Efficient On-the-Fly Model Checking for CTL*

Girish Bhat*, Rance Cleaveland*
Department of Computer Science,
North Carolina State University
Raleigh, NC 27695-8206, USA,
e-mail: {girish,rance}@science.csc.ncsu.edu.

Orna Grumberg
Department of Computer Science, The Technion,
Haifa, Israel,
e-mail: orna@cs.technion.ac.il.

Abstract

This paper gives an on-the-fly algorithm for determining whether a finite-state system satisfies a formula in the temporal logic CTL*. The time complexity of our algorithm matches that of the best existing "global algorithm" for model checking in this logic, and it performs as well as the best known global algorithms for the sublogics CTL and LTL. In contrast with these approaches, however, our routine constructs the state space of the system under consideration in a need-driven fashion and will therefore perform better in practice.

1 Introduction

Researchers have devoted considerable attention to the development of automatic techniques, or *model-checking* procedures, for verifying finite-state systems against specifications given in various temporal logics [3, 7, 8, 12, 16, 19, 22, 25]. These logics permit users to characterize the properties, including safety and liveness, a system should exhibit as it executes over time, and as such they are useful for specifying *reactive* systems that behave in an ongoing manner by interacting with their environments. Among the temporal logics that have been developed, CTL* [15] has proved to be particular interesting in this respect, since it is capable of encoding uniformly a wide variety of different temporal logics, including those based on a linear and branching models of time. Several verification tools include implementations of model-checking algorithms for (subsets of) CTL* [11, 20], and a variety of case studies point to the practical utility of model checking as a basis for system verification [4, 5, 9, 21].

*Research supported by NSF/DARPA grant CCR-9014775, NSF grant CCR-9120995, ONR Young Investigator Award N00014-92-J-1582, NSF Young Investigator Award CCR-9257963, and NSF grant CCR-9402807.

In general, one may identify two basic approaches to model checking. The first, and more traditional, uses *global analysis* to determine if a system satisfies a formula; the entire state space of the system is constructed and subjected to analysis. Researchers have developed algorithms based on this approach that exhibit good worst-case time complexity, and they have been implemented and used in the verification of systems. From a pragmatic perspective, however, these algorithms may be seen to perform unnecessary work: in many cases (especially when a system does *not* satisfy a specification) only a subset of the system states need to be analyzed in order to determine whether or not a system satisfies a formula. *On-the-fly*, or *local*, approaches to model checking attempt to take advantage of this observation by constructing the state space in a demand-driven fashion.

In this paper we present an on-the-fly algorithm for checking finite-state Kripke structures against CTL* formulas. The algorithm has the same worst-case complexity as the best existing global procedure for this logic [16], and it also matches the best worst-case complexity for global and on-the-fly techniques developed for various sublogics [7, 26, 25]. Thus our algorithm may be seen as providing an efficient and unifying on-the-fly framework for temporal logic model checking. The rest of the paper is organized along the following lines. The next section describes the syntax and semantics of CTL*, while Section 3 describes a general approach to determining whether a system satisfies formulas of LTL. The section following shows how this approach may be made "on-the-fly", and Section 5 shows how this algorithm may be generalized to handle all of CTL* and remarks on the complexity of the algorithm on different sublogics. The last section contains our conclusions and directions for future research.

Related work Existing work in on-the-fly model checking has tended to focus on specifications given as various forms of automata. Courcoubetis et al. [6] give an algorithm with the same time complexity as ours for determining when a system satisfies a specification given as a Büchi automaton. In light of the correspondence between such automata and the linear-time fragment, LTL, of CTL*, it follows that the algorithms may be used for LTL model checking also. However, it is not clear how their approach may be extended to handle full CTL*. Jard and Jeron [18] define an on-the-fly approach for determining when a system satisfies a deterministic Büchi specification, but this algorithm cannot be used to verify LTL formulas as LTL is strictly more expressive than deterministic Büchi automata. Our algorithm is also more time-efficient than theirs, as their aim was space efficiency. Results in an extended version of [3] suggest a model-checking algorithm for full CTL* which allows the on-the-fly construction of the state space of the system. However, this approach requires the *a priori* construction of the states of an amorphous Büchi tree automaton from the formula being checked, and the time complexity is worse than ours.

Efficient, non-automata-based, on-the-fly algorithms for the branching-time sublogic CTL and for the alternation-free μ-calculus have been presented in [26] and [2] respectively, but neither approach appears capable of being extended to handle all of CTL*.

2 Syntax and Semantics of CTL*

In what follows we fix a set $\mathcal{A}$ of *atomic propositions*, which will be ranged over by $a, a', \ldots$. The following BNF-like grammar then describes the syntax of CTL*.

$$\begin{array}{lll} \mathcal{S} & ::= & a \mid \neg a \mid \mathcal{S} \wedge \mathcal{S} \mid \mathcal{S} \vee \mathcal{S} \mid A\mathcal{P} \mid E\mathcal{P} \\ \mathcal{P} & ::= & \mathcal{S} \mid \mathcal{P} \wedge \mathcal{P} \mid \mathcal{P} \vee \mathcal{P} \mid \mathsf{X}\mathcal{P} \mid \mathcal{P}\mathsf{U}\mathcal{P} \mid \mathcal{P}\mathsf{V}\mathcal{P} \end{array}$$

We sometimes call formulas of the form a or $\neg a$ *literals*; $\mathcal{L}$ is the set of all literals and will be ranged over by $l, \ldots$. We refer to the formulas generated from $\mathcal{S}$ as *state* formulas and those from $\mathcal{P}$ as *path* formulas; we define the CTL* formulas to be the set of state formulas. We use $p, p_1, q, \ldots$ to range over the set of state formulas and $\phi, \phi_1, \gamma, \ldots$ to range over the set of path formulas. We also call A and E *path quantifiers* and the X, U and V constructs *path modalities*. The following important sublogics of CTL* will also be mentioned on occasion.

CTL CTL (Computation Tree Logic) [7] consists of those CTL* formulas in which every occurrence of a path modality is immediately preceded by a path quantifier.

LTL LTL (Linear Temporal Logic) [19] contains CTL* formulas of the form $A\phi$, where the only state subformulas of ϕ are literals.

CTL* formulas are given meaning relative to *Kripke* structures, which may be defined as follows.

Definition 2.1 *A* Kripke structure *is a triple $\langle S, R, L\rangle$, where S is a set of* states, *$R \subseteq S \times S$ is the* transition relation, *and $L \in S \to 2^{\mathcal{A}}$ is the* labeling.

Intuitively, a Kripke structure encodes the operational behavior of a system, with S representing the possible system states, R describing the "execution steps", and L indicating which atomic propositions hold in a given state. For technical convenience, throughout this paper we assume that R is *total*: for every $s \in S$ there is an $s' \in S$ such that $\langle s, s'\rangle \in R$. We also use the following notions.

Definition 2.2 *Let $M = \langle S, R, L\rangle$ be a Kripke structure.*

1. *A* path *in M is a maximal sequence of states $\langle s_0, s_1, \ldots\rangle$ such that for all $i \geq 0$, $\langle s_i, s_{i+1}\rangle \in R$.*
2. *If $x = \langle s_0, s_1, \ldots\rangle$ is a path in M then $x(i) = s_i$ and $x^i = \langle s_i, s_{i+1}, \ldots\rangle$.*
3. *If $s \in S$ then $\Pi_M(s)$ is the set of paths x in M such that $x(0) = s$.*

Note that $\Pi_M(s)$ contains the paths emanating from s in M. Since R is assumed to be total, all paths in M are infinite. In what follows we use $x, \ldots$ to range over paths.

Let $M = \langle S, R, L\rangle$ be a Kripke structure. Then the meaning of CTL* formulas is given in terms of a relation $\models_M$ relating states to state formulas and paths to path formulas. Intuitively, $s \models_M p$ ($x \models_M \phi$) if state s (path x) satisfies state formula p (path formula ϕ).

Definition 2.3 *Let $M = \langle S, R, L\rangle$ be a Kripke structure, with $s \in S$ and x a path in M. Then $\models_M$ is defined inductively as follows.*

- *$s \models_M a$ if $a \in L(s)$ (recall $a \in \mathcal{A}$).*
- *$s \models_M \neg a$ if $s \not\models_M a$.*
- *$s \models_M p_1 \wedge p_2$ if $s \models_M p_1$ and $s \models_M p_2$.*
- *$s \models_M p_1 \vee p_2$ if $s \models_M p_1$ or $s \models_M p_2$.*
- *$s \models_M A\phi$ if for every $x \in \Pi_M(s), x \models_M \phi$.*
- *$s \models_M E\phi$ if there exists $x \in \Pi_M(s)$ such that $x \models_M \phi$.*
- *$x \models_M p$ if $x(0) \models p$ (recall p is a state formula).*
- *$x \models_M \phi_1 \wedge \phi_2$ if $x \models_M \phi_1$ and $x \models_M \phi_2$.*
- *$x \models_M \phi_1 \vee \phi_2$ if $x \models_M \phi_1$ or $x \models_M \phi_2$.*
- *$x \models_M \mathsf{X}\phi$ if $x^1 \models_M \phi$.*
- *$x \models_M \phi_1 \mathsf{U} \phi_2$ if there exists $i \geq 0$ such that $x^i \models_M \phi_2$ and for all $j < i, x^j \models_M \phi_1$.*
- *$x \models_M \phi_1 \mathsf{V} \phi_2$ if for all $i \geq 0, x^i \models_M \phi_2$ or if there exists $i \geq 0$ such that $x^i \models_M \phi_1$ and for every $j \leq i, x^j \models_M \phi_2$.*

The meaning of most of the constructs is straightforward. A state satisfies $A\phi$ ($E\phi$) if every path (some path) emanating from the state satisfies ϕ, while a path satisfies a state formula if the initial state in the path does. X represents a "next-time operator" in the usual sense, while $\phi_1 \mathsf{U} \phi_2$ holds of a path if ϕ_1 remains true until ϕ_2 becomes true. The constructor V may be thought of as a "release operator"; a path satisfies $\phi_1 \mathsf{V} \phi_2$ if ϕ_2 remains true until ϕ_1 "releases" the path from the obligation. Finally, although we only allow a restricted form of negation in this logic ($\neg$ may only be applied to atomic propositions), we do have the following result.

Lemma 2.4 *Let $M = \langle S, R, L\rangle$ be a Kripke structure.*

1. *For any state formula p there is a state formula $\mathsf{neg}(p)$ such that for all $s \in S, s \models_M \mathsf{neg}(p)$ iff $s \not\models_M p$.*

2. *For any path formula ϕ there is a path formula $\mathsf{neg}(\phi)$ such that for all paths x in M, $x \models_M \mathsf{neg}(\phi)$ iff $x \not\models_M \phi$.*

Proof. Follows from the duality of $\wedge$ and $\vee$, and U and V, and the self-duality of X. Note that $\mathsf{neg}(\phi_1 \mathsf{U} \phi_2)$ is $\mathsf{neg}(\phi_1) \mathsf{V} \mathsf{neg}(\phi_2)$. □

3 Proof Structures for LTL

The goal of this paper is to develop an efficient on-the-fly model-checking procedure for CTL*. To do so, we first give an on-the-fly routine for the sublogic LTL and then show how to modify it to handle all of CTL*. The on-the-fly procedure for LTL is in turn based on a collection of top-down *proof rules* for inferring when a state in a Kripke structure satisfies a LTL formula. In this section we present these proof rules, and in the next section we show how they may be used in support of an efficient local model checker for LTL. In what follows, fix $M = \langle S, R, L \rangle$ to be a finite-state Kripke structure.

The proof rules appear in Figure 1; they are similar to ones devised by Dam [13] for translating CTL* formulas into the modal mu-calculus. In particular, they are "goal-directed" in the sense that the goal of the rule appears above the subgoals, and they work on *assertions* of the form $s \vdash_M A\Phi$, where $s \in S$ and Φ is a *set* of path formulas. Semantically, $s \vdash_M A\Phi$ holds if $s \models_M A(\bigvee_{\phi \in \Phi} \phi)$. We sometimes write $A(\Phi, \phi_1, \ldots, \phi_n)$ to represent a formula of the form $A(\Phi \cup \{\phi_1, \ldots, \phi_n\})$. If σ is an assertion of the form $s \vdash_M A\Phi$ then we use $\phi \in \sigma$ to denote that $\phi \in \Phi$. In the remainder of the paper, we use $\sigma, \sigma_1, \ldots$ to range over Σ_M, the set of all assertions involving states in M.

The remainder of this section is devoted to showing that by using the rules in particular fashion, we may infer $s \vdash_M A\Phi$ exactly when $s \models_M A\Phi$. The following establishes that the rules are "backwards consistent."

Lemma 3.1 *Let $\sigma \equiv s \vdash_M A\Phi$ be an assertion.*

1. *If the subgoals resulting from applying a rule to σ have the form $s_1 \vdash_M A\Phi_1, \ldots, s_n \vdash_M A\Phi_n$, then $s \models_M A\Phi$ iff $s_i \models_M A\Phi_i$ for each i.*

2. *If the subgoal resulting from applying a rule to σ is* true *then $s \models_M A\Phi$.*

3. *If no rule can be applied to σ then $s \not\models_M A\Phi$.*

Proof. Follows from the semantics of CTL* and the fixpoint characterizations of the path modalities. For part 3, note that no rule can be applied to σ iff $\Phi = \emptyset$. □

We now define how one uses the rules to establish that assertions are true. We begin by defining the notion of a *proof structure*.

Definition 3.2 *Let $\Sigma'_M = \Sigma_M \cup true$, $V \subseteq \Sigma'_M$, $E \subseteq V \times V$ and $\sigma \in V$. Then $\langle V, E \rangle$ is a* proof structure *for σ if it is a maximal graph such that for every $\sigma' \in V$: σ' is reachable from σ, and the set $\{\sigma'' | \langle \sigma', \sigma'' \rangle \in E\}$ is the result of applying some rule to σ'.*

Intuitively, a proof structure for σ is a graph that is intended to represent an (attempted) "proof" of σ. In what follows we use traditional graph-theoretic notions, such as path, strongly connected component[1], etc., when speaking of proof structures. Note that in contrast with traditional definitions of proofs, however, proof structures may contain cycles. In order to define when a proof structure represents a valid proof of σ, we use the following notions.

Definition 3.3 *Let $\langle V, E \rangle$ be a proof structure.*

- *$\sigma \in V$ is a* leaf *iff there is no σ' such that $\langle \sigma, \sigma' \rangle \in E$. A leaf σ is* successful *iff $\sigma \equiv true$.*
- *An infinite path $\pi = \langle \sigma_0, \sigma_1, \ldots \rangle$ in $\langle V, E \rangle$ is* successful *iff some assertion σ_i on π satisfies the following: there exists $\phi_1 \mathsf{V} \phi_2 \in \sigma_i$ such that for all $j \geq i, \phi_2 \notin \sigma_j$.*
- *$\langle V, E \rangle$ is* partially successful *iff every leaf is successful. $\langle V, E \rangle$ is* successful *iff it is partially successful and each of its infinite paths is successful.*

Roughly speaking, an infinite path is successful if at some point a formula of the form $\phi_1 \mathsf{V} \phi_2$ is repeatedly "regenerated" by application of Rule R6; that is, the right subgoal (and not the left one) of this rule application appears each time on the path. Note that after $\phi_1 \mathsf{V} \phi_2$ occurs on the path ϕ_2 should not, since, intuitively, if ϕ_2 were true then the success of the path would not depend on $\phi_1 \mathsf{V} \phi_2$, while if it were false then $\phi_1 \mathsf{V} \phi_2$ would not hold. We now have the following.

Theorem 3.4 *Let M be a Kripke structure with $s \in S$ and $A\phi$ an LTL formula, and let $\langle V, E \rangle$ be a proof structure for $s \vdash_M A\{\phi\}$. Then $s \models_M A\phi$ iff $\langle V, E \rangle$ is successful.*

Proof. See appendix. □

One consequence of this theorem is that if σ has a successful proof structure, then all proof structures for σ are successful. Thus, in searching for a successful proof structure for an assertion no backtracking is necessary.

It also turns out that the success of a finite proof structure may be determined by looking at its strongly connected components. Call a strongly connected component S of $\langle V, E \rangle$ *nontrivial* if there exist (not necessarily distinct) $v, v' \in S$ such that there is a path containing at least one edge from v to v'. For any $V' \subseteq V$ we may define the *success set* of V' as follows.

$$Success(V') = \{ \phi_1 \mathsf{V} \phi_2 \mid \exists \sigma \in V'. \phi_1 \mathsf{V} \phi_2 \in \sigma \wedge \forall \sigma' \in V'. \phi_2 \notin \sigma' \}$$

We say that V' is successful iff $Success(V') \neq \emptyset$. We have the following.

Theorem 3.5 *A partially successful proof structure $\langle V, E \rangle$ is successful iff every nontrivial strongly connected component of $\langle V, E \rangle$ is successful.*

Proof. See appendix. □

We close this section by giving a bound on the size of proof structures.

[1] A strongly connected component S of a graph $\langle V, E \rangle$ is a maximal subset of V with the property that for any $v, v' \in S$, there is a path from v to v' in $\langle V, E \rangle$.

$$R1: \frac{s \vdash_M A(\Phi, p)}{true} \; (s \models_M p) \quad R2: \frac{s \vdash_M A(\Phi, p)}{s \vdash_M A\Phi} \; (s \not\models_M p)$$

$$R3: \frac{s \vdash_M A(\Phi, \phi_1 \vee \phi_2)}{s \vdash_M A(\Phi, \phi_1, \phi_2)} \quad R4: \frac{s \vdash_M A(\Phi, \phi_1 \wedge \phi_2)}{s \vdash_M A(\Phi, \phi_1) \quad s \vdash_M A(\Phi, \phi_2)}$$

$$R5: \frac{s \vdash_M A(\Phi, \phi_1 \mathsf{U} \phi_2)}{s \vdash_M A(\Phi, \phi_1, \phi_2) \quad s \vdash_M A(\Phi, \phi_2, \mathsf{X}(\phi_1 \mathsf{U} \phi_2)}$$

$$R6: \frac{s \vdash_M A(\Phi, \phi_1 \mathsf{V} \phi_2)}{s \vdash_M A(\Phi, \phi_2) \; s \vdash_M A(\Phi, \phi_1, \mathsf{X}(\phi_1 \mathsf{V} \phi_2))}$$

$$R7: \frac{s \vdash_M A(\mathsf{X}\phi_1, \ldots, \mathsf{X}\phi_n)}{s_1 \vdash_M A(\phi_1, \ldots, \phi_n) \; \ldots \; s_m \vdash_M A(\phi_1, \ldots, \phi_n)} \{s_1, \ldots, s_m\} = \{s' | (s, s') \in R\}$$

Figure 1: Proof rules for LTL. Note that in R1 and R2, state formula $p \in \mathcal{L}$.

Definition 3.6 *Let ϕ be a path formula such that $A\phi$ is in LTL. Then the* Fischer-Ladner closure *[17] of ϕ, $CL(\phi)$, is the smallest set such that the following hold.*

- $\phi \in CL(\phi)$.
- *If $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\phi_1 \mathsf{U} \phi_2$, or $\phi_1 \mathsf{V} \phi_2$ is in $CL(\phi)$ then $\phi_1, \phi_2 \in CL(\phi)$.*
- *If $\mathsf{X}\phi' \in CL(\phi)$ then $\phi' \in CL(\phi)$.*
- *If $\phi_1 \mathsf{U} \phi_2 \in CL(\phi)$ then $\mathsf{X}(\phi_1 \mathsf{U} \phi_2) \in CL(\phi)$.*
- *If $\phi_1 \mathsf{V} \phi_2 \in CL(\phi)$ then $\mathsf{X}(\phi_1 \mathsf{V} \phi_2) \in CL(\phi)$.*

Let $|\phi|$ represent the length of formula ϕ in the usual sense. One may show that $|CL(\phi)| \leq (3/2) * |\phi|$ (where for a set V, $|V|$ denotes the number of elements in V). Also, if $M = \langle S, R, L \rangle$ is a Kripke structure then let $|M| = |S| + |R|$. We now have the following.

Lemma 3.7 *Let $M = \langle S, R, L \rangle$ be a Kripke structure with $s \in S$, let p be a LTL formula of the form $A\phi$, and let $\langle V, E \rangle$ be a proof structure for $s \vdash_M p$. Then $|V| + |E| < 2^{|p|} * |M|$.*

Proof. Follows from the fact that for every assertion $s' \vdash_M A\Phi'$ in $\langle V, E \rangle$, $\Phi' \subseteq CL(\phi)$. □

4 A Local Model Checker for LTL

In this section we use the results obtained in the previous section to develop an efficient on-the-fly model-checking algorithm for LTL, which we will generalize in the next section to handle all of CTL*. One obvious solution to this problem would be to construct the proof structure for the assertion and then check if the proof structure is successful. Of course, this algorithm is not on-the-fly as it does not check the success of a proof structure until after it is built.

The algorithm in this section, on the other hand, combines the construction of a proof structure with the process of checking whether the structure is successful; as soon as it is determined that the partially constructed structure cannot be extended successfully, the routine halts the construction of the structure and returns the answer *false*. The subtlety of this approach lies in determining when the "strongly connected component condition" for successful proof structures is violated.

Pseudo-code for the algorithm appears in Figures 2 and 3. In essence the routine combines the construction of a proof structure with Tarjan's depth-first-search-based strongly connected components algorithm [1]; additional information is also stored in vertices in the structure that enables the detection of unsuccessful strongly connected components. The procedure uses the following data structures. With each assertion σ we associate three fields, σ.dfsn, σ.low, and σ.valid. The first of these contains the depth-first search number of σ, while the second records the depth-first search number of the "oldest" ancestor of σ that is reachable from σ (this is used to detect strongly connected components). Finally, σ.valid is a set of pairs of the form $\langle \phi_1 \mathsf{V} \phi_2, sp \rangle$. Intuitively, the formula component of such a pair may be used as evidence of the success of the strongly connected component that σ might be in, while *sp* records the "starting point" of the formula, i.e. the depth-first search number of the assertion in which this occurrence of the formula first appeared. The algorithm also maintains a stack onto which assertions are pushed and popped and two sets of assertions: V, which records the assertions that have been encountered so far, and F, which contains a set of assertions that have been determined to be "false". We assume the usual stack operations *push*, *pop*, and *top*; we also use an operation *inStack*, which determines whether or not a given element is in a given stack. Given an assertion σ, *subgoals*(σ) returns a set of assertions resulting from the application of an applicable proof rule; if no rule is applicable the empty set is returned.

The heart of the procedure is the routine dfs in Figure 3, which is responsible for attempting to construct a successful proof structure for its argument assertion σ. Given an assertion σ and a set of formula/number pairs (intuitively, the valid set from σ's parent in the depth-first search tree), the procedure first initializes the "dfsn" and "low" fields of σ appropriately, and it assigns to σ's "valid" field a set of pairs of V-formulas and their "starting points". Note that if $\phi_1 \mathsf{V} \phi_2$ appears in σ.valid and σ's parent then the starting point of the formula is inherited from the parent. After pushing σ

```
procedure modchkLTL(σ)
    var dfn init 0, stack init ε;

    procedure init (σ, valid);
        begin init
            dfn := dfn + 1;
            σ.dfsn := σ.low := dfn;
            σ.valid := {⟨φ1 V φ2, sp⟩|     φ2 ∉ σ ∧ (φ1 V φ2 ∈ σ ∨ X(φ1 V φ2) ∈ σ)
                                            ∧sp = (sp' if ⟨φ1 V φ2, sp'⟩ ∈ valid and dfn otherwise)}
        end init;

    procedure dfs (σ, valid); ... See Figure 3.

    begin modchkLTL
        return (dfs(σ, {}));
    end modchkLTL;
```

Figure 2: An on-the-fly model checker for LTL.

```
procedure dfs (σ, valid);
    var flag init true;
    begin dfs
        init (σ, valid);
        stack := push (σ, stack);
        V := V ∪ {σ};
        case subgoals (σ)
            {true}     : flag := true
            ∅          : flag := false
            else       :
                foreach σ' ∈ subgoals(σ) while flag do
                    if σ' ∈ V then
                        if σ' ∈ F then flag := false
                        else
                            if inStack (σ', stack) then
                                σ.low := min (σ.low, σ'.low);
                                σ.valid := { ⟨φ1 V φ2, sp⟩ ∈ σ.valid | sp ≤ σ'.dfsn };
                                if σ.valid = ∅ then flag := false endif
                            endif
                    else
                        flag := dfs (σ', σ.valid);
                        if σ'.low ≤ σ.dfsn then
                            σ.low := min (σ.low, σ'.low);
                            σ.valid := σ'.valid
                        endif
                endforeach
        endcase ;
        if σ.dfsn = σ.low then
            repeat
                σ' := top (stack);
                stack := pop (stack);
                if flag = false then F := F ∪ {σ'} endif
            until σ' = σ
        endif ;
        return (flag)
    end dfs;
```

Figure 3: The depth-first search procedure.

```
procedure update (b, σ)
    begin update
        flag := b;
        if (not b) then F := F ∪ {σ}
    end update;

procedure modchkCTL*(σ)
    begin modchkCTL*
        if σ ∈ V then
            if σ ∈ F then flag := false else flag := true
        else
            V := V ∪ {σ};
            case σ
                s ⊢_M l:
                        update (s ⊨_M l, σ)
                s ⊢_M p1 ∧ p1:
                        update (modchkCTL*(s ⊢_M p1) and modchkCTL*(s ⊢_M p2), σ)
                s ⊢_M p1 ∨ p2:
                        update (modchkCTL*(s ⊢_M p1) or modchkCTL*(s ⊢_M p2), σ)
                s ⊢_M AΦ:
                        update (modchkLTL(σ), σ)
                s ⊢_M EΦ:
                        update (not (modchkLTL(s ⊢_M neg(EΦ))), σ)
            endcase ;
        return (flag);
    end modchkCTL*;
```

Figure 4: An on-the-fly model-checker for CTL*.

onto the stack and adding σ to the set V, dfs calls the procedure *subgoals* which returns the subgoals resulting from the application of a rule to σ. dfs then processes the subgoals as follows. First, if the only subgoal has the form *true* dfs should return true, while if the set of subgoals is empty then σ is an unsuccessful leaf, and false should be returned. Finally, suppose the set of subgoals is a nonempty set of assertions; we examine each of these in the following fashion. If subgoal σ' has already been examined (i.e. is in V) and found to be false (i.e. is in F), then the proof structure cannot be successful, and we terminate the processing of the subgoals in order to return false. If σ' has already been examined but has not been found false, and if σ' is in the stack (meaning that its strongly connected component is still being constructed), then σ and σ' will be in the same strongly connected component; we reflect this by updating σ.low accordingly. We also update σ.valid by removing formulas whose starting points occur *after* σ'; as we show below, these formulas cannot be used as evidence for the success of the strongly connected component containing σ and σ'. Note that if σ.valid becomes empty then the proof structure cannot be successful and we should return false. On the other hand, if σ' has not been explored, then dfs is invoked recursively on σ', and the low and valid fields of σ updated appropriately if σ' is determined to be in the same strongly connected component as σ. Once the subgoal processing is completed, dfs checks to see whether a new strongly connected component has been detected; if so, it removes it from the stack.

To argue for the correctness of the algorithm we establish that it maintains a particular invariant. Note that the procedure incrementally constructs a graph $\langle V, E\rangle$ that, if successfully completed, constitutes a proof structure for the given assertion. The vertex set V is maintained explicitly, with E defined implicitly by: $\langle \sigma, \sigma'\rangle \in E$ if $\sigma' \in \mathit{subgoals}(\sigma)$. Given σ, let $f(\sigma) = \{\, \phi \mid \exists j.\langle \phi, j\rangle \in \sigma.\mathit{valid}\,\}$. Also, for a set $S \subseteq V$, define $h(S)$ to be the assertion σ with the largest dfs number. We now have the following.

Lemma 4.1 *Algorithm dfs maintains the following invariant. Let $G = \langle V, E\rangle$ be a snapshot of the graph constructed by dfs during its execution. Then for every strongly connected component S in G, $f(h(S)) = \mathit{Success}(S)$.*

Proof. See appendix. □
Using this lemma and Theorem 3.5 we can now establish the correctness of modchkLTL.

Theorem 4.2 *When modchkLTL(σ) terminates, we have that for every $\sigma' \in V$ of the form $s' \vdash_M A\Phi'$, $s' \models_M A\Phi'$ iff $\sigma' \notin F$.*

Corollary 4.3 *modchkLTL($s \vdash_M A\Phi$) returns true iff $s \models_M A\Phi$.*

Regarding the time complexity of modchkLTL($s \vdash_M p$) for the LTL formula p, note that if $G = \langle V, E\rangle$ is the graph at the time the algorithm terminates, then the time consumed by the algorithm is bounded by $|p| * |G|$.

This follows from the fact that the algorithm visits each assertion in V exactly once and that the work done at each node is bounded by the size of the valid set at each assertion, which in turn is bounded by $|p|$. In the worst case G is a complete proof structure; from Lemma 3.7 it follows that the worst-case time complexity of the algorithm is in $O(|p| * 2^{O(|p|)} * |M|) = O(2^{O(|p|)} * |M|)$, which matches that of the best existing algorithms for for LTL [6, 19, 25].

5 On-The-Fly Model Checking for CTL* and CTL

The global model-checking algorithm for CTL* given in [16] processes a CTL* formula p by recursively working on the top-level state subformulas of p and then calling the LTL model checker [19] on the formula p' obtained by replacing these subformulas by new atomic propositions. Our approach to on-the-fly model checking for CTL* follows similar lines and is presented in Figure 4. This routine recursively calls modchkLTL appropriately when it encounters assertions of form $s \vdash_M A\Phi$ or $s \vdash_M E\Phi$. We also make a slight modification to procedure *subgoal*. When *subgoal* encounters an assertion of form $s \vdash_M A(\Phi, p)$, it recursively invokes modchkCTL*($s \vdash_M p$) to determine if $s \models_M p$ and then decides if rule R1 or rule R2 needs to be applied.

Theorem 5.1 *For an assertion $s \vdash_M p$ modchkCTL* returns true iff $s \models_M p$.*

Proof. Follows from Corollary 4.3 by a simple induction on the structure of p. □

Lemma 5.2 *The size of the set of assertions V visited by modchkCTL* is bounded by $2^{O(|p|)} * M$. If p is a CTL formula then the size of V is bounded by $|p| * |M|$.*

Proof. The first part of the lemma can be proved using Lemma 3.7 and induction on the structure of the p. The second part of the lemma also follows by an induction on p. □

Clearly, modchkCTL* visits each assertion in V exactly once. We also know that the work done at each assertion is bounded by $|p|$. It follows from Lemma 5.2 that the time complexity of modchkCTL*is in $O(|p| * 2^{O(|p|)} * |M|)$. Also note that when p is a CTL formula then there is at most one top-level $\bigvee$ formula. This implies that work done at each assertion by modchkLTL (and hence modchkCTL*) is constant. From Lemma 5.2 it follows that when p is a CTL formula modchkCTL* takes $O(|p| * |M|)$ time. This matches the time complexity of the best model-checking algorithms for CTL [7, 26].

6 Conclusions

In this paper we presented an on-the-fly framework for model checking the temporal logics LTL, CTL and CTL*. Our algorithm is uniformly efficient in the sense that when applied to any of these logics, the time complexity of the algorithm matches the complexity of the best known algorithms for these logics. We are in the process of implementing this algorithm in the Concurrency Workbench [11], a tool for verifying finite-state concurrent systems. As future work, we would also like to investigate if the algorithm can be extended to ECTL* [10, 23, 24].

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, 1974.

[2] H.R. Andersen. Model checking and boolean graphs. In *Proceedings of the European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 1–19, Rennes, France, March 1992. Springer-Verlag.

[3] O. Bernholtz, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In Dill [14], pages 142–155.

[4] M.C. Browne, E.M. Clarke, D. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computing*, C-35(12):1035–1044, December 1986.

[5] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[6] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for verification of temporal properties. *Formal Methods in System design*, 1:275–288, 1992.

[7] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[8] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In Dill [14], pages 415–427.

[9] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the futurebus+ cache coherence protocol. In L. Claesen, editor, *11th International Symposium on Computer Hardware Description Languages and their Applications.* North-Holland, April 1993.

[10] E.M. Clarke, O. Grumberg, and R.P. Kurshan. A synthesis of two approaches for verifying finite state concurrent systems. *Manuscript, Carnegie-Mellon University*, 1987.

[11] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[12] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2:121–147, 1993.

[13] M. Dam. CTL* and ECTL* as fragments of the modal mu-calculus. In *Proceedings of the Colloquium on Trees and Algebra in Programming*, volume 581 of *Lecture Notes in Computer Science*, pages 145–164. Springer-Verlag, February 1992.

[14] D.L. Dill, editor. *Computer Aided Verification (CAV '93)*, volume 818 of *Lecture Notes in Computer Science*, Stanford, California, June 1994. Springer-Verlag.

[15] E.A. Emerson and J.Y. Halpern. 'Sometime' and 'not never' revisited: On branching versus linear time temporal logic. *Journal of the Association for Computing Machinery*, 33(1):151–178, 1986.

[16] E.A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8:275–306, 1987.

[17] E.A. Emerson and A.P. Sistla. Deciding full branching time logic. *Information and Control*, 61:175–201, 1984.

[18] C. Jard and T. Jeron. On-line model checking for finite linear temporal logic specifications. In *Automatic Verification Methods for Finite State Systems, Proc. Int. Workshop, Grnoble*, volume 407, pages 189–196. Springer-Verlag, Lecture Notes in Computer Science, June 1989.

[19] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Twelfth Annual ACM Symposium on Principles of Programming Languages (PoPL '85)*, pages 97–107, Orlando, Florida, January 1985. Computer Society Press.

[20] K.L. McMillan. *Symbolic Model Checking: An Approach to the State-Explosion Problem.* PhD thesis, Carnegie-Mellon University, 1992.

[21] K.L. McMillan and J. Schwalbe. *Formal Verification of the Gigamax Cache Consistency Protocol.* MIT Press, 1992.

[22] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th Int'l Symp. on Programming*, volume 137, pages 337–351. Springer-Verlag, Lecture Notes in Computer Science, 1981.

[23] W. Thomas. Computation tree logic and regular ω-languages. *Lecture Notes in Computer Science*, 354:690–713, 1988.

[24] M.Y. Vardi and P. Wolper. Yet another process logic. *Lecture Notes in Computer Science*, 164:501–512, 1984.

[25] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science (LICS '86)*, pages 332–344, Cambridge, Massachusetts, June 1986. Computer Society Press.

[26] B. Vergauwen and J. Lewi. A linear local model-checking algorithm for CTL. In E. Best, editor, *CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 447–461, Hildesheim, Germany, August 1993. Springer-Verlag.

A Appendix

This appendix contains proofs of some of the theorems in this paper.

Proof of Theorem 3.4

This theorem is a consequence of Lemmas A.3 and A.4, which appear below. We first introduce the following definitions.

Definition A.1 *Let $M = \langle S, R, L\rangle$ be a Kripke structure and $\langle V, E\rangle$ be a proof structure for $s \vdash_M A\Phi$. Also let $x = \langle \sigma_0, \ldots\rangle$ be a (finite or infinite) path in $\langle V, E\rangle$.*

1. *If σ is an assertion, then define $s(\sigma) \in S$ to be the state s such that $\sigma \equiv s \vdash_M A\Phi$.*

2. *Rule R* appears *at position i in x if $A = \{\, \sigma \mid \langle x(i), \sigma\rangle \in E\,\}$ is the result of applying R to $x(i)$ and $x(i+1) \in A$.*

3. *$P_M(x) = \langle s_0, \ldots\rangle$ is a sequence of elements from S given as follows. Define a sequence y over $S \cup \{\bullet\}$ from x by:*

$$y(i) = \begin{cases} s(x(0)) & \textit{if } i = 0 \\ s(x(i)) & \textit{if } i > 0 \textit{ and R7 appears} \\ & \textit{at position } i-1. \\ \bullet & \textit{otherwise} \end{cases}$$

Then $P_M(x)$ is the subsequence of y obtained by deleting all occurrences of $\bullet$.

We have the following easy facts about proof structures.

Lemma A.2 *Let $M = \langle S, R, L\rangle$ be a Kripke structure and $\langle V, E\rangle$ be a proof structure for $\sigma \equiv s \vdash_M A\Phi$, with x a path through $\langle V, E\rangle$.*

1. *$P_M(x)$ is a path in M.*

2. *x is infinite iff R7 appears at infinitely many positions in x.*

3. *x is infinite iff $P_M(x)$ is infinite.*

4. *Let $i, j \geq 0$ be such that $j \geq i$, and assume that for all k such that $i \leq k \leq j$, rule R7 does not appear at location k in x. Then $P_M(x^i) = P_M(x^j)$.*

We may now prove one direction of the theorem.

Lemma A.3 *Let M be a Kripke structure with $s \in S$ and Φ be such that $A(\bigvee_{\phi\in\Phi} \phi)$ is a LTL formula with $s \models_M A\Phi$. Also let $\langle V, E\rangle$ be a proof structure for $\sigma \equiv s \vdash_M A\Phi$. Then $\langle V, E\rangle$ is successful.*

Proof. By way of contradiction, assume that $\langle V, E\rangle$ is not successful. There are two cases to consider. First, suppose that $\langle V, E\rangle$ is not partially successful. In this case the result follows from Lemma 3.1. Alternatively, suppose $\langle V, E\rangle$ is partially successful. Then there exists an infinite path x in $\langle V, E\rangle$ such that $x(0) = \sigma$ and x is unsuccessful. We wish to show that $P_M(x) \not\models_M \bigvee \Phi$; to do so we prove the following stronger result.

For any $\phi, i \geq 0$ such that $\phi \in x(i)$, $P_M(x^i) \not\models \phi$.

The proof proceeds by induction on the structure of ϕ. The induction hypothesis states that for any subformula ϕ' of ϕ and $i \geq 0$ such that $\phi' \in x(i)$, $P_M(x^i) \not\models \phi'$. Fix $i \geq 0$; we now perform a case analysis on the form of ϕ. Most cases are routine; we only consider the following in detail.

- $\phi \in \mathcal{L}$. Since x is infinite, Lemma A.2 ensures the existence of a smallest $j \geq i$ such that rule $R7$ appears at location j in x. Moreover, as ϕ is not of the form $\mathsf{X}\phi'$ it follows that $j > i$ and that $\phi \notin x(j)$. The only rules that can remove atomic propositions are $R1$ and $R2$, and since x is infinite it follows that $R2$ must appear at a location k between i and j in x. As Lemma A.2 ensures that $s(x(i)) = s(x(k))$ it follows that $s(x(i)) \not\models_M \phi$, and thus $P_M(x^i) \not\models_M \phi$.

- $\phi \equiv \phi_1 \wedge \phi_2$. The infiniteness of x ensures the existence of a least $j > i$ such that $R7$ appears at position j in x. Moreover, as ϕ is not of the form $\mathsf{X}\phi'$, there must exist a k such that $i \leq k < j$ and $R4$ appears at position k in x. So $x(k)$ must have form $s(x(i)) \vdash_M A\Phi_k$, where either ϕ_1 or ϕ_2 is in Φ_k. In either case we may use the induction hypothesis together with Lemma A.2 to infer that $P_M(x^i) \not\models \phi_i$, and thus $P_M(x^i) \not\models \phi$.

- $\phi \equiv \phi_1 \mathsf{U} \phi_2$. There are two possibilities to consider. In the first, suppose there is a least $j \geq i$ such that $\{\phi_1, \phi_2\} \subseteq x(j)$. Let $k_0 = i < k_1 < \ldots < k_m < k_{m+1} = j$, $m \geq 0$ be such that the k_n, $1 \leq n \leq m$ represent the positions at which $R7$ appears on x between i and j. From the induction hypothesis, it follows that $P_M(x^j) \not\models_M \phi_1$ and $P_M(x^j) \not\models_M \phi_2$. Now, between k_l and k_{l+1}, $0 \leq l \leq m-1$, there is a position n_l such that $\{\phi_2, \mathsf{X}\phi\} \subseteq x(n_l)$; this is a result of the following line of reasoning. First, note that $\phi \in x(k_0)$. If $m = 0$ the result follows; otherwise, k_1 represents one of the positions mentioned above at which $R7$ appears. Now, since ϕ is not of the form $\mathsf{X}\phi'$, it follows that $R5$ must appear at a position p_0 between k_0 and k_1, and as $\{\phi_1, \phi_2\} \not\subseteq x(p_0+1)$, it follows from the form of $R5$ that $\{\phi_2, \mathsf{X}\phi\} \subseteq x(p_0+1)$, and hence $\phi \in x(k_1+1)$. The observation now follows inductively. Using the induction hypothesis and Lemma A.2, then, we have that $P_M(x^{k_l}) \not\models_M \phi_2$. Let $y = P_M(x^i)$. We have demonstrated the existence of an r such that $y^r = P_M(x^j) \not\models_M \phi_2$ and for all q, $0 \leq q \leq r$, $y^q \not\models_M \phi_1$. Therefore, by the semantics of U it follows that $y = P_M(x^i) \not\models \phi$.

  In the second case, assume that no $j \geq i$ exists with $\{\phi_1, \phi_2\} \subseteq x(j)$. Using an argument similar to the one above, one may show that no k exists such that $P_M(x^i)^k \models \phi_2$. Then the semantics of U guarantee that $P_M(x^i) \not\models \phi$.

- $\phi \equiv \phi_1 \mathsf{V} \phi_2$. Because x is unsuccessful we know that there is a least $j \geq i$ such that $\phi_2 \in x(j)$. If $P_M(x^i) = P_M(x^j)$ then we may use the induction hypothesis to infer that $P_M(x^i) \not\models \phi_2$, and hence $P_M(x^i) \not\models \phi_1 \mathsf{V} \phi_2$ by the definition of V. Alternatively, suppose that $P_M(x^i) \neq P_M(x^j)$. We show that for all k such that $i \leq k < j$, $P_M(x^k) \not\models \phi_1$; this, plus the fact that the induction hypothesis ensures that $P_M(x^j) \not\models \phi_2$, guarantees that $P_M(x^i) \not\models \phi$. Since $P_M(x^i) \neq P_M(x^j)$, $R7$ must appear at some positive number of positions between i and j, and as ϕ is not of the form $\mathsf{X}\phi'$ it follows that before each of these appearances of $R7$, $R6$ must appear. Consider one such position k in which $R6$ appears. Since $\phi_2 \notin s(x(k))$ (as $k < j$), it must be that $\phi_1 \in s(x(k))$. Therefore, by the induction hypothesis, it follows that $P_M(x^k) \not\models_M \phi_1$. The desired result now follows from the fact that between appearances of $R7$, Lemma A.2 ensures that the state component of assertions on x cannot change.

$\square$

In order to prove the other direction of the theorem, we first introduce some terminology. Let x be a path in proof structure $\langle V, E\rangle$; we say that x *fails* (or is a failed path) if either x is finite and the last assertion in x is an unsuccessful leaf in $\langle V, E\rangle$ or x is infinite and unsuccessful. It follows that a proof structure is unsuccessful iff it contains a failed path. We now prove the following.

Lemma A.4 *Let M be a Kripke structure with $s \in S$ and $A\Phi$ such that $A(\bigvee_{\phi \in \Phi} \phi)$ is an LTL formula, and let $\langle V, E\rangle$ be a successful proof structure for $\sigma \equiv s \vdash_M A\Phi$. Then $s \models_M A\Phi$.*

Proof. The proof proceeds by contradiction. So assume that $s \not\models_M A\Phi$; this implies the existence of a path $x \in \Pi_M(s)$ such that $x \not\models_M \phi$ for every $\phi \in \Phi$. We will now show that $\langle V, E\rangle$ contains a failed path y such that $y(0) = \sigma$ and such that one of the following holds: either y is finite and $P_M(y)$ is a prefix of x; or y is infinite and $P_M(y) = x$. Thus $\langle V, E\rangle$ will turn out to be unsuccessful, which contradicts our assumption that it is successful.

We construct y recursively in such a way that the following invariants are maintained for all i such that $0 \leq i < |y|$.

1. $P_M(y)$ is a prefix of x.
2. Let $f_y(i)$ be such that $P_M(\langle y(0), \ldots, y(i)\rangle) = \langle s_0, \ldots, s_{f_y(i)}\rangle$. Then for all $\phi \in y(i)$, $x^{f_y(i)} \not\models_M \phi$.

To begin with, take $y = \langle \sigma \rangle$; so $y(0) = \sigma$. Since $x^{f_y(0)} = x$, properties (1) and (2) follow by assumption. Now assume that we have constructed y so that $|y| = i+1$ for some $i \geq 0$; we want to determine if y should be extended to be of length $i+2$. There are two cases. If $y(i)$, the last element in y, is a leaf of $\langle V, E\rangle$, then y is complete. Otherwise, a rule R appears at position i in y. The proof proceeds by a case analysis on R; we consider some of the possibilities here. Note that R cannot be $R1$, since this would imply that $x^{f_y(i)} \models \phi$ for the literal ϕ used in the rule. Let $y(i) \equiv s_i \vdash_M A\Phi_i$.

- $R \equiv R4$. In this case, there is a $\phi \in y(i)$ such that $\phi \equiv \phi_1 \wedge \phi_2$ for some ϕ_1, ϕ_2, and $y(i)$ has two successors in $\langle V, E\rangle$: $s_i \vdash_M A((\Phi_i - \{\phi\}) \cup \{\phi_1\})$ and $s_i \vdash_M A((\Phi_i - \{\phi\}) \cup \{\phi_2\})$. Since we know that $x^{f_y(i)} \not\models \phi$ it follows that $x^{f_y(i)} \not\models \phi_j$ for $j = 1$ or 2; choose $y(i+1) = s_i \vdash_M A((\Phi_i - \{\phi\}) \cup \{\phi_j\})$. It is easy to establish that properties (1) and (2) are maintained.

- $R \equiv R5$. So there is a $\phi \in y(i)$ such that $\phi \equiv \phi_1 \mathsf{U} \phi_2$ for some ϕ_1, ϕ_2, and $y(i)$ has two successors in $\langle V, E\rangle$: $s_i \vdash_M A((\Phi_i - \{\phi\}) \cup \{\phi_1, \phi_2\})$ and $s_i \vdash_M A((\Phi_i - \{\phi\}) \cup \{\phi_2, \mathsf{X}\phi\})$. From the semantics of U and the fact that $x^{f_y(i)} \not\models_M \phi$, it follows either that $x^{f_y(i)} \not\models_M \phi_i$ for $i = 1$ and 2, in which case we should choose the former of the above assertions to be $y(i+1)$, or that $x^{f_y(i)} \models_M \phi_1$ but $x^{f_y(i)} \not\models_M \mathsf{X}\phi$ and $x^{f_y(i)} \not\models_M \phi_2$, in which case we should take $y(i+1)$ to be the latter. In either case, both invariants are maintained.
- $R \equiv R6$. Here there is a $\phi \in y(i)$ such that $\phi \equiv \phi_1 \mathsf{V} \phi_2$ for some ϕ_1, ϕ_2, and $y(i)$ has two successors in $\langle V, E\rangle$: $s_i \vdash_M A((\Phi_i - \{\phi\}) \cup \{\phi_2\})$ and $s_i \vdash_M A((\Phi_i - \{\phi\}) \cup \{\phi_1, \mathsf{X}\phi\})$. From the semantics of V and the fact that $x^{f_y(i)} \not\models_M \phi$, it follows either that $x^{f_y(i)} \not\models_M \phi_1$, in which case we should choose the former of the the above assertions to be $y(i+1)$, or that $x^{f_y(i)} \models_M \phi_2$ but $x^{f_y(i)} \not\models_M \mathsf{X}\phi$ and $x^{f_y(i)} \not\models_M \phi_1$, in which case we should take $y(i+1)$ to be the latter. Again, both invariants are trivially satisfied.
- $R \equiv R7$. In this case $\Phi_i = \{\mathsf{X}\phi_1, \ldots, \mathsf{X}\phi_n\}$ for some $\phi_1, \ldots, \phi_n$, and $y(i)$ has as successors in $\langle V, E\rangle$ assertions $s_1 \vdash_M A\{\phi_1, \ldots, \phi_n\}$, $\ldots s_m \vdash_M A\{\phi_1, \ldots, \phi_n\}$, where $\{s_1, \ldots, s_m\} = \{\, s' \mid \langle s, s'\rangle \in R\,\}$. Consider the state $x(f_y(i)+1)$. Since $x^{f_y(i)} \not\models \mathsf{X}\phi_j$ for $1 \leq j \leq n$, it follows that $x^{(f_y(i)+1)} \not\models \phi_j$ for $1 \leq j \leq n$. So take as $y(i+1)$ the assertion $s_j \vdash_M A\{\phi_1, \ldots, \phi_n\}$ such that $s_j = x(f_y(i)+1)$. Again, the invariants hold.

There are now two cases to consider. In the first, $|y| < \infty$. As the invariant maintained by the construction guarantees that the last element of y is an unsuccessful leaf, y fails. In the second, y is infinite; in this case we must show that y is unsuccessful. Let $i \geq 0$ be such that $\phi_1 \mathsf{V} \phi_2 \in y(i)$ for some ϕ_1, ϕ_2; we must show that for some $j \geq i$, $\phi_2 \in y(j)$. From the way y is constructed, we know that $x^{f_y(i)} \not\models \phi_1 \mathsf{V} \phi_2$, and from the semantics of V it follows that there is a $j \geq f_y(i)$ such that $x^j \not\models \phi_2$ and for all k such that $f_y(i) \leq k < j$, $x^k \not\models \phi_1$. From the construction of y, it follows there is a $k \geq i$ such that $P_M(y^k) = x^j$ and $\phi_2 \in y(k)$. □

Proof of Theorem 3.5

This theorem is a consequence of the following result.

Lemma A.5 *Let $\langle V, E\rangle$ be a proof structure, and let $S \subseteq V$ be a nontrivial strongly connected component. Then S is successful iff every path in S is successful.*

Proof. For the "if" direction, assume that S unsuccessful. In this case we adopt the following strategy to construct an unsuccessful path through S. We start from an arbitrary assertion in S. Since S is unsuccessful, at any assertion σ if there exist ϕ_1, ϕ_2 such that $\phi_1 \mathsf{V} \phi_2 \in \sigma$, then there exists an assertion σ' such that $\phi_2 \in \sigma'$. So in our path construction, if we are at σ then we extend our path by concatenating the path from σ to σ'. It is not hard to show that this path is unsuccessful.

For the "only if" direction, assume that S is successful, and consider an arbitrary infinite path x in S. Since S is successful there exist $\sigma \in S$ and ϕ_1, ϕ_2 such that $\phi_1 \mathsf{V} \phi_2 \in \sigma$ and for every $\sigma' \in S$ $\phi_2 \notin \sigma'$. So all we need to prove is that $\phi_1 \mathsf{V} \phi_2 \in x(i)$ for some $i \geq 0$. Now there exists a path in S from σ to $x(i)$ for some $i \geq 0$, and since ϕ_2 cannot appear in any assertions on this path, it follows that either $\phi_1 \mathsf{V} \phi_2$ or $\mathsf{X}(\phi_1 \mathsf{V} \phi_2) \in x(i)$. In the former case, we are done. Otherwise, let $j \geq i$ be the least number such that $R7$ appears at position j in x; Lemma A.2 guarantees j's existence. It is not hard to show that $\mathsf{X}(\phi_1 \mathsf{V} \phi_2) \in x(j)$, and hence $\phi_1 \mathsf{V} \phi_2 \in x(j+1)$. □

Proof of Lemma 4.1

Proof. We must prove that every time a vertex or an edge is added to G, the resulting graph maintains the invariant. Suppose a vertex is added (by execution of the statement "V := $V \cup \{\sigma\}$"). The strongly connected components of G in this case remain unchanged, and a new component $\{\sigma\}$ is added. From the way σ.valid is initialized, it follows that the invariant is maintained.

Now suppose that an edge $\langle \sigma, \sigma'\rangle$ is added to G. Such an addition can change the strongly connected components of G if σ' is in the stack. In this case a new strongly connected component S' appears in the altered graph, which we refer to as G'. Since $h(S') = \sigma$, we need to prove that $Success(S') = \sigma.valid$. To this end, suppose that $\phi \equiv \phi_1 \mathsf{V} \phi_2 \in Success(S')$; we must show that $\phi \in \sigma$.valid. Let σ_l be the vertex with lowest depth-first search number d in S'. Clearly $\langle \phi, sp\rangle \in \sigma_l$.valid for some $sp \leq d$. From the structure of the algorithm and from the fact that $\phi \in Success(S')$ it can be shown that $\langle \phi, sp\rangle \in \sigma''.valid$ for all $\sigma'' \in S'$, including σ. Since sp is less than the depth-first search number of σ', it follows that $\langle \phi, sp\rangle$ will not be removed from σ.valid.

Now assume that $\phi \equiv \phi_1 \mathsf{V} \phi_2 \notin Success(S')$; we must show that $\phi \notin \sigma$.valid. Let σ_2 be the lowest common ancestor of σ and σ' in the depth-first search tree for G', and let π be the path in this tree from σ_2 to σ, inclusive. Also let SC_G be the set of strongly connected components in G and $\mathcal{S} = \{\, S \in SC_G \mid \exists \sigma'' \in \pi.\ \sigma'' \in S\,\}$. Since the invariant was true for the G, we know that for every $S \in \mathcal{S}$, $h(S).valid := Success(S)$. It is straightforward to show that $S' = \bigcup_{S \in \mathcal{S}} S$ (note that this includes σ and σ_2; the latter is correct, since σ' remains in the stack when σ is explored). Now, since $\phi \notin Success(S')$, there exists $S_2 \in \mathcal{S}$ such that $\phi \notin Success(S_2)$. This latter fact means that $\phi \notin h(S_2).valid$. From the structure of the algorithm, we also know that for some $\sigma_1 \in S_2 \cap \pi$, σ_1.valid $=$ $h(S_2)$.valid. So if $\langle \phi, sp\rangle \in \sigma$.valid then sp is equal to the dfs number of some assertion σ_3 that appears after σ_1 on π. Now, σ_3.dfsn $> \sigma'$.dfsn. Hence, $sp > \sigma'$.dfsn. Therefore, ϕ will be eliminated from σ.valid. □