Model Exploration with Temporal Logic Query Checking

Arie Gurfinkel Benet Devereux Marsha Chechik
Department of Computer Science
University of Toronto
6 King's College Rd
Toronto, Canada M5S 3H5
{arie,benet,chechik}@cs.toronto.edu

ABSTRACT

A temporal logic query is a temporal logic formula with placeholders. Given a model, a solution to a query is a set of assignments of propositional formulas to placeholders, such that replacing the placeholders with any of these assignments results in a temporal logic formula that holds in the model. Query checking, first introduced by William Chan [2], is an automated technique for finding solutions to temporal logic queries. It allows discovery of the temporal properties of the system and as such may be a useful tool for model exploration and reverse engineering.

This paper describes an implementation of a temporal logic query checker. It then suggests some applications of this tool, ranging from invariant computation to test case generation, and illustrates them using a Cruise Control System.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods; Model checking*; D.2.1 [**Software Engineering**]: Requirements/Specifications—*Tools*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Temporal Logic*

General Terms

Documentation; Verification

Keywords

Query-checking; CTL; multi-valued model-checking

1. INTRODUCTION

Temporal logic model-checking [7] allows us to decide whether a property stated in a temporal logic such as CTL [6] holds in a state-based model. Typical temporal logic formulas are $AG(p \wedge q)$: "both p and q hold in every state of the system", or $AG(p \Rightarrow AXq)$: "every state in which p holds is always followed by a state in which q holds".

Model checking was originally proposed as a verification technique; however, it is also extremely valuable for model understanding [2]. We rarely start the study of a design with a complete specification available. Instead, we begin with some key properties, and attempt to use the model-checker to validate them. When the properties do not hold, and they seldom do, what is at fault: the properties or the design? Typically, both need to be modified: the design if a bug was found, and the properties if they were too strong or incorrectly expressed. Thus, this process is aimed not only at building the correct model of the system, but also at discovering which properties it should have.

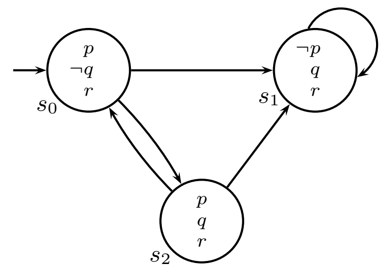

Figure 1: A simple state machine.

Query checking was proposed by Chan [2] to speed up design understanding by discovering properties not known *a priori*. A *temporal logic query* is an expression containing a symbol $?_x$, referred to as the *placeholder*, which may be replaced by any propositional formula[1] to yield a CTL formula, e.g. $AG?_x$, $AG(?_x \wedge p)$. The *solution* to a query is a set of strongest propositional formulas that make the query true. For example, consider evaluating the query $AG?_x$, i.e., "what are the invariants of the system", on a model in Figure 1. $(p \vee q) \wedge r$ is the strongest invariant: all others, e.g., $p \vee q$ or r, are implied by it. Thus, it is the solution to this query. In turn, if we are interested in finding the strongest property that holds in all states following those in which $\neg q$ holds, we form the query $AG(\neg q \Rightarrow AX?_x)$ which, for the model in Figure 1, evaluates to $q \wedge r$.

In solving queries, we usually want to restrict the atomic propositions that are present in the answer. For example, we may not care about the value of r in the invariant computed for the model in Figure 1. We phrase our question as $AG(?_x\{p, q\})$, thus explicitly restricting the propositions of interest to p and q. The answer we get is $p \vee q$. Given a fixed set of n atomic propositions of interest, the query checking problem defined above can be solved by taking all 2^{2^n} propositional formulas over this set, substituting them for the placeholder, verifying the resulting temporal logic formula, tab-

[1]A propositional formula is a formula built only from atomic propositions and boolean operators.

SIGSOFT 2002/FSE-10, November 18–22, 2002, Charleston, SC, USA.

ulating the results and then returning the strongest solution(s) [1]. The number n of propositions of interest provides a way to control the complexity of query checking in practice, both in terms of computation, and in terms of understanding the resulting answer.

In his paper [2], Chan proposed a number of applications for query checking, mostly aimed at giving more feedback to the user during model checking, by providing a partial explanation when the property holds and diagnostic information when it does not. For example, instead of checking the invariant $AG(a \vee b)$, we can evaluate the query $AG?_x\{a, b\}$. Suppose the answer is $a \wedge b$, that is, $AG(a \wedge b)$ holds in the model. We can therefore inform the user of a stronger property and explain that $a \vee b$ is invariant because $a \wedge b$ is. We can also use query checking to gather diagnostic information when a CTL formula does not hold. For example, if $AG(\text{req} \Rightarrow AF\,\text{ack})$ is false, that is, a request is not always followed by an acknowledgment, we can ask *what* can guarantee an acknowledgment: $AG(?_x \Rightarrow AF\,\text{ack})$.

In his work, Chan concentrated on *valid* queries, that is, queries that always yield a single strongest solution. All of the queries we mentioned so far are valid. Chan showed that in general it is expensive to determine whether a CTL query is valid. Instead, he identified a syntactic class of CTL queries such that every formula in the class is valid. He also implemented a query-checker for this class of queries on top of the symbolic CTL model-checker SMV.

Queries may also have multiple strongest solutions. Suppose we are interested in exploring successors of the initial state of the model in Figure 1. Forming a query $EX?_x$, i.e., "what holds in any of the next states, starting from the initial state s_0?", we get two incomparable solutions: $p \wedge q \wedge r$ and $\neg p \wedge q \wedge r$. Thus, we know that state s_0 has at least two successors, with different values of p in them. Furthermore, in all of the successors, $q \wedge r$ holds. Clearly, such queries might be useful for model exploration. Checking queries with multiple solutions can be done using the method of Bruns and Godefroid [1]. They extend Chan's work by showing that the query checking problem with a single placeholder can be solved using alternating automata [17]. In fact, the queries can be specified in temporal logics other than CTL. However, so far this solution remains purely theoretical: no implementation of such a query-checker is available.

The range of applications of query checking can be expanded further if we do not limit queries to just one placeholder. In particular, queries with two placeholders allow us to ask questions about pairs of states, e.g., dependencies between a current and a next state in the system.

This paper describes three major contributions:

1. We enrich the language of queries to include several placeholders. The previous methods only dealt with one placeholder, referring to it as "?". In our framework, placeholders need to be *named*, e.g. "$?_x$", "$?_y$", "$?_{pre}$".
2. We describe the temporal logic query checking tool which we built on top of our existing multi-valued model-checker χChek [4, 5]. The implementation not only allows one to compute solutions to the placeholders but also gives *witnesses* — paths through the model that explain why solutions are as computed.
3. We outline a few uses of the temporal logic query checking, both in domains not requiring witness computation and in those that depend on it.

The rest of this paper is organized as follows: in Section 2, we give the necessary background for this paper, briefly summarizing CTL model-checking, query checking, and multi-valued CTL model-checking. Section 3 defines the reduction of the query checking problem to multi-valued model-checking. Section 4 describes some possible uses of query checking for model exploration. We illustrate these on an example of the Cruise Control System [16]. This section can be read without the material in Sections 2 and 3. We conclude in Section 5 with the summary of the paper and the directions for future work. Proofs of theorems that appear in this paper can be found [11].

2. BACKGROUND

In this section, we briefly outline CTL model-checking, describe the query checking problem, and give an overview of multi-valued CTL model-checking.

2.1 CTL Model-Checking

CTL model-checking [6] is an automatic technique for verifying properties expressed in a propositional branching-time temporal logic called *Computation Tree Logic* (CTL). The system is represented by a Kripke structure, and properties are evaluated on a tree of infinite computations produced by unrolling it. A *Kripke structure* is a tuple (S, s_0, A, R, I) where:

- S is a finite set of states, and s_0 is the initial state;
- A is a set of propositional variables;
- $R \subseteq S \times S$ is the (total) transition relation;
- $I : S \to 2^A$ is a labeling function that maps each state onto the set of propositional variables which hold in it.

CTL is defined as follows:

1. Constants true and false are CTL formulas.
2. Every atomic proposition $a \in A$ is a CTL formula.
3. If φ and ψ are CTL formulas, then so are $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $EX\varphi$, $AX\varphi$, $EF\varphi$, $AF\varphi$, $E[\varphi U \psi]$, $A[\varphi U \psi]$, $EG\varphi$, $AG\varphi$.

The boolean operators $\neg$, $\wedge$ and $\vee$ have the usual meaning. The temporal operators have two components: A and E quantify over paths, while X, F, U and G indicate "next state", "eventually (*future*)", "until", and "always (*globally*)", respectively. Hence, $AX\varphi$ is true in state s if φ is true in the *next* state on *all paths* from s. $E[\varphi U \psi]$ is true in state s if *there exists* a path from s on which φ is true at every step *until* ψ becomes true.

The formal semantics of CTL is given in Figure 2. In this figure, we use a function $[\![\varphi]\!] : S \to \{\mathsf{true}, \mathsf{false}\}$ to indicate the result of checking a formula φ in state s. We further define the set of successors for a state s:

$$R(s) \equiv \{s' | (s, s') \in R\}$$

The more familiar notation for indicating that a property φ holds in a state s of a Kripke structure K ($K, s \models \varphi$) can be defined as follows:

$$\begin{array}{rcl} K, s \models \varphi & \equiv & [\![\varphi]\!](s) = \mathsf{true} \\ K, s \not\models \varphi & \equiv & [\![\varphi]\!](s) = \mathsf{false} \end{array}$$

We also say that a formula φ holds in a Kripke structure K if φ holds in K's initial state. In Figure 2, we used conjunction and disjunction in place of the more familiar universal and existential

$$
\begin{array}{rcl}
[\![\ell]\!](s) & \equiv & \ell, \text{for } \ell \in \{\mathsf{true}, \mathsf{false}\} \\
[\![a]\!](s) & \equiv & a \in I(s), \text{for } a \in A \\
[\![\neg\varphi]\!](s) & \equiv & \neg[\![\varphi]\!](s) \\
[\![\varphi \wedge \psi]\!](s) & \equiv & [\![\varphi]\!](s) \wedge [\![\psi]\!](s) \\
[\![\varphi \vee \psi]\!](s) & \equiv & [\![\varphi]\!](s) \vee [\![\psi]\!](s) \\
[\![EX\varphi]\!](s) & \equiv & \bigvee_{s' \in R(s)} [\![\varphi]\!](s') \\
[\![AX\varphi]\!](s) & \equiv & \bigwedge_{s' \in R(s)} [\![\varphi]\!](s')
\end{array}
$$

Figure 2: Formal semantics of CTL operators.

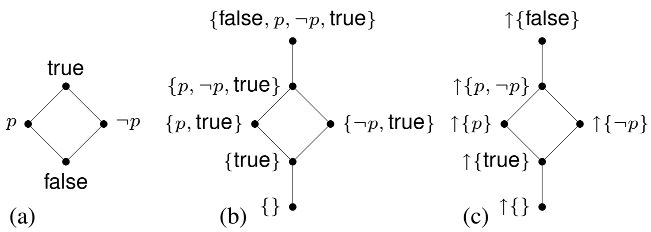

Figure 3: Lattices for set $P = \{p\}$ (from [1]): (a) $(PF(P), \Rightarrow)$; (b) $(U(PF(P), \Rightarrow), \subseteq)$; (c) same as (b), represented using minimal elements.

quantification. The semantics of EX and AX can be alternatively expressed as

$$
\begin{array}{rcl}
[\![EX\varphi]\!](s) & \equiv & \exists s' \in R(s) \cdot [\![\varphi]\!](s') \\
[\![AX\varphi]\!](s) & \equiv & \forall s' \in R(s) \cdot [\![\varphi]\!](s')
\end{array}
$$

Semantics of EG and EU is as follows:

$$
\begin{array}{rcl}
[\![EG\varphi]\!](s_i) & \equiv & \exists \text{ some path } s_i, s_{i+1}, \ldots \text{ s.t.} \\
& & \forall j \geq i \cdot [\![\varphi]\!](s_j) \\
[\![E[\varphi\ U\ \psi]]\!](s_i) & \equiv & \exists \text{ some path } s_i, s_{i+1}, \ldots, \text{ s.t.} \\
& & \exists j \geq i \cdot [\![\psi]\!](s_j) \wedge \\
& & \forall k \cdot i \leq k < j \Rightarrow [\![\varphi]\!](s_k)
\end{array}
$$

Finally, the remaining CTL operators are given below:

$$
\begin{array}{lcl}
A[\varphi\ U\ \psi] & \equiv & \neg E[\neg\psi\ U\ \neg\varphi \wedge \neg\psi]\ \wedge\ \neg EG\neg\psi \\
AX\varphi & \equiv & \neg EX\neg\varphi \\
AF\varphi & \equiv & A[\mathsf{true}\ U\ \varphi] \\
EF\varphi & \equiv & E[\mathsf{true}\ U\ \varphi] \\
AG\varphi & \equiv & \neg EF\neg\varphi
\end{array}
$$

For example, consider the model in Figure 1, where s_0 is the initial state and $A = \{p, q, r\}$. Properties $AG(p \vee q)$ and AFq are true in this model, whereas AXp is not.

2.2 Query Checking Fundamentals

This exposition follows the presentation in [1].

A *lattice* is a partial order $(\mathcal{L}, \sqsubseteq)$, where every finite subset $B \subseteq \mathcal{L}$ has a least upper bound (called "join" and written as $\sqcup B$) and a greatest lower bound (called "meet" and written $\sqcap B$). $\top$ and $\bot$ are the maximal and the minimal elements of a lattice, respectively. A lattice is *distributive* if join distributes over meet and vice versa.

Given a set of atomic propositions P, let $PF(P)$ be the set of propositional formulas over P. For example, $PF(\{p\}) = \{\mathsf{true}, \mathsf{false}, p, \neg p\}$. This set forms a distributive lattice under implication (see Figure 3(a)). Since $p \Rightarrow \mathsf{true}$, p is under true in this lattice. Meets and joins in this lattice correspond to classical $\wedge$ and $\vee$ operations, respectively.

A propositional formula ψ is a *solution* to a query φ in state s if substituting ψ for the placeholder in φ is a formula that holds in the Kripke structure K in state s. A query φ is *positive* [2] if when ψ_1 is a solution to φ and $\psi_1 \Rightarrow \psi_2$, then ψ_2 is also a solution. For example, if $p \wedge q$ is a solution to φ, then so is p. In other words, the set of all solutions to a positive query is a set of propositional formulas that is *upward closed* with respect to the implication ordering: if some propositional formula is a solution, so is every weaker formula. Alternatively, a query is positive if and only if all placeholders in it occur under an even number of negations [2]. Further, for positive queries it makes sense to look just for the strongest solutions because all other solutions can be inferred from them. These notions are formalized below.

Given the ordered set $(\mathcal{L}, \sqsubseteq)$ and a subset $B \subseteq \mathcal{L}$, we define

$$\uparrow B = \{\ell \in \mathcal{L} \mid \exists b \in B \cdot b \sqsubseteq \ell\}$$

For example, for the ordered set $(PF(\{p\}), \Rightarrow)$ shown in Figure 3(a), $\uparrow\{p, \neg p\} = \{p, \neg p, \mathsf{true}\}$. A subset B of $\mathcal{L}$ is an *upset* if $\uparrow B = B$. Thus, $\{p, \neg p\}$ is not an upset whereas $\{p, \neg p, \mathsf{true}\}$ is. We write $U(\mathcal{L}, \sqsubseteq)$ for the set of all upsets of $\mathcal{L}$. The distributive lattice formed by elements of $U(PF(\{p\}), \Rightarrow)$ ordered by set inclusion is shown in Figure 3(b). We refer to this as an *upset lattice*.

Finally, we note that each upset can be uniquely represented by the set of its *minimal elements*. For example, for $P = \{p\}$, $\{p, \neg p\}$ is sufficient to represent the set $\{p, \neg p, \mathsf{true}\}$, and $\{\mathsf{false}\}$ is sufficient to represent $\{p, \neg p, \mathsf{true}, \mathsf{false}\}$. Figure 3(c) shows the lattice $(U(PF(P), \Rightarrow), \subseteq)$ represented using minimal elements. In the remainder of this paper, when we say that X is a solution to a query, we mean that $X \subseteq PF(P)$, X is the set of minimal solutions to this query, and $\uparrow X$ is the set of all of its solutions.

2.3 Multi-Valued Model-Checking

Multi-Valued CTL model-checking [5] is a generalization of the model-checking problem. Let $\mathbb{B}$ refer to the classical algebra with values true and false ("classical logic"). Instead of using $\mathbb{B}$, multi-valued model-checking is defined over any *De Morgan* algebra $L = (\mathcal{L}, \sqsubseteq, \neg)$, where $(\mathcal{L}, \sqsubseteq)$ is a finite distributive lattice and $\neg$ is any operation that preserves involution ($\neg\neg\ell = \ell$) and De Morgan laws. Conjunction and disjunction are defined using meet and join operations of $(\mathcal{L}, \sqsubseteq)$, respectively. In this algebra, we get $\neg\top = \bot$ and $\neg\bot = \top$, but not necessarily the law of non-contradiction ($\ell \wedge \neg\ell = \bot$) or excluded middle ($\ell \vee \neg\ell = \top$).

Properties are specified in a multiple-valued extension of CTL called χCTL. χCTL has the same syntax as CTL, except that any $\ell \in \mathcal{L}$ is also a χCTL formula. However, its semantics is somewhat different. We modify the labeling function of the Kripke structure to be $I : S \times A \rightarrow \mathcal{L}$, so that for each atomic proposition $a \in A$, $I(s, a) = \ell$ means that the variable a has value ℓ in state s. Thus,

$$
\begin{array}{rcl}
[\![\ell]\!](s) & \equiv & \ell, \text{for } \ell \in \mathcal{L} \\
[\![a]\!](s) & \equiv & I(s, a), \text{for } a \in A
\end{array}
$$

The other operations are defined as their CTL counterparts (see Figure 2), where $\vee$ and $\wedge$ are interpreted as lattice $\sqcup$ and $\sqcap$, respectively. In fact, in the rest of this paper we often write "$\vee$" and "$\wedge$" in place of "$\sqcup$" and "$\sqcap$", even if the algebra we use is different from $\mathbb{B}$.

The complexity of model-checking a χCTL formula φ on a Kripke structure $K = (S, s_0, A, R, I)$ over an algebra $L = (\mathcal{L}, \sqsubseteq, \neg)$ is $O(|S| \times h \times |\varphi|)$, where h is the height of the lattice $(\mathcal{L}, \sqsubseteq)$, provided that meets, joins, and quantification operations take constant time [5].

We have implemented a symbolic model-checker χChek [4] that receives a Kripke structure K and a χCTL formula φ and returns an element of the algebra corresponding to the value of φ in K. The exact interpretation of this value depends on the domain. For example, if the algebra is $\mathbb{B}$, χChek returns true if φ holds in K and false if it does not: this is the classical model-checking. For more information about multi-valued model-checking, please consult [4, 5].

3. TEMPORAL LOGIC QUERY-CHECKER

In this section, we describe the computation of query-checking solutions in detail. We express the query-checking problem for one placeholder in terms of the multiple-valued model-checking framework described in Section 2. We then discuss how to deal with queries containing multiple placeholders, and finally what to do in the case of non-positive queries.

Recall that multi-valued model-checking is an extension of model-checking to an arbitrary De Morgan algebra. In our case, the algebra is given by the upset lattice of propositional formulas (see Figure 3). In order to reduce query-checking to multi-valued model-checking, we need to translate a given query into a χCTL formula such that the element of the upset lattice corresponding to the value of the χCTL formula is the set of all solutions to the query.

3.1 Intuition

Consider two simple examples of temporal logic queries, using the model in Figure 1. First, we ask $?_x$, meaning "what propositional formulas are true in a state". Solving this query with respect to s_0, we notice that the formula $p \wedge \neg q \wedge r$ holds in s_0, and all other formulas that hold in s_0 are implied by it. Thus, it is the strongest solution, and the set of all solutions is given by $\uparrow\{p \wedge \neg q \wedge r\}$.

Next, we look at $AX?_x$, which means "what *single* formula holds in all successor states". To solve this query with respect to the state s_0, we must first identify all successors of s_0, solve the query $?_x$ for each of them, and finally take the intersection of the results. The solution to $?_x$ in the two successors of s_0: s_1 and s_2, is $\uparrow\{\neg p \wedge q \wedge r\}$ and $\uparrow\{p \wedge q \wedge r\}$, respectively. The intersection of these solutions is $\uparrow\{q \wedge r\}$; thus, $q \wedge r$ holds in all successors of s_0, and any other solution to $AX?_x$ is implied by it. Notice that this computation is $\bigcap_{s' \in R(s)} [\![?_x]\!](s')$, and since intersection corresponds to meet in the upset lattice, this precisely matches the χCTL semantics of AX from Figure 2. Based on this observation, we show how query-checking is reduced to multi-valued model-checking.

3.2 Reduction to χCTL

The translation is defined top-down. All operators, constants, and propositional variables are translated one-to-one: $\varphi \vee \psi$ is mapped to the disjunction of the translation of φ with the translation of ψ; any variable p is mapped to itself; true is mapped to the constant symbol $\top$; and so forth. For the model in Figure 1,

$$\begin{array}{rcl} [\![p \wedge \neg q]\!](s_0) & = & [\![p]\!](s_0) \sqcap \neg [\![q]\!](s_0) \\ & = & \top \sqcap \neg \bot \\ & = & \top \end{array}$$

We now show how to translate the placeholder $?_x$. Consider the computation of $?_x\{p, q\}$ in state s_0 of the model in Figure 1. The solution $\uparrow\{p \wedge \neg q\}$ to the query is obtained by examining the values of p and q in s_0. We formalize this using the case-statement:

$$\begin{array}{ll} [\![?_x\{p, q\}]\!](s) & \equiv \\ & \textbf{case} \quad ([\![p \wedge \neg q]\!](s) = \top): \quad \uparrow\{p \wedge \neg q\} \\ & \textbf{case} \quad ([\![p \wedge q]\!](s) = \top): \quad \uparrow\{p \wedge q\} \\ & \quad \vdots \end{array}$$

and since all of the cases are disjoint, this yields the (syntactic) translation:

$$\begin{array}{rll} ?_x\{p, q\} \equiv & & (p \wedge \neg q \wedge \uparrow\{p \wedge \neg q\}) \\ & \vee & (p \wedge q \wedge \uparrow\{p \wedge q\}) \\ & \vee & (\neg p \wedge q \wedge \uparrow\{\neg p \wedge q\}) \\ & \vee & (\neg p \wedge \neg q \wedge \uparrow\{\neg p \wedge \neg q\}) \end{array}$$

So, to evaluate $?_x\{p, q\}$, we use the fact that $[\![p \wedge q]\!](s_0) = \top$ to get

$$\begin{array}{rcll} [\![?_x\{p, q\}]\!](s_0) & = & & ([\![p \wedge \neg q]\!](s_0) \sqcap \uparrow\{p \wedge \neg q\}) \\ & & \sqcup & ([\![p \wedge q]\!](s_0) \sqcap \uparrow\{p \wedge q\}) \\ & & \sqcup & \cdots \\ & = & & (\top \sqcap \uparrow\{p \wedge \neg q\}) \\ & & \sqcup & \bot \sqcup \cdots \\ & = & \uparrow\{p \wedge \neg q\} & \end{array}$$

To illustrate this idea further, consider a more complex query $?_x\{p, q\} \wedge EX?_x\{p, q\}$, evaluated in state s_0 of Figure 1. The set of all solutions to the subquery $EX?_x\{p, q\}$ is $\uparrow\{p \wedge q, \neg p \wedge q\}$, and the set of all solutions to $?_x$ is $\uparrow\{p \wedge \neg q\}$. To get the set of all solutions to our query, we intersect the results to get $\uparrow\{p, p \neq q\}$.

THEOREM 1. *Let T be the above translation from CTL queries into χCTL. Then, for any CTL query φ and state s in the model, $[\![T(\varphi)]\!](s)$ contains exactly the solutions to φ in state s.*

As stated in Section 2.3, multiple-valued model-checking has time complexity $O(|S| \times h \times |\varphi|)$, where h is the height of the lattice. Thus, to estimate the complexity of query-checking, we need to compute the height of the upset lattice used in the reduction of query-checking to multi-valued model-checking. If the placeholder is restricted to n atomic propositions $\{p_1, \ldots, p_n\}$, then, since there are 2^n propositional formulas in n variables, the height of the upset lattice $(U(PF(\{p_1, \ldots, p_n\}), \Rightarrow), \subseteq)$ is $2^{2^n} + 1$. The complexity of query-checking is $O(|S| \times 2^{2^n} \times |\varphi|)$

Recall that in traditional model-checking, the height of the model-checking lattice is 2, and the complexity is $O(|S| \times |\varphi|)$. Thus, solving a query is, in the worst case, 2^{2^n} times slower than checking an equivalent model-checking property. However, we find that in practice the running time of the query-checker is much better than the worst case (see Section 4.4).

3.3 More Complex Queries

More than one placeholder can be required to express some properties of interest. In this section, we give an extension of query-checking which allows for multiple placeholders, where each may depend on a different set of propositional variables. Furthermore, we describe how to solve non-positive queries.

3.3.1 Multiple Placeholders

If a query contains multiple placeholders, it is transformed into a CTL formula by substituting a propositional formula for each placeholder. Thus, given a query on n placeholders, with L_i being the lattice of propositional formulas for the ith placeholder, the set of all possible substitutions is given by the cross product $L = L_1 \times \cdots \times L_n$. We can lift the implication order, pointwise, to the elements of L, thus forming a lattice. For two placeholders, let $\varphi_1, \psi_1 \in L_1$ and $\varphi_2, \psi_2 \in L_2$. Then,

$$(\varphi_1, \varphi_2) \Rightarrow (\psi_1, \psi_2) \textbf{ iff } (\varphi_1 \Rightarrow \psi_1) \textbf{ and } (\varphi_2 \Rightarrow \psi_2)$$

Once again, the set of all solutions to a query is an element of the upset lattice over L.

We now show how to translate queries with multiple placeholders to χCTL. Consider the query $?_x \vee (EX?_x \wedge AX?_y)$. Each potential solution to this query is an element of $L = L_1 \times L_2$. To solve this query, we first find solutions to each subformula and then combine the results. Let $B \subseteq L_1$ be the set of all solutions to $?_x$ when viewed as a query with just one placeholder. However, since we have two placeholders, each solution, including the intermediate ones, must be a subset of L. The query $?_x$ does not depend on the placeholder $?_y$; therefore, any substitution for $?_y$ (i.e., any element of L_2) is acceptable. This results in $?_x = B \times L_2$. Similarly, the set of all solutions for $EX?_x$ is $C \times L_2$, and for $AX?_y$ — $L_1 \times D$, for some $C \subseteq L_1$ and $D \subseteq L_2$. Combining these results, we get

$$\begin{array}{ll} & (B \times L_2) \cup ((C \times L_2) \cap (L_1 \times D)) \\ = & (B \times L_2) \cup (C \times D) \end{array}$$

Thus, the set of solutions to this query is $\{(x, y) \mid x \in B \vee (x \in C \wedge y \in D)\}$.

For example, let us compute the solution to the query $?_x\{p, q\} \wedge EX?_y\{p, q\}$ in state s_0 of the model in Figure 1. We know from the example in Section 3.2 with just one placeholder,

$$\begin{array}{lll} [\![?_x]\!](s_0) & = & \uparrow\{p \wedge \neg q\} \\ [\![EX?_y]\!](s_0) & = & \uparrow\{p \wedge q, \neg p \wedge q\} \end{array}$$

Further, recall that $\uparrow\{\mathsf{false}\} = PF(\{p, q\})$. Each solution to $?_x\{p, q\} \wedge EX?_y\{p, q\}$ is an element of the lattice

$$L = PF(\{p, q\}) \times PF(\{p, q\})$$

In this lattice,

$$\begin{array}{lll} [\![?_x]\!](s_0) & = & \uparrow\{p \wedge \neg q\} \times \uparrow\{\mathsf{false}\} \\ [\![EX?_y]\!](s_0) & = & \uparrow\{\mathsf{false}\} \times \uparrow\{p \wedge q, \neg p \wedge q\} \end{array}$$

Putting these together, yields

$$\uparrow\{p \wedge \neg q\} \times \uparrow\{p \wedge q, \neg p \wedge q\}$$

Thus, this query has two minimal solutions: $(p \wedge \neg q, p \wedge q)$, and $(p \wedge \neg q, \neg p \wedge q)$.

3.3.2 Negation

Every query can be converted to its *negation-normal form* — the representation where negation is applied only to atomic propositions and placeholders. A query is positive if and only if all of its placeholders are non-negated when the query is put into its negation-normal form. Furthermore, we say that an occurrence of a placeholder in a query is *negative* if it appears negated in the negation-normal form of the query, and *positive* otherwise.

In this section, we describe how non-positive queries can be solved by transforming them into positive form, query-checking, and post-processing the solution. Note that the solution-set for negated placeholders depend on the *maximal* solutions[2], rather than the minimal ones. We consider two separate cases: (1) when all occurrences of a placeholder are either negative or positive, and (2) when a given placeholder appears in both negative and positive forms.

In case (1), the query is converted to the positive form by removing all of negations that appear in front of a placeholder, and then solved as described in the previous section. Finally, if the ith placeholder occurred in a negative position, the ith formula in the solution is negated to yield the correct result.

THEOREM 2. *If $(\varphi_1, \ldots, \varphi_n)$ is a solution to a query Q, and query Q' is identical to Q except that the ith placeholder appears negated, then $(\varphi_1, \ldots, \neg\varphi_i, \ldots, \varphi_n)$ is a solution to Q'.*

[2] An element φ of a solution-set $X \subseteq PF(P)$ is *maximal* if, for all $\psi \in X$, $\psi \Rightarrow \varphi$.

We sketch the proof by giving an example for a query with a single placeholder. Consider the query $AG\neg?_x$. We obtain a solution-set to $AG?_x$ and choose one formula φ from it. Since $AG\varphi$ holds in the model, so does $AG\neg(\neg\varphi)$; therefore, $\neg\varphi$ is in the solution-set for $AG\neg?_x$.

In case (2), if a placeholder $?_x$ appears both in the positive and the negative forms, we first replace each positive occurrence with $?_{x+}$ and each negative occurrence with $?_{x-}$, and then solve the resulting query. Finally, the set of all solutions to $?_x$ is given by the intersection of solutions to $?_{x+}$ and $?_{x-}$.

The complexity of using multi-valued model-checking for query-checking with multiple placeholders remains determined by the height of the lattice. We show the result for two placeholders: $?_x\{p_1, \ldots, p_n\}$ and $?_y\{q_1, \ldots, q_m\}$. There are 2^{2^n} possible solutions to $?_x$, and 2^{2^m} to $?_y$; therefore, there are $2^{2^n+2^m}$ possible simultaneous solutions. The height of the powerset lattice of solutions is $2^{2^n+2^m} + 1$, and so the complexity is $O(|S| \times 2^{2^n+2^m} \times |\varphi|)$. This result generalizes easily to any number of placeholders. As with the case of a single placeholder, we find that in practice query checking is more feasible than its worst case (see Section 4.4).

4. APPLICATIONS AND EXPERIENCE

In this section, we show two different techniques for model exploration using temporal logic queries. The technique presented in Section 4.2 uses only the solutions to the query-checking problem and is essentially an extension of the methodology proposed by Chan in [2]. The technique presented in Section 4.3 is completely new and is based on the fact that in addition to computing the solution to a query, our model-checker can also provide a witness explaining it. The examples in this section are based on our own experience in exploring an SCR specification of a Cruise Control System [16], described in Section 4.1. Please refer to Table 3 for the running time of various queries used in this section.

4.1 The Cruise Control System (CCS)

The Cruise Control System (CCS) is responsible for keeping an automobile traveling at a certain speed. The driver accelerates to the desired speed and then presses a button on the steering wheel (`Button` = `bCruise`) to activate the cruise control. The cruise control then maintains the car's speed, remaining active until one of the following events occurs: (1) the driver presses the brake pedal (`Brake`); (2) the driver presses the gas pedal (`Accel`); (3) the driver turns the cruise control off (`Button` = `bOff`); (4) the engine stops running (`Running`); (5) the driver turns the ignition off (`Ignition`); (6) the car's speed becomes uncontrollable (`Toofast`). If any of the first three events listed above occur, the driver can re-activate the cruise control system at the previously set speed by pressing a "resume" button (`Button` = `bResume`).

The SCR method [12] is used to specify event-driven systems. System outputs, called *controlled variables*, are computed in terms of inputs from the environment, called *monitored variables*, and the system state. To represent this state, SCR uses the notion of *modeclasses* — sets of states, called *modes*, that partition the monitored environment's state space. The system changes its state as the result of *events* — changes in the monitored variables. For example, an event @T(a) WHEN b, formalized as $\neg a \wedge b \wedge a'$, indicates that a becomes true in the next state while b is true in the current state. We prime variables to refer to their values in the next state.

We use the simplified version of CCS [3] which has 10 monitored variables and 4 controlled variables. One of these, `Throttle`, is described below. The system also has one modeclass `CC`, described in Table 1. Each row of the mode transition table specifies an event that activates a transition from the mode on the left to the mode on the right. The system starts in mode Off if Ignition is false, and transitions to mode Inactive when Ignition becomes true. Table 2 shows the event table for Throttle. Throttle assumes the value tAccel, indicating that the throttle is in the accelerating position, when (1) the speed becomes too slow while the system is in mode Cruise, as shown in the first row of Table 2; or (2) the system returns to the mode Cruise, indicated by @T(Inmode), and the speed has been determined to be too slow (see the second row of the table).

4.2 Applications of Queries without Witnesses

Below we show how temporal logic queries can replace several questions to a CTL model-checker to help express reachability properties and discover system invariants and transition guards.

Reachability analysis. A common task during model exploration is finding which states are reachable. For example, in CCS we may want to know whether all of the modes of the modeclass CC are reachable. This can be easily solved by checking a series of EF properties. For example, $EF(\texttt{CC} = \texttt{Cruise})$ holds if and only if the mode Cruise is reachable. However, queries provide a more concise representation: the solution to the single query $EF?_x\{\texttt{CC}\}$ corresponds to all of the reachable modes, i.e., those values p_i for which $EF(\texttt{CC} = p_i)$ is true. In our example, the solutions include all of the modes; thus, all modes are reachable. Similarly, finding all possible values of Throttle when the system is in mode Cruise is accomplished by the query $EF\big((\texttt{CC} = \texttt{Cruise}) \wedge ?_x\{\texttt{Throttle}\}\big)$. More complex analysis can be done by combining EF queries with other CTL operators. For an example, see the queries in rows 6 and 7 of Table 3.

Discovering invariants. Invariants concisely summarize complex relationships between different entities in the model, and are often useful in identifying errors. To discover all invariants, we simply need to solve the query $AG?_x$, with the placeholder restricted to all atomic propositions in the model. Unfortunately, in all but the most trivial models, the solution to this query is too big to be used effectively [2]. However, it is easy to restrict our attention to different parts of the model. For example, the set of invariants of the mode Inactive, with respect to the variables Ignition and Running, is the solution to the query

$$AG\big((\texttt{CC} = \texttt{Inactive}) \Rightarrow ?_x\{\texttt{Ignition}, \texttt{Running}\}\big)$$

which evaluates to $?_x = \texttt{Ignition}$. Furthermore, using multiple placeholders, we can find all invariants of each mode using a single query. For example, each solution to the query

$$AG(?_x\{\texttt{CC}\} \Rightarrow ?_y\{\texttt{Ignition}, \texttt{Running}\})$$

corresponds to invariants of each individual mode. In our example, the solution

$$?_x = (\texttt{CC} = \texttt{Cruise}), \quad ?_y = \texttt{Ignition} \wedge \texttt{Running}$$

indicates that Ignition and Running remain true while the system is the mode Cruise. Moreover, this query can also help the analyst determine which invariants are shared between modes. From the solution

$$?_x = \big(\texttt{CC} = (\texttt{Inactive} \vee \texttt{Cruise} \vee \texttt{Override})\big)$$
$$?_y = \texttt{Ignition}$$

we see that Ignition not only stays true throughout the mode Inactive, but it is also invariant in the modes Cruise and Override.

The mode invariants for CCS that we were able to discover using query-checking are equivalent to the invariants discovered by the algorithms in [14, 15]. Notice that the strength of the invariants obtained through query-checking depends on the variables to which the placeholder is restricted. The strongest invariant is obtained by restricting the placeholder to *all* of the monitored variables of the system.

Guard discovery. Finally, we illustrate how queries can be used to discover *guards* [18]. Suppose we are given a Kripke structure translation of an SCR model, i.e., events that enable transitions between modes are not explicitely represented. We can reverse-engineer the mode transition table by discovering guards in the Kripke structure.

Formally, a guard is defined as the weakest propositional formula over current (pre-) and next (post-) states such that the invariant $\alpha \wedge \gamma \Rightarrow \beta$ holds, where γ is the guard, and α and β are the pre- and post-conditions, respectively. Notice that since we define the guard to be the weakest solution, the guard does not directly correspond to an SCR event. Later we show that SCR events can be discovered by combining guards with mode invariants. Since guards are defined over pre- and post-states, two placeholders are required to express the query used to discover them, making the guard the weakest solution to the query

$$AG(\alpha \wedge ?_{pre} \Rightarrow AX(?_{post} \Rightarrow \beta))$$

We now show how this query is used to discover an event that causes CCS to switch from the mode Cruise to Inactive. In this case, we let $\alpha = (\texttt{CC} = \texttt{Cruise})$, and $\beta = (\texttt{CC} = \texttt{Inactive})$; furthermore, for practical reasons we restrict the $?_{pre}$ and $?_{post}$ placeholders to the set $\{\texttt{Toofast}, \texttt{Running}, \texttt{Brake}\}$. After solving this query, we obtain two solutions:

$$?_{pre} = \texttt{Brake} \vee \texttt{Toofast} \vee \neg\texttt{Running}, \quad ?_{post} = \mathsf{true}$$
$$?_{pre} = \mathsf{true}, \quad ?_{post} = \neg\texttt{Running} \vee \texttt{Toofast}$$

Before analyzing the result, we obtain the invariant for the mode Cruise:

$$\texttt{CC} = \texttt{Cruise} \Rightarrow \neg\texttt{Brake} \wedge \neg\texttt{Toofast} \wedge \texttt{Running}$$

using the invariant discovery technique presented in Section 4.2. We notice that the first solution violates the invariant, making the antecedent of the implication false; however, from the second solution, it follows that

$$AG\big((\texttt{CC} = \texttt{Cruise}) \Rightarrow AX((\neg\texttt{Running} \vee \texttt{Toofast}) \Rightarrow (\texttt{CC} = \texttt{Inactive}))\big)$$

holds, yielding the guard $\gamma = \neg\texttt{Running}' \vee \texttt{Toofast}'$. Finally, combining this with the invariant for the mode Cruise, we determine that the mode transition is guarded by two independent events, @F(Running) and @T(Toofast), just as indicated in the mode transition table.

4.3 Applications of Queries with Witnesses

Given an existential CTL formula that holds in the model, a model-checker can produce a trace through the model showing *why* the formula holds. This trace is called a *witness* to the formula. Similarly, given an existential query, the query-checker can produce a set of traces, which we also refer to as a *witness*, showing why each of the minimal solutions satisfies the query.

For example, consider the query $EX?_x\{p\}$ for the model in Figure 1. It has two minimal solutions: $?_x = p$ and $?_x = \neg p$; therefore, the witness consists of two traces, one for each solution, as shown in Figure 4. The trace s_0, s_2 corresponds to the solution p, and the trace s_0, s_1 — to the solution $\neg p$.

All of the traces comprising a witness to a query start from the

Old Mode	Event	New Mode
`Off`	@T(`Ignition`)	`Inactive`
`Inactive`	@F(`Ignition`)	`Off`
	@T(`Button` = `bCruise`) WHEN `Ignition` AND `Running` AND NOT (`Toofast`) AND NOT (`Brake`) AND NOT (`Accel`)	`Cruise`
`Cruise`	@F(`Ignition`)	`Off`
	@T(`Toofast`) OR @F(`Running`)	`Inactive`
	@T(`Brake`) OR @T(`Accel`) OR @T(`Button` = `bOff`)	`Override`
`Override`	@F(`Ignition`)	`Off`
	@F(`Running`) WHEN `Ignition`	`Inactive`
	@T(`Button` = `bResume`) WHEN `Ignition` AND `Running` AND NOT (`Toofast`) AND NOT (`Brake`) AND NOT (`Accel`)	`Cruise`
	@T(`Button` = `bCruise`) WHEN `Ignition` AND `Running` AND NOT (`Toofast`) AND NOT (`Brake`) AND NOT (`Accel`)	`Cruise`

Initial Mode: `Off` WHEN NOT `Ignition`

Table 1: Mode transition table for mode class `CC` of the cruise control system.

Modes	**Events**			
`Cruise`	@T(`Speed` = `slow`)	@T(`Speed` = `ok`)	@T(`Speed` = `fast`)	@F(`Inmode`)
`Cruise`	@T(`Inmode`) WHEN (`Speed` = `slow`)	@T(`Inmode`) WHEN (`Speed` = `ok`)	@T(`Inmode`) WHEN (`Speed` = `fast`)	@F(`Inmode`)
`Throttle`′ =	`tAccel`	`tMaintain`	`tDecel`	`tOff`

Initial: `Throttle` = `tOff`

Table 2: Event table for the controlled variable `Throttle`.

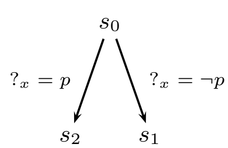

Figure 4: A witness for $EX?_x\{p\}$, in the model in Figure 1.

initial state, so they can be represented as a tree. In addition, our query-checker labels each branch in the tree with the set of solutions that are illustrated by that branch. In the example in Figure 4, the left branch is labeled with $?_x = p$ and the right — with $?_x = \neg p$. The benefit of treating a witness as a tree rather than a set of independent traces is that it becomes possible to *prefer* certain witnesses over others. For example, we may prefer a witness with the longest common prefix, which usually results in minimizing the total number of traces comprising the witness. We now show how witnesses can be used in several software engineering activities.

Guided simulation. The easiest way to explore a model is to simulate its behavior by providing inputs and observing the system behavior through outputs. However, it is almost impossible to use simulation to guide the exploration towards a given objective. Any wrong choice in the inputs in the beginning of the simulation can result in the system evolving into an "uninteresting" behavior. For example, let our objective be the exploration of how CCS evolves into its different modes. In this case, we have to *guess* which set of inputs results in the system evolving into the mode `Cruise`, and then which set of inputs yields transition into the mode `Inactive`, etc. Thus, the process of exploring the system using a simulation is usually slow and error prone.

An interesting alternative to a simple simulation is *guided simulation*. In a guided simulation setting, the user provides a set of objectives, and then only needs to choose between the different paths through the system in cases where the objective cannot be met by a single path. Moreover, each choice is given together with the set of objectives it satisfies.

Query-checking is a natural framework for implementing guided simulations. The objective is given by a query, and the witness serves as the basis for the simulation. For example, suppose we want to devise a set of simulations to illustrate how CCS evolves into all of its modes. We formalize our objective by the query $EF?_x\{\texttt{CC}\}$ and explore the witness. Moreover, we indicate that we prefer a witnesses with the largest common prefix, which results in a single trace through the system going through modes `Off`, `Inactive`, `Cruise`, and finally `Override`. This trace corresponds to a simulation given by the sequence of events: @T(`Ignition`), @T(`Running`), @T(`Button` = `bCruise`), and @T(`Button` = `bOff`). Since our objective was achieved by a single trace, the simulation was generated completely automatically, requiring no user input.

Test case generation. Although the primary goal of model-checking is to verify a model against temporal properties, it has recently been used to generate test cases [10, 9, 13, 18]. Most of the proposed techniques are based on the fact that in addition to computing expected outputs, a model-checker can produce witnesses (or counter-examples) which can be used to construct test sequences. The properties that are used to force the model-checker to generate desired test sequences are called *trap properties* [10].

Gargantini and Heitmeyer [10] proposed a method that uses an SCR specification of a system to identify trap properties satisfying a form of branch coverage testing criterion. Their technique uses both mode transition and condition tables to generate test se-

	Query	Time	Explanation
1	$EF?_x\{\texttt{CC}\}$	$0.596s$	what are all reachable modes
2	$EF(\texttt{CC} = \texttt{Cruise} \wedge ?_x\{\texttt{Throttle}\})$	$0.673s$	what values of `Throttle` are reachable in mode `Cruise`
3	$AGEF?_x\{\texttt{CC}\}$	$0.787s$	what modes are globally reachable
4	$EFEG?_x\{\texttt{CC}\}$	$0.720s$	what modes have self-loops
5	$AG(\texttt{CC} = \texttt{Inactive} \Rightarrow ?_x\{\texttt{Ignition}, \texttt{Running}\})$	$0.267s$	what are the invariants, over `Ignition` and `Running`, of mode `Inactive`
6	$AG(?_x\{\texttt{CC}\} \Rightarrow ?_x\{\texttt{Ignition}, \texttt{Running}\})$	$0.942s$	find all mode invariants, restricted to `Ignition` and `Running`
7	$EF(\texttt{CC} = \texttt{Off} \wedge EX?_{new}\{\texttt{CC}\})$	$0.187s$	what modes can follow `Off`
8	$EF(?_{old}\{\texttt{CC}\} \wedge EX?_{new}\{\texttt{CC}\})$	$1.204s$	what pairs of modes can follow each other
9	$EF\big((\texttt{CC} = \texttt{Cruise}) \wedge ?_x\{\texttt{Toofast}, \texttt{Running}\} \wedge EX(?_y\{\texttt{Toofast}, \texttt{Running}\} \wedge \texttt{CC} = \texttt{Inactive})\big)$	$0.335s$	how do values of `Toofast` and `Inactive` change as the system goes between modes `Cruise` and `Inactive`

Table 3: Summary of queries used in Section 4.

	Query	Time
1	$EF(\texttt{CC} = \texttt{Override})$	$0.265s$
2	$EF?_x\{\texttt{CC}\}$	$0.596s$
3	$EF(\texttt{CC} = \texttt{Cruise} \wedge \texttt{Throttle} = \texttt{tOff})$	$0.257s$
4	$EF(\texttt{CC} = \texttt{Cruise} \wedge ?_x\{\texttt{Throttle}\})$	$0.673s$
5	$EF(\texttt{CC} = \texttt{Cruise} \wedge ?_x\{\texttt{Throttle}, \texttt{Running}, \texttt{Toofast}\})$	$1.648s$

Table 4: Comparison between model-checking and query-checking.

quences. Here, we illustrate how our technique is applicable on mode transition tables; other tables can be analyzed similarly.

The method in [10] assures a form of branch coverage by satisfying the following two rules: (1) for each mode in the mode transition table, test each event at least once; (2) for each mode, test every case when the mode does not change (*no-change*) at least once. For example, the two test sequences need to be generated for mode `Off`, one testing the event @T(`Ignition`), and the other testing the no-change case. These can be obtained using the following trap properties:

$$EF\big((\texttt{CC} = \texttt{Off}) \wedge EX(\texttt{CC} = \texttt{Inactive})\big)$$
$$EF\big((\texttt{CC} = \texttt{Off}) \wedge EX(\texttt{CC} = \texttt{Off})\big)$$

Alternatively, the two test sequences can be obtained from a witness to a single query $EF\big((\texttt{CC} = \texttt{Off}) \wedge EX?_{new}\{\texttt{CC}\}\big)$. Similarly, the set of test sequences that cover the full mode transition table is obtained from the witness of the query $EF(?_{old}\{\texttt{CC}\} \wedge EX?_{new}\{\texttt{CC}\})$.

Since all of the traces comprising a witness to a query are generated at the same time, it is possible to minimize the number of different test sequences that guarantee the full coverage of the mode transition table. Moreover, whenever an EF query has more then one minimal solution, the query-checker can produce each minimal solution, and, if necessary, a witness for it, as soon as the new solution is found. Therefore, even in the cases when the complexity of the model-checking precludes obtaining the results for all of the trap properties, the query-checker can produce a solution to some of the trap properties as soon as possible.

Although the method suggested above generates a set of test sequences that cover every change (and every no-change) in the mode of the system, it does not necessarily cover all of the events. For example, the change from the mode `Cruise` to the mode `Inactive` is guarded by two independent events, @T(`Toofast`) and @F(`Running`); however, the witness for our trap query contains only a single trace corresponding to this change, covering just one of the events. We can first identify the events not covered by the test sequences from the witness to the query, and then use the method from [10] to generate additional test sequences for the events not yet covered.

Alternatively, if we know the variables comprising the event for a given mode transition, we can remedy the above problem by using an additional query. In our current example, the events causing the change from the mode `Cruise` to the mode `Inactive` depend on variables `Toofast` and `Running`. To cover these events, we form the query

$$EF\big((\texttt{CC} = \texttt{Cruise}) \wedge ?_x\{\texttt{Toofast}, \texttt{Running}\} \wedge EX(?_y\{\texttt{Toofast}, \texttt{Running}\} \wedge (\texttt{CC} = \texttt{Inactive}))\big)$$

The witness to this query corresponds to two test sequences: one testing the change on the event @T(`Toofast`) and the other — on the event @F(`Running`).

4.4 Running Time

Theoretical complexity of query-checking in Section 3.2 seems to indicate that query-checking is not feasible for all but very small models. However, our experience (see running times of queries used in this section in Table 3) seems to indicate otherwise. We address this issue in more detail below.

Theoretically, solving a query with a single placeholder restricted to two atomic propositions is slower than model-checking an equivalent CTL formula by a factor of $16 = 2^{2^2}$. To analyze the difference between the theoretical prediction and the actual running times, we verified several CTL formulas and related queries and summarized the results in Table 4. CTL formulas are checked using χChek, parametrized for $\mathbb{B}$. The query in the second row is restricted to two atomic propositions required to encode the enumerated type for `CC`. However, the running time of this query is only double that of the corresponding CTL formula (row 1). A similar picture can be seen by comparing the CTL formula in row 3 with the query in row 4 of the table. Finally, increasing the number of variables that a placeholder depends on, should slow down the analysis significantly. Yet, comparing queries in rows 4 and 5 of the table, we see that the observed slowdown is only three-fold.

Although we have not conducted a comprehensive set of experiments to evaluate the running time of our query-checker, we believe that our preliminary findings indicate that query-checking is in fact feasible in practice.

5. CONCLUSION

In this section, we summarize the paper and suggest venues for future work.

5.1 Summary and Discussion

In this paper, we have extended the temporal logic query-checking of Chan [2] and Bruns and Godefroid [1] to allow for queries with multiple placeholders, and shown the applicability of this extension on a concrete example. We have implemented a query-checker for multiple placeholders using the multi-valued model-checker χChek. Our implementation allows us not only to generate solutions to temporal logic queries, but also to provide witnesses explaining the answers. Further, our preliminary results show that it is feasible to analyze non-trivial systems using query-checking. Please send e-mail to `xchek@cs.toronto.edu` for a copy of the tool.

Building a query-checker on top of our model-checker has two further advantages. First, we allow query-checking over systems that have fairness assumptions. For example, we can compute invariants of CCS under the assumption that `Brake` is pressed infinitely often. As far as we know, Chan's system does not implement fairness. Further, the presentation in this paper used CTL as our temporal logic. However, since the underlying framework of χChek is based on μ-calculus, we can easily extend our query-checker to handle μ-calculus queries.

We are also convinced that temporal logic query-checking has many applications in addition to the ones we explored here. In particular, we see immediate applications in a variety of test case generation domains and hope that practical query-checking can have the same impact as model-checking for model exploration and analysis.

Finally, note that query-checking is a special case of multi-valued model-checking. Multi-valued model-checking was originally designed for reasoning about models containing inconsistencies and disagreements [8]. Thus, the reasoning was done over algebras derived from the classical logic, where the $\sqsubseteq$ relation in $L = (\mathcal{L}, \sqsubseteq, \neg)$ indicates "more true than or equal to". Query-checking is done over lattices, and algebras over them, that have a different interpretation — sets of propositional formulas. We believe that there might be yet other useful interpretations of algebras, making χChek the ideal tool for reasoning over them.

5.2 Future Work

In this paper, we have only considered queries where the placeholders are restricted to sets of atomic propositions. However, through our experience we found that it is useful to place further restrictions on the placeholders. For example, we may want to restrict the solutions to the query $EF?_x\{p, q, r\}$ only to those cases in which p and q are not true simultaneously. From the computational point of view, our framework supports it; however, expressing such queries requires an extension to the query language and some methodology to guide its use. We are currently exploring a query language inspired by SQL, in which the above query would be expressed as follows:

$$EF?_x \quad \textbf{where } ?_x \textbf{ in } PF(\{p,q,r\}) \textbf{ and not } (?_x \Rightarrow p \wedge q)$$

In the future, we plan to conduct further case studies to better assess the feasibility of query-checking on realistic systems. We also believe that the existence of an effective methodology is crucial to the success of query-checking in practice. We will use our case studies to guide us in the development of such a methodology.

6. ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support provided by NSERC and CITO. We also thank members of the UofT Formal Methods reading group for their suggestions for improving the presentation of this work.

7. REFERENCES

[1] G. Bruns and P. Godefroid. "Temporal Logic Query-Checking". In *Proceedings of 16th Annual IEEE Symposium on Logic in Computer Science (LICS'01)*, pages 409–417, Boston, MA, USA, June 2001. IEEE Computer Society.

[2] W. Chan. "Temporal-Logic Queries". In E. Emerson and A. Sistla, editors, *Proceedings of the 12th Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 450–463, Chicago, IL, USA, July 2000. Springer.

[3] M. Chechik. "SC(R)^3: Towards Usability of Formal Methods". In *Proceedings of CASCON'98*, pages 177–191, November 1998.

[4] M. Chechik, B. Devereux, and A. Gurfinkel. "χChek: A Multi-Valued Model-Checker". In *Proceedings of 14th International Conference on Computer-Aided Verification (CAV'02)*, Lecture Notes in Computer Science, pages 505–509, Copenhagen, Denmark, July 2002. Springer.

[5] M. Chechik, S. Easterbrook, and V. Petrovykh. "Model-Checking Over Multi-Valued Logics". In *Proceedings of Formal Methods Europe (FME'01)*, volume 2021 of *Lecture Notes in Computer Science*, pages 72–98. Springer, March 2001.

[6] 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, April 1986.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[8] S. Easterbrook and M. Chechik. "A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints". In *Proceedings of International Conference on Software Engineering (ICSE'01)*, pages 411–420, Toronto, Canada, May 2001. IEEE Computer Society Press.

[9] A. Engels, L. Feijs, and S. Mauw. "Test Generation for Intelligent Networks Using Model Checking". In E. Brinksma, editor, *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 384–398. Springer, 1997.

[10] A. Gargantini and C. Heitmeyer. "Using Model Checking to Generate Tests from Requirements Specifications". In *Proceedings of Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE99)*, pages 146–162, Toulouse, France, September 1999. ACM Press.

[11] A. Gurfinkel, B. Devereux, and M. Chechik. "Temporal Logic Query Checking through Multi-Valued Model Checking". CSRG Technical Report 457, University of Toronto, Department of Computer Science, March 2002.

[12] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. "Automated Consistency Checking of Requirements Specifications". *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[13] H. Hong, I. Lee, O. Sokolsky, and S. Cha. “ Automatic Test Generation from Statecharts Using Model Checking”. In *Proceedings of FATES’01, Workshop on Formal Approaches to Testing of Software*, volume NS-01-4 of *BRICS Notes Series*, pages 15–30, August 2001.

[14] R. Jeffords and C. Heitmeyer. “Automatic Generation of State Invariants from Requirements Specifications”. In *Proceedings of 6th International Symposium on Foundations of Software Engineering (FSE’98)*, pages 56–69, Orlando, FL, November 1998. ACM.

[15] R. Jeffords and C. Heitmeyer. “An Algorithm for Strengthening State Invariants Generated from Requirements Specifications”. In *Proceedings of 5th IEEE International Symposium on Requirements Engineering (RE’01)*, pages 182–191, Toronto, Canada, August 2001. IEEE Computer Society.

[16] J. Kirby. “Example NRL/SCR Software Requirements for an Automobile Cruise Control and Monitoring System”. Technical report, Wang Institute of Graduate Studies, October 1988. Revision 2.0 by Skip Osborne and Aaron Temin.

[17] O. Kupferman, M. Vardi, and P. Wolper. “An Automata-Theoretic Approach to Branching-Time Model Checking”. *Journal of the ACM*, 27(2):312–360, March 2000.

[18] S. Rayadurgam and M. P. Heimdahl. “Coverage Based Test-Case Generation using Model Checkers”. In *Proceedings of 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS’01)*, pages 83–93, Washington, DC, USA, April 2001. IEEE Computer Society.