HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **A:** Research Reports
No. **45**; May 1997

ISSN 0783–5396
ISBN 951–22–3603–6

MODEL CHECKING THE BRANCHING TIME TEMPORAL LOGIC CTL

KEIJO HELJANKO

Digital Systems Laboratory
Department of Computer Science
Helsinki University of Technology
Otaniemi, FINLAND

Helsinki University of Technology
Department of Computer Science
Digital Systems Laboratory
Otaniemi, Otakaari 1
FIN–02150 ESPOO, FINLAND

HELSINKI UNIVERSITY OF TECHNOLOGY
DIGITAL SYSTEMS LABORATORY
Series **A:** Research Reports
No. **45**; May 1997

ISSN 0783–5396
ISBN 951–22–3603–6

Model Checking the Branching Time Temporal Logic CTL

Keijo Heljanko

Abstract: Reachability analysis is a method for analyzing the dynamic behavior of a concurrent system. One way of specifying the properties that the behaviors of the system must fulfill is to use the branching time temporal logic CTL (Computation tree logic). The process of checking whether the behavior of the system fulfills the specified property is called model checking. In this work we analyze several algorithms for CTL model checking. The main contribution of this report is a new model checking algorithm with time complexity matching the best existing algorithms, and memory requirements which are either equal or smaller than the existing algorithms, depending on the structure of the system under consideration. The algorithm has a counterexamples and witnesses facility, which is very valuable when trying to find the cause of incorrect behavior of the system. The presented algorithm is also straightforward to implement efficiently, and we have implemented the algorithm in the PROD tool set.

Keywords: Model checking, temporal logic, CTL, verification, reachability analysis, state-space exploration, PROD

Printing: TKK Monistamo; Otaniemi 1997

Helsinki University of Technology
Department of Computer Science
Digital Systems Laboratory
Otaniemi, Otakaari 1
FIN–02150 ESPOO, FINLAND

Phone: $\frac{90}{+358\text{–}0}$ 4511
Telex: 125 161 htkk fi
Telefax: +358–0–465 077
E-mail: lab@saturn.hut.fi

Contents

1 Introduction **1**

2 Branching Time Temporal Logic **3**

2.1 Branching Time . . . 3

2.2 The Branching Time Temporal Logic CTL . . . 7

2.2.1 Counterexamples and Witnesses . . . 11

2.3 A Small Verification Example . . . 12

3 CTL Model Checking **16**

3.1 The Subformula Evaluation Order . . . 16

3.2 Global Model Checking . . . 19

3.2.1 The General Structure . . . 19

3.2.2 A Naive Implementation . . . 23

3.2.3 The EMC Model Checking Algorithm . . . 26

3.3 Local Model Checking . . . 30

4 A New Global Model Checking Algorithm **44**

4.1 Model Checking the Universal Until Formulas . . . 45

4.1.1 Searching for a Counterexample . . . 46

4.2 Model Checking the Existential Until Formulas . . . 49

4.2.1 Searching for a Witness . . . 51

4.3 Time and Space Requirements . . . 59

5 The PROD Implementation **62**

6 Conclusions **65**

References **67**

Appendix A - Mutex Example PROD Description **i**

Appendix B - Initialization Routines **ii**

1 Introduction

Reachability analysis is method for analyzing the dynamic behavior of a concurrent system. The system is described using one of the numerous available methodologies, for example: concurrent programming languages such as SDL [24] and promela [18], or Petri-Nets [14]. Reachability analysis then provides a way of creating all the possible behaviors of the system starting from a given initial state. All the possible behaviors of the system induce a reachability graph, which is a directed graph in which the nodes are global states of the system and the edges are atomic state transitions between states. Because reachability graph generation is completely mechanical it can be easily automated, and good computer tool support is available.

This thesis concentrates on an automated method to analyze the created reachability graphs using a tool. The user of the tool can of course interactively navigate the reachability graph, but the sheer size of many reachability graphs makes finding errors in this way very difficult. It is easier to specify the properties we want the reachability graph to fulfill using some formalism and let the computer do the mechanical checking. There have been many ad-hoc ways of defining such properties in the past, but different temporal logics [13] seem to be getting wide acceptance. In temporal logic we use logical formulas to specify properties that some or all of the behaviors of the system must fulfill. The process of checking whether a formula is *True* in a specified state of the reachability graph is called model checking, and it can be automated with a tool. In many cases it it also possible for the tool to give further information to help the user to find the cause of errors in the system. For example the tool can give a sequence of states leading to a state in which the specified invariant assertion doesn't hold.

In this thesis we will use the branching time temporal logic CTL (Computation tree logic) [13] for specifying the properties of systems. We are interested in the algorithms that can be used in model checking CTL formulas. The branching time temporal logic CTL has very efficient model checking algorithms. The best algorithms have worst case running time linear both in the size of the formula, and the size of the reachability graph i.e. $O(length(f) \times (|S| + |R|))$, in which the size of the reachability graph is the sum of the number of states and transitions in the reachability graph. This compares favorably to the linear time temporal logic LTL [13], for which the model checking algorithms can take exponential time and space in the size of the formula in the worst case.

The initial goal of this work was to select a CTL model checking algorithm to be implemented in the PROD reachability analyzer tool set developed at the Digital Systems Laboratory. However based on the analysis of the most advanced algorithms it could be seen that no clear winner between the existing algorithms can be found: all algorithms have different strengths. On the other hand it could be seen that a new algorithm having a combination of the good features of the algorithms could be designed.

The main contribution of this thesis is a new CTL model checking algorithm with time complexity matching the best existing algorithms, and memory requirements which are either equal or smaller than the existing algorithms, depending on the structure of the system model checked. The algorithm has a counterexamples and witnesses facility, which is very valuable when trying to find the cause of incorrect behaviors of the system we are model checking. The algorithm is also straightforward to implement efficiently, and we have implemented the algorithm in PROD tool set. The main ideas of this new algorithm were first presented in the paper [17].

In Section 2 we will introduce the branching time temporal logic framework. Section 3 contains description and analysis of existing model checking algorithms. Section 4 introduces the new model checking algorithm developed in this thesis, and compares it to the other algorithms. We will briefly discuss the implementation of the model checker in Section 5. Finally we end this thesis with conclusions and suggestions for further research in Section 6.

2 Branching Time Temporal Logic

In this section we discuss the branching time view of the behavior of the system, and we also try to give an intuitive idea on how to specify properties of a concurrent system using branching time temporal logic. The treatment is somewhat informal, and the reader is directed to the next section for a formal treatment of the subject.

2.1 Branching Time

When looking at a system the reachable states of a system can be seen as a reachability tree. The nodes of the reachability tree are global states of the system and the edges of the reachability tree are atomic state transitions. The root node of the tree is the initial state of the system. The edges beginning from a node represent all the possible state transitions from that state. In this way the tree represents all the possible future behaviors of the concurrent system starting from the initial state. Such an infinite reachability tree is represented in Figure 1. This constitutes a branching time view of the system.

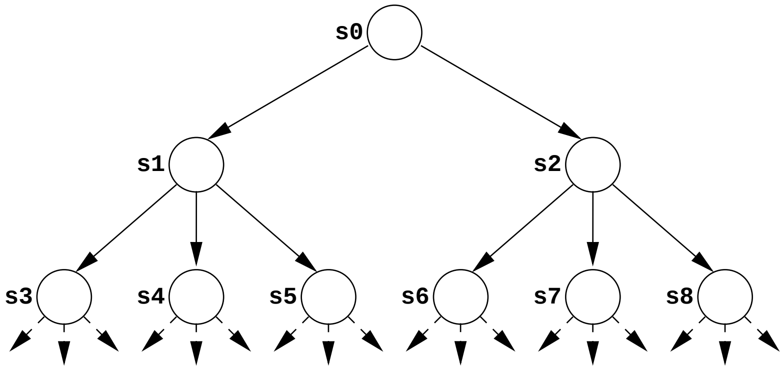

Figure 1: Reachability Tree.

Every state of a system is required to have at least one successor state in the tree, and the tree is therefore infinite and extends downwards without bounds. This is indicated in Figure 1 by the dashed arrows beginning from states **s3** through **s8**. Finite computations can be turned into infinite ones by adding dummy transitions to all states in which there are no enabled transitions. These dummy transitions do not change the state of the system in any way, and their only purpose is to make all the computations in the system infinite. A computation is then an infinite sequence of states $(s_0, s_1, s_2, \dots)$, and it is called a *fullpath*. We sometimes need to consider also finite sequences of states. To mark the difference we will call a finite sequences of states $(s_0, s_1, s_2, \dots, s_n)$ *paths*.

We need a way of specifying properties of these infinite trees to be able to reason about them. As the reachability tree is in fact induced by the state space of the system, we are at the same time specifying properties of the infinite computations of the system we are interested in. One way of specifying properties of infinite trees is to use branching time temporal logic. One such a logic is called CTL (*Computation Tree Logic*).

The branching time temporal logic CTL can conveniently express some important properties of the system we are interested in. We will formally define this logic in Section 2.2, but we'll try to give an intuitive idea of a subset of the logic to help us present examples of branching time properties.

In CTL all formulas refer to states, and the formulas specify behaviors of the system starting from an assigned state in which the formula is evaluated. If the state is omitted, the convention is that the formulas are evaluated from the initial state of the system. The syntax of the CTL is such that the atomic propositions are CTL formulas, and other CTL formulas can constructed by combining a universal (A) or an existential (E) path quantifier with some path formula. Path formulas are of the form: globally ($G(p)$), eventually ($F(p)$), next-state ($X(p)$), or until ($[p\,U\,q]$), where p and q are CTL formulas. Some examples of legal CTL formulas are $AG(p)$, $EF(p)$, and $A\,[p\,U\,EF(q)]$, where p and q are atomic propositions.

The path quantified formula is *True* iff the path formula is *True* for all (A), or some (E) fullpaths starting from the state in which the formula is evaluated. The path formula $G(p)$ is *True*, iff the formula p is *True* for all states in the fullpath. The path formula $F(p)$ is *True*, iff the formula p is *True* for some state in the fullpath. There exists also a more expressive variant of CTL, called CTL^* [13], in which the pairing of a path formula with a path quantifier is not restricted. In this work we use only the logic CTL, because of the much higher algorithmic complexities when using the more expressive logic CTL^* [13].

Using the above quantifiers we can for example verify if some property of the system are *True* invariantly. This means that there is no state in the reachability tree in which the property does isn't *True*. We can express invariant properties of the system with the branching time temporal logic CTL formula $AG(p)$. A reachability tree of a system which satisfies this formula is in Figure 2. In the figure all states which are colored black correspond to states in which the property p is *True*. If there are no states in the tree in which p isn't *True*, then it is *True* invariantly.

A path or a fullpath which forces the existential until formula to be *True* is called a witness, and a path or a fullpath which forces the universal until formula to be *False* is called a counterexample. We will formally define the notions of a counterexamples and witnesses in section 2.2.1.

The requirement specified with the branching time temporal logic CTL formula $AG(p)$ is a typical safety property. It specifies things that should never

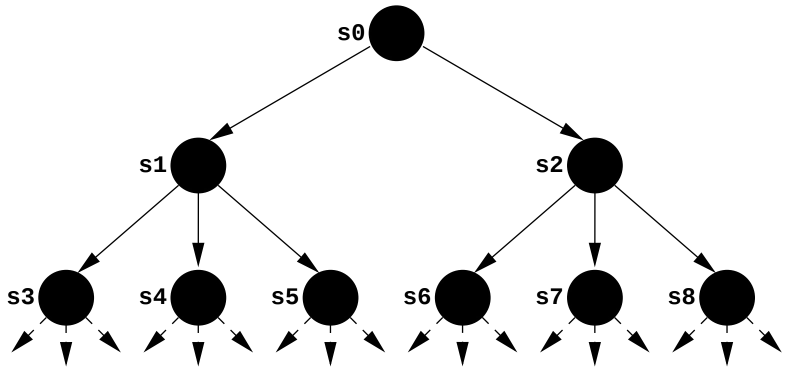

Figure 2: Reachability Tree for $AG(p)$.

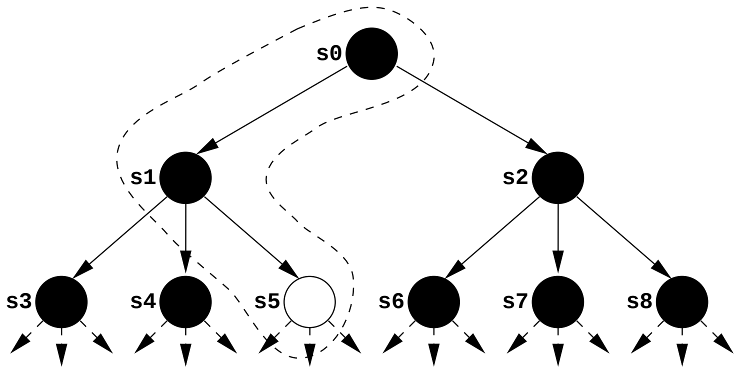

Figure 3: Counterexample for $AG(p)$.

happen in a correctly functioning specification. Sometimes these safety properties are violated, and the requirement isn't satisfied. In Figure 3 there is a state **s5** which violates the property specified by the formula. In state **s5** the invariant property p is not *True*.

The path $(\mathbf{s0}, \mathbf{s1}, \mathbf{s5})$ is a counterexample for the formula $AG(p)$ and is surrounded by a dashed line. A *counterexample* is a path or a fullpath which causes the universally path quantified formula not to be *True*. Clearly there can be several counterexamples for a formula which isn't *True*. A model checker can then output a counterexample to user who can then inspect the execution leading to the error.

Also other kinds of properties can be specified with CTL. One such property is a CTL eventuality property $AF(p)$. It specifies that starting from the initial state one should always eventually reach a state in which the property p is *True*. See Figure 4 for an example. The frontier of tree states marked with a dashed line gives a set of states (**s3, s4, s5,** and **s2**), one of which the system must eventually reach. Because in all of these states p is *True*, the required property is also always eventually *True*. As an example the property p can mean that a message is delivered. Then the CTL formula $AF(p)$ requires that the message must always be eventually delivered. If this were not the case

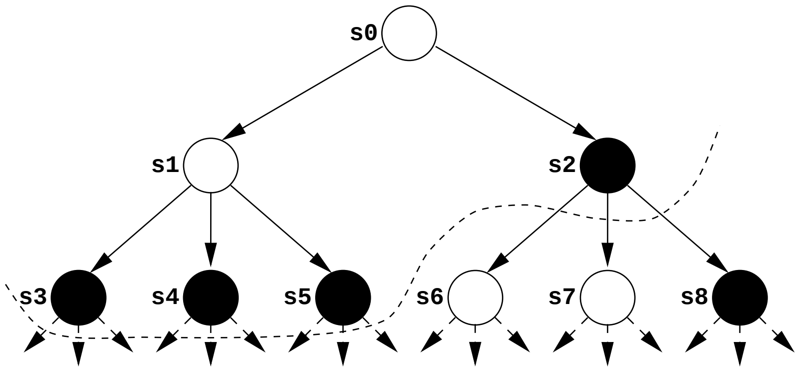

Figure 4: Reachability Tree for $AF(p)$.

then there would be some path in which the message is never delivered.

Let us now move to a more formal definition of CTL.

2.2 The Branching Time Temporal Logic CTL

In this section we will give the formal definition for the branching time temporal logic CTL. This section is largely based on [8]. The formal syntax for CTL is given below:

Definition 2.1 *Given a non-empty underlying set of atomic propositions AP, the formal syntax of CTL is given by the rules:*

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

We will call the set of all formulas which can be formed by the two rules above the set CTL. We will define the semantics of CTL formulas with respect to a labeled state-transition graph M, where the meaning of the word labeled is that states are labeled with atomic propositions, and is not to be confused with transition labels of other formalisms.

Definition 2.2 *Structure M is a labeled state-transition graph, which 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. *$P : S \to 2^{AP}$ assigns to each state atomic proposition true in that state.*

The structure M is generated form the description of the analyzed system, i.e. a program or as in our case a High-Level Petri-Net. This description language essentially defines the syntax of the atomic propositions AP. In this work the terms reachability graph and state-space as a synonyms for the structure M. Note that because the transition relation is total, each state must have at least one successor state. Next we will define the notions of a path and a fullpath. These are the finite and infinite sequences of states, which are also sometimes called computations.

Definition 2.3 *A* fullpath *is an infinite sequence of states $(s_0, s_1, \dots)$ such that $\forall i\,[(s_i, s_{i+1}) \in R]$.*

Definition 2.4 *A* path *is a finite sequence of states $(s_0, s_1, \dots, s_n)$ such that $\forall i\,[0 \leq i < n \implies (s_i, s_{i+1}) \in R]$.*

For any structure $M = (S, R, P)$, and a state $s \in S$, there is an infinite computation tree with root labeled s. $s \rightarrow t$ is an arc in the tree iff $(s, t) \in R$. We will also call this computation tree the reachability tree of the system.

The relation $M, s \models f$ means that the formula f is *True* at state s in structure M. If M is known from the context, it can be omitted.

Definition 2.5 *The relation $\models$ is defined inductively by:*

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

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

- $EX(f) \equiv \neg AX(\neg f)$ means that f is *True* in some immediate successor state of s
- $AF(f) \equiv A\,[\mathit{True}\,U\,f]$ means that f is *True* in some state in the future along every fullpath beginning from s
- $EF(f) \equiv E\,[\mathit{True}\,U\,f]$ means that there is some fullpath beginning from s, which contains a state in which f is *True*
- $AG(f) \equiv \neg EF(\neg f)$ means that f is *True* in every state on every fullpath beginning from s
- $EG(f) \equiv \neg AF(\neg f)$ means that there is some fullpath beginning from s, on which f is *True* in every state

There are other equally valid formulations of CTL, depending on which operators are taken as fundamental. An example of this is the equality presented in [8]: $A\,[f_1\,U\,f_2] \equiv \neg(E\,[\neg f_2\,U\,\neg f_1 \wedge \neg f_2] \vee EG(\neg f_2))$. Using this equality we can replace the definition of $A\,[f_1\,U\,f_2]$ above (definition number 5.) with the definition of the formula $EG(f_2)$. After this the formula $A\,[f_1\,U\,f_2]$ can be defined to be an abbreviation for the equality above. For completeness, here is the required definition of the formula $EG(f_1)$:

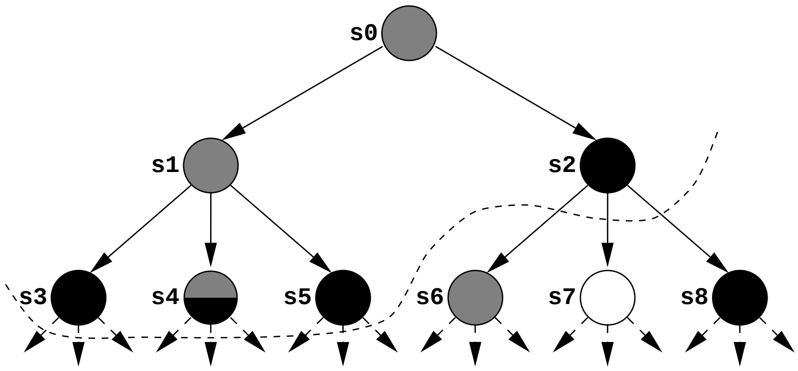

Figure 5: Structure M for $A\,[f_1\,U\,f_2]$.

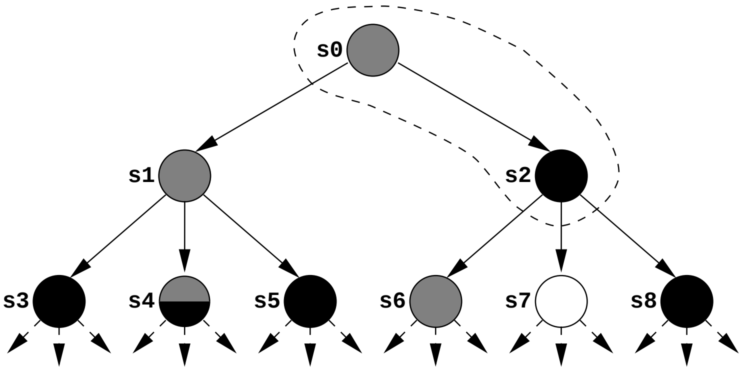

Figure 6: Structure M for $E\,[f_1\,U\,f_2]$.

5. $M, s \models EG(f_1)$ iff for some fullpath $(s_0, s_1, \dots)$ such that $s_0 = s$, $\forall i\,[i \geq 0 \implies M, s_i \models f_1]$

As an example Figure 5 represents a structure M, in which the formula $A\,[f_1\,U\,f_2]$ is *True* in state **s0**. In the figures of this section the states in which f_1 are *True* are black, states in which f_2 *True* are grey, and states in which neither of them are *True* are white. The formula is *True* in Figure 5, because for all fullpaths starting from the state **s0** there is a state in which f_2 is *True*, and f_1 is *True* in all states before it. There is a frontier of states $(\mathbf{s3}, \mathbf{s4}, \mathbf{s5}, \mathbf{s2})$ which prevents the construction of a counterexample for the formula.

As another example Figure 6 represents a structure M, in which the formula $E\,[f_1\,U\,f_2]$ is *True* in state **s0** , because there is a fullpath in which f_2 is *True* in a state, and f_1 is *True* in all states before it. The path $(\mathbf{s0}, \mathbf{s2})$ marked by the dashed outline in Figure 6 is a prefix of such a fullpath. There are also other witnesses for the formula in Figure 6. One such a witness is the path $(\mathbf{s0}, \mathbf{s1}, \mathbf{s3})$.

There can also be witnesses and counterexamples which have an infinite length. One such a witness is a witness for the formula $EG(f)$ in Figure 7. The

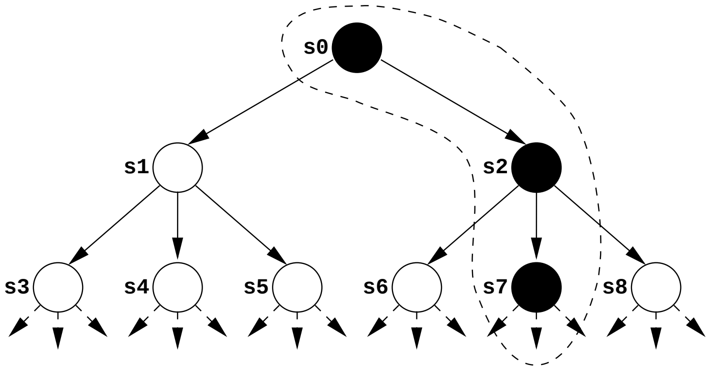

Figure 7: Structure M for $EG(f)$.

fullpath $p = (\mathbf{s0}, \mathbf{s2}, \mathbf{s7}, \dots)$ is a witness for $EG(f)$, iff f is *True* in every state of the fullpath p. Note that if the system we are investigating has finitely many states, then any fullpath of the system must be a path which ends in a loop. In this case we can represented this fullpath by a finite path, which contains all the same states as the fullpath until a loop is reached. Note also that the witness path in Figure 7 is also a counterexample for the formula $A[\mathit{True}\, U \neg f]$. Witnesses and counterexamples are therefore complementary to each other in many cases.

We need some standard definitions to talk about CTL formulas. Here we will define the notions of a set the set of subformulas and the set of of proper subformulas.

Definition 2.6 *Let f be a CTL formula. The set of subformulas $sub(f)$ of f is defined recursively as follows:*

1. *If f is an atomic proposition $p \in AP$, then $sub(f) = \{f\}$.*
2. *If f is of the form $\neg f_1$ or $AX(f_1)$, then $sub(f) = \{f\} \cup sub(f_1)$.*
3. *If f is of the form $f_1 \wedge f_2, A[f_1\, U\, f_2]$ or $E[f_1\, U\, f_2]$, then $sub(f) = \{f\} \cup sub(f_1) \cup sub(f_2)$.*

Definition 2.7 *Let f be a CTL formula. The set of proper subformulas $psub(f)$ of f is defined by*

$$psub(f) = sub(f) \setminus \{f\}$$

We also want to define the length of the formula f to be precise when discussing the complexity of model checking.

Definition 2.8 *Let f be a a CTL formula. The length of the formula f, $length(f)$ is defined by*

$$length(f) = |sub(f)|,$$

where $|sub(f)|$ means the cardinality of the set $sub(f)$.

2.2.1 Counterexamples and Witnesses

In the previous sections we talked informally about counterexamples and witnesses of a CTL formula. We can use the following equality of CTL formulas to define counterexample paths: $A\,[p\,U\,q] \equiv \neg(E\,[\neg q\,U\,\neg p \wedge \neg q] \;\vee\; EG(\neg q))$. This means that for the formula $A\,[p\,U\,q]$ to be *False*, either $E\,[\neg q\,U\,\neg p \wedge \neg q]$ must be *True*, or $EG(\neg q)$ must be *True*. When we combine this equality with the finite state space of the system, we get the following definition:

Definition 2.9 *A path $p_c = s_0, s_1, \ldots, s_n$ is a counterexample for formula $A\,[p\,U\,q]$ from state s_0 of structure M, iff*

$$\exists i\,[i \geq 0 \wedge M, s_i \models (\neg p \wedge \neg q) \wedge \forall j\,[0 \leq j < i \implies M, s_j \models \neg q\,]]\,,\ \textit{or}$$

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

The definition of a witness path is much simpler and comes directly from the CTL formula definition of the existential until formula.

Definition 2.10 *A path $p_w = s_0, s_1, \ldots, s_n$ is a witness for formula $E\,[p\,U\,q]$ from state s_0 of structure M, iff*

$$\exists i\,[i \geq 0 \wedge M, s_i \models q \wedge \forall j\,[0 \leq j < i \implies M, s_j \models p\,]]$$

Next we will present a small verification example to show how CTL is used to specify some interesting properties of the analyzed system.

2.3 A Small Verification Example

In this section we describe a small verification example to demonstrate the usage and also the limitations of the branching time temporal logic CTL. The system we will verify is a small mutual exclusion system. Despite it's small size, it demonstrates many different aspect of modeling and verifying a concurrent system using CTL.

For modeling the system we use a class of high-level Petri-Nets called Predicate/Transition (Pr/T) Nets. The presentation of this formalism in this thesis is very minimal and simplified as we are only interested in the reachability graph the Net model generates. We refer the interested reader to [14] for a more complete description of Pr/T Nets. A Pr/T Net system consists of places, which are graphically noted as circles, and transitions, which are rectangles in the graphical notation. The places are used to represent the local states of the system by markings. A marking of a place consists of multisets of tuples of integers, which are called tokens. The markings of the all places of the net constitute the global state of the system. The Pr/T Net system will also have an initial marking, which is the initial state of the system. The transitions and places of the net are connected with directed edges called arcs to form a connected bipartite-graph. A place is an input place of a transition if there is an arc from the place to the transition, and an output place of transition if there is an arc from the transition to the place.

The transitions represent atomic events. A transition is enabled, iff in all input places of the transition the required tokens specified by the arc annotations are available. These annotations may contain free variables which will become bound to the tokens in the input places. There may be several instances of the same transition enabled if there are different ways to bind the free variables, and such a binding σ defines an instance of the transition and we say that the transition is enabled with mode σ. When an enabled transition fires i.e. an atomic state transition occurs, the bound tuples are removed from the input places, and the annotations on the outgoing arcs of the transition determine what tokens should be added to the markings of the output places of the transition. The reachability graph is generated by recursively firing all enabled transition instances from the initial marking.

A Pr/T-Net of the mutex system is presented in Figure 8, and the PROD Net description can be found from Appendix A. The basic idea of the system is to model a two process mutual exclusion algorithm. The processes are represented by the tuples $\langle 0, NC \rangle$ and $\langle 1, NC \rangle$ in the place P. In the initial states both processes are in their non-critical section, indicated by the constant NC in the tuples. The tuple in place T models a shared variable which is used in the mutex algorithm, and the variable has the initial value indicated by the constant $NONE$.

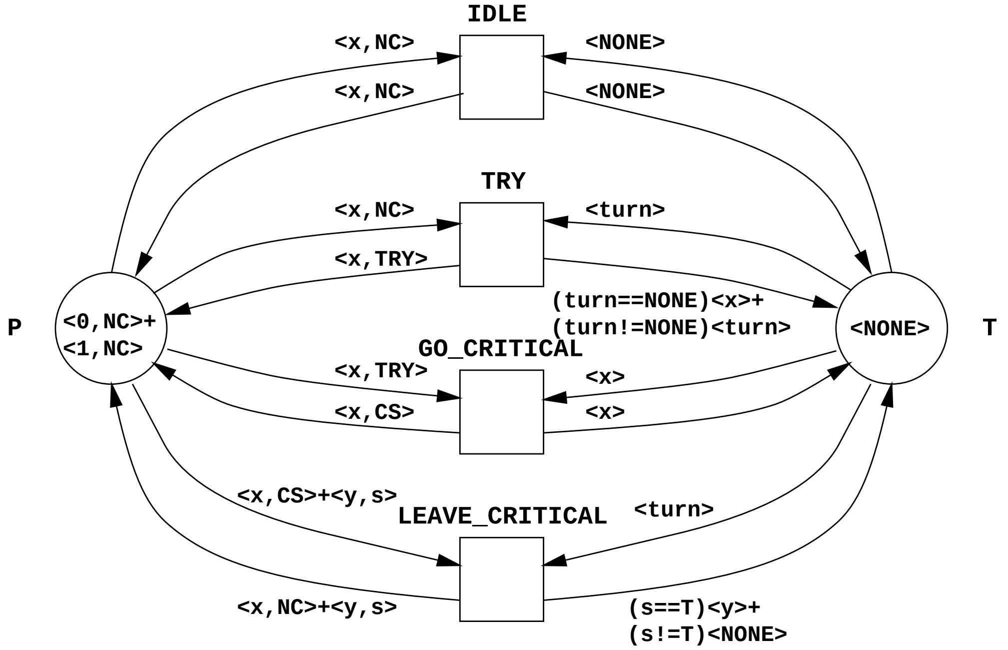

Figure 8: Pr/T-Net Model of the Mutex System.

If neither of the processes want to access the mutual exclusion region, they can stay in their respective non-critical sections. This is modeled by executing the **IDLE**-transition. This transition is enabled in two different modes in the initial marking. In the first mode the transition instance binds the tuple $\langle 0, NC \rangle$ from the place P and the tuple $\langle NONE \rangle$ from the place T. When this instance is fired, the transition atomically first removes both of the bound tuples from the input places, and then uses the arc annotations on the output arcs to generate tuples to the output places. The arc annotations on the output arcs of the transition result in adding a tuple $\langle 0, NC \rangle$ into the place P and a tuple $\langle NONE \rangle$ into the place T. The marking after the transition has been fired is identical to the initial marking, which was to be expected from the transition description. The seconds instance of the transition is symmetric for the other process, and the bound tuples are the tuple $\langle 1, NC \rangle$ from the place P and the tuple $\langle NONE \rangle$ from the place T. All the other transitions behave in a similar way. At any point in time when a process is in it's non-critical section, it can try to enter the critical section by executing the **TRY**-transition. The process which first tries to enter the critical section will get the turn to do so. Also if one of the processes is trying to enter it's critical section, the other process can no longer delay it, because the **IDLE**-transition is no longer enabled. The **GO_CRITICAL**-transition models the process entering a critical section. It is possible only when the process is trying to enter and also it is it's turn to enter the critical section. When a process leaves it's critical section with the **LEAVE_CRITICAL**-transition, it also gives a turn to the other process if it has tried to enter it's critical section. Otherwise it sets turn to the value **NONE**.

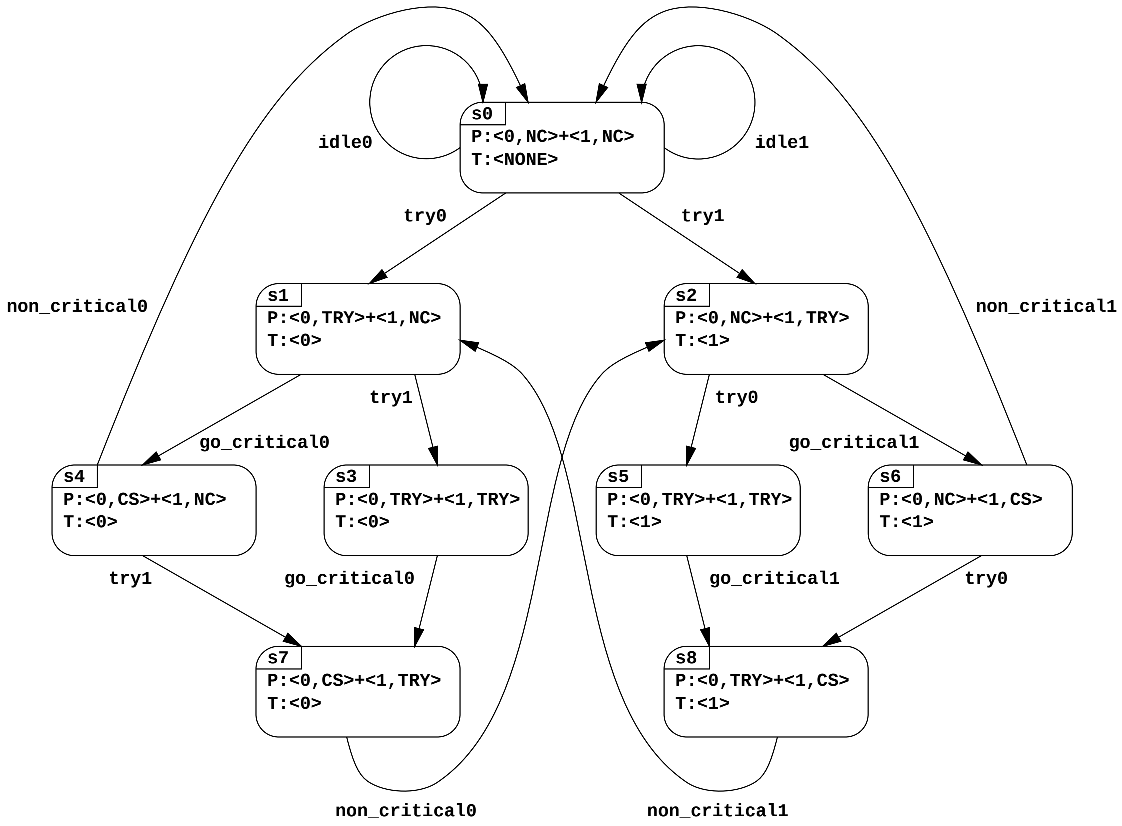

Figure 9: Reachability Graph of the Mutex System.

The graphical representation of the reachability graph of the mutex system can be seen in Figure 9. Note that the transition labels on the arcs are only used for easier readability and are not used in the model checking. The reachability graph can be seen as a more compact presentation of the infinite reachability tree. The tree can be generated by starting from a node in the graph and using the reachability relation to to recursively generate the successor states of each state.

When we have specified a root node of a reachability tree (**s0** in this example), we can verify several properties of the mutex system. We use the notation $P >= \langle 0, CS \rangle$ to express atomic propositions. This atomic proposition is *True*, if the marking of the place P contains the token $\langle 0, CS \rangle$, and *False* otherwise. Some interesting properties of the system are:

1. The processes are never at the same time in the critical section:

$$M, s_0 \models AG(\neg(P >= \langle 0, CS \rangle \wedge P >= \langle 1, CS \rangle))$$

2. If process 0 tries to get into the critical section, it will always eventually get there:

$$M, s_0 \models AG(P >= \langle 0, TRY \rangle \implies AF(P >= \langle 0, CS \rangle))$$

3. The process 0 will always eventually get to critical section. This isn't *True* because the process 0 can remain in the non-critical section forever:

$$M, s_0 \not\models AF(P >= \langle 0, CS\rangle)$$

4. There exists a path starting form the initial state which ends in process 0 getting into it's critical section:

$$M, s_0 \models EF(P >= \langle 0, CS\rangle)$$

5. It is always globally *True* that the system will always eventually reach it's initial state. This isn't *True* because the system can loop forever without returning to the initial state:

$$M, s_0 \not\models AG(AF(P >= \langle 0, NC\rangle \wedge\ P >= \langle 1, NC\rangle))$$

6. It is always globally *True* that the system can always eventually visit it's initial state:

$$M, s_0 \models AG(EF(P >= \langle 0, NC\rangle \wedge\ P >= \langle 1, NC\rangle))$$

7. There is a path starting from the initial state in which the system never leaves the initial state:

$$M, s_0 \models EG(P >= \langle 0, NC\rangle \wedge\ P >= \langle 1, NC\rangle)$$

8. When starting from the initial state, process 0 will never enter it's critical section before going into a trying state:

$$M, s_0 \models A\,[\neg(P >= \langle 0, CS\rangle)\,U\,P >= \langle 0, TRY\rangle]$$

These examples show that quite interesting properties of systems can be expressed with CTL. However CTL has its limitations: For example let's suppose that the model was modified to enable the **IDLE** transition of the process 1 when the process 0 is trying to get into the critical section, and to also include the scheduling of the processes. In this case we would like to ask the following question: If both processes are scheduled infinitely often, will the process 0 eventually reach the critical section if it is trying to get into critical section? This requires the expression of a typical fairness constraint, under which the required property is required to hold. However CTL can't express that some property holds under fairness constraints. If properties under fairness are needed, then the linear time temporal logic LTL, or the full branching time temporal logic CTL* should be used. The complexity of model checking for these logics is exponential in the size of the formula, which is the price to pay for the expressiveness. There is however a way of introducing fairness into the CTL model checking process [7], [8], but also this raises the model checking complexity.

3 CTL Model Checking

In this section we will discuss the model checking problem: Given a structure M and a CTL formula f, determine in which states of M is the CTL formula f *True*. The model checking problem can be formulated in two different ways. Traditionally it is presented as the process in which for each state s of the structure M the truth of the formula f is evaluated. This is called *global model checking* and it gives for each state independently either *True* or *False* as a result. However in many cases we are only interested in the truth value of the formula f in one state s of the model M (in many cases this is the initial state of the system). This is called *local model checking.* A local model checker can sometimes perform less work than a global model checker, because intuitively the local model checker needs to only give a subset of the results required from a global model checker.

Both the worst case time and worst case space complexity of the model checking problem for CTL have been shown to be linear in both the length of the formula f and the size of the structure M [8]. We use $length(f)$ to denote the number of subformulas in the CTL formula f, $|S|$ to denote the number of states in the structure M, and $|R|$ to denote the number of transitions in the structure M. The worst case running times of all of the CTL model checking algorithms discussed in this work require $O(length(f) \times (|S| + |R|))$ running time, with the exception of a naive algorithm, which can require $|S|$ times more time.

3.1 The Subformula Evaluation Order

We need to define in which order the proper subformulas are to be evaluated when evaluating a CTL formula f. There are dependencies between the formulas in the set of all subformulas. We model these by defining a partial order $<_s$ between subformulas in $sub(f)$. As atomic propositions are not proper subformulas of any other CTL formulas, they are the bottom elements of this partial order.

Definition 3.1 *Let f be a CTL formula. We will define the partial order $<_s$ as follows. For all CTL formulas f' and $f'' \in sub(f)$, $(f', f'') \in <_s$ (also marked as $f' <_s f''$), iff $f' \in psub(f'')$ (f' belongs to the set of proper subformulas of f'').*

To demonstrate the use of these notions we proceed with an example. Suppose we need to model check the CTL formula $f = A\,[p\,U\,E\,[q\,U\,\neg p]]$. The set of subformulas of f, $sub(f) = \{p, q, \neg p, E\,[q\,U\,\neg p]\,, A\,[p\,U\,E\,[q\,U\,\neg p]]\}$

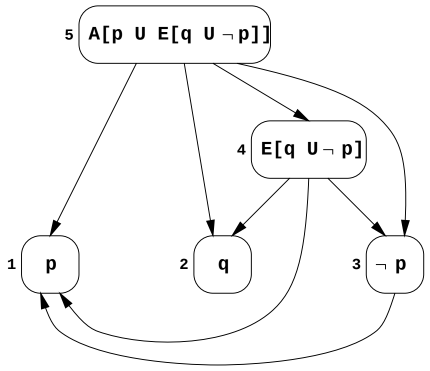

Figure 10: The partial order $<_s$.

The partial order $<_s$ is presented in Figure 10. There is an arrow from a formula f'' to the formula f' when $f' <_s f''$. The arrows between the subformulas form a connected directed acyclic graph, from which we can construct a total order $<_t$ with topological sorting. This can be done by searching the graph in depth-first order starting from the node representing f, and ordering the encountered states in post-order into increasing order. One such acceptable total order is the order is represented in Figure 10 by the numbers beside each subformula.

We can also use this algorithm to construct sequence sub_f, which is a sequence of all the subformulas $\in sub(f)$ in increasing $<_t$ order. We will use the notation $sub_f[i]$, to denote the i:th item of this sequence. The process of constructing the sequence sub_f can be seen as creating a legal evaluation order for the subformulas of f. We'd like to note that there are usually several legal total orders, and we have just presented the most straightforward way of generating one. Actually in the algorithm implementations in this work we use an alternative definition for technical reasons. We will call this partial order $<_{s'}$.

Definition 3.2 *Let f be a CTL formula. For all CTL formulas f' , $f'' \in sub(f)$, $(f', f'') \in <_{s'}$, iff $(f', f'') \in <_s$ or f' is an atomic proposition and f'' is not an atomic proposition .*

We can use the same topological sorting algorithm as before to create the sequence sub_f by using $<_{s'}$ instead of $<_s$. Using this definition all non-atomic formulas are always evaluated after the atomic propositions have been evaluated. The advantages of using this partial order are explained in combination with the model checking algorithm implementations.

As a rule of thumb global model checking algorithms evaluate the formulas $sub_f[i]$ in increasing i order, while local model checking algorithms start from the formula $sub_f[length(f)]$ and proceed in decreasing i order.

To help us define the model checking algorithms, we need a way of accessing the immediate subformulas of a formula, for example the subformulas f_1 and f_2 of the formula $E\,[f_1\,U\,f_2]$. To do this we define two functions $lsub(f)$ and $rsub(f)$, which are defined to be total functions for notational convenience.

Definition 3.3 *Let f be a CTL formula. We define the functions $lsub$: $CTL \rightarrow \{CTL \cup \{null\}\}$ and $rsub : CTL \rightarrow \{CTL \cup \{null\}\}$ by*

1. *If f is an atomic proposition $p \in AP$, then $lsub(f) = null \wedge rsub(f) = null$.*
2. *If f is of the form $\neg f_1$ or $AX(f_1)$, then $lsub(f) = f_1 \wedge rsub(f) = null$.*
3. *If f is of the form $f_1 \wedge f_2, A\,[f_1\,U\,f_2]$ or $E\,[f_1\,U\,f_2]$, then $lsub(f) = f_1 \wedge rsub(f) = f_2$,*

where null is a special place-holder value.

The algorithms presented in this work work by using an index i into the the sequence sub_f to refer to the corresponding subformula $sub_f[i]$. This is also used to specify the left and right subformulas of a formula. To make this more concrete we will use two new subroutines *left(i)* and *right(i)*. These subroutines just return the value indexed by i from the two arrays $left\,[i]$ and $right\,[i]$. These two arrays are initialized by the subroutine *init_formulas*(f). All of these subroutines are represented for completeness in Appendix B, in Figures 48 - 50.

3.2 Global Model Checking

A global model checker evaluates the truth value of a CTL formula f in a *bottom-up* manner. For all subformulas $f' \in sub(f)$ and all states $s \in S$ of the structure M, the truth value of the subformula f' is evaluated in the state s. In the previous section we defined dependencies between the subformulas of f in such a way that if a subformula $f' = sub_f[i]$, then for all it's proper subformulas $f'' \in psub(f') \implies f'' = sub_f[j]$, for some $j < i$. We can use this fact to implement a global model checker. The basic idea is to evaluate the subformulas $sub_f[i]$, $1 \leq i \leq length(f)$ in increasing order of i. Evaluating a subformula f' consists of checking for each state $s \in S$ whether $M, s \models f'$. Because of the way the evaluation sequence sub_f is constructed, when we are evaluating the subformula $sub_f[i]$, we know the truth values of all it's proper subformulas in all states $s \in S$.

If some of these subformula truth values are not needed to evaluate the truth value of the top-level formula f, unnecessary work has been performed. Also if the model checking question the user asked was a local model checking question, then unnecessary results have been obtained in the process. For example if we want to know if some property is *True* in the initial state of the system, the global model checker will check the same property for all states in the system. Another way of seeing global model checking is that it creates new atomic propositions for each subformula $sub_f[i]$. These atomic propositions are then used to model check the subformulas $sub_f[j]$, where $j > i$.

In this section we will first present the general structure of the global model checking algorithms. The presentation includes all of the subroutines common to all the global model checking algorithms. Where the global model checking algorithms differ from each other is the way the existential until and the universal until formulas are model checked. In the next two section we will present two ways of implementing these. The first one is a naive version and the second one is the classical global model checking algorithm EMC.

3.2.1 The General Structure

Here we present the general structure of a global CTL model checking algorithm. We use the same basic algorithm for all the global model checking algorithms we present in this work. We also use the same data structures and subroutines in all the algorithms. The algorithm presented in this section isn't complete, it just contains the routines common to all global model checking algorithms.

The main loop for all global model checking algorithms is presented in Figure 11. It first initializes the subformulas with the subroutine *init_formulas*(f), which was presented in the previous section. Then it starts evaluating all the

```
proc global_model_checker(f) ≡
   init_formulas(f);
   for i := 1 to length(f) do
       foreach s ∈ S do
          reset_label(s, i);                 Initialize subformula i to False
       od;
       label_graph(i);              Evaluate i, access results by labeled(s, i)
   od;
.
```

Figure 11: Main loop of a global model checker.

subformulas in sub_f in increasing order of i>: for each formula its truth value is in all states first set to *False* and then the formula $sub_f[i]$ is evaluated in all the states of the system by the subroutine *label_graph*(i).

We can use a two dimensional bit-array to store all the subformula truth values in all states. We call this bit-array L. It is indexed as $L[s][i]$, where $1 \leq s \leq |S|$, and $1 \leq i \leq length(f)$. We will use the procedure *reset_label*(s, i) to set the bit $L[s][i]$ to *False*, and the procedure *add_label*(s, i) to set the bit $L[s][i]$ to *True*. The procedure *labeled*(s, i) returns *True* or *False* depending on the current value of the bit $L[s][i]$.

How to globally model check one subformula is dependent on the type of the formula. The subroutine *label_graph*(i) chooses between different subroutines depending on the type of the formula, and it is presented in Figure 12. Global model checking of atomic propositions, negations, conjunctions and universal next-state formulas is quite straightforward. The subroutines for all of them loop over all the states in the system and check local conditions or conditions depending only on the immediate successor states to decide the truth value of the formula in each state. The universal next-state formula differs slightly from the others, but all it does is to check that all immediate successor states of the formula indexed by i satisfy the formula indexed by $left(i)$. The universal next-state formula has the worst time complexity of these four routines, and it can be easily seen to be $O(|S| + |R|)$. The subroutine implementations for these routines are in Figures 13 - 16.

```
proc label_graph(i) ≡
  ftype := formula_type(sub_f[i]);
  if ftype = atomic_proposition
    then
        atomic(i);                    Process atomic subformula
  elsif ftype = NOT
      then
          negation(i);                Process negation
  elsif ftype = AND
      then
          conjunction(i);             Process conjunction
  elsif ftype = AX
      then
          ax(i);                      Process formula AX(p)
  elsif ftype = AU
      then
          au(i);                      Process formula A[p U q]
  elsif ftype = EU
      then
          eu(i);                      Process formula E[p U q]
  fi;
.
```

Figure 12: Choosing processing subroutine for different formula types.

```
proc atomic(i) ≡
  foreach s ∈ S do
    if evaluate_proposition(s, i);          Proposition True in state s?
      then
          add_label(s, i);    Change formula labeling to True in s
    fi;
  od;
.
```

Figure 13: Process atomic proposition.

```
proc negation(i) ≡
  foreach s ∈ S do
    if ¬labeled(s, left(i));                         Left formula False?
      then
            add_label(s, i);       Change formula labeling to True in s
    fi;
  od;
.
```

Figure 14: Process negation.

```
proc conjunction(i) ≡
  foreach s ∈ S do
    if labeled(s, left(i)) ∧ labeled(s, right(i))            Both True?
      then
            add_label(s, i);                  Yes. ⟹ Formula True.
    fi;
  od;
.
```

Figure 15: Process conjunction.

```
proc ax(i) ≡
  foreach s ∈ S do
    add_label(s, i);                We suppose formula i is True in s.
    foreach t ∈ successors(s) do
      if ¬labeled(t, left(i))
        then
              reset_label(s, i);       Contradiction! ⟹ i isn't True.
              break;
      fi;
    od;
  od;
.
```

Figure 16: Process universal next-state formula.

3.2.2 A Naive Implementation

To make the global model checking algorithm complete, we also have to process the existential until and universal until -formulas. A naive implementation for doing this is given in [11]. The version presented here looks a bit different but is actually the same basic algorithm. Even this naive implementation has a running time linear in the length of the formula.

The subroutine $au(i)$ in Figure 17 model checks the universal until formulas in all states of the system. We can see from following equivalence of CTL formulas: $A\,[f_1\,U\,f_2] \equiv f_2 \vee (f_1 \wedge AX(A\,[f_1\,U\,f_2]))$, that the universal until formula i is *True* in those states s in which either $right(i)$ is *True*, or $left(i)$ is *True* and i is *True* in all successor states of s.

Now we can first set i to *True* in all states in which $right(i)$ is *True*. Then we can loop over the states to find a state s, in which $left(i)$ is *True* and also i is *True* in all successor states of s. If such a state is found, then in that state i is labeled *True*. If no such a state is found, then a fixpoint has been reached. Because each round of the second part of the algorithm must change the labeling of formula i from *True* to *False* in a some state s, and there are only $|S|$ states, the algorithm will not perform any changes after $|S|$ rounds. This is made more explicit by the counter of the loop on line 7 of the subroutine. The body of this loop takes $O(|S|+|R|)$ time, therefore the whole subroutine needs in the worst case $O(|S| \times (|S|+|R|))$ time.

The handling of the existential until formulas is very similar to universal until formulas. The subroutine $eu(i)$ in Figure 18 model checks the universal until formulas in all states of the system. The CTL formula equivalence we need is: $E\,[f_1\,U\,f_2] \equiv f_2 \vee (f_1 \wedge EX(E\,[f_1\,U\,f_2]))$. We can see from the equivalence that the existential until formula i is *True* in those states s in which either $right(i)$ is *True*, or $left(i)$ is *True* and i is *True* in some successor state t of s. Now we can first set i to *True* in all states in which $right(i)$ is *True*. Then we can loop over the states to find a state s, in which $left(i)$ is *True* and also i is *True* in some successor state of s. If such a state is found, then in that state i is labeled *True*. If no such a state is found, then a fixpoint has been reached.

The global model checking algorithm which uses these naive subroutines to check the existential until and universal until formulas has worst case time complexity of $O(length(f) \times |S| \times (|S|+|R|))$. But the memory requirements for this algorithm are quite small. We needed a bit-array of size $length(f) \times |S|$ bits to hold the subformula truth values in all states. A simple adjacency list representation of the successor relation needs $|R| \times \lceil\log(|S|)\rceil + |S| \times \lceil\log(|R|)\rceil$ bits. The run time book-keeping structures are only a couple of counters and local variables, which need $O(1)$ memory. Therefore to be fair when we compare to the other algorithms presented in this work, we must remember that this this naive algorithm has quite minimal memory requirements.

```
proc au(i) ≡
  foreach s ∈ S do
    if labeled(s, right(i))
      then add_label(s, i);        Label states in which right(i) is True
    fi;
  od;
  for j := 1 to |S| do             Iterate for |S| rounds in the worst case
     changed := False;
     foreach s ∈ S do
       if labeled(s, i) ∨ ¬labeled(s, left(i))
         then continue;              Already labeled or not going to be
       fi;
       all_labeled := True;
       foreach t ∈ successors(s) do
         if ¬labeled(t, i)
           then
                all_labeled := False;
                break;
         fi;
       od;
       if all_labeled
         then                        Local labeling conditions fulfilled
              add_label(s, i);                            Label state
              changed := True;                   No fixpoint found yet
       fi;
     od;
     if ¬changed
       then break;                                  Fixpoint reached
     fi;
  od;
.
```

Figure 17: Process universal until formula.

```
proc eu(i) ≡
  foreach s ∈ S do
    if labeled(s, right(i))
      then add_label(s, i);        Label states in which right(i) is True
    fi;
  od;
  for j := 1 to |S| do             Iterate for |S| rounds in the worst case
     changed := False;
     foreach s ∈ S do
       if labeled(s, i) ∨ ¬labeled(s, left(i))
         then continue;            Already labeled or not going to be
       fi;
       some_labeled := False;
       foreach t ∈ successors(s) do
         if labeled(t, i)
           then
                some_labeled := True;
                break;
         fi;
       od;
       if some_labeled             Local labeling conditions fulfilled
         then
              add_label(s, i);                          Label state
              changed := True;                  No fixpoint found yet
       fi;
     od;
     if ¬changed
       then break;                                 Fixpoint reached
     fi;
  od;
.
```

Figure 18: Process existential until formula.

3.2.3 The EMC Model Checking Algorithm

The EMC algorithm [8] is also a global model checking algorithm, and it has the same general structure as presented in the previous sections. When compared with the naive algorithm presented in the previous section, the EMC algorithm has better worst case time requirements. The time requirements for model checking are reduced to $O(length(f) \times (|S|+|R|))$, which is also the theoretical minimum. On the negative side are increased memory requirements. In this section we present the subroutines for model checking the existential until and universal until formulas, which is where the algorithm differs from other global model checking algorithms. In the original EMC paper only model checking the universal until formula was given in pseudo-code. Here we have constructed the model checking routine also for the existential until formula trying to follow the outline given in the original paper.

The EMC algorithm uses a more efficient algorithm to permit it to globally model check both the universal until and existential until formulas in $O(|S| + |R|)$ time. The subroutines $au(i)$ presented in Figure 19, and $eu(i)$ presented in Figure 21 accomplish this. Both of the subroutines check for each state $s \in S$, whether the formula i has already been evaluated in s. If this is not the case, either the subroutine *check_au*(i, s), or *check_eu*(i, s) is called to evaluate the formula i in state s. Calling this evaluation subroutine might as a side-effect also evaluate the formula i in some other states of the graph. To avoid redoing unnecessary work we need to notice this, and the EMC algorithm uses an additional bit-array *marked* to mark those states in which the formula has already been evaluated. This bit-array has size of $|S|$, and the subroutines *reset_marked*(s) and *add_marked*(s) change the array values to *False* and *True* correspondingly. This bit-array is accessed by the subroutine *marked*(s), which returns the current value of the *marked* array indexed by s.

The subroutine *check_au*(i, s), presented in Figure 20, evaluates the universal until formula i in s, and also in other states whose successor relation it accesses. It uses the *marked*-bitmap to first mark all states it accesses. If the state s was already marked when the subroutine *check_au*(i, s) is called, then the *labeled*-bitmap already contain the truth value of the formula i in state s. We now directly see from the equivalence: $A\,[f_1\,U\,f_2] \equiv f_2 \vee (f_1 \wedge AX(A\,[f_1\,U\,f_2]))$, that if the formula $right(i)$ is *True* in s, then also the formula i will be *True* in s. Also if the formula $right(i)$ isn't *True* in s and also the formula $left(i)$ isn't *True* in s, then i won't be *True* in s either. These two local conditions are checked next in the subroutine. If neither of the two cases apply, then we have the case in which i is *True* in s, iff i is *True* in all successor states of s. We check this case by recursively calling the subroutine *check_au*(i, t) for each successor state t of s until all successors have been checked, or a successor t in which i isn't *True* is found. There is one subtle point we haven't considered yet. The case arises when we have a loop in the reachability graph, in which $left(i)$ is $True$ in all states, and $right(i)$ isn't $True$ in any state. In this case the result of the subroutine *check_au*(i, s) is dependent on the output of another call to the subroutine *check_au*(i, s) for the same state s. What happens in that case is that the second call to the *check_au*(i, s) will return *False*. This is the correct behavior of the algorithm, which we can see from the definition of a counterexample. If we find a loop in the reachability graph, in which $right(i)$ isn't $True$ in any state, then we have found a counterexample. This counterexample path is a witness for the formula i in state s, which is why i isn't $True$ in state s. If we now have checked i in all successor states of s, and haven't found one in which i isn't $True$, then i is labeled $True$ in the state s.

When we talk about time complexity of the algorithm, we can see that the subroutine *check_au*(i, s) takes $O(1)$ time if the state s is already marked. If the state isn't marked, then it is first marked. In that case the time needed by the subroutine is constant except for the loop in which the subroutine is called recursively for all the successor states. The number of these calls and the time needed by their result processing is bounded by $O(|R|)$, when looking at all the calls to *check_au*(i, s) for processing the formula i. The total time complexity of the algorithm $au(i)$ is therefore $O(|S| + |R|)$.

The subroutine *check_eu*(i, s) for model checking the existential until formulas, presented in Figure 22, is quite different from the model checking routine for the universal until formulas. The basic difference between the algorithms is that the existential until model checker uses the predecessor relation during the model checking. The algorithms also differ by the fact that after calling the subroutine *check_eu*(i, s), it is not yet known whether the formula i is $True$ for the state s. This is only known after the model checker has called the subroutine *check_eu*(i, s) for all states $s \in S$. The basic idea of the algorithm can now be explained with the help of the equivalence: $E\,[f_1\,U\,f_2] \equiv f_2 \vee (f_1 \wedge EX(E\,[f_1\,U\,f_2]))$. We can now see that i is $True$ in those states in which $right(i)$ is $True$, or $left(i)$ is $True$ and i is $True$ in some successor state t of s. But this is equivalent to that if i is $True$ in t and there is a predecessor state s in which $left(i)$ is $True$, then i is $True$ also at s. The subroutine *check_eu*(i, s) looks for a state s in which $right(i)$ is found. If such a state is found, i is $True$ in that state and is labeled. After this, it starts the subroutine *label_predecessors*(i, s), presented in Figure 23, which does a recursive depth first search using the predecessor relation. If it encounters a predecessor t in which $left(i)$ is $True$, then i must be $True$ in t and also all predecessors of t must be investigated for states in which i is $True$. The *marked*-bitmap is used to search the predecessor relation of each state only once.

When we talk about time complexity of the algorithm, we can see that the

```
proc au(i) ≡
  foreach s ∈ S do
    reset_marked(s);
  od;
  foreach s ∈ S do
    if ¬marked(s)                         Formula i not evaluated in s?
      then
          check_au(i, s);                 Check formula i in s
    fi;
  od;
.
```

Figure 19: Process universal until formula.

subroutine *check_eu*(i, s) takes $O(1)$ time for it's own processing. The subroutine *label_predecessors*(i, s) takes $O(1)$ time except for the loop over all predecessor states t of s. This loop is executed at most once for each state. The number of the calls to *label_predecessors*(i, s) within a recursive search is bounded by $O(|R|)$. The total time complexity of the algorithm $eu(i)$ is therefore $O(|S| + |R|)$. The global model checking algorithm which uses the EMC subroutines to check the existential until and universal until formulas thus has has worst case time complexity of $O(length(f) \times (|S| + |R|))$.

The memory requirements for this algorithm are as follows. We needed a bit-array of size $(length(f)+1) \times |S|$ bits to hold the subformula truth values and the marked bitmap in all states. A simple adjacency list representation of the successor and predecessor relation needs $2 \times (|R| \times \lceil \log(|S|) \rceil + |S| \times \lceil \log(|R|) \rceil)$ bits. The need for the double reachability relations is a substantial setback, because they usually need to be both present in the memory if there is a mixture of existential until and universal until subformulas in the formula. The predecessor relation can be in theory computed from the successor relation in linear time, but computing it might be awkward if the relation sizes are large. The run time structures are a stack of size $O(|S|)$ for the different recursive calls, which is quite reasonable. The EMC algorithm is quite straightforward to implement, and it has good time complexity. The need for the predecessor relation and the lack of a counterexamples and witnesses facility are drawbacks, which the ALMC algorithm presented in Section 3.3 and the new algorithm presented in Section 4 don't have.

```
proc check_au(i, s) ≡
  if marked(s)                          Formula i evaluated in s or loop found?
    then
          if labeled(s, i)              Labeled gets the right value in both cases!
            then
                return True;
            else
                return False;
          fi;
  fi;
  set_marked(s);                        Mark state s reached and formula evaluated
  if labeled(s, right(i))
    then
          add_label(s, i);                                   right(i) True → i True
          return True;
  elsif ¬labeled(s, left(i))
      then
            return False;                      left(i) ∧ right(i) False → i False
  fi;
  foreach t ∈ successors(s) do          left(i) True, check all successor for i
    if ¬check_au(i, t) then return False; fi
  od;
  add_label(s, i);                                       All successors OK, i True
  return True;
.
```

Figure 20: Process universal until formula for state s.

```
proc eu(i) ≡
  foreach s ∈ S do
    reset_marked(s);
  od;
  foreach s ∈ S do
    if ¬marked(s)                                   Formula i not evaluated in s?
      then
            check_eu(i, s);                                 Check formula i in s
    fi;
  od;
.
```

Figure 21: Process existential until formula.

```
proc check_eu(i, s) ≡
  if labeled(s, right(i))
    then
          add_label(s, i);                      Formula is True directly
          label_predecessors(i, s);     Label states which can reach s
  fi;
.
```

Figure 22: Process existential until formula for state s.

```
proc label_predecessors(i, s) ≡
  set_marked(s);                              Never visit same state twice
  foreach t ∈ predecessors(s) do
    if ¬marked(t) ∧ labeled(t, left(i))        i is True in predecessor ?
      then
            add_label(t, i);                   Yes. → Label predecessor
            label_predecessors(i, t);      Recursively depth-first search
    fi;
  od;
.
```

Figure 23: Propagate label change to predecessor states.

3.3 Local Model Checking

In this section we will discuss an alternative approach to model checking. The model checking question we study here is the local model checking question: Given a structure M, a state $s \in S$ and a CTL formula f, is f true in the state s, i.e. does $M, s \models f$? To answer this question we will take a top-down approach and only evaluate those subformulas that are needed. The subformula truth values in different states are usually needed several times during these recursive evaluations. To keep the time requirements minimal, we will keep track of all the subformulas we have evaluated in each state and also what the results of these evaluations were. A global model checker would in always evaluate all the subformula truth values in all states of the system, and it may thus perform unnecessary work if some of these values are not needed.

The algorithm we will present in this section is based on the ALMC algorithm, published in [22]. The algorithm here is a derivative of that algorithm, and it is one of the contributions of this work. The original ALMC algorithm uses very high-level data-types and some of the operations using these data-types are very difficult to implement using conventional data-types while still maintaining the theoretical complexity of the ALMC algorithm. The algorithm presented here has the same complexity as the original ALMC algorithm does, but it is recast in much more concrete terms, which makes the arguments about the algorithm complexity much clearer. In our opinion it is also easier to understand and implement than the original ALMC algorithm. The time and memory requirements of both algorithms are identical in the worst-case.

The general structure of the algorithm is very similar to the ones presented for global model checking. The part where the algorithm differs most is in the way existential until formulas are model checked. The main loop of the algorithm is presented in Figure 24, and it resembles the global model checking algorithm main loop presented in Figure 11. The main difference is that the ALMC algorithm gets a state s as an input parameter, which is the state in which the formula i should be evaluated, but also the initialization have changed somewhat. The value of a subformula in a state can in ALMC be *Unknown*, *True* or *False*. In initialization all of the subformula values are labeled *Unknown* in all states and also all subformula state pairs are set unmarked. A new depth-first search number array is also initialized to zeroes. The real local model checking algorithm is called for state s and top-level subformula.

We use an array $info[s][i]$ to tell us whether the value of the subformula $sub_f(i)$ is *Unknown*, *True* or *False* in state s. The subroutine $\mathit{info}(s,i)$ returns the current value of the array item $info[s][i]$, and $\mathit{set\_info}(s,i,\mathit{value})$ sets it's value. We also need a bit-array $\mathit{marked}[s][i]$ to keep track of visited states during depth-first searches. We use the subroutines $\mathit{mark\_state}(s,i)$ and $\mathit{unmark\_state}(s,i)$ to manipulate this bit-array, and also the subroutine $\mathit{marked}(s,i)$ to tell whether the state is currently marked or not. A way to save some memory is to code the values of $info[s][i]$ and $\mathit{marked}[s][i]$ into two bits for each state s and subformula i. This is possible because the algorithms never needs to mark a state and subformula whose truth value is already known. Therefore these structure need $|S| \times \mathit{length}(f) \times 2$ bits of memory.

The ALMC algorithm doesn't need the predecessor relation as the EMC algorithm does. A simple adjacency list presentation of the predecessor relation needs $|R| \times \lceil \log(|S|) \rceil + |S| \times \lceil \log(|R|) \rceil$ bits of memory. The ALMC algorithm on the other hand needs an array of depth-first search numbers. The ALMC algorithm has smaller memory requirements, because it only needs $|S| \times \lceil \log(|S|) \rceil$ bits of memory to store one depth-first search number for each state. This is a less memory than that needed by the successor relation, and sometimes even significantly less. Note also that $S < R$ because the structures over which CTL is interpreted are total, i.e. each state has at least one successor state. We use the subroutines $\mathit{set\_dfs\_num}(s,\mathit{num})$ and $\mathit{get\_dfs\_num}(s)$ to set and access these depth-first search numbers.

The subroutine $\mathit{check}(s,i)$ locally model checks the formula $sub_f[i]$ in state s. This subroutine is presented in Figure 25. It looks very similar in structure to the global model checking routine *label_graph*(i) presented in Figure 12. The main difference is that the *check*(s, i) subroutine is only used to evaluate the formula i in state s, when *label_graph*(i) does the evaluation for all states in the graph. Also notice that the first thing *check*(s, i) does is to check whether the formula truth value is already known. If so, the algorithm returns immediately, and uses $O(1)$ time in that case. When we return from the *check*(s, i) procedure, we guarantee that the truth value of the formula i has been evaluated in state s, and can be accessed with the subroutine *info*(s, i).

The Figures 26 - 29 present local model checking subroutines for atomic proposition, negation, conjunction, and universal next-state formulas. They are also very similar to their global model checking counterparts, which were presented in Figures 13 - 16. They use subformulas local to s or to its immediate successor states to determine the truth value of the evaluated formula. Note that calling the subroutine *check*(s, i) can start recursive local model checking evaluations. Obviously there can be at most $length(f)$ evaluation searches active at a given moment, where a single evaluation search corresponds to all active calls to *check*(s, i) with the same subformula i.

The subroutine $au(i, s)$ locally model checks the universal until formula i in state s, and is presented in Figures 30 - 31. When we compare this algorithm to the one for the EMC algorithm presented in Figures 19 - 20, we see that the logic in these routines is almost identical. This might not be obvious when looking at the algorithm pseudo-code, but we can reformulate the ALMC algorithm for model checking the universal until formula to look like the corresponding recursive EMC algorithm. In this section we ourselves try to be faithful to the original representation of the ALMC algorithm.

The major change when comparing to the global model checking algorithm is that we can't be sure if the truth values a needed subformula has been evaluated when we need it. We use the subroutine *check*(s, i) to guarantee that the needed subformula is evaluated before using it. Note that the subroutine *check*_$au(i, s, cst)$ has two different ways of terminating. It can either return normally, or raise the signal $exit\_l$. This signal is caught by the routine $au(i, s)$, which had initialized the current evaluation search. This signal is raised when a counterexample path for formula i has been found, and it causes all recursive calls to *check*_$au(i, s, cst)$ to terminate, and the search to return to the routine $au(i, s)$ for the current value of i. The stack *cst* is passed as a reference variable, and contains the current search path. If a counterexample path was found, the stack *cst* contains a counterexample path, and the states along that path are labeled with *False*.

```
proc almc_scc(f, s) ≡
  init_formulas(f);
  for i := 1 to length(f) do
      foreach t ∈ S do
        unmark_state(t, i);
        set_info(t, i, Unknown);          Initialize formulas to Unknown
      od;
  od;
  foreach t ∈ S do
    set_dfs_num(t, 0);                    Initialize dfs-numbers to 0
  od;
  check(s, length(f));                    Evaluate top-level subformula in state s
  return info(s, length(f));
.
```

Figure 24: Main loop of the ALMC model checker.

```
proc check(s, i) ≡
  if info(s, i) = True ∨ info(s, i) = False
    then return;                          Truth-value known in state s
  fi;
  ftype := formula_type(sub_f[i]);
  if ftype = atomic_proposition
    then
          atomic(i, s);                   Process atomic subformula in state s
  elsif ftype = NOT
      then
            negation(i, s);               Process negation in state s
  elsif ftype = AND
      then
            conjunction(i, s);            Process conjunction in state s
  elsif ftype = AX
      then
            ax(i, s);                     Process formula AX(p) in state s
  elsif ftype = AU
      then
            au(i, s);                     Process formula A[p U q] in state s
  elsif ftype = EU
      then
            eu(i, s);                     Process formula E[p U q] in state s
  fi;
.
```

Figure 25: Choosing processing subroutine for different formula types.

```
proc atomic(i, s) ≡
  if evaluate_proposition(s, i)                 Proposition True in state s?
    then
          set_info(s, i, True);      Change info from Unknown to True
     else
          set_info(s, i, False);     Change info from Unknown to False
  fi;
.
```

Figure 26: Process atomic proposition in state s.

```
proc negation(i, s) ≡
  check(s, left(i));                            Check left(i) in state s
  if info(s, left(i)) = True          info(s, i) ≠ Unknown after check(s, i)
    then
          set_info(s, i, False);
     else
          set_info(s, i, True);
  fi;
.
```

Figure 27: Process negation in state s.

```
proc conjunction(i, s) ≡
  check(s, left(i));                            Check left(i) in state s
  if info(s, left(i)) = True
    then
          check(s, right(i));                   Check right(i) in state s
          if info(s, right(i)) = True
            then
                  set_info(s, i, True);         Both True ⟹ i True
                  return;
          fi;
  fi;
  set_info(s, i, False);
.
```

Figure 28: Process conjunction in state s.

```
proc ax(i, s) ≡
  foreach t ∈ successors(s) do
    check(t, left(i));                         Check right(i) in state t
    if info(t, left(i)) = False
      then
            set_info(s, i, False);
            return;
    fi;
  od;
  set_info(s, i, True);        left(i) True in all successors ⟹ i True
.
```

Figure 29: Process universal next-state formula in state s.

```
proc au(i, s) ≡
  cst := empty_stack;                       Store search path in stack cst
  try begin
          check_au(i, s, cst);              Search for a counterexample
  end;
  catch exit_l begin              check_au() uses signal exit_l to abort
                     ;            Search aborted by a signal, nothing to do
  end;
                                  Iff the search was aborted cst is not empty
  while ¬empty(cst) do                The stack cst is the counterexample
          t := pop(cst);           State t belongs to the counterexample
          unmark_state(t, i);
          set_info(t, i, False);              Set info to False in state t
  od;
.
```

Figure 30: Subroutine au(i, s).

```
proc check_au(i, s, cst) ≡                    // Search for a counterexample containing s
  tmp := info(s, i);
  if tmp = False
    then
        throw exit_l;                         // i False ⟹ Counterexample found
  elsif tmp = True
      then
          return;                             // i True ⟹ No counterexample can be found
  elsif tmp = Unknown
      then
          check(s, right(i));
          if info(s, right(i)) = True
            then
                unmark_state(s, i);           // i True ⟹
                set_info(s, i, True);         // No counterexample
                return ;                      // can contain state s
          fi;
          check(s, left(i));
          if info(s, left(i)) = False
            then                              // i False ⟹
                push_stack(cst, s);           // Counterexample found
                throw exit_l;                 // Abort search
          fi;
          mark_state(s, i);                   // Marking used for loop detection
          push_stack(cst, s);
          foreach t ∈ successors(s) do
            if ¬marked(t, i)                  // Loop found?
              then                            // No, continue searching
                  check_au(i, t, cst);        // Search children
               else                           // Yes, counterexample found
                  throw exit_l;               // Abort search
            fi;
          od;
          pop_stack(cst);
          unmark_state(s, i);                 // No counterexample found ⟹
          set_info(s, i, True);               // i True
  fi;
.
```

Figure 31: Subroutine *check_au*(i, s, cst).

The algorithm for model checking the existential until formulas differs a lot from the EMC algorithm, and it is the most interesting part of the ALMC algorithm. It is also the most difficult part of the algorithm. The algorithm is based on doing two consecutive depth-first searches in the reachability graph for each subformula evaluation search. We will first explain a simple approach which will be then made more elaborate to match the time complexity of the EMC algorithm.

The basic idea of the simple algorithm is to search for a witness path for an existentially quantified until formula using depth-first search starting from state s. It is quite obvious from the definition of a witness path, that a witness path for formula i always ends in a state t in which formula $right(i)$ is *True*. Therefore if we can reach a witness path end state t from s using only states in which $left(i)$ is *True*, we have found a witness for formula i in state s. We can also stop the search for a witness path end state prematurely, if we find a state t in which the formula i has been evaluated *True* in a previous run. In this case we know that there is a witness path starting from the state t and ending in some state t', and the witness path for s can be constructed by the concatenation of the two paths from s to t and from t to t'. We will call the state t, in which the search for witness stopped successfully the goal state. If during the search we find a state r in which i is *False*, or $right(i)$ is *False* and $left(i)$ is *False*, we can conclude that it can't belong to any witness path for i. Therefore the successors of r need not be investigated, and formula i is labeled *False* in r. If a goal state was found, the formula i must be *True* in state s. If no goal state was found, the formula i is *False* in state s.

The problem with the procedure described above is that it only evaluates the truth value of the formula i in one state s. Potentially also all other states of the reachability graph may have been visited with recursive calls to the $check\_eu(i, s)$ subroutine. This obviously makes such a simple algorithm have time requirements quadratic in the number of states of the system, which is quite unacceptable. To get back the linearity of the algorithm we have to evaluate the truth value of the formula i for all states which the current evaluation search has visited. In order to do this we need a much more complicated algorithm. We can easily see from the definition of a witness path that also all other states found during the evaluation search, from which the goal state can be reached using only states in which $left(i)$ is true, have a witness for formula i, and should be labeled *True*.

The algorithm for model checking the existential until formulas can now be divided into two depth-first searches. The first search looks for a goal state using states in which $left(i)$ is *True*, and also performs some additional bookkeeping. The simple algorithm described above is the basis of the first search. We have modified it to keep track of all states which have been reached during the current search and which can potentially belong to some witness path, i.e.

states in which truth value of i is *Unknown*, $right(i)$ is *False*, and $left(i)$ is *True*. The stack *cst* is used in the algorithm to store all these potential witness states. If no goal state could be found, we can't have a witness path for these states either, so formula i is labeled *False* in all states stored in the stack *cst*. This first search of the algorithm is implemented by the subroutine $eu(i, s)$ presented in Figure 32 and the subroutine $check\_eu(i, s, cst)$ presented in the figure 33.

We only need the second search of the algorithm if we found a goal state. The task of the second search is to decide for all states in the stack *cst*, whether they belong to some witness path or not. To do this, we need another depth-first search using exactly the same search order as the first search, but using only potential witness states. These are exactly the states on the stack *cst*, and they are also the states left *Marked* by the first search. When the search is done in exactly the same order, we know that we will eventually reach the goal state. Our aim is to find exactly those potential witness states, from which the goal state can be reached using only other potential witness states. Because we can construct a witness path for i from all such states, the formula i is *True* in them. We can't form a witness path for any other potential witness states, and they are therefore labeled *False*. We use a modified version of an algorithm by R.E. Tarjan [19] to do this. This algorithm finds the maximal strongly connected components of a graph in linear time to the size of the graph. The original ALMC algorithm used another algorithm, which used much more complex data-types and operations for them.

What we do is to search for the maximal strongly connected components in a graph consisting of only the potential witness states until the goal state is reached. Tarjan's algorithm has the property that all states from which the currently visited state can be reached are on the stack *fst*. If we find a maximal strongly connected component before reaching the goal state, there is no way of reaching the goal state from any of the states belonging to that maximal strongly connected component. Therefore we can't construct a witness path for any of these states and the formula i is labeled *False* in them. When we finally encounter the goal state, we abort the execution of Tarjan's algorithm. At that moment of the execution, all states which are stored on the stack *fst* are such that the goal state can be reached from them using only potential witness states. Therefore all states on the stack *fst* are labeled *True* for formula i. The subroutines $label\_eu\_scc(i, s, goal)$ presented in Figure 34, and $eval\_eu\_sccs(i, s, goal, fst)$ presented in Figure 35 implement the second search of the algorithm. Note that the subroutine $label\_eu\_scc(i, s, goal)$ is called by the $eu(i, s)$ subroutine only in the case a goal state was found.

The division into two searches needs to be done to minimize memory requirements. This causes some additional complexity into the algorithm, but it cuts down the memory requirements for depth-first search numbers from

$length(f) \times |S| \times \lceil \log(|S|) \rceil$ to $|S| \times \lceil \log(|S|) \rceil$ bits of memory. The problems caused by this choice is the considerable added complexity of the algorithm, especially in guaranteeing that the two depth-first searches search the reachability graph in exactly the same order as far as potential witness states are concerned. Both of the searches are take linear time in the size of the reachability graph, so they don't increase the worst-case time complexity of the algorithm.

The ALMC algorithm is a local model checking algorithm, whose worst-case time complexity is $O(length(f) \times (|S| + |R|))$, which is the same as the best global model checking algorithms. This comparison isn't quite fair, because the global model checkers always realize this worst-case running time, while a local model checker can perform faster in some cases. The local model checker has to do more book-keeping to accomplish this, so it has bigger book-keeping overheads. The number of bits needed by the formulas and book-keeping bitmaps is $|S| \times length(f) \times 2$ bits, which is almost twice as much as the global model checking algorithms. The ALMC algorithm doesn't need the predecessor relation, which saves $|R| \times \lceil \log(|S|) \rceil + |S| \times \lceil \log(|R|) \rceil$ bits of memory when compared to the EMC algorithm. However the ALMC algorithm needs depth-first search numbers instead, which require $|S| \times \lceil \log(|S|) \rceil$ bits of memory. This is in many cases much less, because many reachability graphs have much more arrows than states, and never less arrows than states. A definite drawback is the amount of stack space needed by the recursive depth-first search routines. This is harder to account for, but it can be seen to be bounded by $O(length(f) \times |S|)$, which is an order of $length(f)$ more than the global model checking routines. We must therefore conclude that the worst case memory requirements of the local model checking algorithm are worse than those of the competing global model checking algorithms. Also the implementation effort needed is considerably more. It remains to be seen whether the local model checking algorithm can make up for this difference by the lazy evaluation of subformulas in practical model checking use.

```
proc eu(i, s) ≡
  set_goal(i, −1);                      Initialize witness path end state to -1
  cst := empty_stack;                   Store potential witness states in cst
  try begin
        check_eu(i, s, cst);            Search for a witness
  end;
  catch exit_p begin                    check_eu() uses signal exit_p to abort
                ;                       Search aborted by a signal, nothing to do
  end;
                                        Iff the search was aborted goal(i) ≠ −1
  if goal(i) = −1
    then                                No witness found
        while ¬empty(cst) do
              t := pop(cst);            t was a potential witness state
              unmark_state(t, i);
              set_info(t, i, False);    i is False in t
        od;
     else                               Witness found
        label_eu_scc(s, i, goal(i));    Label potential witness states
  fi;
.
```

Figure 32: Subroutine *eu*(i, s).

```
proc check_eu(i, s, cst) ≡                    Search for a witness containing s
  tmp := info(s, i);
  if tmp = False
    then
        return;                               i False ⟹ No witness can be found
  elsif tmp = True
      then
          set_goal(i, s);                     i True ⟹ Witness found
          throw exit_p;                       Abort search
  elsif tmp = Unknown
      then
          check(s, right(i));
          if info(s, right(i)) = True
            then                              i True ⟹
                set_goal(i, s);               Witness found
                throw exit_p;                 Abort search
          fi;
          check(s, left(i));
          if info(s, left(i)) = False
            then                              i False ⟹
                unmark_state(s, i);           No witness contains s
                set_info(s, i, False);
                return;                       Search successors later
          fi;
          mark_state(s, i);                   Mark state visited
          push_stack(cst, s);                 Add s to potential witness states
          foreach t ∈ successors(s) do        Use a fixed order
            if ¬marked(t, i)
              then
                  check_eu(i, t, cst);        Search t for a witness
            fi;
          od;
  fi;
.
```

Figure 33: Subroutine *check_eu*(i, s, cst).

```
proc label_eu_scc(i, s, goal) ≡
  dfs := 0;                                          Depth-first search number
  fst := empty_stack;                       Use stack fst in MSCC detection
  try begin
        eval_eu_sccs(i, s, goal, fst);      Label potential witness states
                                             from which goal can be reached
  end;
  catch exit_n begin          eval_eu_sccs() uses signal exit_n to abort
                  ;
  end;
  while ¬empty(fst) do                  Stack fst contains dirtied states
        t := pop_stack(fst);
        set_dfs_num(t, 0);                        Cleanup dfs-number array
  od
.
```

Figure 34: Subroutine *label_eu_scc*(i, s, goal).

```
proc eval_eu_sccs(i, s, goal, fst) ≡                  Label states which can reach goal
  unmark_state(s, i);
  set_info(s, i, True);                               Suppose that s belongs to a witness
  if s = goal
    then
        throw exit_n;                                 Goal state found, abort
  fi;
  dfs := dfs + 1;
  set_dfs_num(s, dfs);
  min := dfs;                                         Initialize minimum reachable dfs-number
  push_stack(fst, s);                                 Store state for MSCC detection
  foreach t ∈ successors(s) do                        Visit successors in same
                                                      fixed order as check_eu()
    m := get_dfs_num(t);
    if m = 0 ∧ marked(t, i)                           Use exactly the same order as check_eu()
      then
          m := eval_eu_sccs(i, t, goal, fst);         dfs-search for goal
    fi;
    if m > 0 ∧ m < min                                A state with smaller dfs-number found?
      then
          min := m;                                   Yes, update minimum dfs-number
    fi;
  od;
  if get_dfs_num(s) = min                             MSCC root state found?
    then                                              Yes, and goal not found either ⟹
        repeat                                        Top of stack fst contains a MSCC,
                                                      from which goal can not be reached
            t := pop_stack(fst);                      t is a state in MSCC
            set_dfs_num(t, 0);                        Clean up dfs-number
            set_info(t, i, False);                    Change info to False
        until t = s;
  fi;
  return min;
.
```

Figure 35: Subroutine *eval_eu_sccs*(i, s, goal, fst).

4 A New Global Model Checking Algorithm

The initial goal of this work was to select a CTL model checking algorithm to be implemented in the PROD reachability analyzer tool set [21]. However based on the analysis of the previous sections we can see that no clear winner between the existing algorithms can be found: all algorithms had different strengths. On the other hand it was discovered that a new algorithm having a combination of the good features of the algorithms could be designed. In this section we will present this new global model checking algorithm, which is based on the algorithms EMC and ALMC presented in Sections 3.2.3 and 3.3 respectively. The main ideas of the algorithm in this section were first presented in the paper [17].

The design goals for our algorithm were to match the best worst case time complexity, and to achieve minimal worst-case memory requirements. Moreover we wanted to deliver an algorithm with a counterexamples and witnesses facility. Without this facility, when asking whether a formula is *True* at a given state, the user will get only a *True* or *False* as the result. But by using this facility, if the result of a universal formula was *False* or the result of an existential formula was *True*, the model checker can give the user a counterexample or a witness path respectively. This can be very valuable when trying to find the cause of unexpected behaviors of the system we are model checking. For example if the model checker can give a counterexample path which forces an universal until formula to be *False*, then the user can more easily figure out why the requirement specified by the formula was not met. Note that the counterexamples and witnesses facility only makes sense in the case the model checking question the user is interested in is a local model checking question i.e. checking whether a formula is *True* in a given state. This is usually the way the user interacts with the PROD model checker.

When evaluating the existing model checking algorithms, the global model checking algorithm EMC was found simple to implement, and it has only small runtime book-keeping memory requirements. However the EMC algorithm requires the predecessor relation to be precomputed and stored in the memory. While creating this relation is quite straightforward and can be implemented in linear running time, it is still somewhat wasteful to store it in memory. The major factor for deciding against not implementing the EMC model checker was the lack of counterexamples and witnesses facility. This feature could be added in the form of another algorithm to be run after the model checking procedure has finished. Taking this approach would need the definition of a separate algorithm for the task, which seems somewhat unnecessary.

The local model checking algorithm ALMC was found to be quite tricky to implement. The correctness of the ALMC algorithm relies on having two depth-first search passes visiting a subset of the reachability graph in exactly the same order. In the previous section we presented a variant of the ALMC algorithm recast with more simple data-types. While this was a difficult task, the algorithm can from this representation be easily implemented. The main drawback of using the ALMC algorithm is the amount of runtime book-keeping memory overhead required in the worst case. For example the search stack memory requirements could in the worst case be in the order of $length(f)$ more than that of a global model checking algorithm. It remains to be seen whether the lazy evaluation of subformulas can make up for the larger runtime book-keeping overhead of a local model checker. The ALMC algorithm can be very easily modified to provide a counterexamples and witnesses facility.

Our new global model checking algorithm combines the good features of the EMC and ALMC algorithms. It has quite small runtime book-keeping memory overheads, as would be expected from a global model checking algorithm. It doesn't need the predecessor relation, and in this respect uses less memory than the EMC algorithm. Another major feature is the availability of a counterexamples and witnesses facility. The algorithm is also somewhat optimized from the implementation perspective, so that quite a straightforward implementation can still have good performance characteristics.

The basic structure of the algorithm is the same as for other global model checking algorithms, which was presented in Section 3.2.1, in Figures 11 - 16. The difference when comparing to the other global model checking algorithms is again the way the existential until and universal until formulas are model checked. In this section we will first describe the model checking of the universal until formulas, followed by the much more demanding part of model checking the existential until formulas.

4.1 Model Checking the Universal Until Formulas

Model checking the universal until formulas in our new algorithm is quite similar to the subroutines presented in EMC and ALMC algorithms. In this chapter the algorithm is presented in a form that makes the memory requirements of the algorithm explicit. This also enables a straightforward implementation of the algorithm with good performance characteristics.

The new algorithm for model checking the universal until formulas is presented in Figure 37. The main difference to other global model checking algorithms is the addition of the counterexamples and witnesses facility. If the global variable *check_global* is set to *True*, the subroutine will calculate the truth value of the formula in all states of the system, just like a normal global model checker would. If global model checking is not wanted, the counterexamples and witnesses facility is used instead. In this case we need to know the state in which the top-level formula is to be evaluated. We use the global variable *evaluation_state* to pass this information to our subroutine. The subroutine

$check\_au(i, evaluation\_state, st)$ is then used to search for a counterexample path for formula i starting from state $evaluation\_state$. If a counterexample path exists, it is left on the stack st, which was passed as a reference to the subroutine. The counterexample path can now be retrieved from the stack st by the procedure $process\_counterexample(st)$.

One important change in our algorithm is the way it initializes the *marked* and *labeled* bitmaps. This subroutine is called $init\_i\_and\_marked(i)$, and is presented in Figure 38. We can see directly from the definition of an universal until formula that in many cases the truth value of the formula i in state s only depends on the truth values of the formulas $left(i)$ and $right(i)$ in the same state. We have enumerated the four different cases in Figure 36.

$labeled(s, left(i))$	$labeled(s, right(i))$	$marked(s)$	$labeled(s, i)$
False	*False*	*True*	*False*
False	*True*	*True*	*True*
True	*False*	*False*	*False*
True	*True*	*True*	*True*

Figure 36: Search bitmap initialization table.

Only those states in which *marked* is *False* need to be processed any further. There is only one such a row in Figure 36. In all other cases the truth value of the formula has been determined without looking at the successor states of s. A very efficient version of the initialization routine can be made by using boolean operations on bitmaps to initialize the *marked* and *labeled* bitmaps.

4.1.1 Searching for a Counterexample

The logic behind model checking the universal until formula i in a given state s is to try to find a counterexample path for the universal until formula i starting from state s. The subroutine $check\_au(i, s, st)$ accomplishes just this, and is presented in the Figure 39. This subroutine is used in two different ways. It is used as a subroutine for a global model checker which evaluates the truth value of the formula i in state s. It is also used as a counterexample generator when counterexamples are required. For this we need is a reformulation of a counterexample path in terms of the search bitmaps *marked* and *labeled*. We can use the table in Figure 36 to substitute values of search bitmaps into the counterexample definition:

```
proc au(i) ≡                                    Model check universal until formulas
  init_i_and_marked(i);                   Initialize i and marked from subformulas
  if i ≠ length(f) ∨ check_global                          Global checking wanted?
    then
        foreach s ∈ S do                              Evaluate formula in all states
          if ¬marked(s)                                     Formula evaluated in s?
            then
                st := empty_stack;
                check_au(i, s, st);                                   Evaluate i in s
          fi;
        od;
     else                                                Counterexample wanted for i
        st := empty_stack;                                    from evaluation_state
        check_au(i, evaluation_state, st);                            Check locally
        if ¬labeled(evaluation_state, i);
          then                                                 Counterexample found
              process_counterexample(st);                                Output it
        fi;
  fi;
.
```

Figure 37: Subroutine for model checking the universal until formula.

```
proc init_i_and_marked(i) ≡                             Initialize search bitmaps
  foreach s ∈ S do                         You should use boolean bitmap operations
    if ¬labeled(s, left(i)) ∨ labeled(s, right(i))
      then
          set_marked(s);
       else
          reset_marked(s);
    fi;
    if labeled(s, right(i))
      then
          add_label(s, i);
       else
          reset_label(s, i);
    fi;
  od;
.
```

Figure 38: Subroutine to initialize i and *marked*.

Definition 4.1 *There is a counterexample path for formula $A[p\,U\,q]$ from state s_0 of structure M, iff there is a path $p_c = s_0, s_1, \ldots, s_n$ such that*

$$\exists i[i \geq 0 \wedge M, s_i \models (marked \wedge \neg labeled) \wedge \\ \forall j[0 \leq j < i \implies M, s_j \models (\neg marked \wedge \neg labeled)]], \textit{ or}$$

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

This definition gives us a direct definition of a search algorithm for a counterexample path from state s_0. We just initialize the search bitmaps and then search for a path satisfying either of the two cases in the definition above. Note also that the definition is formulated in a way which makes it hold even after the truth value of the formula has been evaluated is some states. If the formula was evaluated to *False* in some state s_n i.e. it became marked and unlabeled, then we can construct a counterexample path beginning from it. If we now reach s_n from s_0 using only unmarked and unlabeled states, then there is a counterexample path from s_0. This path is the concatenation of the path from s_0 to s_n and the counterexample path from s_n. In the other case if the formula was evaluated to *True* in some state s_n i.e. it became marked and labeled, then we can't construct a counterexample containing the state s_n and paths containing s_n need not be considered as potential counterexample paths.

The subroutine *check_au*(i, s, st) presented in Figure 39 is a counterexample search algorithm which uses the definition above. The parameter *st* is passed by reference to the subroutine. The subroutine is presented in a non-recursive form, and it is a depth-first search algorithm. It terminates as soon as a counterexample path for the formula i has been found from state s, or no counterexample path for the formula can be found. The search only has to consider those states in which the formula truth value is not yet known, i.e. states which are unmarked and therefore also unlabeled before calling the subroutine. We also need the values of the search bitmaps of the immediate successor states of these states.

The search starts from an unmarked state s, and the algorithm marks unmarked states the first time they are encountered. This gives us a way of detecting loops during the search. Another reason why encountered states are marked is the fact that the formula i is evaluated in all encountered states when the search algorithm terminates.

When we first enter a state, we check whether the concatenation of the current search path stored on stack *st* and the entered state t is a counterexample path. If the entered state t is a marked and unlabeled state, then a counterexample path has been found. We will describe why this is the case and how it can be seen from the counterexample path definition above. The algorithm search path st always only contains states which were initially unmarked and unlabeled. The counterexample definition has two different types of counterexample paths. In the first case the entered state t was marked and unlabeled before the search, in which case we have found a counterexample of the first form. In the second case the state t was unmarked and unlabeled before the search started, but it is marked and not yet labeled by the search, i.e. the state exists on the search stack st and we have found a loop of the second form. When we analyze this subtle detail, we can see that the two different case can be handled identically by the algorithm.

We also need to handle the cases in which no counterexample path was directly found. If the entered state t is marked and labeled then the formula is *True* in t, and no counterexample can be constructed which contains the state t. If the entered state t was not marked, then we will have to continue the search for a counterexample path in depth-first order recursively for all successor states. If no counterexample path was found after all successors have been visited, then the formula i is *True* in the state t, and also no counterexample path can contain the state t.

The logic behind the counterexample search algorithm is actually very similar to the model checking subroutines of the EMC and ALMC algorithms. The major difference is that the algorithm uses the initialization routine to evaluate those formulas, for which the successor state subformula truth values are not needed. This initialization can be implemented very efficiently. In this way the counterexample search algorithm is simplified, which enables a faster implementation of the model checking procedure.

4.2 Model Checking the Existential Until Formulas

Model checking the existential until formulas in our new algorithm has quite similar logic to the corresponding algorithm for the ALMC model checker. The ALMC algorithm has to use a two-pass depth-first search algorithm to model check the existential until formulas for memory requirement reasons. A variant with one pass search would need one set of depth-first search numbers for each recursively evaluated subformula, which would require $length(f)$ times more memory allocated for the depth-first search numbers. Because our algorithm is a global model checking algorithm, there are no recursive subformula evaluations, and only one set of depth-first search numbers suffices. This enables us to use a one pass depth-first search algorithm, which does in one pass everything which the combination of the two passes of the ALMC algorithm does.

```
proc check_au(i, s, st) ≡                                   Search for counterexample for i from s
  while True                                                Non-recursive depth-first search
        do
        while True                                          Visit new states, look for counterexample
              do
              if ¬marked(s)
                then
                      add_marked(s);                        New state found, mark it.
                      succ_num := 0;                        Search successors
                      push(⟨s, succ_num⟩, st);              Save search state
                      s := successor(s, succ_num);          Visit successor 0
                 else
                      if ¬labeled(s, i)                     Loop or illegal state found?
                        then
                              push(⟨s, 0⟩, st);             Counterexample
                              return ;                      path found, i False
                      fi;
                      break;                                s is old, backtrack by exiting loop
              fi;
        od;
        while True                                          Backtracking search for counterexample
              do
              if empty(st)                                  Search complete?
                then
                      return ;                              No counterexample found, i True
              fi;
              ⟨s, succ_num⟩ := pop(st);                     Backtrack to parent state
              succ_num := succ_num + 1;                     Try next successor
              if has_successor(s, succ_num)
                then
                      push(⟨s, succ_num⟩, st);              Save search state
                      s := successor(s, succ_num);          Next succ
                      break;                                Visit new succ by exiting loop
                 else
                      add_label(s, i);                      Succs visited, i True
              fi;
        od;
  od;
.
```

Figure 39: Subroutine *check_au*(i, s, st).

The new algorithm for model checking the existential until formulas is presented in Figure 40. Again the main difference to other global model checking algorithms is the addition of the counterexamples and witnesses facility, which is used in the same fashion as it was used with the universal until formulas. If the global variable *check_global* is set to *True*, the subroutine will calculate the truth value of the formula in all states of the system, just like a normal global model checker would. If global model checking is not wanted, the counterexamples and witnesses facility is used instead. In this case we need to know the state in which the top-level formula is to be evaluated. We use the global variable *evaluation_state* to pass this information to our subroutine. The subroutine $check\_eu(i, evaluation\_state, st, fst)$ is then used to search for a witness path for formula i starting from state *evaluation_state*. If a witness path exists, it is left on the stack *st*, which was passed as a reference to the subroutine. The witness path can now be retrieved from the stack *st* by the procedure $process\_witness(st)$.

The initializations needed differ quite a lot from other global model checking algorithms. We need to initialize the search bitmaps *marked* and *labeled*. We can again look at the definition of an existential until formula, to see in which cases the truth value of the formula can be decided without consulting any successor states, and what the truth formula values should be in those cases. When we construct the needed table and compare it to the one constructed for the universal until formulas, we see that it is by coincidence identical to the one presented earlier in Figure 36. Therefore we can use the same subroutine to perform this initialization. It is called $init\_i\_and\_marked(i)$ and was presented in Figure 38. The algorithm also needs an array of depth-first search numbers *min*, one for each state, and an additional bitmap *modified*. These structures are initialized by the subroutine $clear\_min\_and\_modified()$, which is presented in Figure 41.

4.2.1 Searching for a Witness

The logic behind model checking the existential until formula i in a given state s is to try to find a witness path for the existential until formula i starting from the state s. The subroutine $check\_eu(i, s, st, fst)$ accomplishes just this, and is presented in Figure 42. This subroutine is used in two different ways. It is used as a subroutine for a global model checker which evaluates the truth value of the formula i in state s. It is also used as a witness generator when witnesses are required. For this we need to reformulate the definition of a witness path in terms of the search bitmaps *marked* and *labeled*. We can use the initialization table in Figure 36 to substitute values of search bitmaps *marked* and *labeled* into the witness definition:

```
proc eu(i) ≡                                   Model check existential until formulas
  fst := empty_stack;                                   Initialize father state stack
  init_i_and_marked(i);                  Initialize i and marked from subformulas
  clear_min_and_modified();                             Initialize min and modified
  if i ≠ length(f) ∨ check_global                    Global model checking wanted?
    then
         foreach s ∈ S do                             Evaluate formula in all states
           if ¬marked(s)
             then
                  st := empty_stack;
                  check_eu(i, s, st, fst);                            Evaluate i in s
           fi;
         od;
     else                                                      Witness wanted for i
         st := empty_stack;                                 from evaluation_state
         check_eu(i, evaluation_state, st, fst);                     Check locally
         if labeled(evaluation_state, i)
           then                                                      Witness found
                process_witness(st);                                    Output it
         fi;
  fi;
.
```

Figure 40: Subroutine for model checking the existential until formula.

```
proc clear_min_and_modified() ≡                         Init dfs-number handling
  foreach s ∈ S do
    set_min(s, 0);
    reset_modified(s);
  od;
.
```

Figure 41: Subroutine to initialize *min* and *modified*.

Definition 4.2 *There is a witness path for formula $E\,[p\,U\,q]$ from state s_0 of structure M, iff there is a path $p_w = s_0, s_1, \ldots, s_n$ such that*

$$\exists i[i \geq 0 \wedge M, s_i \models (\mathit{marked} \wedge \mathit{labeled}) \wedge \\ \forall j[0 \leq j < i \implies M, s_j \models (\neg \mathit{marked} \wedge \neg \mathit{labeled})\,]]$$

This definition gives us a direct definition of a search algorithm for a witness path starting from state s. Also this definition holds after the formula has been evaluated to *True* or *False* in some states of the system. If the formula was evaluated to *True* in state s_n i.e. it is marked and labeled, we have a witness path from state s_n. If we can reach the state s_n from s_0 using only unmarked and unlabeled states, then there is also a witness from s_0. If the formula was evaluated to *False* in state s_n i.e. it is marked and unlabeled, then we couldn't construct a witness from s_n and we can construct any witness which contains the state s_n.

Unfortunately the definition above doesn't give a description of a linear time model checking algorithm for the existential until formulas. The reasons are the same why the ALMC algorithm needs two passes. In the first pass the ALMC algorithm checks whether a witness path of the form described above exists. The second pass is used to evaluate the checked formula in all visited potential witness states to either *True* or *False* depending on whether they can or can't reach the witness path using only other visited potential witness states. Note that in the context of our new algorithm the potential witness states are the states which are initially unmarked and unlabeled. The new algorithm described in this section combines these two search passes by using the stack *fst*. When a new state t is first encountered, the stack *fst* contains all visited potential witness states from which the state t can be reached. If a witness is found, the states on the stack *fst* are exactly those visited potential witness states from which some witness path can be constructed. We will first describe the part of the algorithm which searches for a witness path, and after this return to the way the stack *fst* is maintained by the algorithm during the search.

The subroutine $\mathit{check\_eu}(i, s, st, fst)$ presented in Figure 42 is a witness search algorithm which uses the witness definition above. It uses the subroutine $\mathit{check\_new}(i, s, st, fst, dfs)$ presented Figure 43. The parameters *st*, *fst*, and *dfs* are passed by reference to these subroutines. These subroutines together form a non-recursive depth-first search algorithm. The task of the subroutine $\mathit{check\_new}(i, s, st, fst, dfs)$ is to search forward in depth-first order and look for a witness path. The backtracking of search and all book-keeping required during it are handled by the subroutine $\mathit{check\_eu}(i, s, st, fst)$.

The search terminates as soon as a witness path for the formula has been found from state s, or no witness path for the formula can be found. The search only has to consider those states in which the formula truth value is not yet known, i.e. states which are unmarked and therefore also unlabeled before calling the subroutine. We also need the truth values of the search bitmaps of the immediate successor states of these states.

The search starts from an unmarked state s, and the algorithm marks unmarked states the first time they are encountered. This gives us a way of detecting loops during the search. Another reason why encountered states are marked is the fact that the formula i is evaluated in all encountered states when the search algorithm terminates. When we first enter a state, we check whether the concatenation of the current search path stored on stack *st* and the entered state t is a witness path. This part of the algorithm is done by the subroutine $\mathit{check\_new}(i, s, st, fst, dfs)$.

If the entered state t is a marked and labeled state then a witness path has been found. We will describe why this is the case and how it can be seen from the witness path definition above. The algorithm search path *st* always only contains states which were initially unmarked and unlabeled. The state t was marked and labeled before the search and we have found a path of initially unmarked and unlabeled states from s to t. This is obviously a witness path, and we signal this by returning from the subroutine $\mathit{check\_new}(i, s, st, fst, dfs)$ with the return value -1.

We also need to handle the cases in which no witness path was directly found. If the entered state t is marked and unlabeled then we have either found a loop, or a state which was unlabeled before the search started. In either case we don't need to search the successor states of t for a witness, and we backtrack to the parent state by exiting the subroutine $\mathit{check\_new}(i, s, st, fst, dfs)$. If the entered state t was not marked then we will have to continue the search for a witness path in depth-first order recursively for all successor states. If no witness path was found after all successors of t have been visited, we can't yet conclude that there is no witness path which contains t. This is the major difference to checking the universal until formulas, and is the reason why the model checking of an existential until formula is much more complicated. What we do is to backtrack from the state t and let the second part of the algorithm handle the labeling of the state t when it can be decided.

The second part of the algorithm evaluates the truth value of the formula in all potential successor states. Recall that the stack *fst* is used to decide from which potential witness states can some witness path be constructed. When a new state t is encountered, the stack *fst* contains all visited potential witness states from which the state t can be reached. If a witness is found, the states on the stack *fst* are exactly those visited potential witness states from which some witness path can be constructed. Maintaining and using the stack *fst* is what the second part of the model checking algorithm does.

The second part of the algorithm in subroutines *check_eu*(i, s, st, fst) and *check_new*(i, s, st, fst, dfs) was inspired by Tarjan's algorithm [19] for searching the maximal strongly connected components of a graph. Our algorithm uses slightly less memory and was designed specifically for the model checking problem at hand. The difference is in the way depth-first search numbers are used. Tarjan's algorithm uses two sets of depth-first numbers, the second of which is stored on the search stack i.e. $|S| \times \log(\lceil |S| \rceil) \times 2$ bits of memory in the worst case. The algorithm presented in this section uses $|S| \times (\log(\lceil |S| \rceil) + 1)$ bits of memory instead. Tarjan's algorithm could be used with only minor modifications to the algorithm, because the component search stack of Tarjan's algorithm contains exactly the same states as the stack *fst* in our algorithm.

To describe the handling of the stack *fst* handling we need some additional definitions. First we define $<_{dfs}$ to be the total ordering relation on encountered states: $\forall s, s' \in S : s <_{dfs} s'$, iff state s was encountered before s' during the current 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 *check_eu*(i, s, st, fst). The $\overset{*}{\rightsquigarrow}$ is used to represent the reflexive, transitive closure of $\rightsquigarrow$.

Definition 4.3 *Let S be the set of states of the structure M, let $\rightsquigarrow$ the set of edges traversed by the current search, and s a new state encountered by the search. The set of 'father' states is the function $F : S \rightarrow \mathcal{P}(\mathcal{S})$:*

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

Note that because the algorithm will only use the successor edges of potential witness states, only the state $s \in F(s)$ can be something else than a potential witness state. When a new unmarked state s is encountered, we want the father state stack *fst* to contain exactly the states defined by $F(s)$. If this is the case and a witness path is found when we first encounter the state s, then from all states in stack *fst* we can also construct a path of potential witness states ending to state s, because $\forall s' \in fst : s' \overset{*}{\rightsquigarrow} s$, and because apart from s all states in $F(s)$ are potential witness states. Therefore the formula is *True* in them, and the subroutine *check_new*(i, s, st, fst, dfs) in rows 19-25 labels them correspondingly. If the search ended and no witness path was found, the formula is already *False* in all visited potential witness states due to the initialization procedure, and nothing needs to be done.

Here sketch why the states stored on the stack *fst* are exactly those visited potential witness states from which new state s can be reached. When the search starts the stack *fst* is empty. Each time a new state s is encountered, 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 could be found through 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 through the successor, or 0 if there is no way of finding a witness 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 the depth-first search number of some 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: $\exists s' \in F(s) : s \overset{*}{\rightsquigarrow} s' \overset{*}{\rightsquigarrow} s \wedge s' <_{dfs} s$. This loop requires us to keep s 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.

The combination of these two parts of the model checking algorithm is quite straightforward. The algorithm pseudo-code can be found in Figures 42 - 43. These subroutines can quite easily be combined to one larger subroutine for a straightforward non-recursive implementation.

```
proc check_eu(i, s, st, fst) ≡                    Search for witness for i from s
  dfs := 1;
  while True                                      Non-recursive depth-first search
        do
        successor_min := check_new(i, s, st, fst, dfs);       Search new
        if successor_min = −1 then return ;       Witness in st, return
        fi;
        while True                                Backtracking search for witness
              do
              if empty(st)
                then
                      return ;                    No witness found
              fi;
              ⟨s, succ_num⟩ := pop(st);           Backtrack to parent state
              if ((successor_min ≠ 0) ∧           Father state of s reachable
                (get_min(s) > successor_min))     from successor?
                then
                      set_min(s, successor_min);  Update smallest
                      set_modified(s);            reachable father state
              fi;
              succ_num := succ_num + 1;           Try next successor
              if has_successor(s, succ_num)
                then
                      push(⟨s, succ_num⟩, st);    Push search state
                      s := successor(s, succ_num);          Next succ
                      break;                      Visit succ
                else                              All succs visited
                      if ¬modified(s)             Father state reachable?
                        then      No → No witness reachable from s
                            repeat     or any state entered after it
                                  t := pop(fst);
                                  set_min(t, 0);            Cleanup
                                  reset_modified(t);
                            until t = s;          All reachable popped?
                      fi;
              fi;
        od;
  od;
.
```

Figure 42: Subroutine $check\_eu(i, s, st, fst)$.

```
proc check_new(i, s, st, fst, dfs) ≡                  Search new states for witness
  while True                                                        Search forward
      do
      if ¬marked(s)                                      Formula evaluated in s ?
        then
              push(s, fst);                                  Add s to father states
              mark_state(s);                          New state found, mark it.
              set_min(s, dfs);                                  Store dfs number
              reset_modified(s);
              dfs := dfs + 1;              dfs should be passed by reference
              succ_num := 0;                           Search succs for witness
              push(⟨s, succ_num⟩, st);                        Push search state
              s := successor(s, succ_num);                   Visit successor 0
         else
              if labeled(s, i)
                then                                                 Witness found
                      push(⟨s, 0⟩, st);
                      while ¬empty(fst)
                            do                                      ∀t ∈ fst : t ⇝* s
                            t := pop(fst);                  Get father state t
                            set_min(t, 0);                           Cleanup t
                            reset_modified(t);
                            add_label(t, i);                       Label t True
                      od;
                      return −1;               Witness in stack st, return
              fi;
              break;                          Not labeled, backtrack to parent
      fi;
  od;
  return get_min(s);                          Return minimum reachable state
.                                                or 0 if no witness from s found
```

Figure 43: Subroutine *check_new*(i, s, st, fst, dst).

4.3 Time and Space Requirements

The algorithm presented in this section is a global model checking algorithm, whose evaluation routines for universal until and existential until formulas are depth-first searches. The worst case time complexity for these subroutines is therefore $O(|S| + |R|)$ and for the whole model checking algorithm $O(length(f) \times (|S| + |R|))$. This matches the worst-case time complexity of the best global and local model checking algorithms. The worst-case memory requirements are $O(length(f) \times |S| + |R|)$, if we assume storing the formulas truth values, search bitmaps, the successor relation, and also all state and depth-first search numbers require the same constant $O(1)$ amount of memory for each item.

We have made the same analysis using the same assumptions for all the algorithms presented in this work and stated the results in the corresponding sections. When we gather the results of the complexity analysis presented in the different sections, we get the table in Figure 44. To save some space, we use the notation $|f|$ as another way of expressing $length(f)$, i.e. the length of the formula to be model checked. There are no surprises in this table, and apart from the naive algorithm, all the algorithms have similar time and space complexity figures when using this kind of analysis. Another main point to be made is that only the ALMC algorithm and our new algorithm can have the counterexamples and witnesses facility easily incorporated into the algorithm.

Algorithm	Time	Space	Counter-examples
Naive	$O(\|f\| \times \|S\| \times (\|S\| + \|R\|))$	$O(\|f\| \times \|S\| + \|R\|)$	No
EMC	$O(\|f\| \times (\|S\| + \|R\|))$	$O(\|f\| \times \|S\| + \|R\|)$	No
ALMC	$O(\|f\| \times (\|S\| + \|R\|))$	$O(\|f\| \times \|S\| + \|R\|)$	Yes
New	$O(\|f\| \times (\|S\| + \|R\|))$	$O(\|f\| \times \|S\| + \|R\|)$	Yes

Figure 44: Worst-case time and space requirements.

Using only the table in Figure 44, the algorithms might seem to almost be identical in their time and space requirements. The naive algorithm is the only exception to the rule, because it has worse time requirements. However the assumptions made in the space requirement analysis were perhaps not realistic ones. For example it was assumed that the amount of memory needed to store an item of the successor relation is the same as the amount of memory needed by a boolean subformula truth value, which is quite a coarse simplification from an implementation perspective. We will make a more detailed analysis by taking into account the number of bits needed to encode each of the different data-types used by the algorithms. In the environment into which our new model checking algorithm is implemented, the typical use of a model checker is to check formulas, for which $length(f) < 5$, and the cardinality of states and the successor relation can both be > 100000. In this case the amount of memory needed for storing the successor relation, the depth-first search numbers, and the search stacks become the limiting factor before the number of the required subformula truth values does.

A more detailed analysis of worst-case memory requirements of the new algorithm shows that the subformula and search bitmaps need $|S| \times (length(f)+2)$ bits of memory, the depth-first search numbers need $|S| \times \lceil \log(|S|) \rceil$ bits of memory, and the search stacks *st* and *fst* need together $|S| \times \lceil \log(|S|) \rceil \times 3$ bits of memory in the worst case. To sum up the actual memory requirements, the search memory requirement of the new algorithm are $|S| \times ((\lceil \log(|S|) \rceil \times 4) + (length(f) + 2))$ bits of memory in the worst case, and additionally a simple adjacency list representation of the successor relation needs $|R| \times \lceil \log(|S|) \rceil + |S| \times \lceil \log(|R|) \rceil$ bits of memory.

We use Figure 45 to sum up the search memory requirements of the algorithms presented in this work. The search stack usage was not analyzed as carefully for the other algorithms, and therefore only the order of bits of memory required is given. It is expected that the EMC algorithm uses less memory for search stacks than our new algorithm because of its simpler depth-first search algorithm which only needs one stack. The ALMC algorithm is more difficult to analyze, but the effects of the recursive subformula evaluations can be seen in the table as an added $length(f)$ term in the search stack usage.

Algorithm	Bitmaps	DFS-numbers	Search stacks
Naive	$\|S\| \times \|f\|$	-	-
EMC	$\|S\| \times (\|f\| + 1)$	-	$O(\|S\| \times \lceil \log(\|S\|) \rceil)$
ALMC	$\|S\| \times \|f\| \times 2$	$\|S\| \times \lceil \log(\|S\|) \rceil$	$O(\|S\| \times \lceil \log(\|S\|) \rceil \times \|f\|)$
New	$\|S\| \times (\|f\| + 2)$	$\|S\| \times \lceil \log(\|S\|) \rceil$	$O(\|S\| \times \lceil \log(\|S\|) \rceil)$

Figure 45: Search memory requirements in bits.

Algorithm	Successors and predecessors
Naive	$\|R\| \times \lceil \log(\|S\|) \rceil + \|S\| \times \lceil \log(\|R\|) \rceil$
EMC	$(\|R\| \times \lceil \log(\|S\|) \rceil + \|S\| \times \lceil \log(\|R\|) \rceil) \times 2$
ALMC	$\|R\| \times \lceil \log(\|S\|) \rceil + \|S\| \times \lceil \log(\|R\|) \rceil$
New	$\|R\| \times \lceil \log(\|S\|) \rceil + \|S\| \times \lceil \log(\|R\|) \rceil$

Figure 46: Successor and predecessor memory requirements in bits.

We use Figure 46 to sum up the amount of memory needed to store the successor and predecessor relations in a simple adjacency list form. It should be noted that the EMC algorithm needs both the successor and predecessor relations. If cardinality of the successor relation is the same as the number of states i.e $|R| = |S|$, then the actual memory requirements of the EMC algorithm and our new algorithm will differ only by a constant factor, which is very close to 1. In the cases when the cardinality of the successor relation is much larger than the number of states i.e. $|R| \gg |S|$, our algorithm will have smaller memory requirements than the EMC algorithm. The amount of memory needed for the predecessor relation for the EMC algorithm grows in this case faster than the amount of memory needed by the search stacks and depth-first search numbers for our new algorithm.

The ALMC local model checking algorithm has significantly larger actual worst-case memory requirements than the other algorithms, all of which are global model checkers. The difference becomes larger when the length of the formula increases. It is not known, whether this worst-case behavior will occur in practical model checking use. It seems that in many cases the checked formulas are generated by a human and are therefore usually quite short. The lazy evaluation of subformulas could be combined with a good memory management scheme, which might result in a model checker which has good average case memory requirements.

It is easy to make a CTL model checker with smaller memory requirements than our new algorithm: for example the naive implementation of a global model checker presented in this work uses significantly less memory. However the naive algorithm has worse time complexity. It would be interesting to know if there are CTL model checking algorithms which match the same worst case time complexity and have significantly smaller actual memory requirements.

One thing that should be noted is that in the analysis in this thesis we have left out the amount of memory needed to store the state information of each state of the system. The reasons for this are twofold, first this amount of memory is the same for all the algorithms. Secondly the state information can be removed from memory after all the atomic subformulas have been evaluated in all states of the system. For a local model checker this moment might become later than for a global model checker, which can increase memory usage. In global model checker we can use the fact that all atomic subformulas are not dependent on each other, and evaluate all atomic subformulas in all states of the system in one pass through the state information file.

5 The PROD Implementation

Reachability analysis is a formal way to analyze concurrent and distributed systems. Because these systems are complex and errors are often hard to detect, tool support for designing and analyzing these systems is absolutely essential. The Pr/T-Net reachability analysis tool set PROD [21] has been developed at the Digital Systems Laboratory at Helsinki University of Technology. The tool set contains several different ways of performing analysis and verification of concurrent systems. The main emphasis in the PROD tool set is on different reachability analysis methods. To this end PROD contains several different tools which are tied together through a command-line driven tool `prod`. Currently the tool set consist of following components:

1. The control program `prod`, that is used to execute the other components of the tool set.
2. The reachability graph generator generator called `prpp`, a program which reads a net description given in a Net Description Language, and generates a C-language program, which is then compiled, and run to generate the reachability graph.
3. The query program `probe`, a tool for navigating in the reachability graph.
4. A tool called `strong` for calculating the maximal strongly connected components.
5. An interface to the ARA tool, `araprod`, which we won't discuss further in this work.

We can see how the different parts of the tool are combined in Figure 47. The main tool `prod` coordinates this work, so most of the tools don't need to be explicitly used. The description of the system *sys* is given in a Net Description language in a file with a suffix *.net*. This system description is used by the `prpp` program to create a C-language program *sys.c*. This reachability graph generator program is compiled by the C-compiler `gcc`, which generates the reachability graph generator program `sys`. When this program is run it creates the reachability graph of the analyzed system. This reachability graph is stored in the files *sys.adr*, *sys.aws*, and *sys.gph*. One way the reachability graph can be used is to use the program `strong` to compute the maximal strongly connected components of the reachability graph. However in this work we are more interested in model checking the branching time temporal logic CTL.

The new CTL model checking algorithm presented in this work was implemented as a new feature into the *probe* tool. We can use the program `probe`

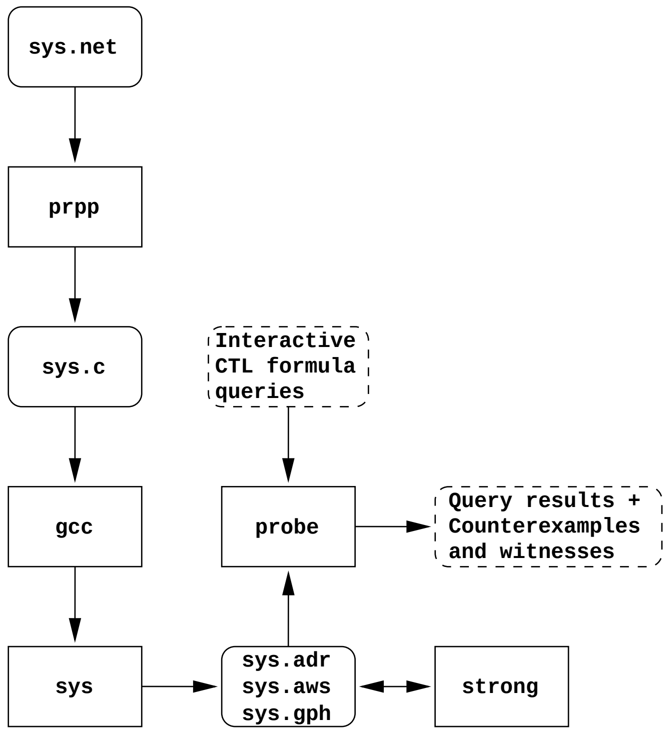

Figure 47: Structure of the PROD tool set.

to investigate the reachability graph. The `probe` program is an interactive reachability graph navigation tool, and it also has it's own query language for more automated analysis. As a new feature the users can also interactively check any CTL formulas they want in any state of the analyzed system. The algorithm was built on top of the existing *probe* tool using existing code whenever possible. For example the algorithm uses the the old probe query language to evaluate the atomic propositions needed in the model checking.

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 in all states in one pass over the markings generated by the reachability graph generator. This is to avoid accessing the large marking file *sys.gph* several times for each state as the different atomic subformulas are evaluated. After all the atomic subformulas have been evaluated in all the states of the system, the state information is no longer needed, and can be removed from the memory. The users of the tool ask local CTL model checking questions, for example they want to know whether the specified formula is *True* in the initial state of the system. Because the CTL model checking questions users ask a local model checking questions, the counterexamples and witnesses facility can be used. A drawback of first evaluating all the atomic subformulas is that even though the model checking for the top-level formula is local, the implementation still needs to evaluate all the atomic subformulas.

The maximal strongly connected components of the reachability graph could be used to optimize the order in which local model checker is called to evaluate a subformula, when global model checking is performed for the subformula. We can use topological sorting to arrange the order in which the formula should be evaluated in different states, starting from the terminal maximal strongly connected components. In this way we would minimize the amount of memory needed for different search stacks at the cost of an extra preprocessing step.

As a side note we'd like to note the old probe query language is expressive enough to implement a CTL model checker [21]. However because the query language wasn't designed for that task, the running times of such a model checker can grow exponentially in the length of the formula model checked. This makes the use of the old probe query language for CTL model checking infeasible for anything but the most simple formulas.

No extensive measurement results are available at the time of writing this, but it seems that the performance of the new model checking algorithm is quite good. For example we tried model checking the dining philosophers problem net description from [21] with 12 philosophers. The system has 531 440 reachable states and 4 251 516 state transitions. The model checking subroutines for all of the temporal subformulas themselves take only under 10 seconds for globally model checking one temporal subformula, after all the required successor relation files have been preloaded into memory. However the evaluation time of even one atomic subformula in all states of the system can take in the order of 1 minute 20 seconds. The reachability graph generation time for this example was approximately 16 minutes. All these results were obtained using a 166Mhz Pentium-PC running Linux with 64MB of memory. From these limited experiments it seems that implementation of the new model checking subroutines themselves are quite capable of handling even larger systems without problems, and that the reachability graph generation and atomic subformula evaluation seems to be the bottlenecks in the model checking process.

6 Conclusions

The initial goal of this work was to select a CTL model checking algorithm to be implemented in the PROD reachability analyzer tool set. For this work we have analyzed different advanced model checking algorithms, and found out that these algorithms have different sets of desired features. However no algorithm having all of these desired features existed, but it was discovered that creating such an algorithm was possible. The main contribution of this thesis is then a new CTL model checking algorithm with time complexity matching the best existing algorithms, i.e. $O(length(f) \times (|S| + |R|))$, and memory requirements which are either equal or smaller than the existing algorithms, depending on the structure of the system model checked. The algorithm has a counterexamples and witnesses facility, which is very valuable when trying to find the cause of incorrect behaviors of the system we are model checking. The algorithm is also straightforward to implement efficiently, and we have implemented the algorithm in PROD tool set. The main ideas of this new algorithm were first presented in the paper [17].

The algorithms analyzed in this thesis are by no means the only presented ones for model checking CTL. All CTL formulas can be expressed with the alternation-free subset of the propositional mu-calculus [12]. This means that all model checking algorithms for the propositional mu-calculus can be used for model checking CTL. Several linear time model checking algorithms for the alternation-free subset of mu-calculus have indeed been presented: Cleaveland and Steffen [10], Cleaveland, Klein and Steffen [9], Andersen [1] and Bhat, Cleaveland and Grunberg [3]. Other approaches which uses the mu-calculus fixed-point representation of CTL formulas are the model checkers based on the BDD data structure [4]. This data structure is used to efficiently compute the fixed-point formulas which can be used to represent CTL formulas. This representation seems to work well especially when verifying digital circuits [5]. Note that using BDD:s doesn't guarantee a linear running time for model checking: BDD's are a way of heuristically computing the fixed point formulas. Yet another way of doing CTL model checking is the automata-theoretic approach. Bernholtz, Vardi and Wolper have developed an automata based CTL model checking algorithm [2]. Also this algorithm has linear time complexity in both the size of the structure and length of the formula.

In their paper Cheng, Christensen and Morgensen [6] discuss a way of using the maximal strongly connected components of the reachability graph when model checking a subset of the CTL. It seems that this method was also independently discovered by Pogrell [20]. The method uses the precomputation of the maximal strongly connected components of the reachability graph, and it doesn't handle full CTL. The methods presented in these works could be used in combination with the model checking algorithm presented here. It would be interesting to compare whether the precomputation of the maximal strongly connected components pays off when compared to the new algorithm presented in this thesis.

Partial order reductions used in the PROD reachability graph generation do not preserve CTL properties. Therefore we need to generate the full reachability graph for the model checking problem at hand. This seems to usually become the bottleneck of the verification process before the CTL model checking process does. 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 in [15], [23]. These reduction also preserves the full branching time temporal logic CTL*-X , with which one can express all formulas of both the linear time temporal logic LTL-X and the branching time temporal logic CTL-X.

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. Such a combination would be truly useful, if the properties we are interested in can't be expressed by the linear time temporal logic LTL-X. For LTL-X there is an on-the-fly model checker combined with partial-order reductions in the PROD tool set. It would be quite a challenging task to design an on-the-fly CTL*-X model checking algorithm, which works in combination with partial-order reductions.

References

[1] Andersen, H. R.: *Model Checking and Boolean Graphs.* Theoretical Computer Science 126 (1994) 1, Elsevier, pp. 3–30.

[2] Bernholtz, O., Vardi, M. Y., Wolper, P.: *An Automata-Theoretic Approach to Branching-Time Model Checking.* In Dill, D. (ed.): Proceedings of the 6th International Conference on Computer Aided Verification, Lecture Notes in Computer Science 818, Springer-Verlag, Berlin, 1994, pp. 142–155.

[3] Bhat, G., Cleaveland, R., Grumberg, O.: *Efficient On-the-Fly Model Checking for CTL*.* In Proceedings of 10th Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, 1995, pp. 388–397.

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

[5] 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.

[6] Cheng, A., Christensen, S., Mortensen, K. H.: *Model Checking Coloured Petri Nets Exploiting Strongly Connected Components.* In Spathopoulos, M. P., Smedinga, G., Kozák, K. (eds.): Proceedings of the International Workshop on Discrete Event Systems, WODES96, Edinburgh, 1996, pp. 169-177.

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

[8] 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.

[9] Cleaveland, R., Klein, M., Steffen, B.: *Faster Model Checking for the Modal Mu-Calculus.* In v. Bochmann, G., Probst D. K. (eds.): Proceedings of the Fourth International Workshop on Computer Aided Verification, Lecture Notes in Computer Science 663, Spriger-Verlag, Berlin, 1992, pp. 410–422.

[10] Cleaveland, R., Steffen, B.: *A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus.* Formal Methods in System Design 2 (1993) 2, pp. 121–147.

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

[12] Emerson, E. A., Lei, C.: *Efficient Model checking in Fragments of the Propositional Mu-Calculus.* In Proceeding of Symposium on Logic in Computer Science, IEEE Computer Society Press, 1986, pp. 267–278.

[13] Emerson, E. A.:*Automated Temporal Reasoning about Reactive Systems.* In Möller, F., Birtwistle, G. (eds.): Logics for Concurrency - Structure versus Automata, Lecture Notes in Computer Science 1043, Springer-Verlag, Berlin, 1996, pp. 41–101.

[14] Genrich, H. J.: *Predicate/Transition Nets.* In Jensen, K., In Rozenberg, G. (eds.): High-level Petri Nets, Springer-Verlag, Berlin, 1991, pp. 3–43.

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

[16] Grönberg, P., Tiusanen, M., and Varpaaniemi, K.: *PROD—A Pr/T-Net Reachability Analysis Tool.* Technical Report B11, Digital Systems Laboratory, Helsinki University of Technology, Espoo, 1993.

[17] Heljanko, K.: *Implementing a CTL Model Checker.* In Proceedings of the Workshop Concurrency, Specification & Programming 1996, Informatik-Bericht Nr. 69, Humboldt-University, Berlin, 1996, pp. 75–84.

[18] Holzmann, G. J.: *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[19] Tarjan, R. E.: *Depth first search and linear graph algorithms.* SIAM Journal of Computing 1 (1972) 2, pp. 146–160.

[20] Pogrell, L.: *A New Model Checking Algorithm For Time Nets.* Informatik-Bericht Nr. 60, Humboldt-University, Berlin, 1996.

[21] Varpaaniemi, K., Halme, J., Hiekkanen, K., and Pyssysalo, T.: *PROD Reference Manual.* Technical Report B13, Digital Systems Laboratory, Helsinki University of Technology, Espoo, 1995.

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

[23] Willems, B., Wolper, P.: *Partial-Order Methods for Model Checking: From Linear Time to Branching Time*. In Proceedings of 11th Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, 1996, pp. 294–303.

[24] *CCITT Specification and Description Language (SDL)*. Technical report Z.100 (1993), ITU-T, 1994.

Appendix A - Mutex Example PROD Description

```
#define NC 0
#define TRY 1
#define CS 2

#define NONE 2

#place P lo(<.0, NC.>) hi (<.1, CS.>) mk(<.0..1, NC.>)
#place T lo(<.0.>) hi(<.NONE.>) mk(<.NONE.>)

#trans idle
   in  { P: <.x, NC.>; T: <.NONE.>; }
   out { P: <.x, NC.>; T: <.NONE.>; }
#endtr

#trans try
   in  { P: <.x, NC.>; T: <.turn.>; }
   out { P: <.x, TRY.>;
         T: (turn == NONE)<.x.> + (turn != NONE) <.turn.>; }
#endtr

#trans critical
   in  { P: <.x, TRY.>; T: <.x.>; }
   out { P: <.x, CS.>; T: <.x.>; }
#endtr

#trans non_critical
   in  { P: <.x, CS.> + <.y, s.>; T: <.turn.>; }
   out { P: <.x, NC.> + <.y, s.>;
         T: (s == TRY)<.y.> + (s != TRY)<.NONE.>; }
#endtr
```

Appendix B - Initialization Routines

```
proc init_formulas(f) ≡
   for i := 1 to length(f) do
        init_left(i);
        init_right(i);
   od
.
```

Figure 48: Subroutine *init_formulas*(f).

```
proc init_left(i) ≡
   if lsub(sub_f[i]) = null
      then
             left[i] := 0;
       else
             for j := 1 to i − 1 do
                  if lsub(sub_f[i]) = sub_f[j]
                     then
                            left[i] := j;
                            break;
                  fi
             od
   fi
.
```

Figure 49: Subroutine *init_left*(i).

```
proc init_right(i) ≡
  if rsub(sub_f[i]) = null
    then
         right[i] := 0;
     else for j := 1 to i − 1 do
              if rsub(sub_f[i]) = sub_f[j]
                then
                     right[i] := j; break;
              fi
         od
  fi
.
```

Figure 50: Subroutine *init_right*(i).