Temporal-Logic Queries

William Chan*

Abstract. This paper introduces temporal-logic queries for model understanding and model checking. A temporal-logic query is a temporal-logic formula in which a placeholder appears exactly once. Given a model, the semantics of a query is a proposition that can replace the placeholder to result in a formula that holds in the model and is as strong as possible. The author defines a class of CTL queries that can be evaluated in linear time, and show how they can be used to help the user understand the system behaviors and obtain more feedback in model checking.

1 Introduction

Although model *checking* was proposed as a verification (or falsification) technique [3], we find it valuable for model *understanding*: The user hypothesizes a behavior of the system, expresses it as a temporal-logic formula, and attempts to use the model checker to validate the hypothesis. The process is iterated while the user gains knowledge about the system. In our opinion, this use of model checking has not been emphasized enough in the literature. To further help the user understand system behaviors, in this paper we introduce temporal-logic queries and use a technique similar to symbolic model checking to *infer* temporal properties as opposed to merely checking them.

This work was partly motivated by the recent interest in deriving invariants of software for comprehension, documentation, or evolution [6, 10, 12]. Inferring invariants is not a new idea, but the traditional objective is to assist in theorem proving [e.g., 2]. We believe that inferring properties is particularly useful for software models, which, unlike hardware, often lack explicit correctness criteria.

Let us first consider inferring invariants. Observe that when the reachable state space can be computed symbolically, we in fact obtain the *strongest invariant* of the system. Although this invariant contains a tremendous amount of useful information, it is likely to be too complex for the user to understand. But note that most interesting invariants in practice involve only a small number of atomic propositions. So, one way to extract useful information from the reachable states is to project its symbolic representation onto a small subset of atomic propositions, thereby deriving a weaker, but more comprehensible, invariant. (In fact, in version 2.5 of CMU's SMV model checker, a user can select a subset of atomic propositions on which to project the reachable states.) We have found situations in which insightful information can be obtained even if we project the reachable states on only singleton sets or pairs of atomic propositions.

* William Chan did this research in the Department of Computer Science and Engineering at the University of Washington. On 26 November 1999, one week after successfully defending his doctoral dissertation, William Chan died in an automobile accident. For requests concerning this paper, please contact Paul Beame, Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, beame@cs.washington.edu

E.A. Emerson and A.P. Sistla (Eds.): CAV 2000, LNCS 1855, pp. 450–463, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Inferring invariants is a special case of evaluating temporal-logic queries. A query is a formula in which a special symbol ?, called a placeholder, appears exactly once. An example CTL query is AG?. The semantics of a query is a proposition p such that replacing ? with p in the query results in a formula that holds in a given model and is as strong as possible. Therefore, the query AG? evaluates to the strongest p that makes AGp hold; in other words, it represents the strongest invariant. More complex examples include AG($req \rightarrow$ A ((?$\vee$AG$\neg ack$)Wack)), which asks what is true between the receipt of a request and the transmission of an acknowledgement, and AF($startup\_complete \vee$ AG?), which roughly asks, what is eventually always true in the case that the startup operation cannot be completed.

As it turns out, however, not every query is meaningful. For example, in most models, the desired proposition defined above cannot be found for the query AF?. Indeed, we show that identifying the "valid" queries is intractable (EXPTIME-complete for CTL queries), so we resort to a conservative approach: We define a class of CTL queries that are guaranteed to be valid, and furthermore can be evaluated in time linear in the size of the model and in the length of the query. That is, the asymptotic worst-case time complexity is the same as that of CTL model checking. Though syntactically quite restricted, the class contains interesting queries such as the two examples given above.

In addition to deriving temporal properties based on a given pattern, the technique can also be used to provide more feedback to the user in model checking, such as providing a partial explanation when the property checked holds, and diagnostic information when it does not. Suppose we would like to check the invariant AG($x \vee y$). We can evaluate the query AG?. Assume that after projecting the strongest invariant on x and y, we obtain AG($x \wedge y$) as an inferred formula. Note that this formula is stronger than the one we wanted to check. Not only can we conclude that AG($x \vee y$) holds, we can also inform the user of the stronger property. This information can either serve as an explanation of the verification result ($x \vee y$ is an invariant because $x \wedge y$ is), or as an indication that the checked property is in a sense vacuously true. Furthermore, in case a formula is falsified, apart from obtaining a counterexample, the user can pose queries to acquire more feedback. For example, if AG($req \rightarrow$ AFack) is false, that is, a request is not always followed by an acknowledgement, we can ask what can guarantee an acknowledgement, AG(? $\rightarrow$ AFack).

The rest of the paper is organized as follows. We review CTL model checking in Section 2, Section 3 gives the main technical results, while Section 4 explains how to simplify a proposition so the user can understand it. Section 5 describes some initial experience of the technique. We conclude in Section 6 with some discussion of future work.

2 Background

This section gives an overview of CTL model checking [3]. Note that, to facilitate our definition of CTL queries, our formulation of CTL is non-standard.

A *model* is a tuple $\langle Q, Q_0, \Delta, X, L\rangle$, where Q is a finite set of *states*, $Q_0 \subseteq Q$ is the set of *initial states*, $\Delta \subseteq Q \times Q$ is the *transition relation*, X is a finite set of *atomic propositions*, and the function $L\colon Q \rightarrow \mathcal{P}(X)$ maps each state to a set of atomic

propositions that are true at the state. The transition relation Δ is assumed to be total; that is, for every $q \in Q$, there exists a *successor* $q' \in Q$ with $\langle q, q' \rangle \in \Delta$. A *path* is an infinite sequence of states in which each consecutive pair of states belongs to Δ. A state is *reachable* if it appears on some path starting from some initial state.

Properties about a model can be specified in the Computation Tree Logic (CTL). CTL formulas consists of atomic propositions, Boolean operators, path quantifiers, and temporal operators. Formally,

- any atomic proposition and *true* are CTL formulas, and
- if ϕ and ψ are CTL formulas, then $\neg\phi$, $\phi \vee \psi$, $\mathsf{AX}\phi$, $\mathsf{A}\,(\phi\,\mathring{\mathsf{W}}\,\psi)$, and $\mathsf{A}\,(\phi\,\mathsf{U}\,\psi)$ are also CTL formulas.

As usual, A is the universal path quantifier, X is the *next-time* operator, and U is the *strong until* operator. We call the operator $\mathring{\mathsf{W}}$ the *overlapping weak until* operator, which is like the dual of U, that is, $\phi\,\mathring{\mathsf{W}}\,\psi \equiv \neg(\neg\psi\,\mathsf{U}\,\neg\phi)$.[1] Intuitively, $\mathsf{X}\phi$ means that ϕ is true in the next state, $\phi\,\mathsf{U}\,\psi$ means that ϕ remains true until ψ becomes true, and $\phi\,\mathring{\mathsf{W}}\,\psi$ means that either ϕ is true forever, or ϕ remains true until both ϕ and ψ become true. A formula is *propositional* (or, simply, abusing terminology, a *proposition*) if it does not contain temporal operators X, $\mathring{\mathsf{W}}$, or, U.

Assume a fixed model M. We write $q \models \phi$ if the CTL formula ϕ is true at state q. The truth value of a CTL formula at a state q_0 is then defined as follows (x is any atomic proposition, and ϕ and ψ are CTL formulas): 1. $q_0 \models \mathit{true}$. 2. $q_0 \models x$ iff $x \in L(q_0)$. 3. $q_0 \models \neg\phi$ iff it is not the case that $q_0 \models \phi$. 4. $q_0 \models \phi \vee \psi$ iff either $q_0 \models \phi$ or $q_0 \models \psi$. 5. $q_0 \models \mathsf{AX}\phi$ iff $q_1 \models \phi$ for every q_1 with $(q_0, q_1) \in \Delta$. 6. $q_0 \models \mathsf{A}\,(\phi\,\mathsf{U}\,\psi)$ iff for every path $q_0, q_1, q_2, \ldots$, there exists an $i \geq 0$ with $q_i \models \psi$, and $q_j \models \phi$ for all $j < i$. 7. $q_0 \models \mathsf{A}\,(\phi\,\mathring{\mathsf{W}}\,\psi)$ iff for every path $q_0, q_1, q_2, \ldots$, for all $i \geq 0$, if $q_j \models \neg\psi$ for all $j < i$, then $q_i \models \phi$.

We write $S \models \phi$ if $q \models \phi$ for each $q \in S$. We say that M satisfies ϕ, written $M \models \phi$, if ϕ is true at every initial state of M. The CTL *model-checking* problem is, given a model and a CTL formula, determine whether the model satisfies the formula. A formula is *valid* if it is satisfied by every model. As usual, we write $\phi \Rightarrow \psi$ if $(\neg\phi) \vee \psi$ is valid, and write $\phi \Leftrightarrow \psi$ if $\phi \Rightarrow \psi$ and $\psi \Rightarrow \phi$. Note that if we have $\phi \Rightarrow \psi$ and $M \models \phi$, then we also have $M \models \psi$.

We define the usual abbreviations $\mathit{false} \equiv \neg\mathit{true}$, $\phi \rightarrow \psi \equiv (\neg\phi) \vee \psi$, and $\phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi)$, as well as

$$\mathsf{AG}\phi \equiv \mathsf{A}\,(\phi\,\mathring{\mathsf{W}}\,\mathit{false}) \qquad\qquad \mathsf{AF}\phi \equiv \mathsf{A}\,(\mathit{true}\,\mathsf{U}\,\phi).$$

The operator G is the *global* operator, and F is the *future* operator. For symmetry, we also define the *weak until* operator W and the *overlapping strong until* operator $\mathring{\mathsf{U}}$:

$$\mathsf{A}\,(\phi\,\mathsf{W}\,\psi) \equiv \mathsf{A}\,(\phi \vee \psi\,\mathring{\mathsf{W}}\,\psi) \qquad\qquad \mathsf{A}\,(\phi\,\mathring{\mathsf{U}}\,\psi) \equiv \mathsf{A}\,(\phi\,\mathsf{U}\,\phi \wedge \psi).$$

[1] In other words, the operator $\mathring{\mathsf{W}}$ is the same as the so-called "release" operator V with the operands swapped: $\phi\,\mathring{\mathsf{W}}\,\psi \equiv \psi\,\mathsf{V}\,\phi$ [9]. Our definition of CTL might remind the reader of the sublogic ACTL. We are actually defining the full CTL here because we allow negating arbitrary formulas. We do not explicitly define the existential fragment because it is not needed in this paper.

In addition it can be shown that

$$\mathsf{A}\,(\phi\,\mathring{\mathsf{W}}\,\psi) \Leftrightarrow \mathsf{A}\,(\phi\,\mathsf{W}\,\phi \wedge \psi) \qquad \mathsf{A}\,(\phi\,\mathsf{U}\,\psi) \Leftrightarrow \mathsf{A}\,(\phi\,\mathsf{W}\,\psi) \wedge \mathsf{AF}\psi.$$

To summarize the different versions of until operators, intuitively $\phi\mathcal{U}\,\psi$ means that ϕ remains true until ψ is true if $\mathcal{U} \in \{\mathsf{W}, \mathsf{U}\}$, and until both ϕ and ψ are true if $\mathcal{U} \in \{\mathring{\mathsf{W}}, \mathring{\mathsf{U}}\}$. If $\mathcal{U} \in \{\mathsf{W}, \mathring{\mathsf{W}}\}$, we also allow ψ to never hold provided ϕ is true forever.

For any CTL formula ϕ, let $[\![\phi]\!]$ denote the set of states in which ϕ is true. The model-checking problem is then equivalent to determining whether the set of initial states is a subset of $[\![\phi]\!]$. For any atomic proposition x, the set $[\![x]\!]$ is easy to find. For any state set S, define $\mathit{pre}_\forall(S)$ as

$$\mathit{pre}_\forall(S) = \{\ q \in Q \mid \forall q'.\ \text{if } \langle q, q'\rangle \in \Delta \text{ then } q' \in S\ \},$$

the set of states with every successor in S. The following equations hold for any CTL formulas ϕ and ψ:

$$[\![\neg\phi]\!] = Q \setminus [\![\phi]\!] \qquad [\![\mathsf{A}\,(\phi\,\mathsf{U}\,\psi)]\!] = \mu Z.\ ([\![\psi]\!] \cup ([\![\phi]\!] \cap \mathit{pre}_\forall(Z)))$$
$$[\![\phi \vee \psi]\!] = [\![\phi]\!] \cup [\![\psi]\!] \qquad [\![\mathsf{A}\,(\phi\,\mathring{\mathsf{W}}\,\psi)]\!] = \nu Z.\ ([\![\phi]\!] \cap ([\![\psi]\!] \cup \mathit{pre}_\forall(Z)))$$
$$[\![\mathsf{AX}\phi]\!] = \mathit{pre}_\forall([\![\phi]\!])$$

where μ and ν are respectively the least fixed-point and greatest fixed-point operators. It can be shown that $\mathit{pre}_\forall$ and these fixed points can be computed in time linear in the size of the model, which is denoted $|M|$ and defined as $|Q| + |\Delta|$. This suggests a model-checking algorithm that given ϕ, recursively converts the subformulas of ϕ to sets of states in an inside-out manner. The algorithm thus runs in time linear in the size of the model and in the length of the formula. Because the formula is evaluated by computing predecessors of states, we say that the algorithm is based on *backward traversals*.

3 CTL Queries

The *placeholder* is a special symbol ?. A *CTL query* is a string in which the placeholder appears exactly once, and for any CTL formula ϕ, substituting ϕ for ? results in a CTL formula. A query is *positive* if the placeholder appears under an even number of negations; otherwise, it is *negative*. For example, the query AG? is positive, while $\mathsf{AG}(? \rightarrow \mathsf{AF}\mathit{ack})$ is negative. We use $\gamma(p)$ to denote the result of replacing the placeholder with p in the query γ.

For any query γ and any formulas ϕ and ψ, we write $\phi \Rightarrow^\gamma \psi$ for $\phi \Rightarrow \psi$ if γ is positive, and for $\psi \Rightarrow \phi$ if γ is negative. Intuitively, the lemma below shows that, if we view a query as a function that maps formulas to formulas, then depending on the polarity of the query, the function is either monotonically increasing or monotonically decreasing.

Lemma 1. *For all formulas ϕ and ψ and every query γ, if $\phi \Rightarrow \psi$, then $\gamma(\phi) \Rightarrow^{\gamma} \gamma(\psi)$.*

Proof Idea. Apply structural induction. Note that the truth of this lemma relies on the fact that we do not allow bi-implication $\leftrightarrow$ in CTL queries. □

If we have $M \models \gamma(p)$ for some proposition p, we say that p is a *solution* to γ in M. We write $\gamma(\top)$ and $\gamma(\bot)$ for $\gamma(\mathit{true})$ and $\gamma(\mathit{false})$ respectively if γ is positive, and $\gamma(\mathit{false})$ and $\gamma(\mathit{true})$ respectively otherwise. Checking whether a given query has a solution in a given model can be reduced to model checking.

Lemma 2. *For every query γ and every model M, we have 1. γ has a solution in M if and only if $M \models \gamma(\top)$, and 2. every proposition is a solution to γ in M if and only if $M \models \gamma(\bot)$.*

Proof. The *if* direction of 1 and the *only-if* direction of 2 are trivial. For the rest of the proof: Consider any proposition p. Because $\mathit{false} \Rightarrow p \Rightarrow \mathit{true}$, $Lemma$ 1 implies $\gamma(\mathit{false}) \Rightarrow^{\gamma} \gamma(p) \Rightarrow^{\gamma} \gamma(\mathit{true})$, or $\gamma(\bot) \Rightarrow \gamma(p) \Rightarrow \gamma(\top)$. Therefore, if $M \models \gamma(\bot)$, then $M \models \gamma(p)$, proving the *if* direction of 2. Also, if $M \models \gamma(p)$, then $M \models \gamma(\top)$, proving the *only-if* direction of 1. □

3.1 Exact Solutions and Valid Queries

Finding an arbitrary solution to a query is not very interesting; finding all solutions does not seem useful either, because there are likely to be too many of them. Instead, we would like to find a single solution that summarizes all solutions. We say that a solution s to γ in M is *exact* if for every solution p, we have $s \Rightarrow^{\gamma} p$. (It is not hard to see from Lemma 1 that, if s is exact, then p is a solution *if and only if* $s \Rightarrow^{\gamma} p$.) Solving a query means finding an exact solution to a given query in a given model.

Note that not every query has an exact solution in a model. Consider the query AF?. Suppose x and y are solutions. Note that $x \wedge y$ is not necessarily a solution because x and y may hold at different states on the paths. In this case the query has no exact solutions in the model. We say that a query is *valid* if it has an exact solution in every model.

For any query γ and any formulas ϕ and ψ, we write $\phi \wedge^{\gamma} \psi$ for $\phi \wedge \psi$ if γ is positive, and for $\phi \vee \psi$ if γ is negative. We say that γ is *distributive over conjunction*, if for any propositions p_1 and p_2, we have $(\gamma(p_1) \wedge \gamma(p_2)) \Leftrightarrow \gamma(p_1 \wedge^{\gamma} p_2)$.

Lemma 3. *A query is valid if and only if it has a solution in every model and is distributive over conjunction.*

Proof. Fix any CTL query γ. For the *if* direction: Consider any model M. Let P be the non-empty set of solutions of γ in M, and let s be $\bigwedge^{\gamma} P$, which we claim is an exact solution. By the definition of P, we know that $M \models \bigwedge_{r \in P} \gamma(r)$. By distributivity, the formula is equivalent to $\gamma(s)$, and therefore s is a solution. Clearly, we have $s \Rightarrow^{\gamma} r$ for every $r \in P$, so s is exact.

For the *only-if* direction: We only need to show that assuming γ is valid , it is distributive over conjunction. That is, we want to show that for any model M, we have $M \models (\gamma(p_1) \wedge \gamma(p_2)) \leftrightarrow \gamma(p_1 \wedge^{\gamma} p_2)$ for all propositions p_1 and p_2. For the $\leftarrow$ direction: Because we have $p_1 \wedge^{\gamma} p_2 \Rightarrow^{\gamma} p_1$, by Lemma 1, we have $\gamma(p_1 \wedge^{\gamma} p_2) \Rightarrow \gamma(p_1)$, which

implies $M \models \gamma(p_1 \wedge^\gamma p_2) \rightarrow \gamma(p_1)$. The argument for p_2 is symmetric. For the $\rightarrow$ direction: If M does not satisfy $\gamma(p_1) \wedge \gamma(p_2)$, we are done. Otherwise, let s be an exact solution to γ in M. By definition, we have $s \Rightarrow^\gamma p_1$ and $s \Rightarrow^\gamma p_2$, and therefore $s \Rightarrow^\gamma (p_1 \wedge^\gamma p_2)$. Lemma 1 now implies that $\gamma(s) \Rightarrow \gamma(p_1 \wedge^\gamma p_2)$. Since by definition $M \models \gamma(s)$, we also have $M \models \gamma(p_1 \wedge^\gamma p_2)$. □

Unfortunately, identifying valid queries is hard in general.

Theorem 1. *Determining whether a given CTL query is valid is complete for EXPTIME.*

Proof. We show that the problem of identifying queries that always admit exact solutions and are distributive over conjunction is equivalent to CTL formula validity, which is complete for EXPTIME [5, 7]. For a reduction to CTL validity: By Lemma 2, a CTL query γ always has an exact solution if and only if the formula $\phi_1 = \gamma(\top)$ is valid, and, by definition, γ is distributive if and only if the formula $\phi_2 = (\gamma(x_1) \wedge \gamma(x_2)) \leftrightarrow \gamma(x_1 \wedge^\gamma x_2)$ is valid, where x_1 and x_2 are atomic propositions not appearing in γ. So we simply check whether the formula $\phi_1 \wedge \phi_2$ is valid. To reduce from CTL validity, observe that a CTL formula ϕ is valid if and only if for an atomic proposition x not appearing in ϕ, the query $\mathsf{A}\,(x \vee \phi\, \mathsf{W}\, ?)$ always has a solution and is distributive over conjunction. □

We can solve a valid query using a naïve approach: Enumerate the exponentially many possible assignments to the atomic propositions. More explicitly, given a set X of atomic propositions, an *assignment* is a proposition of the form

$$\bigwedge_{x \in Y} x \wedge \bigwedge_{x \in X \setminus Y} \neg x$$

for some $Y \subseteq X$. A *satisfying assignment* of a proposition p is an assignment a with $a \Rightarrow p$. Note that every proposition is equivalent to the disjunction of its satisfying assignments. If γ is a positive query and P is the set of every proposition $\neg p$ such that p is an assignment and $\neg p$ is a solution to γ, then it can be shown that $\bigwedge P$ is an exact solution. If γ is negative, it can be shown that $\bigvee P'$ is an exact solution, where P' is the set of every assignment p that is a solution to γ.

Lemma 4. *Given a valid CTL query γ and a model M with atomic propositions X, solving γ in M can be done in time $O(|M|\,|\gamma|\,2^{|X|})$.*

Proof. We assume γ is positive; the case when γ is negative is similar. Let s denote $\bigwedge P$, where P is defined above. We claim that s is an exact solution. It is a solution by the definition of P and the distributivity of γ. To see that it is exact, we want to show $s \Rightarrow r$ for every solution r. We are done if r is a tautology. Otherwise, let A be the non-empty set of all satisfying assignments of $\neg r$. Notice that r is equivalent to $\bigwedge_{a \in A} \neg a$. Consider any $a \in A$. By definition, we have $a \Rightarrow \neg r$, or $r \Rightarrow \neg a$. Because r is a solution, by Lemma 1, $\neg a$ is also a solution. So by the definition of P, we have $\neg a \in P$, and therefore $s \Rightarrow \neg a$, for every $a \in A$. Hence, we have $s \Rightarrow \bigwedge_{a \in A} \neg a$, or equivalently, $s \Rightarrow r$.

Computing P amounts to solving $2^{|X|}$ model-checking problems, each of which takes time $O(|M|\,|\gamma|)$, so the total running time is $O(|M|\,|\gamma|\,2^{|X|})$. □

A time complexity exponential in the number of atomic propositions is hardly desirable. But notice that only backward traversals are used. We now show how, using mixed

forward and backward traversals, the exponential factor can be removed for a subclass of valid queries.

3.2 CTL$^{\mathsf{v}}$ Queries

Although valid queries are hard to identify, we can syntactically define classes of queries that are guaranteed to be valid . Intuitively, there are two major cases in which a query is not distributive over conjunction. The first case is when the placeholder appears within the scope of a temporal operator that is under an odd number of negations, such as $\neg\mathsf{AG?}$. These queries are concerned about what happens on *some* paths. If ϕ_1 and ϕ_2 are true on some paths, we do not know whether $\phi_1 \wedge \phi_2$ holds on any path. (We do know that $\phi_1 \vee \phi_2$ is true on some paths, but this is not sufficient.) The second case is when the placeholder appears on the right hand side of untils, e.g., $\mathsf{AF?}$. Such queries ask what will eventually happen. Even if ϕ_1 and ϕ_2 eventually hold, they may not hold in the same states along the paths. There are many exceptions to the second case, however, such as $\mathsf{A}(\phi\,\mathsf{W}\,\neg\phi \wedge \mathsf{?})$ and $\mathsf{AFAG?}$. Our strategy is to define a class of queries that excludes these known problems while allowing for the exceptions.

We first define two additional until operators, the *disjoint weak until* $\overline{\mathsf{W}}$ and the *disjoint strong until* $\overline{\mathsf{U}}$, as

$$\mathsf{A}(\phi\,\overline{\mathsf{W}}\,\psi) \equiv \mathsf{A}(\phi\,\mathsf{W}\,\neg\phi \wedge \psi) \qquad \mathsf{A}(\phi\,\overline{\mathsf{U}}\,\psi) \equiv \mathsf{A}(\phi\,\mathsf{U}\,\neg\phi \wedge \psi)$$

Formally, we define the class of *CTL$^{\mathsf{v}}$ queries* as the smallest set of queries satisfying the following:

- $\mathsf{?}$ and $\neg\mathsf{?}$ are CTL$^{\mathsf{v}}$ queries.
- If ϕ is a CTL formula and γ is a CTL$^{\mathsf{v}}$ query, then $\phi \vee \gamma$, $\mathsf{AX}\gamma$, $\mathsf{A}(\gamma\,\mathring{\mathsf{W}}\,\phi)$, and $\mathsf{A}(\phi\,\overline{\mathsf{W}}\,\gamma)$ are also CTL$^{\mathsf{v}}$ queries.
- A persistence query is also a CTL$^{\mathsf{v}}$ query.

The class of *persistence queries* is defined as follows:

- If γ is a CTL$^{\mathsf{v}}$ query, then $\mathsf{AG}\gamma$ is a persistence query.
- If γ is a persistence query and ϕ is a CTL formula, then $\phi \vee \gamma$, $\mathsf{AX}\gamma$, $\mathsf{A}(\phi\,\mathsf{W}\,\gamma)$, and $\mathsf{A}(\phi\,\mathsf{U}\,\gamma)$ are persistence queries.

Two queries γ_1 and γ_2 are equivalent, written $\gamma_1 \Leftrightarrow \gamma_2$, if we have $\gamma_1(\phi) \Leftrightarrow \gamma_2(\phi)$ for every ϕ. Additional CTL$^{\mathsf{v}}$ queries are allowed using these equivalences:

$$\gamma \vee \phi \Leftrightarrow \phi \vee \gamma \qquad \mathsf{AG}\gamma \Leftrightarrow \mathsf{A}(\gamma\,\mathring{\mathsf{W}}\,\mathit{false})$$
$$\mathsf{AF}\gamma \Leftrightarrow \mathsf{A}(\mathit{true}\,\mathsf{U}\,\gamma) \qquad \mathsf{A}(\gamma\,\mathsf{W}\,\phi) \Leftrightarrow \mathsf{A}(\phi \vee \gamma\,\mathring{\mathsf{W}}\,\phi).$$

In other words, in CTL$^{\mathsf{v}}$ queries: 1. Negations can only be applied to the placeholder or to CTL formulas. 2. The placeholder cannot appear on either side of $\mathring{\mathsf{U}}$ or $\overline{\mathsf{U}}$, on the left hand side of $\overline{\mathsf{W}}$ or U, or on the right hand side of $\mathring{\mathsf{W}}$. 3. If the placeholder appears on the right hand side of W or U, then there must be an AG between the placeholder and the until. The first restriction on negation is to avoid querying existentially about paths. The second restriction on untils is to ensure that we do not have any eventuality

obligation (which may not be fulfillable in every model). The third restriction rules out queries like AF? but allows valid queries like AFAG?.

Note that $\overline{\mathsf{W}}$ and $\overline{\mathsf{U}}$ are not monotone with respect to the left operand. But since we do not allow the placeholder to appear on their left hand side, this is not a problem and Lemma 1 still holds.

Examples of queries in this class include $\mathsf{A}\,(\neg \mathit{shutdown}\,\overline{\mathsf{W}}\,?)$: what is true when the first shutdown occurs; $\mathsf{AG}(\mathit{shutdown} \rightarrow \mathsf{AG}?)$: what is invariably true after shutdown; $\mathsf{AG}(? \rightarrow \mathsf{AF}\mathit{ack})$: what is true before an acknowledgement is sent; $\mathsf{AG}(? \rightarrow \neg\mathsf{AF}\mathit{ack})$: what is true so that an acknowledgement may never be sent; AFAG?: what is the set of persistent states, or, roughly, the set of states within which the system eventually stays; and the more complex examples given in Section 1. Note that the notion of persistence here is that of the branching time, which seems less useful than the linear-time notion. We will come back to this issue in Section 6.

Not only are these queries guaranteed to be valid , they can be efficiently solved by mixing forward and backward traversals, i.e., by applying $\mathit{pre}_\forall$ and $\mathit{post}_\exists$, where

$$\mathit{post}_\exists(S) = \{\, q' \in Q \mid \exists q.\ \langle q, q' \rangle \in \Delta \text{ and } q \in S \,\},$$

the set of successors of states in S. For any CTL formula ϕ, the set R_ϕ is defined as

$$R_\phi = \mu Z.\ ((S \cup \mathit{post}_\exists(Z)) \cap [\![\phi]\!]),$$

or the set of states reachable from S going through only the states that satisfy ϕ. Figure 1 shows a procedure Solve that takes a CTL$^\forall$ query and a state set, and returns a state set.

$$\begin{aligned}
\mathit{Solve}(?,\ S) &= S \\
\mathit{Solve}(\neg?,\ S) &= Q \setminus S \\
\mathit{Solve}(\phi \vee \gamma,\ S) &= \mathit{Solve}(\gamma,\ S \setminus [\![\phi]\!]) \\
\mathit{Solve}(\,\mathsf{AX}\gamma\,,\ S) &= \mathit{Solve}(\gamma,\ \mathit{post}_\exists(S)) \\
\mathit{Solve}(\,\mathsf{A}\,(\gamma\,\mathring{\mathsf{W}}\,\phi)\,,\ S) &= \mathit{Solve}(\gamma,\ S \cup R_{\neg\phi} \cup \mathit{post}_\exists(R_{\neg\phi})) \\
\mathit{Solve}(\,\mathsf{A}\,(\phi\,\overline{\mathsf{W}}\,\gamma)\,,\ S) &= \mathit{Solve}(\gamma,\ (S \cup \mathit{post}_\exists(R_\phi)) \setminus [\![\phi]\!]) \\
\mathit{Solve}(\,\mathsf{A}\,(\phi\,\mathsf{W}\,\gamma)\,,\ S) &= \mathit{Solve}(\gamma,\ B) \\
&\qquad \text{where } R = R_{\phi \wedge \neg\gamma(\bot)} \\
&\qquad\quad\ B = (S \cup \mathit{post}_\exists(R)) \setminus ([\![\phi]\!] \cup [\![\gamma(\bot)]\!]) \\
\mathit{Solve}(\,\mathsf{A}\,(\phi\,\mathsf{U}\,\gamma)\,,\ S) &= \mathit{Solve}(\gamma,\ B \cup C) \\
&\qquad \text{where } C = \nu Z.\ (R \cap \mathit{post}_\exists(Z)) \\
&\qquad\quad\ R \text{ and } B \text{ are the same as above}
\end{aligned}$$

Fig. 1: Solving CTL$^\forall$ queries (γ is any CTL$^\forall$ query, ϕ is any CTL formula, and $S \subseteq Q$ is any state set)

The idea is that if γ is any CTL$^\forall$ query, M is any model with initial states Q_0, and S is

the result of $Solve(\gamma, Q_0)$, then the *characteristic function* of S, namely

$$\bigvee_{q \in S} \left(\bigwedge_{x \in L(q)} x \wedge \bigwedge_{x \in X \setminus L(q)} \neg x \right),$$

is an exact solution to γ in M. The procedure $Solve$ runs in time linear in the size of the model and linear in the length of the query.

4 Simplification

Recall that our motivation is to help the user understand the system behaviors. Although an exact solution gives complete information, it is likely to be too complex to comprehend. In this section, we suggest a strategy to cope with the problem by decomposing a proposition into a set of conjuncts (for positive queries) or disjuncts (for negative queries) using projection and don't-care minimization. Decomposition is not a new problem in symbolic model checking, but the usual objective is to produce a small number of small, balanced conjuncts or disjuncts to reduce the time or space for the fixed-point computation [e.g., 11]. Our purpose, rather, is to decompose a proposition into a possibly large number of "simple" pieces.

Without loss of generality, we assume positive queries in this section; disjunctive decomposition for negative queries can then be dealt with using DeMorgan's Law. Our conjunctive decomposition is a conservative approximation in the sense that the conjunction obtained may be weaker than the given proposition. Indeed, our method bears some resemblance to the technique of overlapping projections for approximate traversals [8].

Let $\exists \overline{Y}.\, p$ denote the result of projecting the proposition p onto a set Y of atomic propositions. For any symbolic state-set representation for model checking, an implementation of projection is usually available because it is most likely used to implement $pre_\forall$ and $post_\exists$.

For any propositions p and c, let $p \downarrow c$ be any proposition with $p \wedge c \Leftrightarrow (p \downarrow c) \wedge c$. Intuitively, the proposition $\neg c$ represents a don't-care condition. Typically, an implementation of the operation tries to choose a result that minimizes the representation of $p \downarrow c$, and we assume that a minimized representation is also simpler to human users. For BDDs, many operators can be used for this purpose, such as *restrict* [4]. For a set C of propositions, let $p \downarrow C$ be any proposition with $p \wedge (\bigwedge C) \Leftrightarrow (p \downarrow C) \wedge (\bigwedge C)$. In our implementation, we perform $p \downarrow C$ simply by computing $p \downarrow \bigwedge C$ using *restrict*, although there are other possibilities.

Figure 2 shows a greedy algorithm for approximate conjunctive decomposition. We use $atoms(s)$ to denote the set of atomic propositions appearing in s. With increasing j up to the given k, the algorithm finds nontrivial propositions that are weaker than s and contains only j atomic propositions. Redundant information in the result is reduced by simplifying a candidate conjunct using other conjuncts already computed. The algorithm runs in time exponential in k. However, this is not a serious problem in practice, because the result will be too complicated to understand for large k anyway. In our preliminary experience, we have only used $k \leq 4$.

To reduce noise from the output, before we run the decomposition algorithm, it helps to project the proposition s onto the set of atomic propositions which the user is truly

{Input: proposition s
and k with $0 < k \leq |atoms(s)|$ }
$\mathcal{C} := \emptyset$
for $j := 1$ to k
for each $Y \subseteq atoms(s)$ with $|Y| = j$
$r := (\exists \overline{Y}.\, s) \downarrow \mathcal{C}$
if $r \not\Leftrightarrow$ *true* and $r \not\Leftrightarrow s$
$\mathcal{C} := \mathcal{C} \cup \{r\}$
fi
end
end
{Output: $\mathcal{C}$ with $s \Rightarrow \bigwedge \mathcal{C}$}

Fig. 2: Approximate conjunctive decomposition of a proposition

interested in. One way to find out these interesting atomic propositions is to examine the temporal-logic formulas to be checked. Sometimes the number of relevant atomic propositions might appear large, but the user may only want to derive properties of a restricted form. For example, if a subset of the atomic propositions contains x_0, x_1, $\ldots$, x_n, the user may not be interested in each of them individually, but only in their disjunction. In this case, we can create a new atomic proposition d and compute

$$s \wedge (d \leftrightarrow (x_0 \vee x_1 \vee \cdots \vee x_n)).$$

We then project out $x_0, x_1, \ldots, x_n$, and use the result as an input to the decomposition algorithm.

A final remark is that, for simplicity, we have been focusing on only atomic propositions in our discussion. However, a model is often specified as a high-level program with some of its variables ranging over finite domains. In this case, several atomic propositions are used to encode a single source-level variable. When we perform projection and decomposition, we actually operate on these source-level variables instead of the atomic propositions to obtain more meaningful results.

5 Applications

In addition to allowing the user to infer properties based on a given pattern, temporal-logic queries also suggest an alternative model-checking algorithm. Given a model M and a formula ϕ, instead of determining $M \models \phi$ using the standard backward traversals, we can find a query γ with $\phi \Leftrightarrow \gamma(p)$ for some proposition p, and then compute its exact solution s in M. We have $M \models \phi$ if and only if $s \Rightarrow^{\gamma} p$. This gives a model-checking algorithm with mixed forward and backward traversals. An advantage of this over the conventional approach is that, in case ϕ does not hold, the formula $\gamma(s)$ gives insights into why ϕ fails. As an example, suppose that we check $\mathsf{AG}((x \wedge y) \rightarrow \mathsf{AF}\mathit{ack})$ by evaluating $\mathsf{AG}(? \rightarrow \mathsf{AF}\mathit{ack})$, and obtain $x \wedge y \wedge z$ as an exact solution. This tells us that the formula does not hold because, apart from x and y, the condition z is also necessary for *ack* to occur.

Furthermore, the approach can be used for detecting a particular form of vacuity [1]. Let $\widehat{s}$ be $\exists\overline{atoms(p)}.\ s$, that is, the projection of s on to the atomic propositions appearing in p. Because $\widehat{s}$ is the strongest (or weakest, for negative queries) of the solutions that involve only the atomic propositions in $atoms(p)$, we have $M \models \phi$ if and only if $\widehat{s} \Rightarrow^{\gamma} p$. However, if we have $\widehat{s} \Rightarrow^{\gamma} p$ but $\widehat{s} \not\Rightarrow p$, we may say that ϕ trivially holds because the stronger formula $\gamma(\widehat{s})$ also holds. For the previous example, if we obtain x after projecting the exact solution to $\mathsf{AG}(? \rightarrow \mathsf{AF}ack)$ on x and y, we know that y is not needed to produce *ack* and that the original formula holds vacuously. Or the user may suspect that $\mathsf{AG}(x \rightarrow y)$ holds, and can verify this using model checking to learn more about the model.

In the rest of this section, we report on some initial experience of applying temporal-logic queries to two SMV models. We found that the technique is most useful when unexpected properties are inferred.

5.1 A Cache Consistency Protocol

We applied our algorithm to an abstract model of a cache consistency protocol that comes with CMU's SMV 2.5.3 distribution.[2] The model has 3408 reachable states. Among other components, it consists of three processors p_0, p_1, and p_2. The temporal-logic properties specified in the program are concerned with the propositions $p_0.readable$, $p_0.writable$, $p_1.readable$, and $p_1.writable$. For example, an invariant listed is

$$\mathsf{AG}\neg(p_0.writable \wedge p_1.writable). \tag{1}$$

That is, p_0 and p_1 are never simultaneously writable. Note that no properties listed in the code are about p_2. Its correctness was probably assumed by the symmetries in the code.

We asked the query $\mathsf{AG}?$, and projected the exact solution obtained onto $p_i.readable$ and $p_i.writable$ for $i \in \{0, 1, 2\}$. Using the conjunctive decomposition algorithm in Figure 2 with $k = 4$, the following invariants, in addition to Formula (1) above, were inferred:

$$\mathsf{AG}\neg p_2.writable \tag{2}$$

$$\mathsf{AG}\neg p_2.readable \tag{3}$$

$$\mathsf{AG}(p_0.writable \rightarrow p_0.readable) \tag{4}$$

$$\mathsf{AG}(p_1.writable \rightarrow p_1.readable) \tag{5}$$

Formulas (4) and (5) are evident from the code. However, Formulas (2) and (3) are surprising. They indicate that p_2 is never readable nor writable, and therefore p_2 is not symmetric to p_0 or p_1. Upon closer examination of the code, we found a typo in the SMV program that caused p_2's faulty behaviors. We fixed the error, and, as expected, inferred that Formulas (2) and (3) no longer hold, and that the model satisfies

[2] File `gigamax.smv` in `http://www.cs.cmu.edu/~modelcheck/smv/smv.r2.5.3.1d.tar.gz`

$\mathsf{AG}(p_i.writable \rightarrow p_i.readable)$ and $\mathsf{AG}\neg(p_i.writable \wedge p_j.writable)$ for every distinct $i, j \in \{0, 1, 2\}$. In addition, we also discovered

$$\mathsf{AG}((p_i.readable \wedge p_j.readable) \rightarrow \neg p_k.writable) \tag{6}$$

for every distinct $i, j, k \in \{0, 1, 2\}$. It says that if any two of the processors are readable, then the remaining one cannot be writable. This is not a natural property that one would expect from every cache consistency protocol.

5.2 A Shuttle Digital Autopilot

Another example that we looked at was an SMV model of the "shuttle digital autopilot engines out (3E/O) contingency guidance requirements" in the NuSMV 1.1 distribution.[3] There are 70 source-level variables and over 10^{14} reachable states.

One of the properties listed in the SMV program is

$$\mathsf{AG}(\neg cg.idle \rightarrow \mathsf{AF}\ cg.finished), \tag{7}$$

which says that the component cg eventually terminates after it is started. We evaluated the query $\mathsf{AG}(? \rightarrow \mathsf{AF}(cg.finished))$, and projected the exact solution on all singleton sets of variables. In addition to the formula above, we also inferred the following two properties:[4]

$$\mathsf{AG}(\neg cs.idle \rightarrow \mathsf{AF}\ cg.finished) \tag{8}$$

$$\mathsf{AG}(\neg start\_guide \rightarrow \mathsf{AF}\ cg.finished). \tag{9}$$

Formula (9) is easy to see from the SMV program and is not very interesting. Formula (8), however, does not seem obvious; it says that after the component cs is started, cg will eventually finish. Given Formulas (7) and (8), it is natural to ask whether there is any causality relationship between $\neg cg.idle$ and $\neg cs.idle$. So we checked the formulas

$$\mathsf{AG}(\neg cs.idle \rightarrow \mathsf{AF}\neg cg.idle) \tag{10}$$

$$\mathsf{AG}(\neg cg.idle \rightarrow \mathsf{AF}\neg cs.idle) \tag{11}$$

and found that the first formula holds while the second does not. Note how this process of model checking and evaluating queries in tandem allowed us to discover relationships between the two components cs and cg.

Another formula listed is

$$\mathsf{AG}((cg.idle \vee cg.finished) \rightarrow \neg\mathsf{AG}((cg.idle \vee cg.finished) \vee \mathsf{AG}\neg cg.finished)). \tag{12}$$

We evaluated the query $\mathsf{AG}(? \rightarrow \neg\mathsf{AG}((cg.idle \vee cg.finished) \vee \mathsf{AG}\neg cg.finished))$, and, to our surprise, obtained *true* as the exact solution. This indicates that the stronger formula

$$\mathsf{AG}\neg\mathsf{AG}((cg.idle \vee cg.finished) \vee \mathsf{AG}\neg cg.finished) \tag{13}$$

[3] The SMV model was written by Sergey Berezin.
`http://afrodite.itc.it:1024/~nusmv/examples/guidance/guidance.smv`

[4] What we call $cs.idle$ here corresponds to `cs.step = undef` in the SMV program.

holds, and that Formula (12) is in a sense vacuously true.

As our last example, $cs.r$ is an enumerated-type variable with a range of size six. Comments in the SMV program suggest checking the six formulas of the form

$$\neg \mathsf{AG}(cs.r \neq c) \tag{14}$$

for every c in the range of $cs.r$, to ensure that the variable may take on any value in its range. We instead asked only one query $\mathsf{AG?}$, and, after projecting the exact solution on $cs.r$, obtained *true*. This implies that there are no constraints on $cs.r$ in the reachable states, and therefore it may take on any of its possible values.

6 Future Work

One interesting direction for future work is to extend the results to Linear Temporal Logic (LTL). All of our definitions extend to LTL in a straightforward way, and Lemmas 1–3 hold for LTL queries as well. The proof of Theorem 1 can be trivially modified for LTL queries, so it can be easily seen that determining whether an LTL query is valid is equivalent to determining LTL formula validity, which is complete for PSPACE. An advantage of LTL queries over CTL queries is the expressiveness. For example, if the CTL formula $\mathsf{AG}(\mathit{req} \rightarrow \mathsf{AF}\mathit{ack})$ does not hold (or equivalently, the LTL formula $\mathsf{G}(\mathit{req} \rightarrow \mathsf{F}\mathit{ack})$ does not hold), the user can ask the LTL query

$$\mathsf{G}(\mathit{req} \rightarrow \mathsf{F}(\mathit{ack} \vee \mathsf{G?}))$$

to find out what is eventually always true if a request is never followed by an acknowledgement. This in essence gives a summary of all the counterexamples to the formula above. Note that the similar CTL query

$$\mathsf{AG}(\mathit{req} \rightarrow \mathsf{AF}(\mathit{ack} \vee \mathsf{AG?}))$$

is much weaker. However, in general, it is not obvious how to evaluate valid LTL queries.

Bibliography

[1] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In O. Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV'97 Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 279–290, Haifa, Israel, June 1997. Springer-Verlag.

[2] S. Bensalem and Y. Lakhnech. Automatic generation of invariants. *Formal Methods in System Design*, 15(1):75–92, July 1999.

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

[4] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems: International Workshop Proceedings*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373, Grenoble, France, June 1989. Springer-Verlag.

[5] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and Systems Sciences*, 30:1–21, 1985.

[6] M. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In ICSE99, editor, *Proceedings of the 1999 International Conference on Software Engineering: ICSE 99*, pages 213–224, Los Angeles, USA, May 1999. ACM.

[7] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and Systems Sciences*, 18:194–211, 1979.

[8] S. G. Govindaraju, D. L. Dill, A. J. Hu, and M. A. Horowitz. Approximate reachability with BDDs using overlapping projections. In *35th Design Automation Conference, Proceedings 1998*, pages 451–456, San Francisco, USA, June 1998. ACM.

[9] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[10] R. Jeffords and C. Heitmeyer. Automatic generation of state invariants from requirements specifications. In FSE6, editor, *Proceedings of the ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering: FSE-6*, pages 56–69, Lake Buena Vista, Florida, USA, November 1998.

[11] K. Ravi, K. L. McMillan, T. R. Shiple, and F. Somenzi. Approximation and decomposition of binary decision diagrams. In *35th Design Automation Conference, Proceedings 1998*, pages 445–450, San Francisco, USA, June 1998. ACM.

[12] M. Vaziri and G. Holzmann. Automatic invariant deduction in Spin. In J.-C. Gregoire, G. J. Holzmann, and D. A. Peled, editors, *The SPIN Verification System: The 4th International Workshop*, Paris, France, November 1998.