numbers, timeouts, acknowledgments, and a stop-and-wait retransmission strategy in order to ensure that the receiver receives all data packets exactly once and in the correct order. The basic idea is that the sender keeps sending the current data packet until an acknowledgment on this data packet is received. Upon reception of an acknowledgment, the sender starts sending the next data packet. The protocol has two parameters: the number of data packets to transmit and the number of packets (data and acknowledgments) that can be under transmission in the network. We consider two

variants of the protocol: one in which the receiver always sends an acknowledgment when receiving a data packet, and one in which the receiver only sends an acknowledgment when it is the expected data packet arriving. The protocol has been modelled using the Coloured Petri Nets modelling language and CPN Tools. Due to space limitations we do not present the model here but have made it available via [10].

The sequence numbers stored locally by the sender and the receiver are increasing and hence we can use these as a measure of how far the protocol has progressed. Specifically, the sender keeps a sequence number indicating the sequence number of the data packet currently being sent, and the receiver keeps a sequence number indicating the sequence number of the data packet expected next. This means that we can map each state of the system into a pair of integers consisting of the sequence number stored at the sender and the sequence number stored at the receiver. Lexicographical ordering on these pairs then gives a monotonic progress measure for the protocol system.

For the protocol system, we consider verification of the following properties:

P1: $\mathbf{AGEF}\,\phi_{allrcv}$ where ϕ_{allrcv} is a state predicate stating that all data packets have been received in the correct order. Hence, the property states that it is always possible to enter a state in which all data packet have been correctly received.

P2: $\mathbf{AGAF}\,\phi_{dataall}$ where $\phi_{dataall}$ is a state predicate stating that a data packet has been sent or all data packets has been received. Hence, the property states that eventually we either send a data packet or all data packets have been received.

P3: $\mathbf{AGAF}\,\phi_{ack}$ where ϕ_{ack} is a state predicate stating that an acknowledgment has been sent. Hence, the property states that we always eventually send an acknowledgment.

Table 1 summarises selected experimental results for different configurations of the protocol. Configurations are written on the form $N-C$ where N specifies the number of data packets and C specifies the capacity of the network. The States column gives the number of states in the state space and the Edges gives the number of edges. The Peak column lists the peak number of states stored in memory during exploration as percentage of the total number of states. Finally, the column E-Time gives the time in milliseconds for conducting an ordinary sweep-line exploration (without CTL model checking). Column C-Time gives the extra time for doing CTL model checking using our new sweep-line algorithm relative to the E-Time column, i.e., basic sweep-line exploration. Finally, column T-Time gives the extra time relative to the E-Time when we additionally stored a spanning tree in external storage to be able to provide error-traces.

We have not listed the configurations related to the checking of $\mathbf{AGEF}\,\phi_{allrcv}$ and $\mathbf{AGAF}\,\phi_{ack}$ in the case where an acknowledgment is sent only when it is the expected data packet arriving. The reason is that the purpose of these configurations were to check that the our implementation is able to correctly detects violations of a property being verified. Property P1 does not hold in these configurations and the error-trace provided shows how loss of an acknowledgment will cause the sender to keep sending a data packet which is not the one the receiver is expecting. This in turn means that the receiver will not send an acknowledgment and the protocol is no longer able to reach the state in which all data packets have been correctly received. Property P3 is not satisfied in any of the configurations. These results are as expected and contributes to validating the correctness of our implementation.

The results from the verification shows that properties P1 and P2 hold in all $N-C$ configurations (as expected). As can be seen from the Peak column, then the sweep-line

Config	Property	States	Edges	Peak	E-Time	C-Time	T-Time
22-2	$\mathbf{AGEF}\phi_{allrcv}$	7,944	22,419	6.6 %	7 ms	42.9%	485.7%
22-2	$\mathbf{AGAF}\phi_{dataall}$	7,944	22,419	6.6 %	7 ms	42.9%	471.4%
30-2	$\mathbf{AGEF}\phi_{allrcv}$	14,672	41,611	4.9 %	14 ms	42.9%	464.3%
30-2	$\mathbf{AGAF}\phi_{dataall}$	14,672	41,611	4.9 %	15 ms	40.0%	420.0%
10-3	$\mathbf{AGEF}\phi_{allrcv}$	24,052	98,671	19.8 %	35 ms	31.4%	285.7%
10-3	$\mathbf{AGAF}\phi_{dataall}$	24,052	98,671	19.8 %	35 ms	34.3%	308.6%
15-3	$\mathbf{AGEF}\phi_{allrcv}$	78,027	326,906	13.8 %	152 ms	27.0%	241.4%
15-3	$\mathbf{AGAF}\phi_{dataall}$	78,027	326,906	13.8 %	147 ms	32.0%	240.8%
16-3	$\mathbf{AGEF}\phi_{allrcv}$	94,226	395,825	13.0 %	236 ms	55.5%	178.4%
16-3	$\mathbf{AGAF}\phi_{dataall}$	94,226	395,825	13.0 %	182 ms	37.4%	240.7%
17-3	$\mathbf{AGEF}\phi_{allrcv}$	112,525	473,808	12.3 %	306 ms	46.4%	229.4%
17-3	$\mathbf{AGAF}\phi_{dataall}$	112,525	473,808	12.3 %	237 ms	33.3%	219.4%
18-3	$\mathbf{AGEF}\phi_{allrcv}$	133,052	561,415	11.6 %	287 ms	45.6%	231.0%
18-3	$\mathbf{AGAF}\phi_{dataall}$	133,052	561,415	11.6 %	284 ms	38.4%	221.1%
19-3	$\mathbf{AGEF}\phi_{allrcv}$	155,935	659,206	11.1 %	348 ms	38.8%	215.8%
19-3	$\mathbf{AGAF}\phi_{dataall}$	155,935	659,206	11.1 %	357 ms	47.3%	250.1%
20-3	$\mathbf{AGEF}\phi_{allrcv}$	181,302	767,741	10.5 %	437 ms	38.0%	200.7%
20-3	$\mathbf{AGAF}\phi_{dataall}$	181,302	767,741	10.5 %	449 ms	33.2%	186.4%

Table 1: Selected experimental results on the communication protocol example.

method significantly reduces the peak number of states stored (to typically between 5 % - 20 %). Reduction of peak memory usage is the main goal in model checking as it is space which is the limited factor. Furthermore, the C-Time columns shows that the overhead of our SCC-based CTL model checking algorithm is relatively small. The main contributer to overhead in terms of time is the external storage of information to generate error-traces as is evident from the T-Time column. This is, however, not an attribute of our CTL model checking algorithm but applies also when used with the sweep-line algorithms for safety and LTL model checking.

6 Conclusions and Future Work

We have proposed a variant of the sweep-line method that enables on-the-fly model checking of two commonly used CTL properties. The key idea in our approach was to combine the computation of strongly connected components with the least-progress-first exploration of the sweep-line method. Our initial experimental results show that the sweep-line method is able to significantly reduce peak memory usage which is important as space is the limiting factor in model checking. Furthermore, the CTL properties can be checked with a modest overhead in terms of time.

CTL model checking with the sweep-line method has until now been an open research problem, and the algorithm presented represents a first step towards addressing this. The extension of our approach to cover a larger subset of CTL properties is an important direction of future work. A related direction is to develop CTL model checking techniques that can be used for non-monotonic progress measures - and not only monotonic progress measures as presented in this paper.

Our experimental results also showed the relevance of investigating efficient external-memory (I/O) techniques for writing the inverse spanning tree (needed in case of counter example generation) to external storage during state space exploration. A direction for future work is also to evaluate our implementation on a larger set of example models.

References

[1] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[2] S. Christensen, L. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proc. of TACAS'01*, volume 2031 of *Lecture Notes in Computer Science*, pages 450–464. Springer, 2001.

[3] E. Clarke, E. Emerson, and A. Sistla. Automatic Verification of Finite-state Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[4] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 2000.

[5] S. Evangelista and L. Kristensen. Hybrid On-the-fly LTL Model Checking with the Sweep-Line Method. In *Proc. of ICATPN'12*, volume 7347 of *Lecture Notes in Computer Science*, pages 248–267, 2012.

[6] H. Iwashita, T. Nakata, and F. Hirose. CTL Model Checking Based on Forward State Traversal. In *Proc. of Computer-Aided Design*, pages 82–87. IEEE Computer Society, 1996.

[7] K. Jensen, L. Kristensen, and T. Mailund. The Sweep-line State Space Exploration Method. *Theoretical Computer Science*, 429:169–179, 2012.

[8] L. Kristensen and T. Mailund. A Generalised Sweep-Line Method for Safety Properties. In *Proc. of Formal Methods 2002*, volume 2391 of *Lecture Notes in Computer Science*, pages 549–567, 2002.

[9] L. Kristensen and T. Mailund. Efficient Path Finding with the Sweep-Line Method Using External Storage. In *Proc. of Internatioanl Conference on Formal Engineering Methods*, volume 2885 of *LNCS*, pages 319–337. Springer, 2003.

[10] Sweep-Line C++ Implementation Library. Available via: `https://bitbucket.org/exoen/sweepline`, 2017.

[11] R. Tarjan. Depth-first Search and Linear Graph Algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[12] A. Valmari. The State Explosion Problem. In *Lecture on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer, 1998.