Query Checking for Linear Temporal Logic

Samuel Huang and Rance Cleaveland(✉)

University of Maryland, College Park, USA
srhuang@cs.umd.edu, rance@cs.umd.edu

Abstract. The query-checking problem for temporal logic may be formulated as follows. Given a Kripke structure M and a temporal-logic *query* of form $\phi\,[\mathtt{var}]$, which may be thought of as a temporal formula with a missing propositional subformula `var`, find the most precise propositional formula f that, when substituted for `var` in $\phi\,[\mathtt{var}]$, ensures M satisfies the resulting temporal property. Query checking has been used for system comprehension, specification reconstruction, and other related applications in the formal analysis of systems.

In this paper we present an automaton-based methodology for query checking over linear temporal logic (LTL). While this problem is known to be hard in the general case, we show that by exploiting several key observations about the interplay between the input model M and the query $\phi\,[\mathtt{var}]$, we can produce results for many problems of interest. In support of this claim, we report on preliminary experimental data for an implementation of our technique.

1 Introduction

Temporal logics [9] are widely used to specify desired properties of system behavior. Such logics permit the description of how systems should execute over time; tools such as model checkers [4,8] can then be used automatically to determine whether or not certain types of system possess given temporal properties.

The practical utility of model checking and other temporal-logic-based verification technologies relies on the ability of users to define correctly the properties they are interested in. To assist users in this regard, researchers have looked into various forms of automated *temporal-property reconstruction* [1,11,17,18] as a means of helping users to devise temporal specifications from given system specifications. Users may then use these as specifications for the system (useful when systems subsequently have new functionality added, as the new system can be checked against the old specification to ensure backward compatibility); they may also review them as a means of gaining insight into the behavior of a system that may not have been formally specified or verified. One of the most influential lines of work in this area is so-called *temporal logic query checking* [6], which aims to solve the following general problem: given a system, and a temporal formula with a missing (propositional) subformula, "solve" for the missing subformula. As originally formulated by Chan [6], the temporal logic in question was a subset of the branching-time temporal logic CTL [10], for which he gave

© Springer International Publishing AG 2017
L. Petrucci et al. (Eds.): FMICS-AVoCS 2017, LNCS 10471, pp. 34–48, 2017.
DOI: 10.1007/978-3-319-67113-0_3

efficient algorithms for computing most-precise missing formulas. Others have considered different variants of this problem, by considering multiple missing subformulas, for instance, or different logics [5,7,15].

In this paper we consider the problem of *query checking for linear temporal logic* (LTL) [10]. LTL differs from branching-time logics in that one specifies properties of executions, rather than states in a system, and it is often viewed as an easier formalism to master for this reason. It is also the basis for specification languages, such as FORSPEC [2], used in digital hardware design. In the current work we show how automaton-based model-checking techniques may be adapted to yield a solution to the query-checking problem that, while computationally complex in the worst case, exploits structure in the space of possible query solutions to yield better performance. To this end, after reporting on related work and developing needed mathematical preliminaries, we present our technique and report on a preliminary implementation that we are developing.

2 Related Work

Temporal-logic query checking was initially defined and explored by William Chan [6], who considered the problem in the context of the branching-time temporal logic CTL [10]. Chan initially considered a subset of CTL and showed that queries in this subset, which allows the universal path quantifier and places restrictions on the modal operators, can be solved in linear time. This work was subsequently extended to more expressive branching-time logics via alternating-tree automaton constructions [5] and three-valued model checking [15]; this last paper also describes several applications of the technique in areas such as invariant inference and test generation. Other work has studied the problem for classes of infinite-state systems [21].

In contrast to branching time, linear-time query checking has remained relatively unstudied. Chokler et al. [7] consider several variants of LTL query checking and prove complexity results for these problems; however, no implementation or experimental results were reported.

Other researchers have considered the problem of so-called specification mining, in which temporal properties are inferred not from system models, but from execution behavior, using techniques from data mining and machine learning. Such properties hold of the data from which they are generated, but not necessarily of all system behaviors. Emblematic of this work is the dynamic-invariant generation work of Ernst et al. [11], which uses program instrumentation to obtain state information as a program executes and then data mining to identify possible invariants. Other work in this vein couples data mining of execution data with retesting to attempt to remove invalid invariants in the case of Simulink models [1]. Other work has considered the mining of general LTL formulas from run-time data [16].

3 LTL, Kripke Structures and Büchi Automata

This section defines the syntax of LTL and reviews the notions of Kripke structure, Büchi automata, and model checking in LTL. In what follows, we fix a finite non-empty set $\mathcal{A}$ of atomic propositions.

3.1 LTL and Kripke Structures

The syntax of LTL formulas is given by the following grammar.

$$\phi := a \in \mathcal{A} \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{X}\,\phi \mid \phi\,\mathbf{U}\,\phi$$

In addition to the propositional constructs a, $\neg$ and $\vee$, LTL formulas also include the modal operators $\mathbf{X}$, or "next state", and $\mathbf{U}$, or "until". The derived propositional operations $\wedge, \rightarrow$, etc. are defined in the usual manner; we also write $\mathtt{tt}$ as an abbreviation for $a \vee \neg a$ for a designated $a \in \mathcal{A}$ and $\mathtt{ff}$ for $\neg\mathtt{tt}$. We additionally use the following derived temporal operators in the sequel.

$$\mathbf{F}\,\phi \triangleq \mathtt{tt}\,\mathbf{U}\,\phi$$
$$\mathbf{G}\,\phi \triangleq \neg(\mathbf{F}\,\neg\phi)$$
$$\phi_1\,\mathbf{R}\,\phi_2 \triangleq \neg((\neg\phi_1)\,\mathbf{U}(\neg\phi_2))$$

$\mathbf{F}$ and $\mathbf{G}$ are the "eventually" and "always" operators, while $\mathbf{R}$ is sometimes called the "release" operator. We write Φ for the set of LTL formulas.

The semantics of LTL is given as a relation $\models \subseteq (2^{\mathcal{A}})^\omega \times \Phi$. Intuitively, $\pi \models \phi$ holds if $\pi \in (2^{\mathcal{A}})^\omega$, which is an infinite sequence of subsets of $\mathcal{A}$, makes ϕ true. In what follows, if $\pi = A_0A_1\ldots$ then we write $\pi[i] \in 2^{\mathcal{A}}$ for $A_i \subseteq \mathcal{A}$ and $\pi[i..] \in (2^{\mathcal{A}})^\omega$ for the suffix $A_iA_{i+1}\ldots$. The relation $\models$ may now be defined as follows.

- $\pi \models a$ $(a \in \mathcal{A})$ iff $a \in \pi[0]$.
- $\pi \models \neg\phi$ iff $\pi \not\models \phi$.
- $\pi \models \phi_1 \vee \phi_2$ iff $\pi \models \phi_1$ or $\pi \models \phi_2$.
- $\pi \models \mathbf{X}\,\phi$ iff $\pi[1..] \models \phi$.
- $\pi \models \phi_1\,\mathbf{U}\,\phi_2$ iff there is a $j \geq 0$ such that $\pi[j..] \models \phi_1$ and for all $0 \leq i < j$, $\pi[i..] \models \phi_2$.

We often write $[\![\phi]\!] \triangleq \left\{\pi \in (2^{\mathcal{A}})^\omega \mid \pi \models \phi\right\}$ for the set of sequences satisfying ϕ.

LTL formulas are often used to specify properties of systems modeled as *Kripke Structures.*

Definition 1. *A* Kripke Structure *is a quadruple* (S, R, L, i) *where:*

- S *is a non-empty set of* states*;*
- $R \subseteq S \times S$ *is the* transition relation*;*
- $L \in S \rightarrow 2^{\mathcal{A}}$ *is the* labeling function*; and*
- $i \in S$ *is the* initial state.

A Kripke structure encodes the behavior of a system, with S representing system states and the transition relation recording the possible execution steps that are possible when a system is in a given state: when the system is in state s it can evolve in one step to state s' iff $(s, s') \in R$. The labeling function indicates which atomic propositions are true in any given state; if $a \in L(s)$ then a is deemed true s, while if $a \notin L(s)$ it is false. State i is the initial state. In what follows we require Kripke structures to be *left-total*: for every $s \in S$ it must be the case that there is an $s' \in S$ such that $(s, s') \in R$. We also call a Kripke structure *finite-state* if its state set is finite. Semantically, left-total Kripke structure $K = (S, R, L, i)$ gives rise to a subset $[\![K]\!]$ of $(2^{\mathcal{A}})^\omega$ as follows.

- Infinite sequence $s_0 s_1 \ldots \in S^\omega$ is an *execution* of K if $s_0 = i$ and $(s_i, s_{i+1}) \in R$ for all $i \geq 0$.
- K *generates* $\pi = A_0 A_1 \ldots$ iff there is an execution $s_0 s_1 \ldots$ of K such that $A_i = L(s_i)$.
- $[\![K]\!] = \{\pi \in (2^{\mathcal{A}})^\omega \mid K \text{ generates } \pi\}$.

We then write $K \models \phi$, where K is a Kripke structure and ϕ is an LTL formula, iff $[\![K]\!] \subseteq [\![\phi]\!]$.

3.2 Büchi Automata and LTL Model Checking

The LTL model-checking problem may be formulated as follows.

Given: Kripke structure K, LTL formula ϕ
Determine: Does $K \models \phi$?

When K is finite-state the model-checking problem is decidable in time proportional to $|K|$, where $|K|$ is the size of Kripke structure K. A common approach for LTL model checking relies on the use of *Büchi automata*. This section defines these automata and explains their use in LTL model checking.

Büchi automata. Büchi automata are used to recognize so-called ω-regular languages, which are sets of infinite-length sequences of alphabet symbols.

Definition 2. *A Büchi automaton is a quintuple* $(Q, \Sigma, \delta, q_I, F)$*, where:*

- Q *is a finite, non-empty set of* states*;*
- Σ *is a finite, non-empty set of* alphabet symbols*;*
- $\delta \subseteq Q \times \Sigma \times Q$ *is the* transition relation*;*
- q_I *is the* initial state*; and*
- $F \subseteq Q$ *is the set of* accepting states*.*

Let B be a Büchi automaton $(Q, \Sigma, \delta, q_i, F)$. We define the language, $L(B) \subseteq \Sigma^\omega$, of B as follows.

- Given ω-word $w = \alpha_0 \alpha_1 \ldots \in \Sigma^\omega$, define a *run* of B on w to be a sequence $q_0 q_1 \ldots \in Q^\omega$ such that $q_0 = q_I$ and $(q_i, \alpha_i, q_{i+1}) \in \delta$ for all $i \geq 0$.

- A run $q_0 q_1 \ldots \in Q^\omega$ of B on w is *accepting* iff for all $i \geq 0$ there exists $j \geq i$ such that $q_j \in F$.
- $L(B) = \{w \in \Sigma^\omega \mid B \text{ has an accepting run on } w\}$.

The subsets $W \subseteq \Sigma^\omega$ such that $W = L(B)$ for some Büchi automaton B coincide with the so-called ω-regular languages. This class of languages is closed with respect to complementation and intersection; both of these operations can be realized as constructions on Büchi automata. In addition, checking for emptiness of the language of B is decidable in time proportional to the size of B. Algorithmically, this can be done by computing the strongly connected components of B and determining if there is one such component reachable from the start state, containing an accepting state, and having at least one edge from a state in the component to another state in the component. This ensures the existence of at least one accepting run in B, and hence the non-emptiness of $L(B)$.

LTL model checking using Büchi automata. Büchi automata may be used as a basis for *model checking* of finite-state Kripke structures against LTL formulas [19]. Recall the model-checking problem in this case: given finite-state Kripke structure K and LTL formula ϕ, determine whether or not $K \models \phi$. This problem may be solved algorithmically using Büchi automata as follows.

- From K, construct Büchi automaton B_K such that $L(B_K) = [\![K]\!]$.
- From ϕ, construct Büchi automaton $B_{\neg\phi}$ such that $L(B_{\neg\phi}) = [\![\neg\phi]\!]$.
- Construct the Büchi automaton $B_{K,\neg\phi}$ such that $L(B_{K,\neg\phi}) = L(B_K) \cap L(B_{\neg\phi})$ and check if $L(B_{K,\neg\phi}) = \emptyset$. This is true iff $K \models \phi$.

Note that both B_K and $B_{\neg\phi}$ must have alphabet sets $\Sigma = 2^{\mathcal{A}}$. Specifically, transitions in both of these Büchi automata are labeled by subsets of $\mathcal{A}$.

For $K = (S, R, L, i)$, the construction B_K is straightforward: define $B_K = \left(S, 2^{\mathcal{A}}, \delta_K, i, S\right)$, where

$$\delta_K = \{(s, A, s') \mid (s, s') \in R \text{ and } A = L(s)\}.$$

The construction of B_ϕ for LTL formula ϕ is more complex, and a number of approaches may be found in the literature [3, 12–14]. The best techniques yield automata that are $O(3^{|\phi|})$, where $|\phi|$ is the size of formula ϕ.

We close this section by giving an alternative formulation of Büchi automata whose alphabets are $2^{\mathcal{A}}$. The edges in these automata are labeled by propositional formulas constructed from $\mathcal{A}$, rather than subsets of $\mathcal{A}$; the interpretation of such an edge $q \xrightarrow{\gamma} q'$ is that $q \xrightarrow{A} q'$ for every A satisfying γ. These notions are formalized as follows.

- Define the set of propositional formulas Γ over $\mathcal{A}$ by the following grammar.

$$\gamma ::= a \in \mathcal{A} \mid \neg\gamma \mid \gamma \vee \gamma$$

  Note that $\Gamma \subsetneq \Phi$. We use the usual encodings of `tt`, $\wedge$, etc.
- If $A \subseteq \mathcal{A}$ and $\gamma \in \Gamma$ then define $A \models \gamma$ as follows.

- $A \models a$ iff $a \in A$
- $A \models \neg\gamma$ iff $A \not\models \gamma$
- $A \models \gamma_1 \vee \gamma_2$ iff $A \models \gamma_1$ or $A \models \gamma_2$.

We write $[\![\gamma]\!]$ for $\{A \subseteq \mathcal{A} \mid A \models \gamma\}$. Note that $A \models \gamma$ iff $\pi \models \gamma$ for all π such that $\pi[0] = A$.

We sometimes use $A \subseteq \mathcal{A}$ as short-hand for the formula $(\bigwedge_{a\in A} a) \wedge (\bigwedge_{a\notin A} \neg a)$. Note that $[\![A]\!] = \{A\}$ in this case.

Definition 3. *Given $\mathcal{A}$, a Büchi propositional automaton is a tuple (Q, δ, q_I, F), where:*

- *Q is a finite non-empty set of states, with $q_I \in Q$ and $F \subseteq Q$.*
- *$\delta \subseteq (Q \times \Gamma \times Q)$ is the transition relation.*

Based on our interpretation of sets $A \subseteq \mathcal{A}$ as propositions it is easy to see that every Büchi automaton is also a Büchi propositional automaton. An arbitrary Büchi propositional automaton $B = (Q, \delta, q_I, F)$ may also be translated into a traditional Büchi automaton $B' = \left(Q, 2^{\mathcal{A}}, \delta', q_I, F\right)$ by defining

$$\delta' = \{(q, A, q') \mid \exists\gamma.\, (q, \gamma, q') \in \delta \wedge A \in [\![\gamma]\!]\}\,.$$

We define $L(B) = L(B')$. The traditional tableau-based constructions for converting LTL formulas into Büchi automata may easily be adapted to generate Büchi propositional automata with the property that for every pair of automaton states q, q' there is exactly one γ such that $(q, \gamma, q') \in \delta$.

Finally, we give a construction for Büchi propositional automaton B_{12} with $L(B_{12}) = L(B_1) \cap L(B_2)$ for the special case of Büchi propositional automata B_1 and B_2, with every state in B_1 accepting.

Theorem 1. *Let $B_1 = (Q_1, \delta_1, q_1, Q_1)$ and $B_2 = (Q_2, \delta_2, q_2, F_2)$ be Büchi propositional automata. Then $L(B_{12}) = L(B_1) \cap L(B_2)$, where*

$$B_{12} = (Q_1 \times Q_2, \delta_{12}, (q_1, q_2)\,, Q_1 \times F_2)$$

and $((q_1, q_2)\,, \gamma_1 \wedge \gamma_2, (q_1', q_2')) \in \delta_{12}$ iff $(q_1, \gamma_1, q_1') \in \delta_1$ and $(q_2, \gamma_2, q_2') \in \delta_2$.

4 The LTL Query Checking Problem

In LTL query checking we are interested in Kripke structures and LTL formula *queries*, which are formulas containing a missing propositional subformula. The goal in LTL query checking is to construct solutions for the missing subformula. This section defines the problem precisely and proves results that will be used later in our algorithmic solution.

LTL queries correspond to LTL formulas with a missing propositional subformula, which we denote `var`. It should be noted that `var` stands for an unknown proposition over $\mathcal{A}$; it is *not* a propositional variable. The syntax of queries is as follows.

$$\phi := \mathtt{var} \mid a \in \mathcal{A} \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{X}\,\phi \mid \phi\,\mathbf{U}\,\phi$$

In this paper we only consider the case of a single propositional unknown, although the definitions can naturally be extended to multiple such unknowns. We often write $\phi[\mathtt{var}]$ for LTL query with unknown var, and $\phi[\phi']$ for the LTL formula obtained by replacing all occurrences of var by LTL formula ϕ'. We also say that an occurrence of var within $\phi[\mathtt{var}]$ is *positive* if it appears within an even number of instances of $\neg$, and *negative* otherwise. If all occurrences of var in $\phi[\mathtt{var}]$ are positive we say var is *positive* in $\phi[\mathtt{var}]$; if all are negative we say var is *negative* in $\phi[\mathtt{var}]$; if there are both positive and negative occurrences of var in $\phi[\mathtt{var}]$ then var is *mixed* in $\phi[\mathtt{var}]$.

The query-checking problem may now be formulated as follows.

Given: Finite-state Kripke structure K, LTL query $\phi[\mathtt{var}]$
Compute: All $\gamma \in \Gamma$ (i.e. all propositional formulas over $\mathcal{A}$) with $K \models \phi[\gamma]$.

If γ is such that $K \models \phi[\gamma]$, then we call γ a *solution* for K and $\phi[\mathtt{var}]$, and in this case we say that $\phi[\mathtt{var}]$ is solvable for K. Computing all solutions for query checking problem K and $\phi[\mathtt{var}]$ cannot be done explicitly, since the number of propositional formulas is infinite. However, if we define $\gamma_1 \equiv \gamma_2$ to hold if $[\![\gamma_1]\!] = [\![\gamma_2]\!]$, then it is clear that there are only finitely many distinct equivalence classes for Γ. We also say that γ_1 is at least as strong (weak) as γ_2 if $[\![\gamma_1]\!] \subseteq [\![\gamma_2]\!]$ ($[\![\gamma_2]\!] \subseteq [\![\gamma_1]\!]$). We now have the following.

Theorem 2. *Let K be a finite-state Kripke structure and $\phi[\mathtt{var}]$ an LTL query.*

1. *If var is positive in $\phi[\mathtt{var}]$ then there is a finite set (modulo $\equiv$) of strongest solutions for $\phi[\gamma]$.*
2. *If var is negative in $\phi[\mathtt{var}]$ then there is a finite set (modulo $\equiv$) of weakest solutions to $\phi[\gamma]$.*

In some cases these sets of maximal solutions contain a single solution.

Definition 4. *Let $\phi[\mathtt{var}]$ be an LTL query. Then $\phi[\mathtt{var}]$ is:*

- conjunctively covariant *iff for all γ_1, γ_2, $\phi[\gamma_1 \wedge \gamma_2] \equiv \phi[\gamma_1] \wedge \phi[\gamma_2]$; and*
- conjunctively contravariant *iff for all γ_1, γ_2, $\phi[\gamma_1 \vee \gamma_2] \equiv \phi[\gamma_1] \wedge \phi[\gamma_2]$.*

Theorem 3. *Let K be a finite-state Kripke structure, and let $\phi[\mathtt{var}]$ be solvable for K. Then the following hold.*

1. *If var is positive in $\phi[\mathtt{var}]$ and $\phi[\mathtt{var}]$ is conjunctively covariant, then there is a unique strongest solution (modulo $\equiv$) for $\phi[\mathtt{var}]$.*
2. *If var is negative in $\phi[\mathtt{var}]$ and $\phi[\mathtt{var}]$ is conjunctively contravariant, then there is a unique weakest solution (modulo $\equiv$) for $\phi[\mathtt{var}]$.*

As examples, note that $\mathbf{G}\,\mathtt{var}$ is conjunctively covariant and solvable for every K, and that var is positive; it is guaranteed to have a unique strongest solution for any K. So does $\mathbf{G}\,\mathbf{F}\,\mathtt{var}$. On the other hand, $\mathbf{G}(\mathtt{var} \implies \mathbf{F}\,\phi')$ is conjunctively contravariant and solvable for every K, and var appears negatively. Thus, every K has a unique weakest solution for this query.

5 Automaton-Based LTL Query Checking

In this section we show how LTL query checking can be formulated as a problem on Büchi propositional automata whose propositional labels may contain instances of `var`. In this paper we only consider LTL queries in which `var` is either negative or positive; the mixed case will not be dealt with. The approach is based on LTL model checking in that we generate Büchi propositional automata from both a Kripke structure and the negation of an LTL query and compose them; we then search for solutions to `var` that make the language of the composition automaton empty. To formalize these notions, we introduce the following definitions.

5.1 Propositional Queries

Definition 5. *Let* ***var*** *be an unknown proposition. Then* propositional queries *are generated by the following grammar.*

$$\gamma ::= \boldsymbol{var} \mid a \in \mathcal{A} \mid \neg\gamma \mid \gamma \vee \gamma$$

We write $\gamma[\boldsymbol{var}]$ for a generic instance of a propositional template, and $\Gamma[\boldsymbol{var}]$ for the set of all propositional templates involving ***var****.*

It is easy to see that propositional queries form a subset of LTL queries, and that notions of $\gamma[\gamma']$, positive and negative occurrences of `var`, etc., carry over immediately. A *shattering formula* for query $\gamma[\texttt{var}]$ is a propositional formula γ' with the property that $[\![\gamma[\gamma']]\!] = \emptyset$; that is, γ' "makes" $\gamma[\texttt{var}]$ unsatisfiable. We call $\gamma[\texttt{var}]$ *shatterable* if it has a shattering formula. The following is a consequence of the fact that the set of propositional formulas form a Boolean algebra.

Theorem 4. *Let $\gamma[\boldsymbol{var}]$ be shatterable.*

1. *If* ***var*** *is positive in $\gamma[\boldsymbol{var}]$ then there is a unique (modulo $\equiv$) weakest shattering formula for $\gamma[\boldsymbol{var}]$.*
2. *If* ***var*** *is negative in $\gamma[\boldsymbol{var}]$ then there is a unique strongest (modulo $\equiv$) shattering formula for $\gamma[\boldsymbol{var}]$.*

Intuitively, if $\gamma[\texttt{var}]$ is shatterable and `var` is positive, then $\gamma[\texttt{var}]$ can be rewritten as $\texttt{var} \wedge \gamma'$ for some propositional formula γ' (i.e. γ' contains no occurrences of `var`). In this case the weakest shattering formula for $\gamma[\texttt{var}]$ is $\neg\gamma'$. A dual argument holds in the case that `var` is negative in $\gamma[\texttt{var}]$.

5.2 Büchi Query Automata

Büchi query automata are propositional automata with propositional queries labeling transitions.

Definition 6. *Let* $\mathtt{var}$ *be a propositional unknown. A* Büchi query automaton *has form* (Q, δ, q_I, F)*, with finite state set* Q*, initial state* $q_I \in Q$*, accepting states* $F \subseteq Q$*, and transition relation* $\delta \subseteq Q \times \Gamma[\mathtt{var}] \times Q$*.*

Intuitively, a Büchi query automaton is like an LTL query in that it contains a propositional unknown, var, that can be used to change the language accepted by the automaton. Specifically, if var is set to a condition γ' that shatters the edge label $\gamma[\mathtt{var}]$, then any query-automaton edge of form $(q, \gamma[\mathtt{var}], q')$ is no longer available for use in constructing runs of the automaton. Figure 1 illustrates this phenomenon. Thus, by varying var we can thus affect the language accepted by the query automaton.

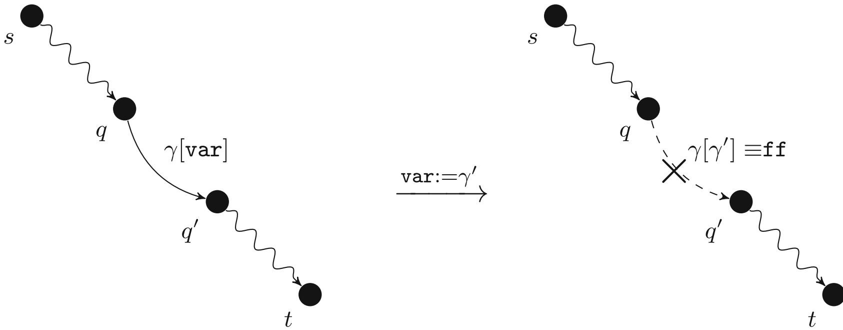

Fig. 1. Shattering edges in a Büchi query automaton. Proposition γ' shatters $\gamma[\mathtt{var}]$, and consequently the edge $(q, \gamma[\mathtt{var}], q')$ is removed.

Formally, if $B[\mathtt{var}]$ is a Büchi query automaton then define $B[\gamma]$ to be the Büchi propositional automaton obtained by replacing all occurrences of var by γ in any edge label within $B[\mathtt{var}]$. We say that γ *shatters* $B[\mathtt{var}]$ if $L(B[\gamma]) = \emptyset$, i.e. if γ renders the language of $B[\mathtt{var}]$ empty. Notions of positive and negative occurrences of var in $B[\mathtt{var}]$, etc., carry over in the obvious manner.

We now note the following correspondence between LTL queries and Büchi query automata.

Theorem 5. *Let* $\phi[\mathtt{var}]$ *be an LTL query. Then there exists a Büchi query automaton* $B_\phi[\mathtt{var}]$ *such that the following hold.*

1. *For all* $\gamma \in \Gamma$, $[\![\phi[\gamma]]\!] = L(B_\phi[\gamma])$.
2. *If* $\mathtt{var}$ *is positive in* $\phi[\mathtt{var}]$ *then* $\mathtt{var}$ *is positive in* $B_\phi[\mathtt{var}]$.
3. *If* $\mathtt{var}$ *is negative in* $\phi[\mathtt{var}]$ *then* $\mathtt{var}$ *is negative in* $B_\phi[\mathtt{var}]$.

The construction of $B_\phi[\mathtt{var}]$ is a straightforward adaptation of the construction of Büchi propositional automata from LTL formulas ϕ.

5.3 LTL Query Checking via Büchi Query Automata

We now explain our approach to LTL query checking. Given finite-state Kripke structure K and LTL query $\phi[\mathtt{var}]$, we perform the following.

1. Construct Büchi (propositional) automaton B_K.
2. Construct Büchi query automaton $B_{\neg\phi}[\mathtt{var}]$.
3. Construct the product query automaton, $B_{K,\neg\phi}[\mathtt{var}]$.
4. Solve for shattering conditions for $B_{K,\neg\phi}[\mathtt{var}]$.

Because of Theorem 5 we know the following. If $\phi[\mathtt{var}]$ is conjunctively covariant and var is positive in $\phi[\mathtt{var}]$, then var is negative in $B_{K,\neg\phi}[\mathtt{var}]$, and the strongest solution for var in $\phi[\mathtt{var}]$ with respect to K coincides with the weakest shattering condition for $B_{K,\neg\phi}[\mathtt{var}]$. The dual result holds in case var is negative in $\phi[\mathtt{var}]$. Thus, solving for shattering conditions in $B_{K,\neg\phi}[\mathtt{var}]$ yields appropriate query solutions for K and $\phi[\mathtt{var}]$.

6 Implementing an LTL Query Checker

Based on the developments given earlier in the paper, to develop a query checker for finite-state Kripke structures and LTL queries $\phi[\mathtt{var}]$ it suffices to construct the product query automaton $B_{K,\neg\phi}[\mathtt{var}]$ and then search for γ that shatter $B_{K,\neg\phi}[\mathtt{var}]$. In this section we highlight some of the algorithmic aspects of this strategy and report on preliminary results of a prototype implementation.

At the outset, we can note that there is one immediate algorithmic solution: enumerate γ and test to see if $L(B[\gamma]) = \emptyset$ by computing the strongly connected components of $B[\gamma]$ and seeing if the start state can reach a successful component (i.e. one with an accepting state and at least one edge from the component back to itself). As there are $2^{2^{|\mathcal{A}|}}$ semantically distinct such γ, this procedure terminates; indeed, this is the basis of the approach outlined in [7]. The complexity of this approach is prohibitive, however, as a sample implementation of ours has shown: even Kripke structures with 10s of states and 10 atomic propositions failed to complete successfully. This is to be expected, given that there are $2^{2^{10}} \geq 1.75 \times 10^{308}$ semantically distinct propositions in this case.

Instead, the approach outlined below pursues two different strategies to reduce the computational effort associated with shattering. One involves exploiting the lattice structure of $2^{2^{\mathcal{A}}}$ to reduce the number of propositions that must be considered; the second combines this idea with a weakening of the problem to require the computation of a single shattering proposition, rather than all such propositions. The next sections provide further details regarding our approach.

6.1 Construct Büchi Automaton B_K

Given a Kripke structure K, constructing the corresponding Büchi automaton B_K is done using the traditional method as described above. There is no query component to the model input, it should be noted.

6.2 Construct Büchi Query Automaton $B_{\neg\phi}[\texttt{var}]$

The LTL3BA package performs translations from standard LTL formulas to Büchi propositional automata. For a given query $\phi[\texttt{var}]$ we convert the formula into a Büchi query automaton by treating `var` as a normal atomic proposition. By default, LTL3BA attempts to remove non-determinism from the output Büchi query automaton, which can increase the number of edges in the automaton containing `var` on their labels. We configure LTL3BA so that removal of non-determinism is not required in order to avoid this extra overhead.

6.3 Construct Product Query Automaton $B_{K,\neg\phi}[\texttt{var}]$

As mentioned before, there is a well-known product construction for composing two Büchi automata into a single one accepting the intersection of the languages of the component automata. We adapt this composition operation to automaton B_K and query automaton $B_{\neg\phi}[\texttt{var}]$, yielding composite query automaton $B_{K,\neg\phi}[\texttt{var}]$, as follows. States in $B_{\neg\phi}[\texttt{var}]$ are pairs of states from B_K and $B_{\neg\phi}[\texttt{var}]$. Tuple $((q_1, q_2), A \wedge \gamma[\texttt{var}], (q_1', q_2'))$ is a transition in $B_{K,\neg\phi}[\texttt{var}]$ iff (q_1, A, q_1') is a transition in B_K and $(q_2, \gamma[\texttt{var}], q_2')$ is a transition in $B_{\neg\phi}[\texttt{var}]$. It should be noted that the transition label in this case, $A \wedge \gamma[\texttt{var}]$, has a special property: for any `var`, either $[\![A \wedge \gamma[\texttt{var}]]\!] = \{A\}$, or $[\![A \wedge \gamma[\texttt{var}]]\!] = \emptyset$. This is a consequence of the fact that our treatment of A as a proposition means that $[\![A]\!] = \{A\}$. The initial state of $B_{K,\neg\phi}[\texttt{var}]$ is the pair consisting of the start states of B_K and $B_{\neg\phi}[\texttt{var}]$, respectively; states are accepting in $B_{K,\neg\phi}[\texttt{var}]$ if and only if the state component coming from $B_{\neg\phi}[\texttt{var}]$ is accepting.

6.4 Solve for Shattering Conditions of $B_{K,\neg\phi}[\texttt{var}]$

Given $B_{K,\neg\phi}[\gamma]$, we now must find a proposition γ such that $L(B_{K,\neg\phi}[\gamma]) = \emptyset$. One approach [7] is to enumerate all possible γ and compute whether or not $L(B_{K,\neg\phi}[\gamma]) = \emptyset$ for each such γ. Because of the number of possible γ, this approach is infeasible for all but trivial $\mathcal{A}$.

Our approach instead focuses on determining when sets of edges in $B_{K,\neg\phi}[\texttt{var}]$ can be shattered via a common proposition γ in such a way that $L(B_{K,\neg\phi)}[\gamma])$ is empty. Our procedure may be summarized as follows.

1. Pre-process $B_{K,\neg\phi}[\texttt{var}]$ to eliminate all strongly connected components that have no outgoing edges from the component and that do not contain any accepting states. Call the reduced query automaton $B'[\texttt{var}]$.
2. Identify all unique edge labels $S = \{\gamma_1[\texttt{var}], \ldots, \gamma_n[\texttt{var}]\}$ in $B'[\texttt{var}]$.
3. Process Γ appropriately to determine how $B'[\texttt{var}]$ can be shattered.

We now expand on the last step of the above procedure. In this work our interest is only for LTL queries $\phi[\texttt{var}]$ in which `var` appears only positively or only negatively; we do not consider queries in which `var` is mixed. Based on the construction of $B_{K,\neg\texttt{var}}[\texttt{var}]$ it follows that `var` is either positive in all of the

$\gamma_i[\mathtt{var}]$ or negative in all of the $\gamma_i[\mathtt{var}]$. In what follows we assume that var is positive; the negative case is dual.

The first step in processing the $\gamma_i[\mathtt{var}]$ (var is positive) is to determine if $\gamma_i[\mathtt{var}]$ is shatterable, and if so, to compute its weakest shattering condition γ_i'. Propositional queries $\gamma_i[\mathtt{var}]$ that are not shatterable are removed from future consideration, as they cannot contribute to shattering $B'[\mathtt{var}]$. In what follows we assume that each $\gamma_i[\mathtt{var}]$ is shatterable, with weakest shattering condition γ_i'.

The next step S is to search for subsets of S that, when all shattered, shatter $B'[\mathtt{var}]$. More specifically, suppose $S' \subseteq S$ and γ'' is such that γ'' shatters each $\gamma'[\mathtt{var}] \in S'$. In $B'[\gamma'']$ none of the edges labeled by elements of S' would be present; if enough edges are eliminated, $L(B[\gamma'']) = \emptyset$, and γ'' would shatter $B'[\mathtt{var}]$. In this case we say that S' *shatters* $B'[\mathtt{var}]$. This search procedure is facilitated by the following observations.

1. If S' shatters $B'[\mathtt{var}]$ and $S' \subseteq S''$, S'' also shatters $B'[\mathtt{var}]$.
2. If S' does not shatter $B'[\mathtt{var}]$ and $S'' \subseteq S'$, S'' does not shatter $B'[\mathtt{var}]$.

These observations can be exploited to develop a modified breadth-first search (BFS) strategy for finding all minimal subsets of S that shatter $B'[\mathtt{var}]$. The BFS algorithm maintains a work set, $W \subseteq 2^S$, of subsets of S that need processing. Initially, $W = \{\emptyset\}$. The algorithm then repeatedly does the following. It selects a minimum-sized $S' \in W$ and checks if S' shatters $B[\mathtt{var}]$. If it does, then it removes all supersets of S' from W and adds S' to the set of minimal shattering subsets of S. If it does not, then every superset of S' that contains one more element than S' is added to W. The procedure terminates when W is empty. Note that the approach does not add to W when S' is found to be a shattering set; the correctness of this approach is based on the first observation above.

The BFS algorithm in the worst-case can still require examination of all subsets of S, so we also consider a different algorithm whose goal is to compute a single minimal shattering subset of S. This approach, which we call GREEDY_SET_SEARCH (GSS), first locates a (not necessarily minimal) shattering set using a depth-first search strategy as follows. The procedure maintains a set $R \subseteq S$ that is initially $\emptyset$. It then repeatedly checks to see if R shatters $B'[\mathtt{var}]$; if so, it terminates, otherwise, it adds a new element from S into R. The observations above guarantee that the above procedure will terminate after at most $|S|$ iterations. The second stage of the procedure then locates a minimal subset of the shattering set R returned by the first stage as follows. Each edge (except the last one added) is removed from R, and the set without this edge is checked for shattering. If the newly modified set R', consisting of R with this single edge removed, shatters $B'[\mathtt{var}]$ then the edge is permanently removed from R; otherwise, the edge is left in R. When this procedure terminates the resulting value of R is guaranteed to be a minimal shattering subset of S.

6.5 Implementation and Evaluation

We have developed prototype implementations of the BFS and GSS algorithms. Kripke structures are read in as directed graph data containing node labels,

and LTL formulas are represented as simple strings. As stated previously, the LTL3BA routine was used to generate Büchi query automata from LTL queries.

For a proof-of-concept assessment of the techniques we use a modified version of NuSMV to extract the explicit Kripke structures from a sample `.smv` model files included in the NuSMV distribution. For each choice of model used, we considered property queries that were conceivably of interest based upon grounded properties known to be true of the systems already. These always took one of the following forms: $\mathbf{G}\,a$, $\mathbf{G}\,\mathbf{F}\,a$ or $\mathbf{G}\,(a \rightarrow \mathbf{F}\,b)$. The models we considered in our evaluation are the following.

- Counter[k] - An implementation of a k-bit counter.
- Semaphore[k] - An implementation of a semaphore access control scheme for k different processes.
- Production cell - A production cell control model, first presented as an SMV model by Winter [20]. The original intent of this model concerned safety and liveness specifications.

Figure 2 contains relevant data about sizes of these models, and about the size of the Büchi query automata formed when composing the models with the query automaton $B_{\neg\,\mathbf{G}\,\mathtt{var}}$. For our purposes, the following measures are relevant: (1) number of states, (2) number of transitions, (3) number of atomic propositions in the Kripke structure, (4) number of transition labels containing variable labels in the composite automaton, and (5) number of unique transition labels.

Dataset	# States	# Transitions	$\|\mathcal{A}\|$	# `var`-present edges	# distinct edge labels
counter[3]	17	26	3	9	8
counter[4]	33	50	4	17	16
counter[5]	65	98	5	33	32
counter[10]	2049	3074	10	1025	1024
semaphore[2]	25	98	9	33	12
semaphore[3]	65	314	13	105	32
semaphore[4]	161	917	17	305	80
semaphore[5]	385	2498	21	833	192
semaphore[6]	897	6530	25	2177	448
semaphore[7]	2049	16514	29	5505	1024
production-cell	163	245	76	82	81

Fig. 2. Statistics for Büchi product query automata when composed with $\mathbf{G}(\mathtt{var})$.

Figure 3 contains performance data for both BFS and GSS. Algorithms were implemented in Java, and experiments were conducted on a single machine with a 3.5 GHz processor containing 32 GB of memory. Individual experiments were allowed to run for up to 2 h before being stopped and considered timed out. BFS yielded minimal success, as most datasets timed out. The GSS approach to find a single minimal shattering set proved much more effective.

Dataset	Time (s)	# Queries
counter[3]	0.2	257
counter[4]	7.2	65537
counter[5]	timeout	2^{32} (*)
counter[10]	timeout	2^{1024} (*)
semaphore[2]	1.3	4097
semaphore[3]	timeout	2^{32} (*)
semaphore[4]	timeout	2^{80} (*)
semaphore[5]	timeout	2^{192} (*)
semaphore[6]	timeout	2^{448} (*)
semaphore[7]	timeout	2^{1024} (*)
production-cell	timeout	2^{81} (*)

(a)

Dataset	Time (s)	# Queries
counter[3]	0.2	17
counter[4]	0.2	33
counter[5]	0.6	65
counter[10]	22.4	2049
semaphore[2]	0.2	25
semaphore[3]	0.4	65
semaphore[4]	1.4	161
semaphore[5]	6.9	385
semaphore[6]	44.1	897
semaphore[7]	296.9	2049
production-cell	1.3	163

(b)

Fig. 3. Timing results for finding (a) all shattering sets via breadth first search, and (b) one minimal shattering set via GREEDY SET SEARCH. The number of total shattering queries that are made for each experiment are also reported. Query counts marked with a (*) are estimates based on our understanding of the models.

7 Conclusions and Directions for Future Research

In this paper we have considered the problem of query checking for Linear Temporal Logic (LTL). An LTL query checker takes a query, or LTL formula with a missing propositional subformula, together with a Kripke structure and computes a solution for the missing subformula. We have shown how this problem may be solved using automata-theoretic techniques that rely on the use of Büchi automata and the computation of so-called shattering conditions that make the languages of these automata empty. An implementation and preliminary performance data are also given.

As future work, we intend to fully develop the implementation and extend the experimental results we have so far. We also would like to extend the results to handle queries involving multiple missing subformulas, as well as ones in which the missing subformula can appear both positively and negatively. Finally, we would like to leverage relationships between different edge labels containing variables, such as in cases where one label implies another.

References

1. Ackermann, C., Cleaveland, R., Huang, S., Ray, A., Shelton, C., Latronico, E.: Automatic requirement extraction from test cases. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 1–15. Springer, Heidelberg (2010). doi:10.1007/978-3-642-16612-9_1
2. Armoni, R., et al.: The ForSpec temporal logic: a new temporal property-specification language. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 296–311. Springer, Heidelberg (2002). doi:10.1007/3-540-46002-0_21

3. Babiak, T., Křetínský, M., Řehák, V., Strejček, J.: LTL to Büchi automata translation: fast and more deterministic. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 95–109. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28756-5_8
4. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
5. Bruns, G., Godefroid, P.: Temporal logic query checking. In: 16th Annual IEEE Symposium on Logic in Computer Science, pp. 409–417. IEEE, June 2001
6. Chan, W.: Temporal-logic queries. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 450–463. Springer, Heidelberg (2000). doi:10.1007/10722167_34
7. Chockler, H., Gurfinkel, A., Strichman, O.: Variants of LTL Query Checking. In: Barner, S., Harris, I., Kroening, D., Raz, O. (eds.) HVC 2010. LNCS, vol. 6504, pp. 76–92. Springer, Heidelberg (2011). doi:10.1007/978-3-642-19583-9_11
8. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
9. Emerson, E.A.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 995–1072. MIT Press (1990)
10. Emerson, E.A., Halpern, J.Y.: "Sometimes" and "not never" revisited: On branching versus linear time temporal logic. JACM **33**(1), 151–178 (1986)
11. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007)
12. Etessami, K., Holzmann, G.J.: Optimizing Büchi automata. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 153–168. Springer, Heidelberg (2000). doi:10.1007/3-540-44618-4_13
13. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001). doi:10.1007/3-540-44585-4_6
14. Giannakopoulou, D., Lerda, F.: From states to transitions: improving translation of LTL formulae to Büchi automata. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 308–326. Springer, Heidelberg (2002). doi:10.1007/3-540-36135-9_20
15. Gurfinkel, A., Chechik, M., Devereux, B.: Temporal logic query checking: a tool for model exploration. IEEE Trans. Soft. Eng. **29**(10), 898–914 (2003)
16. Lemieux, C., Park, D., Beschastnikh, I.: General LTL specification mining. In: 30th IEEE/ACM International Conference on Automated Software Engineering, pp. 81–92. IEEE, Lincoln, November 2015
17. Li, W., Forin, A., Seshia, S.A.: Scalable specification mining for verification and diagnosis. In: 47th Design Automation Conference, pp. 755–760. ACM, Anaheim, June 2010
18. Shoham, S., Yahav, E., Fink, S.J., Pistoia, M.: Static specification mining using automata-based abstractions. IEEE Trans. Soft. Eng. **34**(5), 651–666 (2008)
19. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: First Symposium on Logic in Computer Science, pp. 322–331. IEEE Computer Society, Boston, June 1986
20. Winter, K.: Model checking for abstract state machines. J. Univ. Comput. Sci. **3**(5), 689–701 (1997)
21. Zhang, D., Cleaveland, R.: Efficient temporal-logic query checking for Presburger systems. In: 20th IEEE/ACM International Conference on Automated Software Engineering, pp. 24–33. ACM, Long Beach, November 2005