Implementing a CTL Model checker

Keijo Heljanko

Helsinki University of Technology, Digital Systems Laboratory
Otakaari 1, FIN-02150 Espoo, Finland
Keijo.Heljanko@hut.fi

July 29, 1996

Abstract

This paper discusses the implementation of a branching time temporal logic *CTL model checker* for the *PROD* Pr/T-Net Reachability analysis tool. A *new algorithm* for model checking CTL is presented. This algorithm doesn't need the converse of the transition relation as the EMC algorithm does [4]. The algorithm also provides a *counterexample and witness facility* using *one-pass* reachability graph traversal. The ALMC *local model checking* algorithm as presented in [10] uses a two-pass algorithm. The new algorithm presented here is a *global model checking* algorithm and requires *less memory* in the worst case than the local model checking ALMC algorithm.

Classification: Concurrency, distributed systems

1 Introduction

This paper discusses the implementation of a branching-time temporal logic *CTL model checker* for the *PROD* Pr/T-Net Reachability analysis tool [7], [9] . The need for a new branching time model-checker was raised by the users of PROD. The model checker would be implemented in the reachability graph traversal tool PROBE, so the implementation would have the reachability graph generated for it.

The partial order methods implemented in the PROD tool set reachability graph generation do not preserve the branching time temporal logic CTL, and can't be applied to the reachability graph generation when checking CTL properties. Therefore the full reachability graph needs to be generated. The full reachability graph generation is the bottleneck in using the CTL model checker.

Model checking CTL has been know to have a linear complexity in terms of the size of the reachability graph to be checked and also in the size of the CTL formula to be checked [4]. We studied two existing algorithms for model checking CTL: the EMC algorithm [4] by Clarke, Emerson and Sistla, and the ALMC algorithm [10] by Vergauwen and Lewi.

What was discovered was that while both of the algorithms could be used as a basis for a linear time model checker, they both had some drawbacks in our application.

The EMC model checker needs both the forward transition relation, and the *converse of the transition relation* during model checking. While the converse of the transition relation can be theoretically computed in linear time, it requires additional storage space which is *proportional to the number of edges* in the reachability graph. Another problem with the EMC algorithm is that it doesn't contain a *counterexample and witness facility* [3]. These are in our experience very valuable when for example verifying an unfinished design that contains faults. The model checker can give a counterexample path along which a universally quantified formula does not hold, or a witness path along which an existentially quantified the formula holds.

The ALMC model checker is a so called *local model checker*. It evaluates the truth value of the formula to be checked in a *top-down* manner evaluating only those subformulas that are needed to decide the truth value of the top level formula. The ALMC algorithm stores information about which subformulas it has already evaluated and the truth values of those subformulas. Evaluating a subformula in ALMC algorithm requires space for search structures which is propositional to the number of nodes in the reachability graph in the worst case. Because evaluating a subformula might trigger the evaluating its subformulas, the search structures for the ALMC algorithm can require

several times more memory than a *global model checker* in the worst case. The added memory used for book-keeping enables a local model checker to evaluate only a *subset* of those formulas that a global model checker does. In the worst case it needs to evaluate all the subformulas in the same way a global model checker does.

Because the model checking algorithms presented in the literature were not a perfect match for our needs, we developed a new model checking algorithm. It is based on the same basic ideas as the EMC and ALMC algorithm, but also contains some new ideas. The resulting new algorithm is *a global model checker with a counterexample and witness facility.*

Our algorithm uses directed graph *depth-first search* as the basic algorithm on which the model checking algorithm is implemented. It enables us to do CTL model checking memory-efficiently with a counterexample and witness facility provided without increasing the time or space complexity of the algorithm.

One implementation option was to make the CTL model checker a *BDD-data structure* [1], [8] based one. It was discovered that we couldn't get all the benefits of the BDD-based model checking without a major rewrite of the PROD tools. We decided that a more conventional approach would fit the current tools better and we could still be able to obtain very significant improvements in CTL model checking compared to the older versions of the PROD tools.

In *chapter 2* we shortly define the branching time temporal logic CTL. *Chapter 3* discusses CTL model checking complexity and the local vs. global model checking. In *chapter 4* we present our *new algorithm. Chapter 5* discusses the PROD implementation issues.

2 Branching time temporal logic CTL

Here we briefly discuss the branching time temporal logic CTL. For a more thorough discussion on different temporal logics we direct the reader to [5]. This section is largely based on the paper by Clarke, Emerson and Sistla [4]. We discuss model checking finite-state systems that are described as a labeled state-transition graph. This structure is a triple $M = (S, R, P)$ where

1. S is a finite set of states
2. R is a binary relation on $S (R \subseteq S \times S)$ which gives the possible transitions between states. It must be total; $\forall x \in S \, \exists y \in S \, [(x, y) \in R]$
3. $P : S \rightarrow 2^{AP}$ assigns to each state atomic proposition true in that state

The formal syntax of CTL is given below:

1. Every atomic proposition $p \in AP$ is a CTL formula
2. If f_1 and f_2 are CTL formulas, so are $\neg f_1, f_1 \wedge f_2, EX f_1, A[f_1 U f_2]$ and $E[f_1 U f_2]$.

A *path* is an infinite sequence of states $(s_0, s_1, \ldots)$ such that $\forall i \, [(s_i, s_{i+1}) \in R]$. For any structure $M = (S, R, P)$ there is an infinite computation tree with root labeled s_0. $s \rightarrow t$ is a arc in the tree iff $(s, t) \in R$. The relation $M, s_0 \models f$ means that the formula f holds at state s_0 in structure M. The relation $\models$ is defined inductively as follows:

1. $M, s_0 \models p$ iff $p \in P(s_0)$
2. $M, s_0 \models \neg f$ iff not$(M, s_0 \models f)$
3. $M, s_0 \models f_1 \wedge f_2$ iff $M, s_0 \models f_1$ and $M, s_0 \models f_2$
4. $M, s_0 \models EX f_1$ iff $M, t \models f_1$ for some state t such that $(s_0, t) \in R$
5. $M, s_0 \models A[f_1 U f_2]$ iff for all paths $(s_0, s_1, \ldots)$,
   $\exists i \, [i \geq 0 \wedge M, s_i \models f_2 \wedge \forall j \, [0 \leq j < i \implies M, s_j \models f_1]]$
6. $M, s_0 \models E[f_1 U f_2]$ iff for some path $(s_0, s_1, \ldots)$,
   $\exists i \, [i \geq 0 \wedge M, s_i \models f_2 \wedge \forall j \, [0 \leq j < i \implies M, s_j \models f_1]]$

We can use some abbreviation to help us in writing CTL formulas more easily:

1. $AX(f) \equiv \neg EX(\neg f)$ means that f holds in all the immediate successor states of s
2. $AF(f) \equiv A[\text{True} \, U \, f]$ means that f holds in the future along every path
3. $EF(f) \equiv E[\text{True} \, U \, f]$ means that there is some path which leads to a state at which f holds
4. $AG(f) \equiv \neg EF(\neg f)$ means that f holds in every state on every path
5. $EG(f) \equiv \neg AF(\neg f)$ means that there is some path on which f holds in every state

3 CTL Model checking

CTL model checker is a program that answers the problem stated as follows: Given a structure M and a CTL formula f, determine which states of M satisfy f. We may be only interested whether the formula is satisfied in a state s of the structure M. This is called *local model checking* [10]. Another possibility is that we want to know the satisfiability of formula f in all the states of the structure M. This is called *global model checking* correspondingly.

The model checking problem for CTL has been shown to be linear in both the length of the formula f and the size of the structure M [4]. Therefore in the worst case CTL model checkers discussed here require $O(length(f) \times (card(S) + card(R)))$ running time.

A global model checker evaluates the truth value of the top-level formula in a *bottom-up* manner. It always evaluates the truth values of subformulas in all states of a graph before trying to evaluate a formula that depends on these subformulas. If some of these subformula truth values are not needed to evaluate the truth value of the top-level formula, unnecessary work has been performed. One way of looking at globally evaluating a subformula is that it creates a new atomic proposition that is evaluated to either `true` or `false` for all the states of the reachability graph.

A local model checker, which only wants to know the satisfiability of a formula in one state can evaluate only those of the subformulas it requires to show whether the top-level formula is satisfied or not. The drawback is that it needs to do more complicated *book-keeping* to lazily evaluate only those subformulas it needs. This translates into a slower execution when compared to a simpler global model checking algorithm. The ALMC [10] local model checking algorithm also has to keep in memory the search structures needed for evaluating a top-level formula when encountering a non-evaluated subformula. Evaluating this subformula can recursively cause an evaluation of its subformulas. The size of each of these search structures are proportional to the number of nodes in the reachability graph for each subformula.

We decided to develop a global model checking algorithm because it requires *less memory* for both storing the subformulas and the subformula evaluation search structures in the worst case. This combined with the *reduced book-keeping overhead* of the model checking algorithm will hopefully make up for the potentially larger number of subformulas required to be evaluated.

4 A Global Model Checking Algorithm

Here we present a new algorithm for model checking CTL. It owes a lot to the algorithms EMC [4] and ALMC [10]. It presents an algorithm which has the reduced book-keeping overhead of the EMC algorithm, without the need for the backward transition relation. Another major benefit is the counterexample and witness facility, which is provided without affecting the complexity of the algorithm. The paper presenting the ALMC algorithm uses a *two-pass* depth-first search method to implement model checking the temporal subformula $E\,[f_1\,U\,f_2]$. The basic idea of used by our algorithm for model checking the temporal formula $E\,[f_1\,U\,f_2]$ is the same as used in the ALMC algorithm, but our algorithm uses simpler datatypes and completes the model checking using only *one-pass* depth-first search.

Our algorithm implements local model checking for the top-level subformula. This can be seen as an optimization which is provided because it required no extra memory or running-time overhead.

Because of space considerations we won't duplicate the parts of the model checking algorithm that are identical to the ones represented in [4] and [10]. We present here the procedures needed for model checking only the temporal subformulas $A\,[f_1\,U\,f_2]$ and $E\,[f_1\,U\,f_2]$. The main program and the algorithms needed for model checking other temporal subformulas are either trivial or identical to the algorithms presented in [4], [10].

4.1 Model checking the formula $A\,[f_1\,U\,f_2]$

Implementation of the subroutine *check_au*, which is used to model check the temporal formula $A\,[f_1\,U\,f_2]$, is very similar to the one used in the ALMC algorithm. Because our algorithm is a global model checker, some of the logic needed in evaluating a formula can be replaced by initialization which uses only simple boolean operations on subformula truth values.

Subroutine *check_au* is provided with truth values of the subformulas *f1* and *f2* evaluated in all states. It labels states *marked* when the truth of the formula has been evaluated in that state, and it stores the truth values of the formula to be evaluated for each state in *labeled*. All of these can be simply implemented with bit-arrays containing as many bits as there are states.

The actual subroutine *check_au* is called with additional parameters *state* and *check_global*. The parameter *state* is the state we want the formula $A\,[f_1\,U\,f_2]$ to be evaluated in, and *check_global* is

`false` when the formula to be checked is a top-level subformula, and a counterexample path is to be produced instead. Here is the pesudo-code for the subroutine *check_au*:

```
1  proc check_au(state, f1, f2, check_global) ≡
2        init_f_and_marked(f1, f2);                          Initialize from subformula values
3        if check_global then
4                              foreach s ∈ S do              Evaluate formula in all states
5                                 if ¬marked(s) then
6                                                  st := empty_stack;
7                                                  visit_au(s, st);
8                                 fi;
9                              od;
10                       else
11                             st := empty_stack;            Evaluate formula in state
12                             if ¬visit_au(state, st) then
13                                                 process_counterexample(st);
14                             fi;
15       fi;
16       return (labeled(state));
17 .
```

4.1.1 Initialization

Apart from the counterexample facility the subroutine *check_au* also differs from other model checkers in the initialization of the *marked* and *labeled*. We want the bit-array *marked* to be initialized to `true` when we can decide the truth value of the formula without searching any successor states. We want the bit-array *labeled* to be initialized to `true` only when we can decide that the formula is `true` without searching any successor states. From the definition of the temporal formula $A[f_1 U f_2]$ we can create a truth table presenting all the possible cases:

f1	f2	marked	labeled
false	false	true	false
false	true	true	true
true	false	false	false
true	true	true	true

From the column *marked* in the truth table we see that only those states in which *f1* holds but *f2* doesn't hold require further processing. Implementing these initializations can be easily be made fast by processing several states at a time in one machine word. Also if the subformulas *f1* and *f2* are no longer needed, they can be overwritten as *marked* and *labeled* respectively.

4.1.2 Searching for a counterexample

The subroutine *visit_au* is a non-recursive subroutine with explicit stack manipulation to implement depth-first search. The logic of the *visit_au* is to try to find a counterexample that shows that the formula $A[f_1 U f_2]$ doesn't hold in an unmarked state s. If no such counterexample can be found, the formula must be `true` in that state.

A *counterexample* is either a path of initially unmarked states to a state in which neither *f1* nor *f2* holds, or an infinite path (i.e. a path ending in a loop) of initially unmarked states along which *f1* holds, but *f2* does not hold. We do not need to tell these two cases apart, if we do not want to. If we have found a counterexample, all states on the search stack are part of this counterexample path and we must conclude they should be labeled `false`, which is their initialization value in the truth table above. A formal definition of the two possible cases of counterexample paths is below:

A path $\pi_c = s_0, s_1, \ldots$ is a counterexample for formula $A[f_1 U f_2]$ in state s_0 iff:

$$\exists i\,[i \geq 0 \wedge M, s_i \models (\neg f_1 \wedge \neg f_2) \wedge \forall j\,[0 \leq j < i \implies M, s_j \models (f_1 \wedge \neg f_2)]]$$

or

$$\exists i\,[i \geq 0 \wedge \exists j\,[j > 0 \wedge s_i = s_{i+j} \wedge \forall k\,[0 \leq k < i+j \implies M, s_k \models (f_1 \wedge \neg f_2)]]]$$

The definition above is suitable for our purposes when noticing that one can replace in the definition $(\neg f_1 \wedge \neg f_2)$ with $(marked \wedge \neg labeled)$, and $(f_1 \wedge \neg f_2)$ with $(\neg marked \wedge \neg labeled)$. This definition of counterexample paths is derived from the following equality of CTL formulas presented in [4]: $_1\,A\,[f_1\,U\,f_2] \equiv \quad _1\,\neg(E\,[\neg f_2\,U\,\neg f_1 \wedge \neg f_2] \vee EG(\neg f_2))$.

We only need to evaluate the formula in those states in which it is unknown i.e. which are not marked. If we find a marked and labeled state, it can't belong to a counterexample path, and therefore we do not need to consider paths going trough it as potential counterexamples.

Each time we encounter a new unmarked state, we mark it. By doing this we accomplish two things: We know the truth value of the formula in all visited states after returning from *visit_au*, and therefore no state needs to be visited more than once. Secondly we notice if a counterexample path ending in a loop is found by encountering a state marked when it was first seen by *visit_au*, and which is not yet labeled.

When all successor states of a state have been visited and no counterexample has been found, we must conclude that the formula is `true` in that state and label the state accordingly. If the search stack *st* is empty, all states reachable from the state *s* have been visited and the formula $A[f_1\,U\,f_2]$ must be `true` in state *s*.

Here is the pseudo-code for visit_au:

```
1  proc visit_au(s, st) ≡
2       while TRUE do
3            while TRUE do
4                 if ¬marked(s) then
5                      mark_state(s);                    Unmarked state found, mark it.
6                      succ_num := 0;                    Search succs for counterexample
7                      push(⟨s, succ_num⟩, st);                              Push state
8                      s := successor(s, succ_num);                  Get first successor
9                 else
10                     if ¬labeled(s) then
11                          push(⟨s, 0⟩, st);                          Counterexample
12                          return (FALSE);                                    found
13                     fi;
14                     break ;                    Already labeled, backtrack to parent
15                fi;
16           od;
17           while TRUE do
18                if empty(st) then
19                     return (TRUE);                        No counterexample found
20                fi;
21                ⟨s, succ_num⟩ := pop(st);                                 Get parent
22                succ_num := succ_num + 1;                    Try next successor
23                if has_successor(s, succ_num) then
24                     push(⟨s, succ_num⟩, st);                             Push state
25                     s := successor(s, succ_num);                         Next succ
26                     break ;                                              Visit succ
27                else
28                     add_label(s);          Succs visited, formula true
29                fi;
30           od;
31      od;
32 .
```

The stack *st* contains tuples $\langle s, succ\_num\rangle$, the *succ_num* is a index to the successor state table of state *s*. The subroutine *has_successor(s,succ_num)* returns `true` if the state *s* has over *succ_num+1* successors. The subroutine *successor(s,succ_num)* returns the successor state indexed by *succ_num*. Note that because the structures over which CTL is interpreted are total, each state has at least one successor.

4.2 Model checking the formula $E\,[f_1\,U\,f_2]$

Also the implementation of the subroutine *check_eu*, which is used to model check the temporal formula $E\,[f_1\,U\,f_2]$, is very similar to the one used in the ALMC algorithm. The difference here is that our algorithm is a global model checker that uses one-pass depth-first traversal of the reachability graph. The ALMC algorithm as presented in [10] is a local model checking algorithm which uses a two-pass depth-first traversal of the reachability graph.

Another major difference is that ALMC algorithm uses linked lists and sharing to keep track of those states which have been encountered during the depth-first search and from which the currently visited state can be reached. Our algorithm uses a stack to store these states.

Here is the pseudo-code for *check_eu*:

```
1  proc check_eu(state, f1, f2, check_global) ≡
2        fst := empty_stack;
3        clear_min_and_modified();                    Initialize min to zeroes and modified to false
4        init_f_and_marked(f1, f2);                   Initialize from subformula values
5        if check_global then
6                          foreach s ∈ S do           Evaluate formula in all states
7                            if ¬marked(s) then
8                                                st := empty_stack;
9                                                visit_eu(s, st, fst);
10                           fi;
11                         od;
12                   else
13                         st := empty_stack;         Evaluate formula in state
14                         if visit_eu(state, st, fst) then
15                                                   process_witness(st);
16                         fi;
17       fi;
18       return (labeled(state));
19 .
```

4.2.1 Initialization

As before we use the bit-array *marked* to mark those states for which the final truth value of the evaluated formula is known. Also we use the bit-array *labeled* to mark those states in which the formula is `true`.

We can see from the definition of the formula $E\,[f_1\,U\,f_2]$ that the bit-arrays marked and labeled should be initialized with exactly the same truth table as for the formula $A\,[f_1\,U\,f_2]$:

f1	f2	marked	labeled
false	false	true	false
false	true	true	true
true	false	false	false
true	true	true	true

4.2.2 Searching for a witness

The logic used in evaluating the subformula $E\,[f_1\,U\,f_2]$ is the following: We try to find a path of initially unmarked states to a state which is both marked and labeled. If no such path can be found the formula must be `false`.

For the formula $E\,[f_1\,U\,f_2]$ defining which paths are witness paths is straightforward. We can get this almost directly from the CTL definition:

A path $\pi_w = s_0, s_1, \ldots$ is a witness for formula $E\,[f_1\,U\,f_2]$ in state s_0 iff:

$$\exists i\,[i \geq 0 \wedge M, s_i \models f_2 \wedge \forall j\,[0 \leq j < i \implies M, s_j \models (f_1 \wedge \neg f_2)]]$$

Again using the initialization truth table we can replace in the definition above ($f_1 \wedge \neg f_2$) with ($\neg marked \wedge \neg labeled$), and f_2 with ($marked \wedge labeled$).

Unfortunately creating an algorithm for model checking $E\,[f_1\,U\,f_2]$ while keeping the model checking algorithm linear in the number of states is much more challenging than model checking $A\,[f_1\,U\,f_2]$.

When we find the last state of a witness path, we must decide for all visited states, whether the formula is `true` or `false` in them.

The EMC algorithm in [4] depends on the converse of the transition relation to implement this. It first globally finds those states in which f_2 holds and then does a depth-first starting from these states using states in which f_1 holds. All these states are such that $E\,[f_1\,U\,f_2]$ holds in them and are labeled when found.

The drawback of using the EMC method is that the amount of additional storage needed to store the converse of the transition relation is proportional to the number of transitions in the reachability graph.

The method we use here is to continuously keep track of those states encountered before state s the depth-first search from which the currently visited state can be reached. First we define $<$ to be the total ordering relation on visited states: $\forall s, s' \in S : s < s'$, iff state s was encountered before s' during the depth-first search. We use the symbol $\rightsquigarrow$ to represent those edges of the reachability graph which have been traversed by the current call to the subroutine *visit_ eu*. The $\overset{*}{\rightsquigarrow}$ is used to represent the reflexive, transitive closure of $\rightsquigarrow$. The set of 'father' states is defined here to be function $F : S \rightarrow \mathcal{P}(\mathcal{S})$:

$$F(s) = \left\{ s' \leq s \mid s' \overset{*}{\rightsquigarrow} s \right\}$$

When a state which is the last state of a witness path is encountered in the depth-first search, all states in $F(s)$ must also be labeled. Our algorithm *visit_eu* keeps track of the set $F(s)$ by using a stack *fst*. At the time the state s is first visited the stack *fst* contains all those visited states s' from which s can be reached: $\forall s' \in fst[s' \overset{*}{\rightsquigarrow} s]$.

Here we present only a sketch why the states stored on the stack *fst* are exactly those visited states from which the currently visited state s can be reached.

Each time a state s is visited for the first time, the stack *fst* contains the visited states from which the state s can be reached. The state s always belongs to the set $F(s)$ and is added to stack *fst* before visiting any successor state t. After this the array *min* is set to the depth-first search number of the state s and min value is marked unmodified.

Next we visit successor states of s one at a time. We return from searching a successor state t only if no witness path end state could be found from t.

When returning from a successor state t back to the parent state s, the array *min* value for the successor contains the depth-first search number of the smallest state reachable from the successor, or 0 if there is no way of finding a counterexample which contains the successor state. If the smallest state reachable through the successor t is smaller than the current smallest depth-first number stored in the variable min of s, the following is true: $\exists s' \in F(s)[s \rightsquigarrow t \overset{*}{\rightsquigarrow} s']$. The *min* value of s is in this case updated to the value of the successor t i.e depth first search number of the state s' and the min value of s is marked modified.

After all successor states of state s have been visited, and the *min* value of state s hasn't been modified, there can't exist a witness path that contains state s or any state reachable from it. Before we return to the parent state of s we must update the stack *fst* by removing s and all states found after s from the stack *fst*.

If the *min* value has been modified, we have found a loop: $\exists s' \in F(s)[s \overset{*}{\rightsquigarrow} s' \overset{*}{\rightsquigarrow} s \wedge s' < s]$. This loop requires us to keep s and all states found after it on the stack *fst*. The logic for this is that through s we can get to a state s' found earlier in the depth-first search. We can now from s' get to any new state found during the depth-first search at least as long as state s' has not been fully processed. Now we can return to the parent state of s and the stack *fst* has been updated for the next unmarked state to be visited.

All that is left for us to do in *visit_ eu* is to combine the updating of the stack *fst* with detection of the end state of a witness path. When an end state of a witness path is encountered, the stack *fst* will contain all the initially unmarked states from which it can be reached. All that is left for us to do is to label the formula `true` for all the states in the stack *fst*. If no witness can be found from the start state of the search, no witness can be found and the formula must therefore be `false` in all visited states. This is the initialized value, so nothing more needs to be done in this case.

Here is the pseudo-code for *visit_eu*:

```
1  proc visit_eu(s, st, fst) ≡
2        dfs := 1;
3        while TRUE do
4              while TRUE do
5                    if ¬marked(s) then
6                                 mark_state(s);                    Unmarked state found, mark it.
7                                 push(s, fst);                     Store state on father state stack
8                                 set_min(s, dfs);          Store the state depth-first search number
9                                 reset_modified(s); Mark minimum reachable state as unmodified
10                                dfs := dfs + 1;
11                                succ_num := 0;                    Search succs for witness
12                                push(⟨s, succ_num⟩, st);                          Push state
13                                s := successor(s, succ_num);                 Get first successor
14                           else
15                                if labeled(s) then                                Witness found
16                                      push(⟨s, 0⟩, st);                          ∀t ∈ fst[t ⇝* s]
17                                      while ¬empty(fst) do
18                                            t := pop(fst);                       Get state
19                                            set_min(t, 0);                       Cleanup
20                                            reset_modified(t);
21                                            add_label(t);                        Label state
22                                      od;
23                                      return (TRUE);
24                                fi;
25                                break ;                        Not labeled, backtrack to parent
26                   fi;
27             od;
28             while TRUE do
29                   if empty(st) then
30                         return (FALSE);                                  No witness found
31                   fi;
32                   successor_min := get_min(s);                Get smallest dfs of reachable states
33                   ⟨s, succ_num⟩ := pop(st);                                      Get parent
34                   if ((successor_min ≠ 0) ∧ (get_min(s) > successor_min))
35                     then
36                           set_min(s, successor_min);          Update smallest dfs of reachable states
37                           set_modified(s);          Mark the smallest dfs reachable form s modified
38                   fi;
39                   succ_num := succ_num + 1;                                  Try next successor
40                   if has_successor(s, succ_num)
41                     then
42                           push(⟨s, succ_num⟩, st);                                  Push state
43                           s := successor(s, succ_num);                              Next succ
44                           break ;                                                   Visit succ
45                      else                                                   All succs visited
46                           if ¬modified(s) then      Can a state with a smaller dfs number be reached?
47                                 repeat              No. → No witness can be reached from s
48                                       t := pop(fst);  or any states t reachable from it
49                                       set_min(t, 0);                              Cleanup
50                                       reset_modified(t);
51                                 until t = s;          All states reachable form s popped?
52                           fi;
53                   fi;
54             od;
55       od;
56 .
```

5 PROD implementation of the algorithm

We have implemented the model checking algorithm presented in the previous section of this paper and integrated it into the PROD tool set reachability graph traversal tool PROBE. It will be distributed in a future release of the PROD tool set.

The implementation of the CTL model checking algorithm presented in previous section has been straightforward. Initializing the bit-arrays *marked* and *labeled* several states at a time enables the model checker to avoid calling the *visit_ au* and *visit_ eu* subroutines for many states.

In our experience most of the time in CTL model checking is spent in evaluating the atomic subformulas for all states in the reachability graph. The current implementation evaluates all atomic subformulas for all states in one pass over the markings generated by the PROD reachability graph generator. This is to avoid accessing the large marking file several times for each state as atomic subformulas are evaluated. This is possible, because the amount of memory needed to store the truth values of the atomic subformulas is usually very small when compared to for example memory needed to store the markings of the Pr/T-Net reachability graph. The current code used to evaluate the atomic subformulas has been reused from the old PROBE query language. One way of speeding up the model checking would be to rewrite this code to better fit the CTL model checking algorithm.

Model checking for the top-level formula is local in the current implementation, but the implementation still evaluates the needed atomic subformulas in all states. We could easily implement a local model checker with small modifications to the algorithms presented in the previous section. The user of the tool could then choose the method better suited for his or her purpose, based on intuition of the problem at hand. A global model checker might have smaller worst case memory requirements, while a local model checker might require evaluating less subformulas.

The algorithm presented in the previous section always gives an counterexample or witness path when one can be found. This path is the first one found during the depth-first search. In some cases this path might be longer than necessary. Another algorithm could be added to the tool set, which would try to find shorter counterexample and witness paths.

6 Conclusions

Partial order reductions used in the PROD reachability graph generation do not preserve the branching time temporal logic CTL properties. Therefore we need to generate the full reachability graph for the problem at hand. This limits the usefulness of the CTL model checking algorithm presented here. Partial order reductions can be used while preserving the temporal logic CTL-X, the branching time logic CTL without the next-state operator, as demonstrated by [6], [11]. We do not currently know of any implementation combining the CTL-X preserving partial-order reductions with an on-the-fly branching time model checker.

The CTL model checker presented in this paper has *linear time complexity* both in the size of the formula to be checked and the reachability graph to be model checked. The algorithm is a *global model checking* algorithm with a *counterexample and witness facility*. It is *more memory efficient* than the ALMC algorithm [10] in the worst case. The algorithm is a *one-pass* depth-first search algorithm with small book-keeping overhead.

In their paper Cheng, Christensen and Morgensen [2] discuss a way of using the strongly connected components of a graph when model checking some often used CTL queries. The methods presented in their paper can be used also with the model checking algorithm presented here. The strongly connected components of a graph could also be used to optimize the order in which the global model checking is performed for the unmarked states in the reachability graph. We can use topological sorting to arrange the order in which the strongly connected components should be marked, starting from the terminal components. In this way we would minimize the amount of memory needed for different search stacks.

References

[1] Bryant, R. E.: *Graph-Based Algorithms for Boolean Function Manipulation.* IEEE Transactions on Computers, C-35 (1986) 8. pp. 677–691

[2] Cheng, A., Christensen, S., Mortensen, K. H.: *Model Checking Coloured Petri Nets; Exploiting Strongly Connected Components.* Proceedings of 3rd Workshop on Discrete Event Systems, Edinburgh, Scotland, UK, August 1996. To appear.

[3] Clarke, E., Grumberg, O., McMillan, K., Zhao, X.: *Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking.* Pittsburg, October 1994, Technical Report, Carnegie Mellon University, School of Computer Science, TR CMU-CS-94-204, 15 p.

[4] Clarke, E. M., Emerson, E. A., and Sistla, A. P.: *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications.* ACM Transactions on Programming Languages and Systems 8 (1986) 2. pp. 244–263.

[5] Emerson, E. A.: Temporal and modal logic, in van Leeuwen, J. (ed): *Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics*, 1990, Elsevier, pp. 995–1072

[6] Gerth, R., Kuiper, R., Peled, D., Penczek, W.: *A Partial Order Approach to Branching Time Logic Model Checking.* Proceedings of the 3rd Israel Symposium on the Theory of Computing and Systems, Tel Aviv, Israel, 1995, IEEE Computer Society Press, pp. 130–139

[7] Grönberg, P., Tiusanen, M., and Varpaaniemi, K.: *PROD—A Pr/T-Net Reachability Analysis Tool.* Otaniemi 1993, Digital Systems Laboratory, Helsinki University of Technology, Series B: Technical Reports, No. 11. 44 p.

[8] Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., Hwang, L. J.: *Symbolic Model Checking:* 10^{20} *States and Beyond.* Information and Computation, 98 (1992) 2. pp. 142–170

[9] Varpaaniemi, K., Halme, J., Hiekkanen, K., and Pyssysalo, T.: *PROD Reference Manual.* Otaniemi 1995, Digital Systems Laboratory, Helsinki University of Technology, Series B: Technical Reports, No. 13. 56 p.

[10] Vergauwen, B., Lewi, J.: *A Linear Local Model Checking Algorithm for CTL.* Best, E. (ed), Proceedings of 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 1993, Lecture Notes in Computer Science 715, Springer-Verlag, pp. 447–461

[11] Willems, B., Wolper, P.: *Partial-Order Methods for Model Checking: From Linear Time to Branching Time.* Proceedings of 11th Annual IEEE Symposium on Logic in Computer Science, New Jersey, USA, July 1996. To appear.