Temporal Logic Query Checking
(Extended Abstract)

Glenn Bruns Patrice Godefroid
Bell Laboratories, Lucent Technologies
263 Shuman Boulevard, Naperville, IL 60566, U.S.A.
Email: {grb,god}@bell-labs.com

Abstract

A temporal logic query checker takes as input a Kripke structure and a temporal logic formula with a hole, and returns the set of propositional formulas that, when put in the hole, are satisfied by the Kripke structure. By allowing the temporal properties of a system to be discovered, query checking is useful in the study and reverse engineering of systems.

Temporal logic query checking was first proposed in [2]. In this paper, we generalize and simplify Chan's work by showing how a new class of alternating automata can be used for query checking with a wide range of temporal logics.

1 Introduction

As pointed out by Chan in [2], model checking is as often used for understanding a design as for verifying its correctness. One rarely begins the study of a design with a complete specification in hand. Instead, one identifies a few key properties, expresses them in temporal logic, and checks them against the design. Some of the properties usually fail to hold, so the properties (and possibly the design) are revised and rechecked. As this process iterates one develops a more detailed picture of the properties the design satisfies or should satisfy.

To speed the process of design understanding, Chan proposed temporal logic query checking [2]. Here one works with a temporal logic formula containing a placeholder, or hole. A query checker returns the strongest propositional formula that, when put into the hole, is satisfied by the design. For example, given a design and the CTL query $AG?$, the query checker will return the strongest invariant of the system; i.e. the strongest propositional formula that is satisfied in every state of the design. Thus, a query checker allows the mechanization of much of the trial-and-error work done while analyzing a design.

The aim of this paper is to extend and simplify Chan's work. Chan studied CTL query checking, and was interested in queries for which a single strongest solution exists, called *valid* queries in [2]. He showed that it is expensive to determine whether a CTL query is valid, and identified a syntactic class of CTL queries such that every formula in the class is valid. His query-checking algorithm works only with queries in this class. In contrast, we are interested in all CTL queries, even those that have multiple maximally-strong solutions. Furthermore, we do not restrict our attention to CTL. Our query-checking approach is defined for an arbitrary temporal logic.

We simplify Chan's work by showing that query checking can be accomplished by adapting existing model-checking algorithms. In particular, we show how to adapt the automata-theoretic approach to model checking of Kupferman, Vardi and Wolper [8] to solve the query-checking problem.

In the following section of the paper we define the query-checking problem and compare it to model checking. In Section 3, we present some properties of lattices that are central to understanding the solution space of query checking. In Section 4, we outline our approach to query checking and introduce a new class of alternating automata. In Section 5, we show how a query-checking algorithm can be obtained for any logic having a translation to alternating automata, and we describe the application of this approach to CTL queries. In Section 6 we present some examples. Proofs of most theorems are omitted in this extended abstract.

2 Problem Statement

In this section we define the query-checking problem. Our definition is relative to any temporal logic that is interpreted on Kripke structures and that allows atomic propositions as formulas. We write $(K, s) \models \phi$ if state s of Kripke structure K satisfies temporal logic formula ϕ.

A *query* is an expression obtained by replacing a single atomic proposition in a temporal logic formula by the symbol ?, which is referred to as the *placeholder* (or *hole*) of the query. Substituting the placeholder of a temporal logic query by a propositional formula (i.e., a formula built only from atomic propositions and boolean operators) yields a temporal logic formula. We write $\phi[\psi]$ for the formula obtained by substituting propositional formula ψ for the placeholder in query ϕ. We also accept temporal logic formulas themselves as queries. If ϕ is a temporal logic formula, then $\phi[\psi]$ is identical to ϕ.

A propositional formula ψ is a *solution* to a query ϕ, relative to state s of Kripke structure K, if $(K, s) \models \phi[\psi]$.

A *positive* query is a query ϕ that is monotonic with respect to its placeholder: if $\psi_1 \Rightarrow \psi_2$ then $\phi[\psi_1] \Rightarrow \phi[\psi_2]$ (where $\Rightarrow$ denotes logical implication). In what follows we consider only positive queries. With such a query it makes sense to compute only maximally strong solutions, because from these solutions all others can be inferred[1]. Formally, let $PF(P)$ stand for the set of propositional formulas that can be built from a set P of atomic propositions. The ordering $\leq$ on set $PF(P)$ is defined as $\psi_1 \leq \psi_2$ iff $\psi_1 \Rightarrow \psi_2$. The resulting ordered set $\langle PF(P), \leq\rangle$ is a boolean lattice, which we refer to as L_P. For any ordered set $\langle A, \leq\rangle$ and $B \subseteq A$, we define $\min(B)$ by $\{b \in B \mid \forall b' \in B. b' \leq b \Rightarrow b' = b\}$. A subset B of A is *minimal* if $\min(B) = B$.

Definition 1 Let P be a set of atomic propositions, and let P' be a subset of P. Let K be a Kripke structure containing state s, and let ϕ be a query, both defined over P. The *query-checking problem* is to compute the set $\min\{\psi \in PF(P') \mid (K, s) \models \phi[\psi]\}$ of strongest solutions to ϕ. ■

We write $[(K, s), \phi]_{P'}$, or $[(K, s), \phi]$ for short, for the set of strongest solutions to query ψ relative to state s of Kripke structure K and set P' of atomic propositions.

[1] Our restriction to positive queries does not reduce generality. Suppose we had a query with a negated placeholder. We could compute the solution set for this query by removing the negation on the placeholder, computing the solution set for the resulting query, negating each formula in this set, and then interpreting the result as the set of *weakest* solutions to the query.

For a query ϕ without a placeholder, query checking reduces to model checking. If $(K, s) \not\models \phi$, then $(K, s) \not\models \phi[\psi]$ for all propositional formulas ψ, and hence $[(K, s), \phi] = \emptyset$. Otherwise $(K, s) \models \phi$, so $(K, s) \models \phi[\psi]$ for all propositional formulas, and hence $[(K, s), \phi] = \{\mathit{false}\}$. Since query checking is a generalization of model checking, it is at least as hard. Conversely, it is easy to show that query checking itself can be reduced to several model-checking problems.

Theorem 2 *Given a fixed set P' of atomic propositions and a temporal logic TL, the query-checking problem and the model-checking problem for TL have the same complexity in the size of the Kripke structure and in the size of the query/formula.*

Proof: A naive query-checking algorithm for solving $[(K, s), \phi]_{P'}$ consists of enumerating all $L = 2^{2^{|P'|}}$ possible solutions ψ, checking whether $(K, s) \models \phi[\psi]$ for each such ψ, and then returning only the minimal elements from that set. Query checking is thus reduced to at most $2^{2^{|P'|}}$ model-checking problems with a formula of length at most $|\phi| + O(2^{|P'|})$. ■

Since there can be $O(2^{2^{|P'|}})$ minimal solutions to a query-checking problem, parameter P' provides a way to control the complexity of query checking in practice, by specifying the atomic propositions that will appear in solutions computed for the query.

In the remainder of this paper, we develop a constructive algorithm for solving the query-checking problem that can converge directly to its minimal solutions, instead of guessing and checking exponentially-many individual potential solutions one by one as done with the above naive algorithm.

We illustrate the query-checking problem and our ideas to solve it by presenting examples of queries in the temporal logic CTL [5, 10]. Let p range over a set P of atomic propositions. The abstract syntax of CTL is defined from *state formulas* ϕ and *path formulas* ψ as follows:

$$\begin{array}{lcl} \phi & ::= & p \mid \neg p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid A\psi \mid E\psi \\ \psi & ::= & X\phi \mid \phi_1 \,\mathcal{U}\, \phi_2 \mid \phi_1 \,\tilde{\mathcal{U}}\, \phi_2 \end{array}$$

A CTL formula is a state formula. The *closure* of a CTL formula ϕ, written $cl(\phi)$, is defined as the set of all state subformulas of ϕ. The *size* $|\phi|$ of a formula ϕ is defined as the number of elements of $cl(\phi)$.

A CTL formula is interpreted with respect to a *Kripke structure* $K = (P, S, s_0, R, L)$ where P is a finite set of atomic propositions, S is a finite set of states, s_0 in S is the initial state, $R \subseteq S \times S$ is a total transition relation on states, and $L : S \to 2^P$ is a labeling

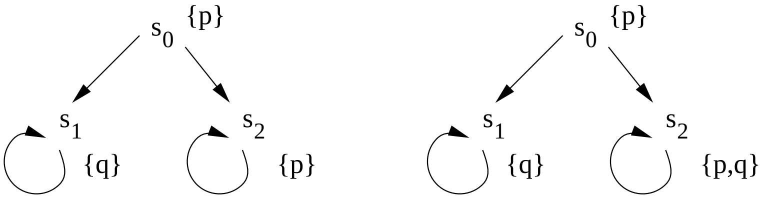

Figure 1. Example Kripke structures K_1 and K_2

function that maps each state to a set of atomic propositions. A *path* $w = s_0, s_1, \ldots$ of a Kripke structure is an infinite sequence of states such that $(s_i, s_{i+1}) \in R$ for all $i \geq 0$. We write w^i for the ith state of path w, with w^0 the first state. Also, we write paths(s) for the set of all paths w in K such that w^0 is s.

Given a Kripke structure $K = (P, S, s_0, R, L)$, a state formula ϕ satisfies a state s of K, and a path formula ψ satisfies a path w of K, according to the following inductive definitions.

$$
\begin{aligned}
(K,s) \models p &\stackrel{\text{def}}{=} p \in L(s)\\
(K,s) \models \neg p &\stackrel{\text{def}}{=} p \notin L(s)\\
(K,s) \models \phi_1 \wedge \phi_2 &\stackrel{\text{def}}{=} (K,s) \models \phi_1 \text{ and } (K,s) \models \phi_2\\
(K,s) \models \phi_1 \vee \phi_2 &\stackrel{\text{def}}{=} (K,s) \models \phi_1 \text{ or } (K,s) \models \phi_2\\
(K,s) \models A\psi &\stackrel{\text{def}}{=} \forall w \in \text{paths}(s).(K,w) \models \psi\\
(K,s) \models E\psi &\stackrel{\text{def}}{=} \exists w \in \text{paths}(s).(K,w) \models \psi
\end{aligned}
$$

$$
\begin{aligned}
(K,w) \models X\phi &\stackrel{\text{def}}{=} (K,w^1) \models \phi\\
(K,w) \models \phi_1 \,\mathcal{U}\, \phi_2 &\stackrel{\text{def}}{=} \exists i.(K,w^i) \models \phi_2 \text{ and}\\
&\quad\ \forall j < i.(K,w^j) \models \phi_1\\
(K,w) \models \phi_1 \,\tilde{\mathcal{U}}\, \phi_2 &\stackrel{\text{def}}{=} \forall i.(K,w^i) \models \phi_2 \text{ or}\\
&\quad\ \exists j < i.(K,w^j) \models \phi_1
\end{aligned}
$$

The class of CTL queries we allow are those for which negation is not applied to the placeholder. All such queries are positive.

Consider the CTL query $A(\mathit{false}\,\tilde{\mathcal{U}}\,?)$ (sometimes written $AG?$) and Kripke structure K_1, which is shown on the left of Figure 1. The formula $A(\mathit{false}\,\tilde{\mathcal{U}}\,\phi)$ holds if formula ϕ holds everywhere along all paths of a structure. A solution to the query is therefore a maximally-strong propositional formula that holds everywhere in the Kripke structure. Informally, the strongest solution of $\mathit{true}\,\mathcal{U}\,?$ for the left path in the example is $p \neq q$, and strongest solution for the right path is $p \wedge \neg q$. Therefore, the strongest solution that holds for all paths is $p \neq q$.

Consider the same query and Kripke structure K_2. Here the strongest solution on the left branch is $p \neq q$ and the strongest solution on the right branch is p. The strongest solution for both paths is therefore $p \vee q$.

Now, consider the CTL query $E(\mathit{true}\,\mathcal{U}\,?)$ (sometimes written $EF?$) and Kripke structure K_1. A solution to this query is a maximally-strong propositional formula that holds anywhere in the Kripke structure. This query on K_1 has two maximally-strong solutions: $p \wedge \neg q$ and $q \wedge \neg p$. The same query evaluated on K_2 has three maximally-strong solutions: $p \wedge \neg q$, $q \wedge \neg p$, and $p \wedge q$.

3 Solutions to Queries

In model checking with alternating automata, conjunction and disjunction operations are performed on truth values. In our algorithm for query checking, analogous operations are performed on sets of maximally-strong propositional formulas. These operations are defined as the meet and join operations of a lattice. In this section we define this lattice and show properties of the meet and join operations.

To begin, recall that we write $\psi_1 \leq \psi_2$ for propositional formulas ψ_1 and ψ_2 if $\psi_1 \Rightarrow \psi_2$. Also, given a set P of atomic propositions we write L_P for the boolean lattice $\langle PF(P), \leq \rangle$ having as its elements the propositional formulas built from elements of P. The left-most lattice of Figure 2 is L_P, where P contains only the single atomic proposition p.

Before going directly to the definition of a lattice on sets of maximally-strong propositional formulas, we will define a related lattice. Consider the set of *all* solutions to a query, not just the minimal ones. Because our queries are positive, the set of all solutions to a query is a set of propositional formulas that is closed under "going-up" with respect to $\leq$. In other words, if some propositional formula belongs to the the set, then so does every weaker formula. Given an ordered set $\langle A, \leq \rangle$ and a subset B of A, we define

$$\uparrow B \stackrel{\text{def}}{=} \{a \in A \mid \exists b \in B. b \leq a\}$$

A subset B of A is an *up-set* if $\uparrow B = B$. We write $U(A)$ for the set of all up-sets of A. Lattice theory (see

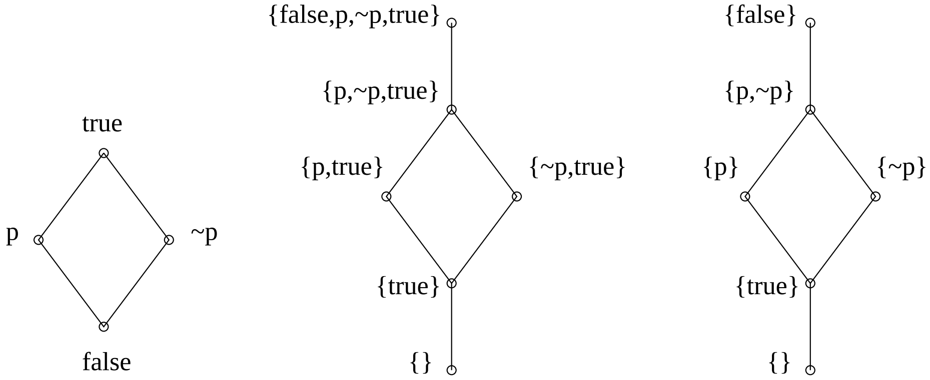

Figure 2. Lattices L_P, $L_P^{\uparrow}$, and L_P^{min} for $P = \{p\}$

Sec. 8.20 of [6]) tells us that if A is finite then $U(A)$ is a finite, distributive lattice, with elements ordered by set inclusion. It is easy to see that the meet and join operations of $U(A)$ are just set intersection and union.

Given a set P of atomic propositions, let $L_P^{\uparrow}$ be the lattice $U(PF(P))$, which is finite and distributive, but not boolean (see Lemma 8.21 of [6]). The middle lattice of Figure 2 is $L_P^{\uparrow}$ for $P = \{p\}$. Each element of this lattice is a possible set of solutions to a query in which the set of atomic propositions contains only atomic proposition p. Although not evident from Figure 2, lattice $L_P^{\uparrow}$ grows much faster than L_P as the set P of atomic propositions grows.

Each element of $L_P^{\uparrow}$ can be represented by its minimal elements. Recall from Section 2 that $\min(A)$ stands for the minimal elements of an ordered set A.

Proposition 3 *Let $\langle A, \leq \rangle$ be an ordered set with $B, C \subseteq A$. Then*

$$\begin{aligned} \mathit{min}(\uparrow\! B) &= \mathit{min}(B) \\ \mathit{min}(B \cup C) &= \mathit{min}(\mathit{min}(B) \cup \mathit{min}(C)) \end{aligned}$$

From $L_P^{\uparrow}$ we get an isomorphic lattice L_P^{min} by applying min to each element. Each element of L_P^{min} represents a set of maximally-strong propositional formulas, i.e., a candidate set of solutions to a query. The ordering of L_P^{min} is derived from the ordering of $L_P^{\uparrow}$: $A \leq B$ in $L_P^{\uparrow}$ if $\uparrow\! A \subseteq \,\uparrow\! B$. Similarly, the meet and join operations of L_P^{min} (which we write as $\underline{\wedge}$ and $\underline{\vee}$) are derived from $L_P^{\uparrow}$:

$$\begin{aligned} A \,\underline{\wedge}\, B &\stackrel{\text{def}}{=} \min(\uparrow\! A \cap \uparrow\! B) \\ A \,\underline{\vee}\, B &\stackrel{\text{def}}{=} \min(\uparrow\! A \cup \uparrow\! B) \end{aligned}$$

The right-most lattice of Figure 2 is L_P^{min} for $P = \{p\}$.

Defining $\underline{\wedge}$ and $\underline{\vee}$ as the meet and join operations of a distributive lattice is helpful because we immediately learn some properties of $\underline{\wedge}$ and $\underline{\vee}$.

Proposition 4 *Let A, B, and C be elements of L_P^{min}. Then*

$$\begin{aligned} A \,\underline{\wedge}\, B &= B \,\underline{\wedge}\, A \\ A \,\underline{\vee}\, B &= B \,\underline{\vee}\, A \\ A \,\underline{\wedge}\, (B \,\underline{\wedge}\, C) &= (A \,\underline{\wedge}\, B) \,\underline{\wedge}\, C \\ A \,\underline{\vee}\, (B \,\underline{\vee}\, C) &= (A \,\underline{\vee}\, B) \,\underline{\vee}\, C \\ A \,\underline{\wedge}\, (B \,\underline{\vee}\, C) &= (A \,\underline{\wedge}\, B) \,\underline{\vee}\, (A \,\underline{\wedge}\, C) \\ A \,\underline{\vee}\, (B \,\underline{\wedge}\, C) &= (A \,\underline{\vee}\, B) \,\underline{\wedge}\, (A \,\underline{\vee}\, C) \end{aligned}$$

It is awkward to compute $A \,\underline{\wedge}\, B$ and $A \,\underline{\vee}\, B$ using the definitions of $\underline{\wedge}$ and $\underline{\vee}$ directly because they first expand A and B to $\uparrow\! A$ and $\uparrow\! B$. The following characterizations allow $\underline{\wedge}$ and $\underline{\vee}$ to be computed directly using minimal sets.

Theorem 5 *Let A and B be elements of L_P^{min}. Then*

$$\begin{aligned} A \,\underline{\wedge}\, B &= \mathit{min}(\{a \vee b \mid a \in A \text{ and } b \in B\}) \\ A \,\underline{\vee}\, B &= \mathit{min}(A \cup B) \end{aligned}$$

4 Extended Alternating Automata

Inspired by the automata-theoretic approach to model checking of [8], we propose the following approach to query checking. Given a temporal logic query ϕ and a Kripke structure K, we (1) build an alternating automaton representing ϕ, (2) compute the product of this automaton with K, and finally (3) check whether the language accepted by the product automaton is empty. A key step in developing this approach is to discover a kind of alternating automaton appropriate for representing a temporal logic query. In this section we introduce a new type of alternating automata for this purpose, which we call *extended alternating automata* (EAA).

The novel aspect of alternating automata [3] is that the transition function maps an automaton state and an input tree value to a boolean expression. For example, suppose we have an alternating automaton in state q that is reading node x of an input tree with label a, and that the transition function of the automaton maps (q, x) to expression $((1, q') \wedge (1, q'')) \vee (2, q')$. Roughly, this means that the automaton can split into two copies, one of which is in state q' and is reading the first successor of x, and the second is in state q'' and is reading the first successor of x, *or* the automaton can instead evolve to state q'' without splitting and read the second successor of x. Constants *true* and *false* can also appear in an expression mapped to by the transition function.

The idea behind EAA is to generalize the expressions of the transition function in alternating automaton. Instead of allowing disjunction and conjunction operations, and constants *true* and *false*, we allow meet and join operations of an arbitrary finite lattice, and any values of the lattice as constants. For query checking, we use EAAs based on the lattice $L^{min}_{P'}$. Thus, the expression of the transition function involves operators $\underline{\wedge}$ and $\underline{\vee}$, and sets of maximally-strong propositional formulas. A standard alternating automaton is an EAA based on the lattice $\langle\{\mathit{true}, \mathit{false}\}, \leq\rangle$.

Semantically, EAA generalize the notion of a run in alternating automata. A run of an alternating automaton on an input tree is itself a tree, with each node of the run labeled by a node of the input tree. The labels on a node of the run and its children must satisfy the automaton's transition function. In an EAA, each node of the run is additionally labeled with a value of the underlying lattice, and the values of a node of a run and its children must again satisfy the transition function. Every run itself has a value, which is the value labeling the root node of the run. A run is accepting if it satisfies the acceptance condition of the EAA and also has no node labeled with the bottom element of the lattice. In a standard alternating automaton each node of a run is implicitly labeled with value *true*.

Now for definitions. A *tree* τ is a subset of $\mathbb{N}^*$ such that if $x \cdot c \in \tau$ then $x \in \tau$ and $x \cdot c' \in \tau$ for all $1 \leq c' < c$. The elements of τ are called its *nodes*, with ϵ called the *root*. Given a node x of τ, values of the form $x \cdot i$ in τ are called the *children* or *successors* of x. The number of successors of x is called the *degree* of x. A node with no successors is called a *leaf*. Given a set $D \subset \mathbb{N}$, a *D-tree* is a tree in which the degree of every node is in D. A *Σ-labeled* tree is a pair (τ, T) in which τ is a tree and $T : \mathbb{N}^* \rightarrow \Sigma$ is a labeling function. A 2^P-labeled tree, where P is a set of atomic propositions, is called a *computation tree*.

Let $L = (B, \wedge, \vee)$ be a finite lattice with its bottom element denoted $\perp$, and let $\mathcal{B}^+(X)$ stand for terms built from elements in a set X using $\wedge$ and $\vee$. A *tree EAA over L* is a tuple $A = (\Sigma, D, S, s_0, \rho, F)$, where Σ is a nonempty alphabet, S is a nonempty set of states, $s_0 \in S$ is the initial state, F is an acceptance condition, $D \subset \mathbb{N}$ is a finite set of arities, and $\rho : S \times \Sigma \times D \rightarrow \mathcal{B}^+((\mathbb{N} \times S) \cup B)$ is a transition function, where $\rho(s, a, k) \in \mathcal{B}^+((\{1, \ldots, k\} \times S) \cup B)$ is defined for each s in S, a in Σ, and k in D. Various acceptance conditions can be used with EAA, just as in alternating automata. In what follows we use only the *Büchi* acceptance condition, where $F \subseteq S$ is a set of accepting states.

A *v-run* of a tree EAA A on a Σ-labeled leafless D-tree (τ, T) is an $\mathbb{N}^* \times S \times B$-labeled D-tree (τ_σ, T_σ). A node in τ_σ labeled by (x, s, v) describes a copy of automaton A that reads the node x of τ_σ in the state s of A and has value $v \in B$ associated with it. Formally, a v-run (τ_σ, T_σ) is an $\mathbb{N}^* \times S \times B$-labeled tree, defined as follows.

- $T_\sigma(\epsilon) = (\epsilon, s_0, v)$
- Let $y \in \tau_\sigma$, $T_\sigma(y) = (x, s, v')$, $\mathit{arity}(x) = k$, and $\rho(s, T(x), k) = \theta$. Then there is a (possibly empty) set $Q = \{(c_1, s_1, v_1), \ldots, (c_n, s_n, v_n)\} \subseteq \{1, \ldots, k\} \times S \times B$ such that
  - for all $1 \leq i, j \leq n$, $c_i = c_j$ and $s_i = s_j$ implies $v_i = v_j$,
  - $\mathit{Eval}(Q, \theta) = v'$, and
  - for all $1 \leq i \leq n$, we have $y \cdot i \in \tau_\sigma$ and $T_\sigma(y \cdot i) = (x \cdot c_i, s_i, v_i)$

$\mathit{Eval}(Q, \theta)$ denotes the value of the expression θ obtained by replacing each term (c_i, s_i) in θ by v_i if $(c_i, s_i, v_i) \in Q$ and replacing each term (c_i, s_i) in θ by $\perp$ if $(c_i, s_i, v_i) \notin Q$.

A v-run σ is *accepting* (here assuming a Büchi acceptance condition F) if (1) the value associated with each node of the run is not $\perp$ and (2) all infinite branches of the run intersect the set F of accepting states infinitely often. Note that an accepting run can have finite branches: if, for some $y \in \tau_\sigma$, $T_\sigma(y) = (x, s, v)$ and $\rho(s, T(x), \mathit{arity}(x)) = v$ with $v \neq \perp$, then y does not need to have any successor.

A tree EAA accepts a Σ-labeled leafless D-tree (τ, T) with value v if there exists an accepting v-run of the automaton on that tree. We define the language $\mathcal{L}_v(A)$ as follows (for $v \neq \perp$):

$$\mathcal{L}_v(A) \stackrel{\text{def}}{=} \{(\tau, T) \mid A \text{ accepts } (\tau, T) \text{ with value } v\}$$

For notational convenience, we define $\mathcal{L}_\perp(A)$ as the set $\{(\tau, T) \mid A \text{ has no accepting run on } (\tau, T)\}$.

Note that if D is a singleton, then A runs over trees with a fixed branching degree. In particular, an *EAA over infinite words* is simply an EAA over infinite trees in which $D = \{1\}$.

We now recall the definition of a class of alternating automata that we will consider in the next section. A *weak* alternating automaton [9, 8] is an alternating Büchi automaton that has a partition of its set S of states into disjoints sets $S_1, \ldots, S_n$ satisfying the following two properties:

- for each set S_i, either $S_i \subseteq F$, in which case S_i is called an *accepting set*, or $S_i \cap F = \emptyset$, in which case S_i is called a *rejecting set*; and
- there exists a partial order $\leq$ on the set $\{S_1, \ldots, S_n\}$ such that, for every $s \in S_i$ and $s' \in S_j$ for which s' occurs in $\rho(s, a, k)$ for some $a \in \Sigma$ and $k \in D$, we have $S_j \leq S_i$.

In other words, transitions from states in S_i leads to states either in S_i or to some lower S_j in the partition. As shown in [8], this property makes it possible to check the emptiness problem for weak alternating automata more efficiently than for general alternating automata. This definition applies to alternating automata on either words or trees, extended or not.

5 A Query-Checking Algorithm

Three results are needed for model checking with alternating automata. First, in defining a translation from a temporal logic formula ϕ to an alternating automaton A_ϕ, one must show that A_ϕ accepts exactly all the trees that satisfy ϕ. Second, in defining the product of an alternating automaton A_ϕ and a Kripke structure K, one must show that the product automaton is nonempty just if K satisfies ϕ. Finally, one must show how to compute nonemptiness of the product automaton.

We need similar results for query checking. First, in defining a translation from a temporal logic query ϕ to an EAA A_ϕ, we show that the maximum value v labeling an accepting run of A_ϕ on a given tree is the set of all strongest propositional formulas ψ such that the tree satisfies $\phi[\psi]$ for every ψ in v. Second, in defining the product of an EAA A_ϕ and a Kripke structure K, we show that the maximum value v labeling an accepting run of the product automaton is the set of all strongest propositional formulas ψ such that K satisfies $\phi[\psi]$ for all ψ in v. Finally, we show how to compute the solution to the query ϕ using the product automaton.

In this section we present the steps of our query-checking algorithm and show these results. For the translation step, we translate CTL queries to EAA. Before presenting the translation we present a result needed in showing its correctness. This theorem shows that the checking of a CTL query can be reduced to checking its subqueries and then combining the results.

Theorem 6 *Let K be a Kripke structure with state s, let $\{s_i \mid i \in I\}$ be the (finite) set of successors of s, and let ϕ, ϕ_1 and ϕ_2 be CTL queries. Then*

$$\begin{array}{rcl}
[(K,s),\phi_1 \wedge \phi_2] & = & [(K,s),\phi_1] \underline{\wedge} [(K,s),\phi_2] \\
[(K,s),\phi_1 \vee \phi_2] & = & [(K,s),\phi_1] \underline{\vee} [(K,s),\phi_2] \\
[(K,s),AX\phi] & = & \underline{\bigwedge}_{i\in I}[(K,s_i),\phi] \\
[(K,s),EX\phi] & = & \underline{\bigvee}_{i\in I}[(K,s_i),\phi] \\
[(K,s),A(\phi_1 \,\mathcal{U}\, \phi_2)] & = & [(K,s),\phi_2] \underline{\vee} ([(K,s),\phi_1] \underline{\wedge} \\
 & & \underline{\bigwedge}_{i\in I}[(K,s_i),A(\phi_1 \,\mathcal{U}\, \phi_2)]) \\
[(K,s),E(\phi_1 \,\mathcal{U}\, \phi_2)] & = & [(K,s),\phi_2] \underline{\vee} ([(K,s),\phi_1] \underline{\wedge} \\
 & & \underline{\bigvee}_{i\in I}[(K,s_i),E(\phi_1 \,\mathcal{U}\, \phi_2)]) \\
[(K,s),A(\phi_1 \,\tilde{\mathcal{U}}\, \phi_2)] & = & [(K,s),\phi_2] \underline{\wedge} ([(K,s),\phi_1] \underline{\vee} \\
 & & \underline{\bigwedge}_{i\in I}[(K,s_i),A(\phi_1 \,\mathcal{U}\, \phi_2)]) \\
[(K,s),E(\phi_1 \,\tilde{\mathcal{U}}\, \phi_2)] & = & [(K,s),\phi_2] \underline{\wedge} ([(K,s),\phi_1] \underline{\vee} \\
 & & \underline{\bigvee}_{i\in I}[(K,s_i),E(\phi_1 \,\mathcal{U}\, \phi_2)])
\end{array}$$

Now we define the translation of CTL queries to EAA.

Theorem 7 *Given a CTL query ϕ, a set $P' \subseteq P$ of atomic propositions, and a set $D \subset I\!N$, one can construct in linear time a weak tree EAA $A_{(D,\phi)} = (2^P, D, cl(\phi), \phi, \rho, F)$ over $L_{P'}^{min}$ such that, for every 2^P-labeled leafless D-tree (τ, T), the value $v = [(\tau, T), \phi)]$ is the maximum value in $L_{P'}^{min}$ such that*

$$(\tau, T) \in \mathcal{L}_v(A_{(D,\phi)})$$

Proof: (Sketch) Our construction of $A_{(D,\phi)}$ is similar to the construction in [8] of a weak alternating tree automaton that accepts exactly the computation trees of Kripke structures satisfying ϕ. The set F of accepting states of $A_{(D,\phi)}$ consists of all the $\tilde{\mathcal{U}}$-formulas in $cl(\phi)$. The transition function ρ is defined as follows. For all a in 2^P and k in D, we have:

- $\rho(p, a, k) = \begin{cases} \{\mathit{false}\} & \text{if } p \in a \\ \emptyset & \text{otherwise} \end{cases}$
- $\rho(\neg p, a, k) = \begin{cases} \{\mathit{false}\} & \text{if } p \notin a \\ \emptyset & \text{otherwise} \end{cases}$
- $\rho(?, a, k) = \{\bigwedge\{p \mid p \in (a \cap P')\} \wedge \bigwedge\{\neg p \mid p \notin (a \cap P')\}\}$.

When the query is an atomic proposition p, the set $\{false\}$ is returned if $p \in a$ (since all elements in L_P are solutions to the query and *false* is the least value in L_P). Otherwise, if $p \notin a$, the empty set is returned since the query has no solutions. For the case of the placeholder symbol ?, the strongest propositional formula that holds at the current state a is returned since it represents the minimal solution in L_P holding in a.

For the other types of formulas in $cl(\phi)$, ρ is defined as in [8] except that we replace everywhere the symbol $\wedge$ by $\underline{\wedge}$ and $\vee$ by $\underline{\vee}$. We thus have, for all a in 2^P and k in D:

- $\rho(\phi_1 \wedge \phi_2, a, k) = \rho(\phi_1, a, k) \,\underline{\wedge}\, \rho(\phi_2, a, k)$.
- $\rho(\phi_1 \vee \phi_2, a, k) = \rho(\phi_1, a, k) \,\underline{\vee}\, \rho(\phi_2, a, k)$.
- $\rho(AX\phi, a, k) = \underline{\bigwedge}_{c=1}^{k}(c, \phi)$.
- $\rho(EX\phi, a, k) = \underline{\bigvee}_{c=1}^{k}(c, \phi)$.
- $\rho(A(\phi_1 \,\mathcal{U}\, \phi_2), a, k) = \rho(\phi_2, a, k) \,\underline{\vee}\, (\rho(\phi_1, a, k) \,\underline{\wedge}\, \underline{\bigwedge}_{c=1}^{k}(c, A(\phi_1 \,\mathcal{U}\, \phi_2)))$.
- $\rho(E(\phi_1 \,\mathcal{U}\, \phi_2), a, k) = \rho(\phi_2, a, k) \,\underline{\vee}\, (\rho(\phi_1, a, k) \,\underline{\wedge}\, \underline{\bigvee}_{c=1}^{k}(c, E(\phi_1 \,\mathcal{U}\, \phi_2)))$.
- $\rho(A(\phi_1 \,\tilde{\mathcal{U}}\, \phi_2), a, k) = \rho(\phi_2, a, k) \,\underline{\wedge}\, (\rho(\phi_1, a, k) \,\underline{\vee}\, \underline{\bigwedge}_{c=1}^{k}(c, A(\phi_1 \,\tilde{\mathcal{U}}\, \phi_2)))$.
- $\rho(E(\phi_1 \,\tilde{\mathcal{U}}\, \phi_2), a, k) = \rho(\phi_2, a, k) \,\underline{\wedge}\, (\rho(\phi_1, a, k) \,\underline{\vee}\, \underline{\bigvee}_{c=1}^{k}(c, E(\phi_1 \,\tilde{\mathcal{U}}\, \phi_2)))$.

The weakness partition and partial order on the set of states of $A_{(D,\phi)}$ is defined as in [8]. Specifically, each formula $\phi_1 \in cl(\phi)$ constitutes a singleton set $\{\phi_1\}$ in the partition, and the partial order $\leq$ is defined by $\{\phi_1\} \leq \{\phi_2\}$ iff $\phi_1 \in cl(\phi_2)$. Since each transition of the automaton from a state ϕ leads to states in $cl(\phi)$, the weakness conditions holds.

The correctness of our construction is proved in two steps. For soundness we show that, given an accepting v-run (τ_σ, T_σ) of $A_{(D,\phi)}$ over a 2^P-labeled leafless D-tree (τ, T), then for every $y \in \tau_\sigma$ such that $T_\sigma(y) = (x, \psi, v')$, we have $(\tau(x), T) \models \psi[\psi']$, for all $\psi' \in v'$ and where $\tau(x)$ denotes the subtree of τ rooted in x. For completeness we show that, given a 2^P-labeled leafless D-tree (τ, T) and $v \in L_{P'}^{min}$ such that $v = [(\tau, T), \phi]$, there exists an accepting v-run (τ_σ, T_σ) of $A_{(D,\phi)}$ over (τ, T). ■

In the next step of the algorithm we take the product of a Kripke structure and an EAA representing a query. In the following theorem, $A_{(D,\phi)}$ is an EAA that represents query ϕ according to the condition in Theorem 7.

Theorem 8 *Let $K = (P, S, s_0, R, L)$ be a Kripke structure with degrees in D, let $P' \subseteq P$ be a set of atomic propositions, and let $A_{(D,\phi)} = (2^P, D, Q_\phi, q_0, \rho_\phi, F_\phi)$ be a tree EAA over $L_{P'}^{min}$. Then one can construct a word EAA $A_{K,\phi}$ over a 1-letter alphabet with at most $O(|S| \cdot |Q_\phi|)$ states in such a way that the value $v = [(K, s_0), \phi)]$ is the maximum value in $L_{P'}^{min}$ such that*

$$\mathcal{L}_v(A_{K,\phi}) \neq \emptyset$$

Proof: (Sketch) We define the product automaton $A_{K,\phi}$ as done in [8]. Given Kripke structure $K = (P, S, s_0, R, L)$ and EAA $A_{(D,\phi)} = (2^P, D, Q_\phi, q_0, \rho_\phi, F_\phi)$, the product word EAA is $A_{K,\phi} = (\{a\}, S \times Q_\phi, (s, q_0), \rho, F)$, where ρ and F are defined as follows. If $q \in Q_\phi$, $s \in S$, the successors of s in R are $\{s_1', \ldots, s_n'\}$, and $\rho_\phi(q, L(s), n) = \theta$, then $\rho((s, q), a) = \theta'$ where θ' is obtained from θ by replacing each term (c, q') in θ by (s_c', q'). If $F_\phi \subseteq Q_\phi$ is a Büchi condition, then $F = S \times F_\phi$. When $A_{(D,\phi)}$ has a weakness partition $\{Q_1, Q_2, \ldots, Q_n\}$, then $A_{K,\phi}$ has a weakness partition $\{S \times Q_1, S \times Q_2, \ldots, S \times Q_n\}$.

In proving the correctness of the construction we view a Kripke structure $K = (P, S, s_0, R, L)$ as a computation tree (τ, T) that corresponds to the unwinding of K from s_0. Following the usual construction, we define (τ, T) using function $\nu : N^* \rightarrow S$ that maps the nodes of τ to states of K: initially, $\nu(\epsilon) = s_0$ and $T(\epsilon) = L(s_0)$; then, for any node x in τ, letting $s_1', \ldots, s_n'$ denote the successors of state $\nu(x)$ in K, we have $x \cdot i \in \tau$, $\nu(x \cdot i) = s_i'$, and $T(x \cdot i) = L(s_i')$, for all $1 \leq i \leq n$.

We then show that an accepting v-run of $A_{K,\phi}$ can be build from any accepting v-run of $A_{(D,\phi)}$ over (τ, T), and vice versa. ■

In the final step of the algorithm, we compute the solution to the query using the product EAA. By Theorem 8, the value of $[(K, s_0), \phi]$ is equal to the maximum value v in $L_{P'}^{min}$ such that $\mathcal{L}_v(A_{K,\phi}) \neq \emptyset$ (and hence is equal to $\emptyset$ if the product automaton does not have any accepting run). Checking nonemptiness of the languages $\mathcal{L}_v(A_{K,\phi})$ simultaneously for all values v can be done in time linear in the size of the product EAA $A_{K,\phi}$ when this automaton is weak, as is the case for CTL queries (since the EAA $A_{(D,\phi)}$ corresponding to a CTL query ϕ is itself weak).

Theorem 9 *Given a weak word EAA* $A_{K,\phi} = (\{a\}, Q, q_0, \rho, F)$ *over* $L_{P'}^{min}$ *and a 1-letter alphabet, computing the maximum value* v *in* $L_{P'}^{min}$ *such that* $\mathcal{L}_v(A_{K,\phi}) \neq \emptyset$ *can be done in time linear in the size of the automaton.*

Proof: (Sketch) The algorithm is similar to the algorithm for checking emptiness of a 1-letter weak alternating word automaton, which can be decided in linear time [8]. The only difference is that the algorithm dealing with EAA propagates values in the set $L_{P'}^{min}$ and manipulate these using the operations $\underline{\vee}$ and $\underline{\wedge}$, instead of propagating values in $\{true, false\}$ manipulated with $\vee$ and $\wedge$ as done in [8].

Specifically, the algorithm labels each reachable state of $A_{K,\phi} = (\{a\}, Q, q_0, \rho, F)$ with a value in $L_{P'}^{min}$. The correctness of the algorithm is established by showing that a state $q \in Q$ is labeled with v if and only if v is the maximum value in $L_{P'}^{min}$ such that $\mathcal{L}_v(A_q) \neq \emptyset$, where A_q denotes $A_{K,\phi}$ with q as initial state. The value labeling the initial state q_0 when the algorithm terminates is thus the solution to $[(K, s_0), \phi]$.

Since $A_{K,\phi}$ is weak, there exists a partition $\{Q_1, \ldots, Q_n\}$ of Q into disjoint sets Q_i such that there exists a partial order $\leq$ on the collection of the Q_i's and such that for for every $q \in Q_i$ and $q' \in Q_j$ for which q' occurs in $\rho(q, a)$, we have $Q_j \leq Q_i$. Moreover, each set Q_i is either accepting if $Q_i \subseteq F$, or rejecting otherwise.

The algorithm works in phases and proceeds up the partial order, starting from Q_1. First, states that are in Q_1 are temporarily labeled with $\{false\}$ if Q_1 is accepting, or with $\emptyset$ otherwise. Then, transition functions $\rho(q, a)$ in which $q' \in Q_1$ occurs are simplified by replacing the occurrence of q' by the label associated with q'. Once a transition function $\rho(q, a)$ is simplified to a value $v \in L_{P'}^{min}$, this value v becomes the permanent label of state q, and the simplification propagates further up the partial order. The successive temporary labeling of states in Q_2 upto Q_n according to their acceptance condition and the simplification process is repeated until the initial state q_0 is eventually labeled with some value v, which is the solution to the query-checking problem considered.

The entire algorithm can be implemented in time linear in the size of the EAA as discussed in [8]. ■

Although the worst-case complexity of our algorithm is in general no better than the naive algorithm of Section 2 with respect to the size of the set P' of atomic propositions, our algorithm has better complexity than the naive algorithm for many common queries. For example, consider query $A(false\, \tilde{\mathcal{U}}\, ?)$. Since the transition function of the EAA representing this query contains only $\underline{\wedge}$ operations, and since $|A| \leq 1$ and $|B| \leq 1$ implies $|A \,\underline{\wedge}\, B| \leq 1$, our algorithm will always associate a set containing at most one element of $L_{P'}$ with each state of the product automaton, and hence will manipulate singleton sets only. Since every element of $L_{P'}$ can be encoded using $2^{|P'|}$ bits only, the algorithm using EAA will run in time $O(2^{|P'|})$ instead of $O(2^{2^{|P'|}})$ for the naive algorithm.

6 Example

Consider the CTL query $A(false\, \tilde{\mathcal{U}}\, ?)$. By applying the construction of Theorem 7 with $P' = P = \{p, q\}$, we obtain a weak tree EAA with Büchi acceptance condition $\{A(false\, \tilde{\mathcal{U}}\, ?)\}$ and the following transition function:

$$
\begin{aligned}
\rho(A(false\, \tilde{\mathcal{U}}\, ?), \{\}, k) &= \{\neg p \wedge \neg q\} \,\underline{\wedge} \\
&\quad \underline{\bigwedge}_{c=1}^{k} (c, A(false\, \tilde{\mathcal{U}}\, ?)) \\
\rho(A(false\, \tilde{\mathcal{U}}\, ?), \{p\}, k) &= \{p \wedge \neg q\} \,\underline{\wedge} \\
&\quad \underline{\bigwedge}_{c=1}^{k} (c, A(false\, \tilde{\mathcal{U}}\, ?)) \\
\rho(A(false\, \tilde{\mathcal{U}}\, ?), \{q\}, k) &= \{q \wedge \neg p\} \,\underline{\wedge} \\
&\quad \underline{\bigwedge}_{c=1}^{k} (c, A(false\, \tilde{\mathcal{U}}\, ?)) \\
\rho(A(false\, \tilde{\mathcal{U}}\, ?), \{p, q\}, k) &= \{p \wedge q\} \,\underline{\wedge} \\
&\quad \underline{\bigwedge}_{c=1}^{k} (c, A(false\, \tilde{\mathcal{U}}\, ?))
\end{aligned}
$$

Taking the product of this automaton with Kripke structure K_1 of Figure 1, as described in Theorem 8, we obtain a weak word EAA over a 1-letter alphabet with all states accepting and the following transition function:

$$
\begin{aligned}
\rho((s_0, A(false\, \tilde{\mathcal{U}}\, ?)), a, 1) &= \{p \wedge \neg q\} \,\underline{\wedge} \\
&\quad (s_1, A(false\, \tilde{\mathcal{U}}\, ?)) \,\underline{\wedge} \\
&\quad (s_2, A(false\, \tilde{\mathcal{U}}\, ?)) \\
\rho((s_1, A(false\, \tilde{\mathcal{U}}\, ?)), a, 1) &= \{q \wedge \neg p\} \,\underline{\wedge} \\
&\quad (s_1, A(false\, \tilde{\mathcal{U}}\, ?)) \\
\rho((s_2, A(false\, \tilde{\mathcal{U}}\, ?)), a, 1) &= \{p \wedge \neg q\} \,\underline{\wedge} \\
&\quad (s_2, A(false\, \tilde{\mathcal{U}}\, ?))
\end{aligned}
$$

Applying the algorithm of Theorem 9 to this automaton can be viewed as solving this system of equations:

$$
\begin{aligned}
x_1 &= \{p \wedge \neg q\} \,\underline{\wedge}\, x_2 \,\underline{\wedge}\, x_3 \\
x_2 &= \{q \wedge \neg p\} \,\underline{\wedge}\, x_2 \\
x_3 &= \{p \wedge \neg q\} \,\underline{\wedge}\, x_3
\end{aligned}
$$

The algorithm starts in state x_1 and then visits states x_2 and x_3. In these states the occurrences of x_2 and

x_3 respectively on the right-hand side are replaced by $\{false\}$, since the state is accepting. Then, the values for x_2 and x_3 are computed by applying the definition of $\underline{\wedge}$: one obtains $x_2 = \{q \wedge \neg p\}$ and $x_3 = \{p \wedge \neg q\}$. The algorithm then backs up to x_1 and computes the value of x_1, which is $\{p \neq q\}$. This value is the solution to the query $[(K_1, s_0), A(\mathit{false}\, \tilde{\mathcal{U}}\, ?)]$.

7 Discussion

We have presented a general automata-theoretic approach to temporal logic query checking. The approach is general in the sense that if one has a translation from queries to EAA in the sense of Theorem 7, then checking nonemptiness of the product automaton gives the solution to the query. For CTL we showed how this translation can be derived directly from the translation of CTL to alternating automata. Translations for queries in other temporal logics (such as the modal mu-calculus) can be derived similarly.

We have defined EAA relative to an arbitrary finite lattice, although for query checking we need only EAA based on a lattices of the form L_P^{min}. A general definition for EAA was chosen because it is simpler, and also because we can imagine other uses for the more general form. For example, EAA could be used for model checking multi-valued temporal logics [7, 1, 4].

References

[1] G. Bruns and P. Godefroid. Model Checking Partial State Spaces with 3-Valued Temporal Logics. In *Proceedings of the 11th Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 274–287, Trento, July 1999. Springer-Verlag.

[2] W. Chan. Temporal-Logic Queries. In *Proceedings of the 12th Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 450–463, Chicago, July 2000. Springer-Verlag.

[3] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[4] M. Chechik, W. Easterbrook, and V. Petrovykh. Model-Checking over Multi-Valued Logics. In *Proceedings of FME '01*, pages 72–98, March 2001.

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

[6] B. Davey and H. Priestly. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[7] M. Fitting. Many-valued modal logics I. *Fundamenta Informaticae*, 15:235–254, 1992.

[8] O. Kupferman, M. Y. Vardi, and P. Wolper. An Automata-Theoretic Approach to Branching-Time Model Checking. *Journal of the ACM*, 47(2):312–360, March 2000.

[9] D. Muller, A. Saoudi, and P. Schupp. Alternating automata and the weak monadic theory of the tree and its complexity. In *Proceedings of ICALP '86, Lecture Notes in Computer Science*, pages 275–283. Springer-Verlag, 1986.

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