Using BDDs to Decide CTL

Will Marrero
DePaul Univeristy
wmarrero@cs.depaul.edu

Abstract

Computation Tree Logic (CTL) has been used quite extensively and successfully to reason about finite state systems. Algorithms have been developed for checking if a particular model satisfies a CTL formula (model checking) as well as for deciding if a CTL formula is valid or satisfiable. Initially, these algorithms explicitly constructed the model being checked or the model demonstrating satisfiability. A major breakthrough in CTL model checking occurred when researchers started representing the model implicitly via Boolean formulas. The use of ordered binary decision diagrams (OBDDs) as an efficient representation for these formulas led to a large jump in the size of the models that can checked. This paper presents a way to encode the satisfiability algorithms for CTL in terms of Boolean formulas as well, so that symbolic model checking techniques using OBDDs can be exploited.
Keywords: CTL, satisfiability, validity, BDDs, tableau

1 Introduction

Temporal logic has been used quite extensively and successfully to reason about finite state systems, including both hardware and software systems. While there are different logics to choose from, this discussion focuses on Computation Tree Logic (CTL) proposed by Clarke and Emerson [4]. (For a survey of various temporal logics, see Chapter 16 of *Handbook of Theoretical Computer Science, Volume B* [8].)

Initial efforts with CTL focused on algorithms for checking if a particular structure satisfies a formula (model checking) as well as algorithms for checking if there exists a structure that satisfies the formula (satisfiability) [4]. These algorithms required the explicit construction of a finite-state transition system either to check that it satisfies the formula or in an attempt to prove that the formula is satisfiable. In general, the size of the finite-state transition system is exponential in the number of atomic propositions in the case of model checking and exponential in the size of the formula in the case of satisfiability. This placed a severe limitation on the size of the problems that could be handled by these algorithms.

A major breakthrough occurred when researchers started using boolean formulas to represent the transition relation of finite-state system as well as sets of states in the system implicitly. This technique, called symbolic model checking, avoids explicitly constructing the graph for the system [2, 5]. The key was to use Ordered Binary Decision Diagrams (OBDDs), which are a canonical representation for quantified boolean formulas [1]. This allowed researchers to verify systems with more than 10^{20} states [2].

While much effort continued to be focused on symbolic model checking, very little effort was placed on using these symbolic techniques in the area of CTL satisfiability checking. In particular, the algorithms for CTL satisfiability rely on the construction of an explicit model for the formula in question either by constructing a *tableau* [4] or by constructing a *Hintikka structure* [9]. This paper presents a satisfiability algorithm for CTL which uses OBDDs to implicitly construct a model satisfying the formula. This work depends heavily on the explicit-state Hintikka structure algorithm presented in [9] and is inspired by a similar use of OBDDs for LTL satisfiability and model checking presented in [2, 3].

2 Syntax and Semantics

2.1 Syntax

We provide the syntax and semantics of CTL in a slightly non-standard way which will be useful later when describing the algorithm. The set of well formed formulas (hereafter shortened to formulas) are defined inductively as follows:

- If p is an atomic proposition, then p is a formula.
- If f is a formula, then so is $\neg f$.
- If f and g are formulas, then so are $f \wedge g$ and $f \vee g$.
- If f is a formula, then so are $\mathbf{EX}f$ and $\mathbf{AX}f$.
- If f and g are formulas, then so are $\mathbf{E}[f\ \mathbf{U}\ g]$, $\mathbf{A}[f\ \mathbf{U}\ g]$, $\mathbf{E}[f\ \mathbf{R}\ g]$, and $\mathbf{A}[f\ \mathbf{R}\ g]$.

We will also use the following common abbreviations: $f \rightarrow g$ for $\neg f \vee g$, $p \leftrightarrow q$ for $(f \rightarrow g \wedge g \rightarrow f)$, $\mathbf{EF}f$ for $\mathbf{E}[\mathbf{tt}\ \mathbf{U}\ f]$, $\mathbf{AF}f$ for $\mathbf{A}[\mathbf{tt}\ \mathbf{U}\ f]$, $\mathbf{EG}f$ for $\mathbf{E}[\mathbf{ff}\ \mathbf{R}\ f]$, and $\mathbf{AG}f$ for $\mathbf{A}[\mathbf{ff}\ \mathbf{R}\ f]$.

2.2 Structures

CTL formulas are interpreted over Kripke structures. A Kripke structure $M = (S, L, R)$ consists of

- S - a set of states
- L - a labeling of each state with formulas true in the state
- $R \subseteq S \times S$ - a transition relation

Note that the transition relation R is required to be *total* which means every state has a successor. In other words $\forall s \in S\ .\ \exists s' \in S\ .\ R(s, s')$). A *path*, $\pi = s_0, s_1, s_2, \ldots$ is an infinite sequence of states such that $(s_i, s_{i+1}) \in R$ for all $i \geq 0$.

2.3 Semantics

The truth or falsity of a formula f in a state s of a structure $M = (S, L, R)$ is given by the inductively defined relation $\models$. In the definition below, p is an atomic proposition while f and g are arbitrary formulas.

- $M, s \models p$ iff $p \in L(s)$.
- $M, s \models \neg f$ iff $M, s \not\models f$.
- $M, s \models f \wedge g$ iff $M, s \models f$ and $M, s \models f$.
- $M, s \models f \vee g$ iff $M, s \models f$ or $M, s \models f$.
- $M, s \models \mathbf{EX}f$ iff $\exists s' \,.\, R(s, s') \in R$ and $M, s' \models f$.
- $M, s \models \mathbf{AX}f$ iff $\forall s' \,.\, (s, s') \in R$ implies $M, s' \models f$.
- $M, s \models \mathbf{E}[f \ \mathbf{U} \ g]$ iff there exists a path $s_0, s_1, s_2, \ldots$ where $s_0 = s$ and there exists k such that $M, s_k \models g$ and $M, s_i \models f$ for all $0 \leq i < k$.
- $M, s \models \mathbf{A}[f \ \mathbf{U} \ g]$ iff for all paths $s_0, s_1, s_2, \ldots$ where $s_0 = s$, there exists k such that $M, s_k \models g$ and $M, s_i \models f$ for all $0 \leq i < k$.
- $M, s \models \mathbf{E}[f \ \mathbf{R} \ g]$ iff there exists a path $s_0, s_1, s_2, \ldots$ where $s_0 = s$, such that for all $k > 0$, if $M, s_i \not\models f$ for all $0 \leq i < k$, then $M, s_k \models g$
- $M, s \models \mathbf{A}[f \ \mathbf{R} \ g]$ iff for all paths $s_0, s_1, s_2, \ldots$ where $s_0 = s$, and for all $k > 0$, if $M, s_i \not\models f$ for all $0 \leq i < k$, then $M, s_k \models g$

Note that we have the following dualities:

- $\neg(f \wedge g) \equiv \neg f \vee \neg g$
- $\neg \mathbf{EX} f \equiv \mathbf{AX} \neg f$
- $\neg \mathbf{E}[f \ \mathbf{U} \ g] \equiv \mathbf{A}[\neg f \ \mathbf{R} \ \neg g]$
- $\neg \mathbf{E}[f \ \mathbf{R} \ g] \equiv \mathbf{A}[\neg f \ \mathbf{U} \ \neg g]$

Also note the following semantic identities which will be used when trying to construct a model that satisfies a formula.

- $\mathbf{E}[f \ \mathbf{U} \ g] \equiv g \vee (f \wedge \mathbf{EXE}[f \ \mathbf{U} \ g])$
- $\mathbf{A}[f \ \mathbf{U} \ g] \equiv g \vee (f \wedge \mathbf{AXA}[f \ \mathbf{U} \ g])$
- $\mathbf{E}[f \ \mathbf{R} \ g] \equiv g \wedge (f \vee \mathbf{EXE}[f \ \mathbf{R} \ g])$
- $\mathbf{A}[f \ \mathbf{R} \ g] \equiv g \wedge (f \vee \mathbf{AXA}[f \ \mathbf{R} \ g])$

The modalities **EU** and **AU** are the until operator. For example, $\mathbf{E}[f \ \mathbf{U} \ g]$ is interpreted to mean there is a path on which g eventually holds and on which f holds *until* g holds. The abbreviation $\mathbf{EF}f$ ($\mathbf{AF}f$) is interpreted to mean that there is a path (on all paths) f eventually holds at some point in the future while the abbreviation $\mathbf{EG}f$ ($\mathbf{AG}f$) means that there is a path (on all paths) f holds globally. The modalities **ER** and **AR** are not as common nor as intuitive as the other modalities. The modality $\mathbf{R}$ is often translated as "release". $\mathbf{E}[f \ \mathbf{R} \ g]$ can be understood to mean that f releases g in the sense that along some path g is required to be true unless f becomes true in which case g is no longer required to be true after that state. In other words, $\mathbf{E}[f \ \mathbf{R} \ g]$ has the same meaning as $\mathbf{EG}g \vee \mathbf{E}[g \ \mathbf{U} \ f \wedge g]$. The importance of **ER** and **AR** is that they are the duals of the until operator and they can be used to define **EG** and **AG**. The **R** modality was introduced in [6] (although **V** was used instead of the currently popular **R**) precisely because a dual for the until operator was required.

3 Hintikka Structures

We now proceed to describe the algorithm for trying to construct a satisfying model for a formula f given in [8]. First, we assume that the formula f is in *negation normal form*, which means all negations are pushed inward as far as possible using the dualities in Section 2.3. For any formula g, we will use $\sim g$ to represent the formula $\neg g$ after being converted into negation normal form.

The *closure* of a formula f, denoted $cl(f)$, is the smallest set of formulas such that

- Every subformula of f is a member of $cl(f)$.
- If $\mathbf{E}[f \ \mathbf{U} \ g] \in cl(f)$ or $\mathbf{E}[f \ \mathbf{R} \ g] \in cl(f)$, then, $\mathbf{EXE}[f \ \mathbf{U} \ g] \in cl(f)$ or $\mathbf{EXE}[f \ \mathbf{R} \ g] \in cl(f)$ respectively.
- If $\mathbf{A}[f \ \mathbf{U} \ g] \in cl(f)$ or $\mathbf{A}[f \ \mathbf{R} \ g] \in cl(f)$, then, $\mathbf{AXA}[f \ \mathbf{U} \ g] \in cl(f)$ or $\mathbf{AXA}[f \ \mathbf{R} \ g] \in cl(f)$ respectively.

For example,

$$cl(\mathbf{AF}p \wedge \mathbf{EX}q) = \{\mathbf{AF}p \wedge \mathbf{EX}q, \mathbf{AF}p, \mathbf{AXAF}p, p, \mathbf{EX}q, q\}$$

We will use $\phi = \mathbf{AF}p \wedge \mathbf{EX}q$ as a running example. The *extended closure* of f, denoted $ecl(f)$, is defined to be $cl(f) \cup \{ \sim g \mid g \in cl(f) \}$. For example,

$$ecl(\phi) = cl(\phi) \cup \{\mathbf{EG}\neg p \vee \mathbf{AX}\neg q, \mathbf{EG}\neg p, \mathbf{EXEG}\neg p, \neg p, \mathbf{AX}\neg q, \neg q\}$$

An *elementary* formula is one which is either a literal, a negated literal, or a formula whose main connective is $\mathbf{EX}$ or $\mathbf{AX}$. We will use $el(f)$ to denote the elementary formulas of f (the members of $ecl(f)$ that are elementary). Any other formula is said to be *nonelementary*. For example,

$$el(\phi) = \{\mathbf{AXAF}p, p, \mathbf{EX}q, q, \mathbf{EXEG}\neg p, \neg p, \mathbf{AX}\neg q, \neg q\}$$

Table 1: Classification of nonelementary formulas

$\alpha = f \wedge g$	$\alpha_1 = f$	$\alpha_2 = g$
$\alpha = \mathbf{E}[f\ \mathbf{R}\ g]$	$\alpha_1 = g$	$\alpha_2 = f \vee \mathbf{EXE}[f\ \mathbf{R}\ g]$
$\alpha = \mathbf{A}[f\ \mathbf{R}\ g]$	$\alpha_1 = g$	$\alpha_2 = f \vee \mathbf{AXA}[f\ \mathbf{R}\ g]$
$\beta = f \vee g$	$\beta_1 = f$	$\beta_2 = g$
$\beta = \mathbf{E}[f\ \mathbf{U}\ g]$	$\beta_1 = g$	$\beta_2 = f \wedge \mathbf{EXE}[f\ \mathbf{U}\ g]$
$\beta = \mathbf{A}[f\ \mathbf{U}\ g]$	$\beta_1 = g$	$\beta_2 = f \wedge \mathbf{AXA}[f\ \mathbf{U}\ g]$

Recall that arbitrary negation is not allowed, so all nonelementary formulas have a binary main connective. By using the semantic identities from Section 2.3, every nonelementary formula can be viewed as either a conjunctive formula $\alpha \equiv \alpha_1 \wedge \alpha_2$ or as a disjunctive formula $\beta \equiv \beta_1 \vee \beta_2$. Table 1 contains the classifications for all nonelementary formulas.

The model we will try to build for the formula f will have states labeled with formulas in *ecl*(f). We will need to impose certain consistency requirements when constructing the model:

- *Propositional Consistency Rules:*

  (PC0) $\sim\!p \in L(s)$ implies $p \notin L(s)$.

  (PC1) $\alpha \in L(s)$ implies $\alpha_1 \in L(s)$ and $\alpha_2 \in L(s)$.

  (PC2) $\beta \in L(s)$ implies $\beta_1 \in L(s)$ or $\beta_2 \in L(s)$.

  (PC3) $\mathbf{tt} \in L(s)$

  (PC4) $\mathbf{ff} \notin L(s)$

- *Local Consistency Rules:*

  (LC0) $\mathbf{AX}f \in L(s)$ implies that for every successor t of s, $f \in L(t)$.

  (LC1) $\mathbf{EX}f \in L(s)$ implies that there is a successor t of s, such that $f \in L(t)$.

A *fragment* is a triple $(\hat{S}, \hat{R}, \hat{L})$. It is similar to a structure, except that $\hat{R}$ need not be total. In fact, the fragments we choose will be directed acyclic graphs contained within a particular structure $M = (S, R, L)$ that is under consideration. So $\hat{S} \subseteq S$, $\hat{R} \subseteq R$, and $\hat{L} = L|_{\hat{S}}$. In addition, all nodes in a fragment satisfy rules **PC0-PC4** and **LC0** above, and all interior nodes also satisfy **LC1**.

It turns out that we do not have to construct the full model for a formula f to determine satisfiability. Instead, we will construct a *pseudo-Hintikka structure* for f. A pseudo-Hintikka structure for f is a structure $M = (S, R, L)$ where:

1. $f \in L(s)$ for some state $s \in S$.
2. All states satisfy the consistency rules **PC0-PC4** and **LC0-LC1**.
3. All eventualities are *pseudo-fulfilled* as follows:

- $\mathbf{A}[f \ \mathbf{U} \ g] \in L(s)$ implies there is a fragment contained in M and rooted at s such that for all frontier nodes t, $g \in L(t)$ and for all interior nodes u, $f \in L(u)$.
- $\mathbf{E}[f \ \mathbf{U} \ g] \in L(s)$ implies there is a fragment contained in M and rooted at s such that for some frontier node t, $g \in L(t)$ and for all interior nodes u, $f \in L(u)$.

In [8], Emerson proves that a formula f is satisfiable if and only if there is a finite pseudo-Hintikka structure for f. He proceeds to give the following algorithm for deciding the satisfiability of a formula f:

1. Build an initial tableau $T = (S, R, L)$ for f as follows:
   - Define S to be the collection of maximal, propositionally consistent subsets of $ecl(f)$. In other words, $\forall s \in S \,.\forall g \in ecl(f) \,.\, \{g, \sim g\} \cap s \neq \emptyset$ and s satisfies **PC0 - PC4**.
   - Define R to be $S \times S - \{\ (s,t) \mid \ \mathbf{AX}g \in s$ and $g \notin t$ for some $\mathbf{AX}g \in ecl(f)\ \}$. This ensures that T satisfies **LC0**.
   - Define $L(s) = s$ for all $s \in S$.
2. Ensure the tableau also satisfies pseudo-fulfillment of eventualities and **LC1**. This can be done by repeatedly applying the following rules until a fixpoint is reached:
   - Delete any state that has no successors.
   - Delete any state that violates **LC1**.
   - Delete any state s that is labeled with an eventuality that does not have a fragment certifying pseudo-fulfillment of r.
3. The formula f is satisfiable if and only if the final tableau has a state labeled with f.

It is important to note that the final tableau is *not* necessarily a model for the formula, although Emerson does describe how a satisfying model could be extracted from this final tableau [8].

4 OBDD Encoding

The algorithm presented in Section 3 assumes an explicit representation of the tableau as a finite-state transition system. OBDDs have been used as an efficient, implicit representation for transition systems and for sets of states in both CTL and LTL model checking as well as in deciding satisfiability for LTL formulas [2, 3]. We use similar techniques to decide satisfiability for CTL formulas by encoding the initial tableau (step 1) and the fixpoint computation (step 2) in terms of OBDDs. When the final tableau is computed, its states and transition relation will also be represented as OBDDs and we can simply ask if the conjunction of the OBDD for the states with the OBDD for the formula f is satisfiable (is not the false OBDD).

First, we observe that when constructing the initial tableau $T = (S, R, L)$ for a formula f, the propositional consistency rules **PC0-PC2** mean that the labeling on the elementary formulas in a state completely determines the labeling on all formulas in $ecl(f)$ in that state, so we could define S to be $2^{el(f)}$. In fact, we only need half of the elementary formulas since the label on g also determines the label on $\sim g$. (Recall that $\sim g$ is the result of pushing in the negation in the formula $\neg g$.) Therefore, we get the same set of states by letting $S = 2^{el^+(f)}$ where $el^+(f)$ are the formulas in $ecl(f)$ that are either atomic propositions or have $\mathbf{EX}$ as the main connective. Again, using $\phi = \mathbf{AF}p \wedge \mathbf{EX}q$, we have

$$el^+(\phi) = \{p, \mathbf{EXEG}\neg p, q, \mathbf{EX}q\}$$

The set $el^+(f)$ forms the set of boolean state variables. Each unique assignment to these state variables yields a unique state in the tableau. To help avoid confusion, we use $\langle g \rangle$ to denote the state variable for $g \in el^+(f)$ and $\mathcal{V}_f = \{ \langle g \rangle \mid g \in el^+(f) \}$ to denote the set of all state variables in the tableau for f. We then encode states of the tableau as well as the transition relation of the tableau using quantified boolean formulas (QBF) over $\mathcal{V}_f$ which will be represented using OBDDs. For example, any QBF formula over $\mathcal{V}_f$ can be used to encode the set of states in which that formula evaluates to true.

The labeling function L can easily be implemented using OBDDs. Using Table 1, and the fact that all $\mathbf{EX}$ and all $\mathbf{AX}$ formulas will correspond to variables (possibly negated) in $\mathcal{V}_f$, we can translate any formula in $g \in ecl(f)$ into an equivalent boolean formula $\overline{g}$ over $\mathcal{V}_f$ without temporal operators as follows:

- $\overline{p} = \langle p \rangle$ for all atomic propositions p.
- $\overline{\neg p} = \neg\langle p \rangle$ for all atomic propositions p.
- $\overline{g \wedge h} = \overline{g} \wedge \overline{h}$
- $\overline{g \vee h} = \overline{g} \vee \overline{h}$
- $\overline{\mathbf{EX}g} = \langle \mathbf{EX}g \rangle$
- $\overline{\mathbf{AX}g} = \neg\langle \mathbf{EX} \sim g \rangle$
- $\overline{\mathbf{E}[g \,\mathbf{U}\, h]} = \overline{h} \vee (\overline{g} \wedge \langle \mathbf{EXE}[g \,\mathbf{U}\, h] \rangle)$
- $\overline{\mathbf{A}[g \,\mathbf{U}\, h]} = \overline{h} \vee (\overline{g} \wedge \neg\langle \mathbf{EXE}[\sim g \,\mathbf{R}\, \sim h] \rangle)$
- $\overline{\mathbf{E}[g \,\mathbf{R}\, h]} = \overline{h} \wedge (\overline{g} \vee \langle \mathbf{EXE}[g \,\mathbf{R}\, h] \rangle)$
- $\overline{\mathbf{A}[g \,\mathbf{R}\, h]} = \overline{h} \wedge (\overline{g} \vee \neg\langle \mathbf{EXE}[\sim g \,\mathbf{U}\, \sim h] \rangle)$

With this translation, we can determine if a state is labeled with g by checking if $\overline{g}$ evaluates to true in the state. In other words, $L(s) = \{ g \mid s \models \overline{g} \}$. Clearly, this definition for L satisfies the propositional consistency rules **PC0-PC4**.

We now have constructed S and L for the tableau. To construct the transition relation R, we create a second copy of state variables, $\mathcal{V}_f' = \{ v' \mid v \in \mathcal{V}_f \}$, to represent the next state in a transition. In what follows, V and V' are boolean vectors representing a truth assignment to the variables in V_f and V_f' respectively. A boolean vector V is identified with the state s that is equal to the set of variables in V_f assigned true by V. (Recall that in our tableau, $S = 2^{el^+(f)}$ and so each state is a subset of $el^+(f)$). The tableau transition relation $\overline{R}(V, V')$ is encoded as a QBF formula over $V_f \cup V_f'$ that evaluates to true whenever there is a transition from the state encoded by the assignment V to the state encoded by the assignment V'.

Recall that the transition relation for the tableau has a transition between every pair of states except where this would violate rule **LC0**. In other words, there should be a transition from s to s' whenever

$$\bigwedge_{\mathbf{AX}g \in ecl(f)} \mathbf{AX}g \in L(s) \Rightarrow g \in L(s')$$

is satisfied. This can be translated into a boolean formula over $\mathcal{V}_f \cup \mathcal{V}_f'$ as

$$R(V, V') = \bigwedge_{\langle \mathbf{EX}g \rangle \in \mathcal{V}_f} \langle \mathbf{EX}g \rangle \vee \overline{\sim g}'$$

where for any QBF formula h over $\mathcal{V}_f$, h' is identical to h except that every occurrence of a variable $v \in \mathcal{V}_f$ is replaced by the corresponding next state variable $v' \in \mathcal{V}_f'$. So $\langle \mathbf{EX}g \rangle$ is a variable in $\mathcal{V}_f$ and would have a value assigned to it by the boolean vector V while $\overline{\sim g}'$ is a formula over the variables in $\mathcal{V}_f'$ which would be assigned values from the boolean vector V'. Note that this formula for R restricts outgoing transitions from any state s labeled with $\mathbf{AX}g$, since then $\overline{\mathbf{AX}g} = \neg\langle \mathbf{EX} \sim g \rangle$ would be true. This would require all outgoing transitions from the state to satisfy $\overline{\sim\sim g}' = \overline{g}'$ thus satisfying rule **LC0**.

Figure 1 shows the states in the initial tableau for $\phi = \mathbf{AF}p \wedge \mathbf{EX}q$. So that it is easier to follow the discussion, the figure uses the label $\mathbf{AXAF}p$ which is actually not in $el^+(\phi)$ instead of $\mathbf{EXEG}\neg p$ which is. Since $\mathbf{AXAF}p \equiv \neg\mathbf{EXEG}\neg p$, states in which $\mathbf{AXAF}p$ does *not* appear are exactly the states where $\mathbf{EXEG}\neg p$ should appear. Recall that $\mathbf{AF}p$ is characterized by the disjunctive formula $p \vee \mathbf{AXAF}p$. Therefore, states which are labeled with neither p nor $\mathbf{AXAF}p$ (and which are drawn with dashed lines for ease of identification) are the states that would also not be labeled with $\mathbf{AF}p$. In this example, there is a transition from every state to every state, except that states labeled with $\mathbf{AXAF}p$ do not have transitions to dashed states (states that would not labeled with $\mathbf{AF}p$).

Step two of the satisfiability algorithm requires us to remove "bad" states from the tableau. We use a QBF formula to encode the states of the original tableau that have not been removed. Let $S(V)$ be a QBF formula over $\mathcal{V}_f$ encoding of all the valid states currently in the tableau. The initial value for $S(V)$ is the formula $\mathtt{tt}$. We now now show how to update $S(V)$ to remove "bad" states as defined in the explicit tableau algorithm. In what follows, $R(V, V')$ is the QBF encoding of the transition relation described earlier.

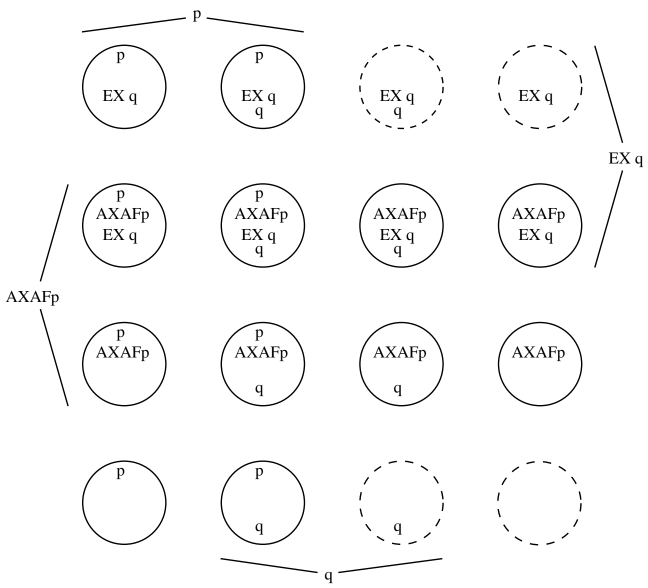

Figure 1: Constructing a tableau for $\phi = \mathbf{AF}p \wedge \mathbf{EX}q$

To remove all states that have no successors, we place a new restriction on valid states. The formula

$$\mathit{SUCC}(V) = \exists V' \,.\, R(V, V') \wedge S(V')$$

is satisfied by all states with an outgoing transition to a valid state. In other words, this formula encodes states that have a successor that is a valid state. To restrict ourselves to such states (and thus delete dead end nodes), we take the conjunction of the current set of valid states together with the formula to get an updated set of valid states:

$$S_1(V) = S(V) \wedge \mathit{SUCC}(V)$$

Note that in our example in Figure 1, all states have valid successors and so $S_1 = S$ in this example.

To remove all states that violate **LC1**, we again place a new restriction on the set of valid states. The set of states satisfying **LC1** is exactly the states satisfying:

$$\bigwedge_{\mathbf{EX}g \in ecl(f)} \mathbf{EX}g \in L(s) \Rightarrow \exists s' \in S \,.\, R(s, s') \wedge g \in L(s')$$

which can be encoded as a QBF formula over $\mathcal{V}_f \cup \mathcal{V}'_f$ as

$$\mathit{LC1}(V) = \bigwedge_{\langle \mathbf{EX}g \rangle \in \mathcal{V}_f} \neg \langle \mathbf{EX}g \rangle \vee \exists V' [S(V') \wedge R(V, V') \wedge \overline{g}']$$

Again, to update the set of valid states, simply take the conjunction of the current valid states with this formula

$$S_2(V) = S_1(V) \wedge \mathit{LC1}(V).$$

Again, in our example, all states labeled with $\mathbf{EX}q$ have at least one transition to a state labeled with q and so $S_2 = S_1 = S$.

Finally, to remove all states that are labeled with eventualities that are not pseudo-fulfilled, we need predicates for states labeled with eventualities and predicates for states at the root of a fragment certifying pseudo-fulfillment. Assume that a predicate $\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]}$ ($\mathit{frag}_{\mathbf{A}[g\ \mathbf{U}\ h]}$) over the state variables $\mathcal{V}_f$ exists such that $\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]}$ ($\mathit{frag}_{\mathbf{A}[g\ \mathbf{U}\ h]}$) evaluates to true exactly in those states which are roots of fragments certifying pseudo-fulfillment of the formula $\mathbf{E}[g\ \mathbf{U}\ h]$ ($\mathbf{A}[g\ \mathbf{U}\ h]$). To encode the states labeled with an eventuality, we must recall that there are no variables associated with formulas of the form $\mathbf{E}[g\ \mathbf{U}\ h]$ or $\mathbf{A}[g\ \mathbf{U}\ h]$ since neither is an elementary formula. However, states satisfying $\langle \mathbf{EXE}[g\ \mathbf{U}\ h]\rangle \wedge \overline{g}$ would be labeled with $\mathbf{E}[g\ \mathbf{U}\ h]$ in the original algorithm. Similarly, states satisfying $\neg\langle \mathbf{EXE}[g\ \mathbf{R}\ h]\rangle \wedge \overline{\sim g}$ would be labeled with $\mathbf{A}[\ \sim g\ \mathbf{U}\ \ \sim h]$ in the original algorithm. Such states also need to satisfy $\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]}$ or $\mathit{frag}_{\mathbf{A}[\sim g\ \mathbf{U}\ \sim h]}$ respectively. The corresponding formulas for states that respect the pseudo-fulfillment requirement are:

$$E(V) = \bigwedge_{\langle \mathbf{EXE}[g\ \mathbf{U}\ h]\rangle \in \mathcal{V}_f} \left[\neg\langle \mathbf{EXE}[g\ \mathbf{U}\ h]\rangle \vee \neg\overline{g} \vee \mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]}\right]$$

$$A(V) = \bigwedge_{\langle \mathbf{EXE}[g\ \mathbf{R}\ h]\rangle \in \mathcal{V}_f} \left[\langle \mathbf{EXE}[g\ \mathbf{R}\ h]\rangle \vee \overline{g} \vee \mathit{frag}_{\mathbf{A}[\sim g\ \mathbf{U}\ \sim h]}\right]$$

Note that states satisfying $\overline{h}$ would also be labeled with $\mathbf{E}[g\ \mathbf{U}\ h]$ and states satisfying $\overline{\sim h}$ would also be labeled with $\mathbf{A}[\ \sim g\ \mathbf{U}\ \ \sim h]$, but these states have the trivial fragment consisting only of the state itself and so do not need to be checked. Once again, the current set of valid states is restricted to those satisfying these predicates:

$$S_3(V) = S_2(V) \wedge E(V) \wedge A(V)$$

Recall that in our example there are no elementary formulas of the form $\mathbf{EXE}[g\ \mathbf{U}\ h]$. The only eventuality, $\mathbf{AF}p$, results in the elementary formula $\neg\mathbf{EXE}[\mathbf{ff}\ \mathbf{R}\ \neg p]$. Therefore, $E(V) = \mathtt{tt}$ since the conjunction is empty. There is only one elementary formula of the form $\mathbf{EXE}[g\ \mathbf{R}\ h]$ and so

$$A(V) = \langle \mathbf{EXE}[\mathbf{ff}\ \mathbf{R}\ \neg p]\rangle \vee \mathit{frag}_{\mathbf{A}[\mathtt{tt}\ \mathbf{U}\ p]}$$

In our example, states in which the variable $\langle \mathbf{EXE}[\mathbf{ff}\ \mathbf{R}\ \neg p]\rangle$ is assigned true are the ones that are not labeled with $\mathbf{AXAF}p$. In other words, in order for a state in our example to satisfy $A(V)$, it must either not be labeled with $\mathbf{AXAF}p$ or it must satisfy the predicate $\mathit{frag}_{\mathbf{A}[\mathtt{tt}\ \mathbf{U}\ p]}$ It turns out that every state in the initial tableau satisfies $\mathit{frag}_{\mathbf{A}[\mathtt{tt}\ \mathbf{U}\ p]}$. Figure 2 illustrates certifying fragments for two states in the tableau. Along the bottom is a fragment rooted at the bottom right node certifying pseudo-fulfillment of $\mathbf{AF}p$ for that node. The fragment consists of only three nodes and two

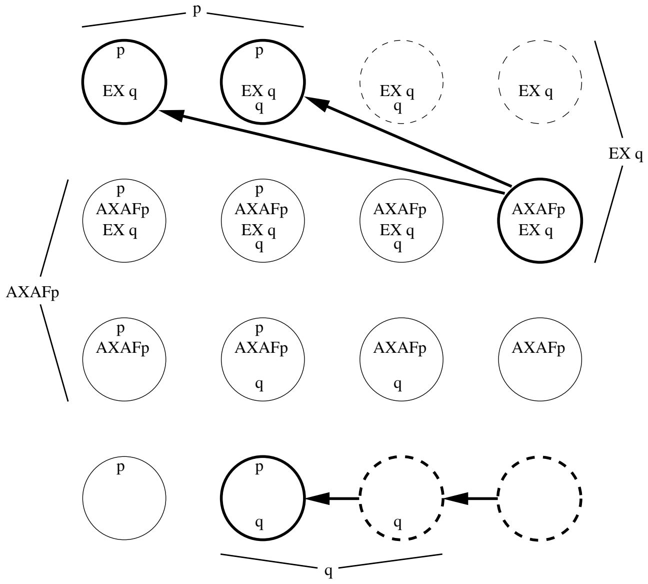

Figure 2: Example fragments certifying pseudo-fulfillment of $\mathbf{AF}p$

transitions even though in the original tableau, all three nodes had transitions to every state in the tableau. Similarly, near the top right is a fragment consisting of three nodes that is rooted at the rightmost node in the second row and certifying pseudo-fulfillment of $\mathbf{AF}p$ for that node. Note that in both cases, many other certifying fragments were possible, and some would have been smaller (would have only contained 2 states). For example, in the top fragment, the leftmost transition could be removed and what remains is a valid 2 state fragment rooted at the same node. Note that because of the $\mathbf{EX}q$ label in the root node, we could not keep the left transition and remove the right transition instead.

All that remains is to give definitions for $\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]}$ and $\mathit{frag}_{\mathbf{A}[\neg g\ \mathbf{U}\ \neg h]}$. These predicates cannot be encoded directly in QBF; however, they can be encoded as fixpoints of QBF formulas. These fixpoints can then be computed iteratively as is done for μ-calculus model checking [2]. The correct definitions for $\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]}$ and $\mathit{frag}_{\mathbf{A}[g\ \mathbf{U}\ h]}$ are given in the theorems stated below which are proved in the appendix.

Theorem 1 *The set of states at the root of a fragment certifying pseudo-fulfillment of* $\mathbf{E}[g\ \mathbf{U}\ h]$ *equals the set of states satisfying the fixpoint equation:*

$$\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]} = \mu Z\ .\ \left[\overline{h} \vee \left(\overline{g} \wedge \exists V' \big[R(V,V') \wedge S(V') \wedge Z(V')\big]\right)\right]$$

Readers familiar with modal μ-calculus model checking may be more familiar with the following formulation

$$\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]} = \mu Z\ .\ \left[\overline{h} \vee (\overline{g} \wedge \Diamond Z)\right]$$

where for any formula f, $\Diamond f$ is true in a state if there exists a next state where f is true. The μ-calculus formula $\Diamond f$ can be translated into a QBF formula as $\exists V' \,.\, R(V, V') \wedge S(V') \wedge f'$. The extra term $S(V')$ is required to ensure the next state satisfying f is also a valid state, since unlike model checking, the structure is not fixed. Recall that the satisfiability algorithm begins with a tableau that includes too many states. As states are pruned, we need to ensure that these pruned states are not used to satisfy the predicate for pseudo-fulfillment. Therefore, the entire formula reads as follows: a state is at the root of a certifying fragment for $\mathbf{E}[g \ \mathbf{U} \ h]$ iff

- the state satisfies h or
- the state
  1. satisfies g and
  2. the state has a successor that is a valid state at the root of a certifying fragment for $\mathbf{E}[g \ \mathbf{U} \ h]$.

The definition for $\mathit{frag}_{\mathbf{A}[g \ \mathbf{U} \ h]}$ is a little more complicated. The difference arises because in the case of $\mathit{frag}_{\mathbf{E}[g \ \mathbf{U} \ h]}$, states serving as witnesses to $\mathbf{EX}$ formulas did not themselves have to satisfy $\mathbf{E}[g \ \mathbf{U} \ h]$. But for $\mathit{frag}_{\mathbf{A}[g \ \mathbf{U} \ h]}$, any state witnessing an $\mathbf{EX}$ formula must also satisfy $\mathbf{A}[g \ \mathbf{U} \ h]$. The theorem below provides the correct μ-calculus formula to use.

Theorem 2 *The set of states at the root of a fragment certifying pseudo-fulfillment of* $\mathbf{A}[g \ \mathbf{U} \ h]$ *equals the set of states satisfying the fixpoint equation:*

$$\mathit{frag}_{\mathbf{A}[g \ \mathbf{U} \ h]} = \mu Z \,.\, \left[\begin{array}{c} \overline{h} \\ \vee \\ \Big(\overline{g} \wedge \exists V' \big[R(V, V') \wedge S(V') \wedge Z(V') \big] \\ \wedge \\ \displaystyle\bigwedge_{\langle \mathbf{EX} i \rangle \in \mathcal{V}_f} \Big(\langle \mathbf{EX} i \rangle \rightarrow \exists V' \big[R(V, V') \wedge S(V') \wedge Z(V') \wedge \overline{i}' \big] \Big) \Big) \end{array} \right]$$

The corresponding modal μ-calculus formula is

$$\mathit{frag}_{\mathbf{A}[g \ \mathbf{U} \ h]} = \mu Z \,.\, \overline{h} \vee \Big(\overline{g} \wedge \Diamond Z \wedge \bigwedge_{\langle \mathbf{EX} i \rangle \in \mathcal{V}_f} \big[\langle \mathbf{EX} i \rangle \rightarrow \Diamond (Z \wedge \overline{i}) \big] \Big)$$

This formula reads as follows: a state is at the root of a certifying fragment for $\mathbf{A}[g \ \mathbf{U} \ h]$ iff

- the state satisfies h or
- the state
  1. satisfies g and

2. the state has at least one successor that is a valid state at the root of a certifying fragment for $\mathbf{A}[g\ \mathbf{U}\ h]$ and
3. for every label of the form $\mathbf{EX}i$ in the state, there is a valid successor that satisfies i and that is at the root of a of a certifying fragment for $\mathbf{A}[g\ \mathbf{U}\ h]$.

It seems counterintuitive that the predicate for $\mathit{frag}_{\mathbf{A}[g\ \mathbf{U}\ h]}$, should contain existential quantification. However the first one appears because the models for CTL formulas are Kripke structures which must have a transition relation that is *total*. Without the first existential, any **AU** eventuality could be trivially certified by not including any outgoing transitions; however, such dead end states are not allowed in a Kripke structure. The second existential is required because of the presence of other existential formulas ($\mathbf{EX}$ formulas) in the label for the state.

We now have all the machinery we need to give a fixpoint characterization for the full algorithm. Given the construction of the QBF transition relation described earlier, the states in the final tableau can be computed as the greatest fixpoint of the predicate transformer:

$$T(S) = S(V) \wedge SUCC(V) \wedge LC1(V) \wedge E(V) \wedge A(V).$$

This means that the original CTL formula f is satisfiable if and only if

$$\overline{f} \wedge \nu S\ .\ S(V) \wedge SUCC(V) \wedge LC1(V) \wedge E(V) \wedge A(V)$$

is satisfiable. Once a BDD for this formula is constructed, checking for satisfiability involves verifying that the BDD is not equal to the *false* BDD.

5 Experimental Results

Satisfiability for CTL is known to be EXPTIME complete [8]. The μ-calculus formula we need to check has alternation depth 2 and can be checked in time $O(|f| \cdot |M|^3)$ using the result in [7]. Note that the f here is our μ-calculus formula and not the original f we were checking for satisfiability. Also, M is the tableau we are constructing whose size is exponential in the size of the CTL formula we are checking. Like model checking, the limiting factor is really the exponential size of the model. As is the case for symbolic model checking, the practicality of the technique is supported by experimental results.

Table 2 contains the results of some experiments conducted with our satisfiability checker. The first column contains the name of the experiment. The second column lists the amount of time taken in seconds. The third column lists the number of BDD variables required (twice the size of $el^+(f)$). The last column displays how much memory was allocated by the BDD library while running the experiment. All experiments were performed on a 600MHz Pentium III machine with 512 MB running Linux. Experiments consisted of checking for the validity of a formula by checking that its negation is not satisfiable. The experiments named induction_n successfully verified the validity of formulas of the form

$$\left[p_0 \wedge \bigwedge_{i=0}^{n-1} \mathbf{AG}\big(p_i \rightarrow \mathbf{AX}p_{[i+1]_n}\big)\right] \rightarrow \mathbf{AGAF}p_0$$

Table 2: Experimental Results

Experiment	Time (sec.)	BDD Variables	BDD Memory (KB)
induction_16	30.3	100	544
induction_20	111	124	608
induction_24	286	148	736
induction_28	710	172	960
precede_16	0.6	102	544
precede_32	11.3	198	544
precede_64	101	390	992
precede_128	794	774	2912
fair_8	0.2	50	512
fair_16	1.9	98	512
fair_32	20	194	576
fair_64	166	386	672
fair_128	1370	770	1856
nobase_16	33.9	100	544
nobase_20	106	124	608
nobase_24	308	148	768
nobase_28	728	172	960

The experiments named precede_n successfully verified the validity of formulas of the form

$$\left[\mathbf{AF}p_n \wedge \bigwedge_{i=0}^{n} \neg p_i \wedge \bigwedge_{i=0}^{n-1} \mathbf{AG}(\neg p_i \rightarrow \mathbf{AX}\neg p_{i+1})\right] \rightarrow \mathbf{AF}p_0$$

The experiments named fair_n successfully verified the validity of formulas of the form

$$\left[\mathbf{AGAF}p_0 \wedge \bigwedge_{i=0}^{n-1} \mathbf{AG}(p_i \rightarrow \mathbf{AXAF}p_{[i+1]_n})\right] \rightarrow \mathbf{AGAF}p_{n-1}$$

Finally, the experiments named nobase_n verified that the induction formulas were not valid without the base case. In other words, these experiments successfully verified that the formulas

$$\neg\left[\bigwedge_{i=0}^{n-1} \mathbf{AG}\big(p_i \rightarrow \mathbf{AX}p_{[i+1]_n}\big) \rightarrow \mathbf{AGAF}p_0\right]$$

are satisfiable.

6 Conclusion

We have implemented a symbolic algorithm for determining whether a CTL formula is satisfiable or not. The algorithm avoids constructing an explicit pseudo-Hintikka structure for the formula by using OBDDs (boolean formulas) to encode the structure.

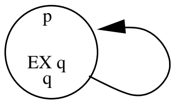

Figure 3: Minimal model for $\phi = \mathbf{AF}p \wedge \mathbf{EX}q$

The procedure has exponential time complexity; however, we have been able to use it check a number of complex formulas (on the order of 100 atomic propositions). We are confident that this algorithm will work for other non-trivial formulas.

While this algorithm seems sufficient to check the very structured formulas in the experiments, it remains to be seen how practical this approach is if used on formulas that may arise from some problem domain. Identifying problem domains where this kind of satisfiability checker would prove useful would be an excellent avenue for future work.

Perhaps the most obvious avenue for future work is the development of a model synthesis facility for CTL. Not only could it be useful to be able to construct concrete models that satisfy certain properties, but it would also be very useful be able to construct a concrete model that fails to satisfy some property. This could serve as a counterexample facility while doing validity checking. In other words, to check the validity of f, one checks if $\neg f$ is satisfiable. If $\neg f$ is not satisfiable, then f is valid. However, if $\neg f$ is satisfiable, an example model satisfying $\neg f$ would help us to understand why f is not valid. Often when trying experiments, we would try to verify formulas we thought were valid. When our algorithm reported back that the formula was not valid, it often took a significant amount of work to determine if this was an error in our algorithm or a mistake in our formula. In all cases, it was a mistake in the formula. However, an automatically generated counter-example would have drastically reduced the time and thought involved in trying to uncover the mistake. It must be noted that there is some cause for concern regarding the practicality of synthesizing the model. In particular, the size of the model generated by Emerson's algorithm is bounded by $m2^n$ where n is the length of the formula and m is the number of eventualities appearing in the formula [8]. Some minimization would most likely be necessary. In particular, it would be extremely helpful to be able to find small certifying fragments. In our running example, the smallest fragment for $\mathbf{AF}p$ for a state that also satisfies $\mathbf{EX}q$ is shown in Figure 3 which is clearly much smaller than the initial tableau. In the model checking community, there is already a need to find small counterexamples and perhaps there work could once again prove helpful.

References

[1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

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

[3] E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1):47–71, 1997.

[4] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[5] Olivier Coudert, Jean Christophe Madre, and Christian Berthet. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of Automatic Verification Methods for Finite State Systems*, pages 365–373, 1989.

[6] E.M.Clarke, O.Grumberg, and D.E.Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[7] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the 1st Annual Symposium on Logic in Computer Science*, pages 267–278. IEEE Computer Society Press, 1986.

[8] E. Allen Emerson. *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. MIT Press, 1990.

[9] E. Allen Emerson and Jospeh Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 169–180, 1982.

A Proof of Correctness

Recall that the correctness of our algorithm relies on the theorems about the fixpoint characterization of the $\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]}$ and $\mathit{frag}_{\mathbf{A}[g\ \mathbf{U}\ h]}$ predicates. This appendix provides proofs for these two theorems. In what follows, we will often abuse notation and use a formula (such as $\mathit{frag}_{\mathbf{A}[g\ \mathbf{U}\ h]}$) to refer both to the predicate and to the set of states that satisfy the predicate.

Definition 1 *A* satisfying path *for* $\mathbf{E}[g\ \mathbf{U}\ h]$ *is a path where the last node satisfies* h *and every other node satisfies* g*.*

Note that any fragment certifying pseudo-fulfillment for $\mathbf{E}[g\ \mathbf{U}\ h]$ at node s looks like a satisfying path starting at s and with enough other nodes "hanging from the path" to satisfy all formulas of the form $\mathbf{EX}i$ that label the nodes on the path. This is to insure that internal nodes satisfy rule **LC1**. All other consistency rules are automatically satisfied by the way the tableau is constructed. Since these extra nodes are terminal nodes, they have no additional restrictions placed on them by the definition of a fragment for $\mathbf{E}[g\ \mathbf{U}\ h]$. Also note that any node along a satisfying path is itself at the beginning of a satisfying path and is also the root of a fragment for $\mathbf{E}[g\ \mathbf{U}\ h]$. In other words, all such path nodes are also in $\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]}$.

Lemma 1.1 *The set of nodes at the root of a fragment for* $\mathbf{E}[g\ \mathbf{U}\ h]$ *(i.e. the set* $\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]}$*) is a fixpoint of the predicate transformer*

$$T(Z) = \left[\overline{h} \vee \left(\overline{g} \wedge \exists V'.R(V,V') \wedge S(V') \wedge Z(V')\right)\right]$$

Proof:
$\underline{\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]} \subseteq T(\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]})}$
Let $s \in \mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]}$ be the root of a fragment for $\mathbf{E}[g\ \mathbf{U}\ h]$. If s is the only node in that fragment, it must satisfy h and so $s \in T(\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]})$. Otherwise s satisfies g and it must be at the beginning of a satisfying path for $\mathbf{E}[g\ \mathbf{U}\ h]$. It must have a successor node s' that is also at the beginning of a satisfying path for $\mathbf{E}[g\ \mathbf{U}\ h]$. Therefore, $s' \in \mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]}$. But then s satisfies $T(\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]})$ and so $s \in T(\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]})$.

$\underline{T(\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]}) \subseteq \mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]}}$
Let $s \in T(\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]})$. If s satisfies h then it is at the beginning of a satisfying path of length 0 for $\mathbf{E}[g\ \mathbf{U}\ h]$ and so $s \in \mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]}$. If s does not satisfy h, it must satisfy g and it must have a transition to another node in the tableau s' that satisfies $\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]}$. This means that s' is at the beginning of a satisfying path for $\mathbf{E}[g\ \mathbf{U}\ h]$, and so s is as well and $s \in \mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]}$.

Lemma 1.2 *The set of nodes at the root of a fragment for* $\mathbf{E}[g\ \mathbf{U}\ h]$ *(i.e. the set* $\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]}$*) is contained in all fixpoints of the predicate transformer*

$$T(Z) = \left[\overline{h} \vee \left(\overline{g} \wedge \exists V'.R(V,V') \wedge S(V') \wedge Z(V')\right)\right]$$

Proof: Let Y be an arbitrary fixpoint of T and let $F = \mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]}$. To see that that $F \subseteq Y$, assume $F \not\subseteq Y$. Then the set $F - Y$ is not empty. All nodes in $F - Y$ are at the beginning of a satisfying path. Choose s from $F - Y$ with a satisfying path of minimal length. Note that s cannot satisfy h because all nodes satisfying h are in Y. Therefore s satisfies g and s must have a successor s' on the satisfying path. Since s' has a shorter satisfying path than s, $s' \in Y$ otherwise $s' \in F - Y$ and s' would have been chosen instead of s. But then s satisfies $T(Y)$ and $s \in T(Y)$. Since Y is a fixpoint, $T(Y) = Y$ and $s \in Y$. This contradicts our choice of s, so $F - Y$ must be empty and $F \subseteq Y$.

Theorem 1 *The set of nodes at the root of a fragment certifying pseudo-fulfillment of* $\mathbf{E}[g\ \mathbf{U}\ h]$ *(i.e. the set* $\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]}$*) equals the set of nodes satisfying the fixpoint equation:*

$$\mathit{frag}_{\mathbf{E}[g\ \mathbf{U}\ h]}(V) = \mu Z\ .\ \left[\overline{h} \vee \left(\overline{g} \wedge \exists V'.R(V,V') \wedge S(V') \wedge Z(V')\right)\right]$$

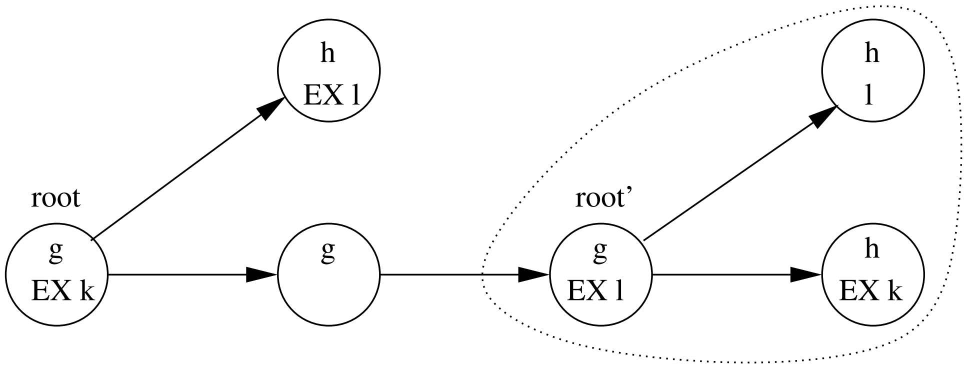

Figure 4: Example fragments certifying pseudo-fulfillment of $\mathbf{A}[g\ \mathbf{U}\ h]$

Proof: Together, lemma 1.1 and lemma 1.2 imply that the set of nodes that are roots of fragments for $\mathbf{E}[g\ \mathbf{U}\ h]$ (i.e. the set *frag*$_{\mathbf{E}[g\ \mathbf{U}\ h]}$) is equal to the least fixpoint of T.

In a manner analogous to the case for $\mathbf{E}[g\ \mathbf{U}\ h]$, we first describe what a fragment for $\mathbf{A}[g\ \mathbf{U}\ h]$ looks like. All terminal nodes satisfy h and all internal nodes satisfy g. Additionally, all internal nodes must satisfy rule **LC1**. (Again, the other consistency rules are trivially satisfied by the way the tableau is constructed.) Consider any node s in the fragment. The node s and its descendants in the fragment also form a fragment for $\mathbf{A}[g\ \mathbf{U}\ h]$. For example, consider Figure 4. The entire graph is a fragment for $\mathbf{A}[g\ \mathbf{U}\ h]$ at the node labeled root. The portion bounded by the dotted lines is a fragment for $\mathbf{A}[g\ \mathbf{U}\ h]$ at the node labeled root'. Note that every node in the figure is at the root of a fragment for $\mathbf{A}[g\ \mathbf{U}\ h]$. This must be the case in general because otherwise the whole thing would be a valid fragment. We will make use of the fact that nodes in a fragment for $\mathbf{A}[g\ \mathbf{U}\ h]$ are themselves roots of fragments for $\mathbf{A}[g\ \mathbf{U}\ h]$ in what follows.

Lemma 2.1 *The set of nodes at the root of a fragment for* $\mathbf{A}[g\ \mathbf{U}\ h]$ *(i.e. the set frag*$_{\mathbf{A}[g\ \mathbf{U}\ h]}$*) is a fixpoint of the predicate transformer*

$$T(Z) = \left[\begin{array}{c} \overline{h} \lor \Big(\overline{g} \land \exists V'\big[R(V,V') \land S(V') \land Z(V')\big] \land \\ \displaystyle\bigwedge_{\langle \mathbf{EX} i\rangle \in \mathcal{V}_f} \Big(\langle \mathbf{EX} i \rangle \rightarrow \exists V'\big[R(V,V') \land S(V') \land Z(V') \land \overline{i}'\big]\Big)\Big) \end{array} \right]$$

Proof:

<u>*frag*$_{\mathbf{A}[g\ \mathbf{U}\ h]} \subseteq T($*frag*$_{\mathbf{A}[g\ \mathbf{U}\ h]})$</u>

Let $s \in$ *frag*$_{\mathbf{A}[g\ \mathbf{U}\ h]}$. If s satisfies h then it satisfies $T($*frag*$_{\mathbf{A}[g\ \mathbf{U}\ h]})$ and so $s \in T($*frag*$_{\mathbf{A}[g\ \mathbf{U}\ h]})$. If s does not satisfy h then it must satisfy g and it must have at least one successor in *frag*$_{\mathbf{A}[g\ \mathbf{U}\ h]}$ since it cannot be terminal. Since s is internal, it must also satisfy rule **LC1**. Namely, each formula of the form $\mathbf{EX}i$ must be satisfied by some successor in *frag*$_{\mathbf{A}[g \ \mathbf{U} \ h]}$ that satisfies i. So s satisfies $T(\mathit{frag}_{\mathbf{A}[g \ \mathbf{U} \ h]})$ and $s \in T(\mathit{frag}_{\mathbf{A}[g \ \mathbf{U} \ h]})$.

$\underline{T(\mathit{frag}_{\mathbf{A}[g \ \mathbf{U} \ h]}) \subseteq \mathit{frag}_{\mathbf{A}[g \ \mathbf{U} \ h]}}$

Let $s \in T(\mathit{frag}_{\mathbf{A}[g \ \mathbf{U} \ h]})$. If s satisfies h then $s \in \mathit{frag}_{\mathbf{A}[g \ \mathbf{U} \ h]}$. If s does not satisfy h, it must satisfy g, it must have a transition to another node in the tableau that satisfies *frag*$_{\mathbf{A}[g \ \mathbf{U} \ h]}$, and for every formula of the form $\mathbf{EX}i$ that labels s, s must have a successor s' that satisfies i as well as *frag*$_{\mathbf{A}[g \ \mathbf{U} \ h]}$ (*i.e.* s' is the root of a fragment for $\mathbf{A}[g \ \mathbf{U} \ h]$). Therefore, s together with the fragments for its children form a fragment for $\mathbf{A}[g \ \mathbf{U} \ h]$ rooted at s. Therefore, $s \in \mathit{frag}_{\mathbf{A}[g \ \mathbf{U} \ h]}$.

Definition 2 *For any fragment, the* height *of a node n is the length of the longest path from n to a terminal node in the fragment.*

Lemma 2.2 *The set of roots of fragments for $\mathbf{A}[g \ \mathbf{U} \ h]$ (i.e. the set frag$_{\mathbf{A}[g \ \mathbf{U} \ h]}$) is contained in all fixpoints of the predicate transformer*

$$T(Z) = \left[\begin{array}{c} \overline{h} \vee \Big(\overline{g} \wedge \exists V' \big[R(V,V') \wedge S(V') \wedge Z(V') \big] \wedge \\ \displaystyle\bigwedge_{\langle \mathbf{EX}i \rangle \in \mathcal{V}_f} \Big(\langle \mathbf{EX}i \rangle \rightarrow \exists V' \big[R(V,V') \wedge S(V') \wedge Z(V') \wedge \overline{i}' \big] \Big) \Big) \end{array} \right]$$

Proof: Let Y be an arbitrary fixpoint of T and let $F = \mathit{frag}_{\mathbf{A}[g \ \mathbf{U} \ h]}$. To see that $F \subseteq Y$, assume $F \not\subseteq Y$. Then $F - Y$ is not empty. Let s be a node in $F - Y$ with minimal height. The node s cannot satisfy h since all nodes satisfying h are in Y. But $s \in F$, so it must satisfy g, it must have a successor in *frag*$_{\mathbf{A}[g \ \mathbf{U} \ h]}$ and each formula of the form $\mathbf{EX}i$ labeling s must be satisfied by a successor satisfying i that is also in *frag*$_{\mathbf{A}[g \ \mathbf{U} \ h]}$. However, all these successors of s must be in Y because they have smaller height than s and would have been chosen instead s if they were not in Y. Therefore, s satisfies $T(Y)$ and $s \in T(Y)$. Since Y is a fixpoint of T, $T(Y) = Y$ and $s \in Y$. This contradicts our choice of s. Therefore, $F - Y$ must be empty and $F \subseteq Y$.

Theorem 2 *The set of nodes at the root of a fragment certifying pseudo-fulfillment of $\mathbf{A}[g \ \mathbf{U} \ h]$ (i.e. the set frag$_{\mathbf{A}[g \ \mathbf{U} \ h]}$) equals the set of nodes satisfying the fixpoint equation:*

$$\mathit{frag}_{\mathbf{A}[g \ \mathbf{U} \ h]} = \nu Z. \left[\begin{array}{c} \overline{h} \vee \Big(\overline{g} \wedge \exists V' \big[R(V,V') \wedge S(V') \wedge Z(V') \big] \wedge \\ \displaystyle\bigwedge_{\langle \mathbf{EX}i \rangle \in \mathcal{V}_f} \Big(\langle \mathbf{EX}i \rangle \rightarrow \exists V' \big[R(V,V') \wedge S(V') \wedge Z(V') \wedge \overline{i}' \big] \Big) \Big) \end{array} \right]$$

Proof: Together, lemma 2.1 and lemma 2.2 imply that the set of nodes that are roots of fragments for $\mathbf{A}[g \ \mathbf{U} \ h]$ (i.e. the set *frag*$_{\mathbf{A}[g \ \mathbf{U} \ h]}$) is equal to the least fixpoint of T.