Temporal Logic Query Checking: A Tool for Model Exploration

Arie Gurfinkel, Marsha Chechik, Benet Devereux
Department of Computer Science, University of Toronto,
Toronto, ON M5S 3G4, Canada.

***Abstract*— Temporal logic query checking was first introduced by William Chan to speed up design understanding by discovering properties not known *a priori*. A query is a temporal logic formula containing a special symbol $?_1$, known as a placeholder. Given a Kripke structure and a propositional formula φ, we say that φ satisfies the query if replacing the placeholder by φ results in a temporal logic formula satisfied by the Kripke structure. A solution to a temporal logic query on a Kripke structure is the set of all propositional formulas that satisfy the query.**

Query checking helps discover temporal properties of a system, and as such is a useful tool for model exploration. In this paper, we show that query checking is applicable to a variety of model exploration tasks, ranging from invariant computation to test case generation. We illustrate these using a Cruise Control System.

Additionally, we show that query checking is an instance of a multi-valued model checking of Chechik et al. This approach enables us to build an implementation of a temporal logic query checker, TLQSolver, on top of our existing multi-valued model checker χChek. It also allows us to decide a large class of queries and introduce witnesses for temporal logic queries – an essential notion for effective model exploration.

***Index Terms*— CTL, query checking, model checking, TLQSolver, model understanding, multi-valued logic.**

I. Introduction

Temporal logic model checking [1] allows us to decide whether a property stated in a temporal logic such as CTL [2] holds in a 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, complete specifications are typically not available at the start of a design analysis phase. 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, the verification process is aimed not only at building the correct model of the system, but also at discovering which properties it should have. It is therefore closely linked to the process of *model exploration* or *model understanding* [3] – understanding the behaviour of the model as well as well as the properties it preserves and the reasons for this preservation.

Temporal logic query checking was first introduced by William Chan [3] to speed up design understanding by discovering properties not known *a priori*. A query is a temporal logic formula containing a special symbol $?_1$, known as a *placeholder*, e.g., $AG(?_1)$, $AG(?_1 \wedge p)$. Given a Kripke structure and a propositional formula ψ, we say that ψ satisfies the query if replacing the placeholder by ψ results in a temporal logic formula satisfied by the Kripke structure. For example, r, $p \vee q$ and $(p \vee q) \wedge r$ satisfy the query $AG(?_1)$ evaluated on the model given in Figure 1. A solution to a temporal logic query on a Kripke structure is the set of all propositional formulas that satisfy the query.

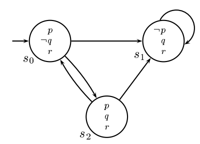

Fig. 1. A simple state machine.

In query checking, we often 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 $AG(?_1\{p, q\})$, thus explicitly restricting the propositions of interest to p and q. The solution to this query only includes expressions on p and q. Given a fixed set of k atomic propositions of interest, the query checking problem defined above can be solved by taking all 2^{2^k} propositional formulas over this set, substituting them for the placeholder, verifying the resulting temporal logic formula, tabulating the results and then returning the set of maximally strong solutions [3], [4]. In this paper, we will refer to this approach as the *naive* or *direct* solution. The number k of propositions of interest provides a way to control the complexity of query checking in practice, both in terms of understanding the result and in terms of the computation.

In his paper [3], 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 $AG?_1\{a, b\}$. 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 $AG(\mathrm{req} \Rightarrow AF\,\mathrm{ack})$ is false, that is, a request is not always followed by an acknowledgment, we can ask *what* does follow a request: $AG(\text{req} \Rightarrow AF\,?_1)$

Since the complexity of the general query checking problem looked infeasible, Chan [3] concentrated on *valid* queries, that is, queries where all solutions are derivable from a single solution. The complexity of query checking valid queries on k atomic propositions is exponential in k (as opposed to double exponential in the general case). Chan showed that in general, it is expensive to determine whether a given CTL query is valid. Instead, he identified a set of syntactic restrictions that guaranteed that the resulting CTL query is valid. He also implemented a query checker for valid queries with a single placeholder, on top of the symbolic model checker SMV [5].

On the other hand, many queries are not valid. Suppose we are interested in exploring successors of the initial state of the model in Figure 1. For example, a query $EX?_1$, i.e., "what holds in any of the next states, starting from the initial state s_0?", results in two incomparable sets of solutions (e.g., with elements $p \wedge q \wedge r$ 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. Bruns and Godefroid [4] provide a mechanism for computing all solutions to arbitrary queries with a single placeholder, occurring either in a negative or a positive position in the query, using extended alternating automata (EAA) [6]. Some examples of such queries are: $EX?_1$ (positive position), $EF(EX(\neg?_1) \vee p)$ (negative position). Hornus and Schnoebelen [7] study the problem of efficiently producing some of the maximally strong solutions for positive queries with a single placeholder. Their algorithm computes one solution using a linear number of calls to the model checker, two solutions using a quadratic number of calls to the model checker, etc. In both of the above methods, queries can be specified in temporal logics other than CTL, but, to our knowledge, neither of the methods has been implemented.

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 the current and a next state in the system. For example, suppose we are interested in exploring the transition relation of the model in Figure 1. We can achieve that by forming a query $EF(?_1 \wedge EX?_2)$, i.e., "compute pairs of reachable states". We may also want to produce a complete set of test cases to cover the transition relation of this model. To do so, solving the query is insufficient: we also need paths to each state in the model. The problem could be addressed if the query checker were to return *witnesses* – paths through the model that explain why the solution is as computed.

In this paper, we show that query checking is applicable to a variety of model exploration tasks, ranging from invariant computation to test case generation. Some of these tasks do not require witness computation, whereas others depend on it. We illustrate these using a specification of a Cruise Control System.

Additionally, we show that query checking is an instance of multi-valued model checking [8] – a generalization of a classical model-checking problem from a classical logic to arbitrary *De Morgan* algebras $(\mathcal{L}, \sqsubseteq, \neg)$, 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. This approach enables us to build an implementation of a temporal logic query checking tool, TLQSolver, on top of our existing symbolic multi-valued model checker χChek [9]. This implementation also computes witnesses for temporal logic queries.

The encoding proposed in this paper also gives us a formal framework for extending the query language to include multiple dependent and independent placeholders, occurring in both positive and negative positions in the query. Some examples of queries we can decide are: $AG(?_1 \wedge AX?_2)$ and $AG(?_1 \wedge AX(p \vee \neg?_1))$. In the first query, placeholders are independent. In the second, they are dependent; further, one instance is in a positive position, and the other is in a negative position. It also allows us to study the complexity of query checking from the symbolic representation point of view.

Note that our approach is similar to the one proposed by Bruns and Godefroid [4], where the query checking problem is decided using extended alternating automata. Yet in this work we raise the level of abstraction – our reduction works independently of the implementation of the multi-valued model checking machinery (i.e., using EAA or decision diagrams), and can benefit from both approaches to the implementation.

The rest of this paper is organized as follows: in Section II, we give the necessary background for this paper and fix the notation. Section III defines the language of queries. Section IV describes the implementation of our query checker TLQSolver. In Section V, we use TLQSolver in a variety of model exploration tasks. We describe the use of multi-valued model checking for solving queries with single and multiple placeholders, respectively, in Sections VI and VII, and address complexity issues in Section VIII. We conclude in Section IX with the summary of the paper and directions for future work.

II. Background

In this section we provide background information on lattice theory, fix the notation, briefly outline CTL model checking, and describe multi-valued CTL model checking.

A. Fundamentals

Here we give the basics of lattice theory, review the concept of join-irreducibility and define upward- and downward-closed sets.

Definition 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 maximum and the minimum elements of the lattice, respectively.

In this paper, if $(\mathcal{L}, \sqsubseteq)$ is a lattice, and the ordering operation $\sqsubseteq$ is clear from the context, we refer to it as $\mathcal{L}$. Furthermore, only *finite* lattices are considered.

Definition 2: A lattice is *distributive* if for all lattice elements a, b, c,

$$\begin{aligned} a \sqcup (b \sqcap c) &= (a \sqcup b) \sqcap (a \sqcup c) \\ a \sqcap (b \sqcup c) &= (a \sqcap b) \sqcup (a \sqcap c) \end{aligned}$$

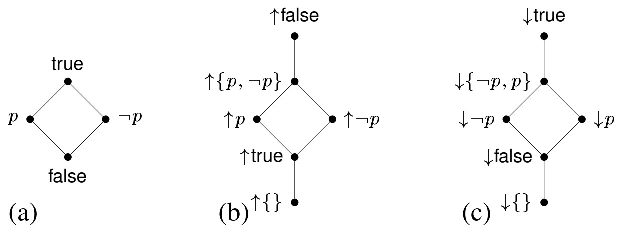

Fig. 2. Lattices for set $P = \{p\}$ (from [4]): (a) $(PF(P), \Rightarrow)$; (b) $(U(PF(P), \Rightarrow), \subseteq)$; (c) $(D(PF(P), \Rightarrow), \subseteq)$.

Definition 3: 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 finite distributive lattice under implication (see Figure 2(a)). Since $p \Rightarrow \mathsf{true}$, p is under true in this lattice. Meets and joins in this lattice correspond to logical $\wedge$ and $\vee$ operations, respectively.

Definition 4: Given the ordered set $(\mathcal{L}, \sqsubseteq)$ and a subset $B \subseteq \mathcal{L}$, the *upward closure* of B is

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

For example, for the ordered set $(PF(\{p\}), \Rightarrow)$ shown in Figure 2(a), $\uparrow\{p, \neg p\} = \{p, \neg p, \mathsf{true}\}$.

Definition 5: A subset B of $\mathcal{L}$ is *upward closed*, (or an *upset*) if $\uparrow B = B$.

Thus, $\{p, \neg p\}$ is not an upset whereas $\{p, \neg p, \mathsf{true}\}$ is. We write $U(\mathcal{L})$ for the set of all upsets of $\mathcal{L}$, i.e. if $A \subseteq \mathcal{L}$ then $\uparrow A \in U(\mathcal{L})$. Note that $U(\mathcal{L})$ is closed under union and intersection, and therefore forms a lattice ordered by set inclusion. We call the lattice $(U(\mathcal{L}), \subseteq)$ an *upset* lattice of $\mathcal{L}$. The upset lattice of $PF(\{p\})$ is shown in Figure 2(b), where every element is encoded as an upset. For singleton sets, we often write $\uparrow a$ instead of $\uparrow\{a\}$. Let a, b be elements of a lattice $\mathcal{L}$, then

$$\begin{array}{rcll} \uparrow a \sqcap \uparrow b & = & \uparrow(a \sqcup b) & \text{(distribution of meet)} \\ \uparrow a \sqsupseteq \uparrow b & = & a \sqsubseteq b & \text{(anti-monotonicity of } \uparrow) \end{array}$$

Proposition 1: If a lattice $\mathcal{L}$ is distributive, then so is $U(\mathcal{L})$.
Note that $U(PF(P))$ is distributive for any set P of atomic propositions.

Definition 6: A subset B of $\mathcal{L}$ is a *downset* if

$$\downarrow B \triangleq \{\ell \in \mathcal{L} \mid \exists b \in B \cdot b \sqsupseteq \ell\}$$

A set B is *downward-closed* if $\downarrow B = B$. We refer to such sets as *downsets*, and write $D(\mathcal{L}, \sqsubseteq)$ for the *downset lattice* of $\mathcal{L}$. The downset lattice of $PF(\{p\})$, ordered by set inclusion, is given in Figure 2(c).

Definition 7: [10] An element j in a lattice $\mathcal{L}$ is *join-irreducible* iff $j \neq \bot$ and for any x and y in $\mathcal{L}$, $j = x \sqcup y$ implies $j = x$ or $j = y$.
In other words, j is join-irreducible if it cannot be further decomposed into a join of other elements in the lattice. For example, the join-irreducible elements of the lattice in Figure 2(b) are $\{\mathsf{true}\}$, $\{p, \mathsf{true}\}$, $\{\neg p, \mathsf{true}\}$ and $\{\mathsf{false}, p, \neg p, \mathsf{true}\}$. We denote the set of all join-irreducible elements of a lattice $\mathcal{L}$ by $\mathcal{J}(\mathcal{L})$.

Every element of a finite lattice can be uniquely decomposed as a join of all join-irreducible elements below it.

Theorem 1: [10] Let ℓ be any element in a lattice $(\mathcal{L}, \sqsubseteq)$. Then $\ell = \bigsqcup\{j \in \mathcal{J}(\mathcal{L}) \mid j \sqsubseteq \ell\}$.
Note that in the case of the lattice of propositional formulas $PF(P)$, a join-irreducible element is a conjunction of literals of P. For example, the join-irreducible elements of $PF(\{p, q\})$ are: $p \wedge q$, $p \wedge \neg q$, $\neg p \wedge q$, and $\neg p \wedge \neg q$.

Proposition 2: The set of join-irreducible elements of the lattice $U(\mathcal{L})$ is isomorphic to $\mathcal{L}$, i.e.,

$$\mathcal{J}(U(\mathcal{L})) \quad = \quad \{\uparrow\ell \mid \ell \in \mathcal{L}\}$$

B. CTL Model Checking

CTL model checking is an automatic technique for verifying properties expressed in a propositional branching-time temporal logic called *Computation Tree Logic* (CTL) [2]. A model is a Kripke structure $K = (S, R, s_0, A, I)$, where S is a set of states, $R \subseteq S \times S$ is a (total) transition relation, $s_0 \in S$ is the initial state, A is a set of atomic propositions, and $I : S \to 2^A$ is a labeling function, providing the set of atomic propositions which hold in each state. CTL properties are evaluated on a tree of infinite computations produced by K.

The formal semantics of CTL is given in Figure 3. In this figure, we use the 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: $R(s) \triangleq \{s' | (s, s') \in R\}$. 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:

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

We say that a formula φ holds in a Kripke structure K if φ holds in its initial state. In Figure 3, we used conjunction and disjunction in place of the more familiar universal and existential quantification. The semantics of EX and AX can be alternatively expressed as

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

Note that our definition preserves the duality between EX and AX operators: $[\![AX\varphi]\!](s) = [\![\neg EX\neg\varphi]\!](s)$. Semantics of EG and EU are also given in Figure 3, where $\mu Z.f(Z)$ and $\nu Z.f(Z)$ are least and greatest fixpoints of f, respectively. Alternatively, we can define a *path* (or a *trace*) from a state s as a sequence of states $s = s_0, s_1, s_2, \ldots$ such that $(s_i, s_{i+1}) \in R$, and define an equivalent semantics for EG and EU as:

$$\begin{array}{rcl} [\![EG\varphi]\!](s) & \triangleq & \text{exists a path } s = s_0, s_1, \ldots \text{ s.t.} \\ & & \forall i \in \mathbb{N} \cdot [\![\varphi]\!](s_i) \\ [\![E[\varphi\ U\ \psi]]\!](s) & \triangleq & \text{exists a path } s = s_0, s_1, \ldots \text{ s.t.} \\ & & \exists j \in \mathbb{N} \cdot [\![\psi]\!](s_j) \wedge \forall\, 0 \leq i < j \cdot [\![\varphi]\!](s_i) \end{array}$$

The remaining CTL operators are given below:

$$\begin{array}{rcl} A[\varphi\ U\ \psi] & \triangleq & \neg E[\neg\psi\ U\ \neg\varphi \wedge \neg\psi] \ \wedge\ \neg EG\neg\psi \\ AF\varphi & \triangleq & A[\mathsf{true}\ U\ \varphi] \\ EF\varphi & \triangleq & E[\mathsf{true}\ U\ \varphi] \\ AG\varphi & \triangleq & \neg EF\neg\varphi \end{array}$$

$$
\begin{aligned}
[\![\ell]\!](s) &\triangleq \ell, \text{ for } \ell \in \{\mathsf{true}, \mathsf{false}\}\\
[\![a]\!](s) &\triangleq a \in I(s), \text{ for } a \in A\\
[\![\neg\varphi]\!](s) &\triangleq \neg[\![\varphi]\!](s)\\
[\![\varphi \wedge \psi]\!](s) &\triangleq [\![\varphi]\!](s) \wedge [\![\psi]\!](s)\\
[\![\varphi \vee \psi]\!](s) &\triangleq [\![\varphi]\!](s) \vee [\![\psi]\!](s)\\
[\![EX\varphi]\!](s) &\triangleq \textstyle\bigvee_{s' \in R(s)} [\![\varphi]\!](s')\\
[\![AX\varphi]\!](s) &\triangleq \textstyle\bigwedge_{s' \in R(s)} [\![\varphi]\!](s')\\
[\![EG\varphi]\!](s) &\triangleq [\![\nu Z.\varphi \ \wedge \ EXZ]\!](s)\\
[\![E[\varphi\, U\, \psi]]\!](s) &\triangleq [\![\mu Z.\psi \vee (\varphi \wedge EXZ)]\!](s)
\end{aligned}
$$

Fig. 3. Formal semantics of CTL operators.

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.

We further define two fragments of CTL: an existential fragment (ECTL) that allows only existential temporal operators, and a dual universal fragment (ACTL) that allows only universal temporal operators. Both fragments restrict the application of negation to atomic propositions. It is easy to see that for every ECTL property that holds in the model there exists a *witness* – a collection of paths exemplifying why the property is true. Dually, every ACTL property that fails to hold in the model can be explained by a *counter-example* – a collection of paths exemplifying why the negation of the property is true.

C. Multi-Valued Model Checking

Multi-Valued CTL model checking [8] is a generalization of the model-checking problem from a classical logic to an arbitrary *De Morgan* algebra $(\mathcal{L}, \sqsubseteq, \neg)$, 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 ($\ell \vee \neg\ell = \top$). When the ordering and the negation operation of an algebra $(\mathcal{L}, \sqsubseteq, \neg)$ are clear from the context, we refer to it as $\mathcal{L}$.

The model-checking problem can be extended in a number of ways: (a) transitions of a Kripke structure can be allowed to take on values from the algebra, (b) propositions of the Kripke structure can be evaluated to an element from the algebra, (c) lattice values can be introduced as constants to CTL. A general theory of multi-valued model checking that allows for all of the above extensions and their combinations is described in [11]. For the purpose of this paper we assume a multi-valued model-checking problem where the model remains classical (i.e. both the transition relation and atomic propositions are two-valued), but lattice values are allowed to appear as constants in temporal logic formulas.

Properties are specified in a multi-valued extension of CTL called χCTL. Given a De Morgan algebra $\mathcal{L}$, χCTL($\mathcal{L}$) has the same syntax as CTL, except that it extends the scope of allowable constants from $\{\mathsf{true}, \mathsf{false}\}$ to all $\ell \in \mathcal{L}$. It is assumed that the boolean values true and false occurring in the model are mapped to $\top$ and $\bot$ of the lattice, respectively. Thus, boolean formulas can be interpreted in each De Morgan algebra. The semantics of χCTL($\mathcal{L}$) are the same as CTL; the exception is the interpretation of constants:

$$[\![\ell]\!](s) \quad \triangleq \quad \ell, \quad \text{for } \ell \in \mathcal{L}$$

The other operations are defined as their CTL counterparts (see Figure 3), 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 "$\sqcap$". In what follows, we often write $[\![\varphi]\!] = [\![\psi]\!]$ to mean $\forall s \in S \cdot [\![\varphi]\!](s) = [\![\psi]\!](s)$.

For every $\mathcal{L}$, every CTL formula is also a χCTL($\mathcal{L}$) formula, but not vice versa. For example, $AGq \wedge \uparrow p$ is a χCTL formula over $(U(PF(\{p\}))$, but this formula is not in CTL.

Multi-valued model checking can be reduced to several classical model-checking problems [12]. Recall that each element of a lattice can be uniquely decomposed using join-irreducible elements. Thus, a solution to a χCTL($\mathcal{L}$) property φ in a state s of a Kripke structure can be obtained by solving $\varphi \sqsupseteq j$ for every join-irreducible element of the lattice and composing the results. The reduction comes from the following theorem.

Theorem 2: [12] For every De Morgan algebra $\mathcal{L}$, every χCTL($\mathcal{L}$) formula φ and a join-irreducible j, there exists a CTL formula $\varphi \Uparrow j$, called *the j-cut of* φ, such that

$$[\![\varphi \Uparrow j]\!] \quad = \quad ([\![\varphi]\!] \sqsupseteq j)$$

In our case, a CTL formula $\varphi \Uparrow j$ can be obtained from φ as follows: given a lattice value ℓ, replace it by true if $\ell \sqsupseteq j$, and by false otherwise. For example, a cut of $AG(q \wedge \uparrow p)$ with respect to $\uparrow p$ and $\uparrow\neg p$ is $AG(q \wedge \mathsf{true})$ and $AG(q \wedge \mathsf{false})$, respectively. Furthermore, since every CTL formula is also in χCTL, a cut of any CTL formula φ is itself ($\varphi \Uparrow j = \varphi$). Finally, Theorem 3 shows how to combine the results of the cuts to obtain the solution to the multi-valued model-checking problem.

Theorem 3: [12] Let $\mathcal{L}$ be a De Morgan algebra and φ be a χCTL($\mathcal{L}$) formula. Then,

$$[\![\varphi]\!] \quad = \quad \textstyle\bigsqcup_{j_i \in \mathcal{J}(\mathcal{L})} (j_i \wedge [\![\varphi \Uparrow j_i]\!])$$

In our example, if the $\uparrow p$-cut of $AG(q \wedge \uparrow p)$ is true but the $\uparrow\neg p$-cut is false, then the model-checking solution is $\uparrow p$. On the other hand, if both cuts are true, the solution becomes $\uparrow p \sqcup \uparrow\neg p = \uparrow\{p, \neg p\}$.

Given a De Morgan algebra $\mathcal{L}$, the complexity of model checking a χCTL($\mathcal{L}$) formula φ on a Kripke structure $K = (S, s_0, A, R, I)$ is $O(|S| \times |\varphi|)$, provided that meets, joins, and quantification operations take constant time [8], [13].

III. The Language of Queries

In this section we review query checking fundamentals and define the language of queries.

A. Queries with One Placeholder

Let P be a set of propositional variables, and let $PF(P)$, defined in Section II, be the set of propositional formulas over P.

Definition 8: A *temporal logic query* with a single placeholder $?_1$, denoted $\varphi[?_1]$, is an expression containing a symbol

$?_1$, where replacing $?_1$ by a propositional formula yields a CTL formula.

Note that this definition allows for multiple occurrences of the same placeholder symbol in the query. For example, both $AG?_1$, $?_1 \wedge AX?_1$ are valid queries with a single placeholder $?_1$. We denote the set of all temporal logic queries with one placeholder as TLQ.

We denote substituting a propositional formula ψ for a placeholder in a query $\varphi[?_1]$ over P as $\varphi[\psi]$. When we write $\varphi[?_1]$ on a Kripke structure K, we mean that all variables in A can be used in the solution. Otherwise, we name P explicitly. For convenience, we write $?_1\{p, q\}$ for a placeholder restricted to $\{p, q\}$, e.g. $AG?_1\{p, q\}$. Thus, every query (implicitly or explicitly) restricts the set of variables that can be substituted for the placeholder.

Definition 9: A substitution ψ satisfies a query $\varphi[?_1]$ over P in state s if and only if $[\![\varphi[\psi]]\!](s) = \mathsf{true}$.

For example, $(p \vee q) \wedge r$ satisfies the query $AG?_1$ in state s_0 of the model in Figure 1, whereas $p \vee q$ satisfies the query $AG?_1\{p, q\}$.

Definition 10: The *solution* to $\varphi[?_1]$ over P in state s, denoted $[\![\varphi[?_1]]\!](s)$, is the set of all substitutions from $PF(P)$ that satisfy φ in state s.

Thus, for a given Kripke structure K, a query $\varphi[?_1]$ over P can be thought of as a function from $PF(P)$ to $\{\mathsf{true}, \mathsf{false}\}$. We refer to the set of variables restricting a placeholder $?_1$ as $var(?_1)$. Thus, solutions to $?_1$ are formed from elements of $PF(var(?_1))$, which, abusing notation, we refer to as $PF(?_1)$.

A placeholder can appear in multiple places of a query, but each occurrence has to be restricted to the same set of propositional variables. For example, $AG(?_1\{p, q\}) \vee A[?_1\{p, q\}\, U\, r]$ is legal whereas $AG(?_1\{p, q\}) \vee A[?_1\{p, r\}\, U\, r]$ is not.

Definition 11: [3] A query φ is *positive* 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 an element of the solution, so is every weaker formula. Thus, the set of all possible solutions to queries over P is given by the upset lattice $U(PF(P))$, defined in Section II-A. Note that this means that the solution to a positive query $\varphi[?_1]$ can be uniquely represented by the set of maximally strong propositional formulas satisfying it. In what follows, we refer to this set as *the set of maximally strong solutions*. Moreover, if the set of maximally strong solutions to $\varphi[?_1]$ is a singleton, say $\{p\}$, we say that p is the (strongest) solution to $\varphi[?_1]$.

For simplicity of the following presentation, we assume that our queries are in *negation normal form* – negations appear only in front of atomic propositions and placeholders. We say that an occurrence of a placeholder $?_1$ in a query $\varphi[?_1]$ is *negative* if $?_1$ is preceded by a negation. Otherwise, an occurrence of $?_1$ is *positive*.

Theorem 4: [3] A query $\varphi[?_1]$ is positive if and only if every occurrence of $?_1$ is positive.

Definition 12: A query $\varphi[?_1]$ is *negative* if every occurrence of $?_1$ is negative. A query $\varphi[?_1]$ is *mixed* if it is neither positive nor negative.

For example, the query $AG(?_1 \wedge r)$ is positive, the query $EF(\neg ?_1)$ is negative, and the query $AG(?_1 \wedge r) \vee EF(\neg ?_1)$ is mixed. Note that as a function $\varphi[?_1] : PF(P) \rightarrow \{\mathsf{true}, \mathsf{false}\}$, $\varphi[?_1]$ is *monotone* for positive and negative queries. Thus, we refer to the set of all positive and negative queries as *monotone* queries.

The set of all possible solutions to negative queries over P is given by the downset lattice $D(PF(P))$ defined in Section II-A. Similarly to positive queries, the solution to a negative query $\varphi[?_1]$ can be uniquely represented by the set of weakest propositional formulas satisfying it, referred to as the set of *weakest solutions*.

B. Queries with Multiple Placeholders

A number of interesting model-exploration properties, such as the ones in Section V, cannot be formed using queries with just a single placeholder. We therefore extend the query language to include multiple independent placeholders. We denote the set of all queries with at most n placeholders by TLQ^n. A *substitution* for a query with multiple placeholders is a tuple of propositional formulas, one for each placeholder. Given a query $\varphi[?_1, \ldots, ?_n]$ on n placeholders, with $PF(?_i)$ being the lattice of propositional formulas for the ith placeholder, the set of all possible substitutions is the cross product $L = PF(?_1) \times \cdots \times PF(?_n)$. The set L forms a lattice under the implication ordering lifted pointwise to the elements of L. For example, if $L = PF(?_1) \times PF(?_2)$; $\varphi_1, \psi_1 \in PF(?_1)$ and $\varphi_2, \psi_2 \in PF(?_2)$, then:

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

Given a placeholder $?_i$, we refer to its name and variables restricting it as $name(?_i)$ and $var(?_i)$, respectively. The rest of the notation given in Section III-A trivially extends to queries with multiple placeholders.

A query can be *positive*, *negative* or *mixed* in a given placeholder. Further, a query is *positive* (*negative*, *monotone*) if it is positive (negative, monotone) in every placeholder. Queries that are not monotone are referred to as *non-monotone* or *mixed*. For example, $AG(?_1 \wedge AX?_2)$ is positive (and therefore monotone) whereas $AG(?_1 \Rightarrow AG(?_2 \Rightarrow AX?_1))$ is mixed.

Table I summarizes the different types of queries addressed in this paper. In column "Comments", we use "Chan" and "B&G" to indicate the queries that can be decided using William Chan's approach [3] and Bruns and Godefroid's approach [4], respectively. Note that the latter approach subsumes the former. For example, positive valid queries can be solved by both Chan's and B&G approaches, whereas general positive queries can only be solved using the B&G approach. Decision procedures for query types marked "new" are some of the technical contributions in this paper. In addition, the "Comments" column includes references to sections in this paper that describe the solutions to corresponding types of queries. For example, monotone queries with multiple placehoders are addressed in Section VII-B.2.

	Type	Example	Comment
Single placeholder			
1	positive, valid	$AG?_1$	Chan, Sec. VI
2	positive	$EF?_1$	B&G, Sec. VI
3	positive, several occurrences	$EF(?_1 \wedge EX?_1)$	new, Sec VI
4	negative, valid	$AG\neg?_1$	Chan, Sec. VII-B.1
5	negative	$EF\neg?_1$	B&G, Sec. VII-B.1
6	negative several occurrences	$EF(\neg?_1 \wedge EX\neg?_1)$	new, Sec. VII-B.1
7	mixed	$AG(?_1 \wedge r) \vee EF(\neg?_1)$	new, Sec. VII-C
Multiple placeholders			
8	positive	$AG(?_1 \wedge AX?_2)$	new, Sec. VII-B.2
9	negative	$EF(\neg?_1 \wedge EX\neg?_2)$	new, Sec. VII-B.2
10	monotone	$AG(?_1 \Rightarrow AF?_2)$	new, Sec. VII-B.2
11	mixed	$EF(?_1 \Rightarrow AG(?_2 \Rightarrow AX?_1))$	new, Sec. VII-C

TABLE I
SUMMARY OF TYPES OF QUERIES.

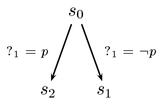

Fig. 4. A witness for $EX?_1\{p\}$, in the model in Figure 1.

C. Queries and Witnesses

Given an ECTL formula that holds in a model, a *witness* is a trace through the model showing *why* the formula holds. Similarly, a *witness* to an existential query is a set of traces showing why each of the solutions satisfies the query.

For example, consider the query $EX?_1\{p\}$ for the model in Figure 1. It has two maximally strong solutions: $?_1 = p$ and $?_1 = \neg p$; therefore, the witness consists of two traces, one for each maximally strong solution, as shown in Figure 4. The trace s_0, s_2 corresponds to the solution p, meaning that this trace explains why all propositional formulas in $\uparrow p$ satisfy the query. The trace s_0, s_1 corresponds to the solution $\neg p$.

All of the traces comprising a witness to a query start from the initial state, so they can be represented as a tree. In addition, each branch of the tree can be labeled with the set of propositional formulas whose membership in the solution set is illustrated by that branch. In the example in Figure 4, the left branch is labeled with $?_1 = p$, and the right – with $?_1 = \neg p$. 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 whose traces have the longest common prefix, which usually results in minimizing the total number of traces comprising the witness [14].

IV. TEMPORAL LOGIC QUERY CHECKER: TLQSOLVER

In this section, we describe the design and the implementation of the temporal logic query checker TLQSolver. The architecture of TLQSolver is given in Figure 5.

TLQSolver is implemented in Java on top of our χCTL model checker χChek [9] using the TLQ to χCTL reduction technique presented in Section VI and Section VII. χChek is a symbolic χCTL model checker that provides support for both model checking with fairness and the generation of witnesses (and counter-examples). Given a Kripke structure K (the model), a De Morgan algebra $\mathcal{L}$ and a χCTL($\mathcal{L}$) formula φ, χChek returns the element of the algebra corresponding to the value of φ in the initial state of K. The model can be represented in a variety of modeling languages; in particular, the SMV modeling language is supported. Internally, χChek can use one of several decision diagram packages for symbolic state space representation. In particular, it can use Algebraic Decision Diagrams (ADD) [15], also known as Multi-Terminal Binary Decision Diagrams (MBTDD) [16], that are part of CUDD – a state-of-the-art CU decision diagram package [15].

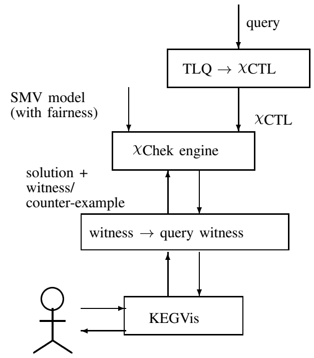

Fig. 5. Architecture of TLQSolver.

The witness generation facility of χChek is handled by interactive witness visualization and exploration tool KEGVis [14]. KEGVis presents the witnesses graphically using *daVinci Presenter* [17] for layout and exploration. In addition to simply browsing witnesses, the user can define a number of strategies for their exploration [14]: forward and backward, setting step granularity, choosing witnesses based on size, branching factor, etc.

TLQSolver uses the combination of χChek and KEGVis as a black box. Given a temporal logic query and an SMV model, it first converts the query into a corresponding χCTL formula. The model and the resulting formula are then passed to χChek for model checking. Finally, the answer and the witness returned by χChek are transformed from multi-valued logic into sets of propositional formulas and presented to the user through KEGVis.

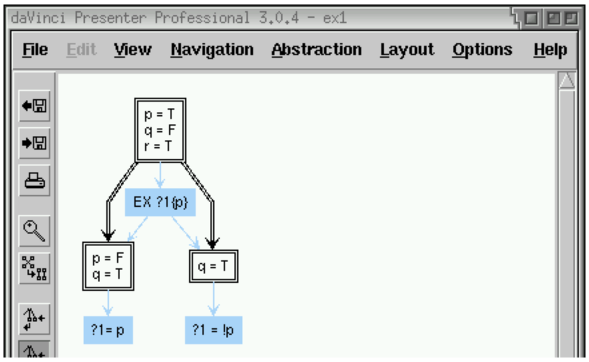

Fig. 6. KEGVis presentation of the witness for $EX?_1\{p\}$.

Consider for example the query $EX?_1\{p\}$ evaluated in the state s_0 of the Kripke structure in Figure 1; its witness is shown in Figure 4. A screen-shot of TLQSolver displaying the same witness is given in Figure 6. The presentation follows the proof-like [14], [18], [19] style where state nodes (double lines) are labeled with proof steps that depend on them. The double lines correspond to transitions in the Kripke structure, while single lines indicate dependencies between proof nodes and state nodes. For example, the proof node labeled $\texttt{EX ?1}\{\texttt{p}\}$ is connected with the two states required to explain the solution.

For a copy of either tool, please send e-mail to xchek@cs.toronto.edu.

V. Applications and Experience

In this section, we show two different techniques for model exploration using temporal logic queries. The technique presented in Section V-B only uses solutions to the query-checking problem and is essentially an extension of the methodology proposed by Chan in [3] to solving queries with multiple placeholders, in both positive and negative positions. The technique presented in Section V-C is completely new and is based on the fact that in addition to computing the solution to a query, TLQSolver can also provide a witness explaining it. The examples in this section are based on our experience in exploring an SCR specification of a Cruise Control System [20], described in Section V-A. Please refer to Table IV for the running time of TLQSolver on various queries used in this section.

A. 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 ($\texttt{Button} = \texttt{bCruise}$) 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 ($\texttt{Button} = \texttt{bOff}$); (4) the engine stops running (Running); (5) the driver turns the ignition off (Ignition); (6) the car's speed becomes uncontrollable. 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 ($\texttt{Button} = \texttt{bResume}$). In addition, the system keeps track of the number of miles the car traveled to determine if it is due for service or its oil or air filter need to be replaced.

In our case study, we use the functional part of the SCR specification of CCS done by Jim Kirby [20]. The SCR method [21] 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.

Our version of CCS [22] has 12 monitored and 4 controlled variables and one modeclass CC. In specifying CCS, we made two major assumptions: (1) we did not use some features of the SCR method such as terms and condition tables and (2) we ensured that all variables used in the system are boolean or enumerated type, for the ease of model checking. In particular, referring to the speed to be maintained by CCS as Sp_Des and the speed of the car as Sp_Veh, we defined the following predicates:

Toofast	the car's speed is out of control: $\texttt{Sp\_Veh} > \texttt{MAX\_SPEED}$
$\texttt{Speed} = \texttt{fast}$	$\texttt{Sp\_Veh} > \texttt{Sp\_Des} + \texttt{THRESHOLD}$
$\texttt{Speed} = \texttt{slow}$	$\texttt{Sp\_Veh} < \texttt{Sp\_Des} - \texttt{THRESHOLD}$
$\texttt{Speed} = \texttt{ok}$	$(\texttt{Sp\_Des} - \texttt{THRESHOLD} \leq \texttt{Sp\_Veh}) \wedge (\texttt{Sp\_Veh} \leq \texttt{Sp\_Des} + \texttt{THRESHOLD})$

The modeclass CC is described in Table II. 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. Formally, the two events, the change in Ignition and the mode transition, occur at the same time: $\neg\texttt{Ignition} \wedge \texttt{CC} = \texttt{Off} \wedge \texttt{Ignition}' \wedge \texttt{CC}' = \texttt{Inactive}$. In order to facilitate the computation of the next state, the SCR method assumes that at most one monitored variable can change during a time interval (*Single Input Assumption*) and that there is a partial order of dependencies between the variables in the system.

Table III 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 III; 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).

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 II

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 III

EVENT TABLE FOR THE CONTROLLED VARIABLE Throttle.

	Query	**Time**	**Explanation**
1	$EF?_1\{\texttt{CC}\}$	$0.23s$	what are all reachable modes
2	$EF(\texttt{CC} = \texttt{Cruise} \wedge ?_1\{\texttt{Throttle}\})$	$0.23s$	what values of Throttle are reachable in mode Cruise
3	$EF(\texttt{CC} = \texttt{Off} \wedge EX?_1\{\texttt{CC}\})$	$0.032s$	what modes can follow Off
4	$EF(?_1\{\texttt{CC}\} \wedge EX?_2\{\texttt{CC}\})$	$0.33s$	what pairs of modes can follow each other
5	$EF\big((\texttt{CC} = \texttt{Cruise}) \wedge ?_1\{\texttt{Toofast}, \texttt{Running}\} \wedge EX(?_2\{\texttt{Toofast}, \texttt{Running}\} \wedge \texttt{CC} = \texttt{Inactive})\big)$	$0.18s$	how do values of Toofast and Inactive change as the system goes between modes Cruise and Inactive
6	$AG(\texttt{CC} = \texttt{Inactive} \Rightarrow ?_1\{\texttt{Ignition}, \texttt{Running}\})$	$0.17s$	what are the invariants, over Ignition and Running, of mode Inactive
7	$AG(?_1\{\texttt{CC}\} \Rightarrow ?_2\{\texttt{Ignition}, \texttt{Running}\})$	$0.33s$	find all mode invariants, restricted to Ignition and Running
8	$AGEF?_1\{\texttt{CC}\}$	$0.24s$	what modes are globally reachable
9	$EFEG?_1\{\texttt{CC}\}$	$0.23s$	in what modes can the system stay forever

TABLE IV

RUNNING TIME OF A FEW QUERIES ON THE CRUISE CONTROL SYSTEM.

B. Applications of Queries without Witnesses

We have converted the SCR specification of the Cruise Control System into the SMV format [5] using the tool SC(R)[3] [22]. One can analyze the resulting model using several questions to the symbolic CTL model checker SMV; instead, we show how temporal logic queries can be used to help express reachability properties and discover system invariants and transition guards.

1) 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 $EF?_1\{\texttt{CC}\}$ 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 solution

$$?_1 = \{\texttt{CC} = \texttt{Off}, \texttt{CC} = \texttt{Cruise}, \texttt{CC} = \texttt{Inactive}, \texttt{CC} = \texttt{Override}\}$$

includes 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 $EF\big((\texttt{CC} = \texttt{Cruise}) \wedge ?_1\{\texttt{Throttle}\}\big)$, whose solution is $?_1 = (\texttt{Throttle} \neq \texttt{tOff})$.

More complex analysis can be done by combining EF queries with other CTL operators. For example, a query $EF(\texttt{CC} = \texttt{Off} \wedge EX?_1\{\texttt{CC}\})$ can be used to find the modes which can follow Off; a query $EF(?_1\{\texttt{CC}\} \wedge EX?_2\{\texttt{CC}\})$ can be used to determine which pairs of modes can follow each other; and a query

$$EF\big((\texttt{CC} = \texttt{Cruise}) \wedge ?_1\{\texttt{Toofast}, \texttt{Running}\} \wedge EX(?_2\{\texttt{Toofast}, \texttt{Running}\} \wedge (\texttt{CC} = \texttt{Inactive}))\big)$$

Mode	Invariant
Off	$\neg$Ignition $\wedge$ $\neg$Toofast $\wedge$ $\neg$Running
Inactive	Ignition
Cruise	Ignition $\wedge$ $\neg$Brake $\wedge$ $\neg$Toofast $\wedge$ Running
Override	Ignition $\wedge$ Running

TABLE V

MODE INVARIANTS OF THE CRUISE CONTROL SYSTEM.

Event	Correspondence
@T(Ignition)	transition to mode Inactive from the mode Off.
@T(Running)	essential to enabling a transition to mode Cruise.
@T(Button = bCruise)	transition to mode Cruise.
@T(Button = bOff)	transition to mode Override.

TABLE VI

A SEQUENCE OF EVENTS FOR GUIDED SIMULATION.

can be used to explore how the values of Toofast and Inactive change as the system goes between modes Cruise and Inactive.

2) Discovering invariants: Invariants concisely summarize complex relationships between different entities in the model, and are often useful in identifying errors. This was one of the original motivations behind the introduction of query checking by William Chan [3]. To discover all invariants, we simply need to solve the query $AG?_1$, 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 [3]. 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

$$AG\big((\texttt{CC} = \texttt{Inactive}) \Rightarrow ?_1\{\texttt{Ignition}, \texttt{Running}\}\big)$$

which evaluates to $?_1 = \texttt{Ignition}$. Furthermore, using multiple placeholders, we can find all invariants of each mode using a single query. For example, each solution to the query

$$AG(?_1\{\texttt{CC}\} \Rightarrow ?_2\{\texttt{Ignition}, \texttt{Running}\})$$

corresponds to invariants of each individual mode with respect to variables Ignition and Running. In our example, the solution

$$?_1 = (\texttt{CC} = \texttt{Cruise}) \qquad ?_2 = \texttt{Ignition} \wedge \texttt{Running}$$

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

$$?_1 = (\texttt{CC} = \texttt{Inactive}) \vee (\texttt{CC} = \texttt{Cruise}) \vee (\texttt{CC} = \texttt{Override})$$
$$?_2 = \texttt{Ignition}$$

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, restricted to the variables Ignition, Brake, Toofast and Running, that we were able to discover using query checking are shown in Table V. They are equivalent to the invariants discovered by the algorithms in [23], [24], [25]. Notice that the strength of the invariants obtained through query checking depends on the variables to which the placeholder is restricted. For example, the result of the query

$$AG\big((\texttt{CC} = \texttt{Cruise}) \Rightarrow ?_1\{\texttt{Ignition}, \texttt{Running}\}\big)$$

is not the strongest invariant for mode Cruise (e.g. the one in Table V is stronger). The strongest invariants can be obtained by restricting the placeholder to *all* of the monitored variables of the system.

Combining reachability and invariant computation results in many interesting queries. For example, $AGEF?_1\{\texttt{CC}\}$ can be used to determine which modes of CC are globally reachable, whereas $EFEG?_1\{\texttt{CC}\}$ can be used to find all reachable modes in which the system can stay forever.

3) Guard discovery: Finally, we illustrate how queries can be used to discover *guards* [26]. For example, we would like to know the change in the environment that prompted the mode transition from $\texttt{CC} = \texttt{Off}$ to $\texttt{CC}' = \texttt{Inactive}$. Clearly, guards that enable transitions between modes can be easily obtained from examining SCR tables: from the first raw of Table II we see that this mode transition is caused by @T(Ignition). Yet, guards are not readily available once SCR tables have been converted into the Kripke structure format of SMV. We can reverse-engineer the event tables by discovering guards in the resulting Kripke structure.

Formally, a guard γ is defined as a weakest propositional formula over current (pre-) and next (post-) states such that the invariant $\alpha \wedge \gamma \Rightarrow \beta$ holds, where α and β are the pre- and post-conditions, respectively [26]. 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 $AG(\alpha \wedge ?_1 \Rightarrow AX(?_2 \Rightarrow \beta))$.

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, we restrict the $?_1$ and $?_2$ placeholders to the set $\{\texttt{Toofast}, \texttt{Running}, \texttt{Brake}\}$. Note that we did not include Ignition since it is invariantly true in both modes (see Table V). After solving this query, we obtain two weakest solutions:

$$?_1 = \texttt{Brake} \vee \texttt{Toofast} \vee \neg\texttt{Running} \qquad ?_2 = \textsf{true}$$
$$?_1 = \textsf{true} \qquad ?_2 = \neg\texttt{Running} \vee \texttt{Toofast}$$

Before analyzing the result, we note that the first solution violates the invariant for mode Cruise (see Table V), 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.

C. Applications of Queries with Witnesses

We now present a few applications of query checking that use witnesses.

1) 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 simulation is usually slow and error prone.

An interesting alternative to simple simulation is *guided simulation*. In a guided simulation setting, the user provides a set of objectives to be demonstrated by the simulation, and then only needs to *guess* the inputs in cases where the objective cannot be met by a single execution of the system. Moreover, each choice is presented together with the set of objectives it satisfies.

Query checking is a natural framework for implementing guided simulations. The objective is given by an existential 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 $EF?_1\{\texttt{CC}\}$ and explore the witness. Moreover, we indicate that we prefer witnesses with the longest 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 given in Table VI. Since our objective was achieved by a single trace, the simulation was generated completely automatically, requiring no user input.

2) 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 [27], [28], [29], [26]. Most of the proposed techniques are based on the fact that in addition to computing values of the given properties, 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* [27].

Gargantini and Heitmeyer [27] 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 sequences. We illustrate the method on mode transition tables; other tables can be analyzed similarly.

The method in [27] 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, 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 $EF\big((\texttt{CC} = \texttt{Off}) \wedge EX?_1\{\texttt{CC}\}\big)$. Similarly, the set of test sequences that cover the full mode transition table is obtained from the witness of the query $EF(?_1\{\texttt{CC}\} \wedge EX?_2\{\texttt{CC}\})$.

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 a query has more then one minimal solution, the query checker can produce each minimal solution, and, if necessary, a witness, as soon as it is found. Therefore, even in the cases when the complexity of 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. One solution to this problem is: (a) examine the query witness to discover what events are not yet covered by the test sequences, and (b) generate the remaining test sequences by explicit trap properties following the methodology of [27]. For example, if the witness covers the event @T(`Toofast`), then the test sequence covering the event @F(`Running`) is obtained by the trap property

$$\begin{aligned}EF\big((&\texttt{CC} = \texttt{Cruise}) \ \wedge \ \texttt{Running} \\ &\wedge EX(\neg\texttt{Running} \wedge (\texttt{CC} = \texttt{Inactive}))\big)\end{aligned}$$

A better solution, provided that we know the variables comprising the event for a given mode transition, is to use 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

$$\begin{aligned}EF\big(&(\texttt{CC} = \texttt{Cruise}) \wedge ?_1\{\texttt{Toofast}, \texttt{Running}\} \ \wedge \\ &EX\,(?_2\{\texttt{Toofast}, \texttt{Running}\} \wedge (\texttt{CC} = \texttt{Inactive}))\big)\end{aligned}$$

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`).

VI. Deciding Temporal Logic Queries with a Single Placeholder

We now turn our attention to deciding temporal logic queries. In this section, we consider the query checking problem for a positive query with a single placeholder, potentially occurring in several places in the query, e.g., $EF(?_1\{p,q\} \wedge p)$, $A[?_1\{p,q\}\ U\ ?_1\{p,q\}]$. We show that this problem is an instance of a multi-valued model-checking problem and thus can be expressed in terms of the framework described in Section II-C. A more complex class of queries is considered in Section VII.

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 2). In order to reduce query checking to multi-valued model checking, we need to define a syntactic translation $T_\uparrow$ from positive temporal logic queries with a single placeholder to χCTL formulas over the lattice $U(PF(?_1))$ such that the solution to the χCTL formula $T_\uparrow(\varphi)$ is the solution to the query φ: $[\![\varphi[?_1]]\!] = [\![T_\uparrow(\varphi[?_1])]\!]$. The lattice $U(PF(?_1))$ is the basis for our De Morgan algebra.

A. Reduction to χCTL

Consider a query $?_1\{p\}$ evaluated in the model in Figure 1. Solving this query with respect to s_0, we notice that the formula p holds in s_0, and all other formulas from $PF(\{p\})$ that hold in s_0 are implied by it. Thus, $\uparrow\{p\}$ is the solution to the query. For an arbitrary state s of a Kripke structure K, the solution to the query $?_1\{p\}$ is given by

$$[\![?_1\{p\}]\!](s) = \begin{cases} \uparrow\{p\} & \text{if } [\![p]\!](s) = \mathsf{true} \\ \uparrow\{\neg p\} & \text{if } [\![\neg p]\!](s) = \mathsf{true} \end{cases}$$

Thus, a χCTL($\mathcal{L}$) formula corresponding to this query is $p \wedge \uparrow\{p\} \vee \neg p \wedge \uparrow\{\neg p\}$. In general, the solution to the query $?_1$ restricted to an arbitrary set of atomic propositions is

$$[\![?_1]\!](s) = \uparrow j \quad \textbf{iff} \quad [\![j]\!](s) = \mathsf{true}$$

where j is an element of $\mathcal{J}(PF(?_1))$. For example, the solution to the query $?_1\{p,q\}$ in state s_0 of the model in Figure 1 is $\uparrow\{p \wedge \neg q\}$ because $p \wedge \neg q$ is a join-irreducible of the lattice $PF(\{p,q\})$, and it is true in s_0.

The syntactic translation $T_\uparrow$ of the query $?_1$ into a χCTL($\mathcal{L}$) formula over the lattice $U(PF(?_1))$ is

$$T_\uparrow(?_1) \quad \triangleq \quad \bigvee_{j \in \mathcal{J}(PF(?_1))} (j \wedge \uparrow j)$$

This transformation is correct if the solution to a query $?_1$ in state s is the same as the value of the resulting χCTL($\mathcal{L}$) formula in the same state, i.e. if $[\![?_1]\!](s) = [\![T_\uparrow(?_1)]\!](s)$. This equality follows trivially from the definition of $T_\uparrow$.

One can think of the transformation $T_\uparrow$ as a two-stage transformation. The first stage transforms $?_1$ into a disjunction, where each disjunct corresponds to a unique join-irreducible element of $PF(?_1)$. The second stage assigns a value from the lattice to each disjunct. The values used by the transformation determine the model-checking lattice $\mathcal{L}$. In our case, the value assigned to each disjunct $j \in \mathcal{J}(PF(?_1))$ is its upward closure $\uparrow j$, but in principle we can replace $\uparrow$ by a function $f : \mathcal{J}(PF(?_1)) \to \mathcal{L}$ which maps each element $j \in \mathcal{J}(PF(?_1))$ to a lattice value corresponding to the solution of $?_1$ in a state satisfying j. Thus, our generalized transformation function becomes:

$$T_f(?_1) \quad \triangleq \quad \bigvee_{j \in \mathcal{J}(PF(?_1))} (j \wedge f(j))$$

We define several different functions f in Section VII.

Finally, we need to extend T_f from the query $?_1$ to the set of all temporal logic queries:

$$T_f(\varphi[?_1]) \triangleq \varphi[T_f(?_1)]$$

i.e., to translate a query, we just need to translate the placeholder. For example, suppose we are interested in computing $EX?_1\{p,q\}$ in state s_0 of the model in Figure 1. First, we apply the transformation T_f and obtain $T_f(EX?_1\{p,q\}) = EX\ T_f(?_1\{p,q\})$. Expanding $T_f(?_1\{p,q\})$, we get

$$\begin{aligned} T_f(?_1\{p,q\}) \quad = \quad & p \wedge q \wedge \uparrow\{p \wedge q\} \vee \\ & p \wedge \neg q \wedge \uparrow\{p \wedge \neg q\} \vee \\ & \neg p \wedge q \wedge \uparrow\{\neg p \wedge q\} \vee \\ & \neg p \wedge \neg q \wedge \uparrow\{\neg p \wedge \neg q\} \end{aligned}$$

Finally, we evaluate the resulting formula on state s_0:

$$\begin{aligned} & [\![EX\ T_f(?_1\{p,q\})]\!](s_0) \\ = \ & [\![T_f(?_1\{p,q\})]\!](s_1) \vee [\![T_f(?_1\{p,q\})]\!](s_2) \\ = \ & \uparrow\{\neg p \wedge q\} \vee \uparrow\{p \wedge q\} \\ = \ & \uparrow\{p \wedge q, \neg p \wedge q\} \end{aligned}$$

Thus, $\uparrow\{p\wedge q, \neg p \wedge q\}$ is the solution to the query $EX?_1\{p,q\}$ in state s_0 of the model in Figure 1.

B. Proof of Correctness

We now prove that our transformation $T_\uparrow$ is in fact the reduction from query checking into multi-valued model checking. To do so, we show that for all states s of a given Kripke structure, the solution using $T_\uparrow$ is equivalent to the *naive* (see Section I) solution to the query checking problem, i.e. $[\![\varphi[?_1]]\!](s) = [\![T_\uparrow(\varphi[?_1])]\!](s)$.

The proof of correctness is based on the reduction from multi-valued model checking into classical model checking, described in Section II-C. Given an algebra $(\mathcal{L}, \sqsubseteq, \neg)$, let φ be a χCTL($\mathcal{L}$) formula. The reduction to classical CTL is based on computing the solutions to each cut $\varphi \Uparrow j$ of φ with respect to join-irreducible elements $j \in \mathcal{J}(\mathcal{L})$ [30]. In our case, the model-checking lattice is $U(PF(?_1))$, and its join-irreducible elements are $\mathcal{J}(U(PF(?_1))) = \{\uparrow\ell \mid \ell \in PF(?_1)\}$. To avoid confusion: we use join-irreducible elements of $PF(?_1)$ to translate a query into a χCTL($\mathcal{L}$) formula, and join-irreducible elements of $U(PF(?_1))$ are used in the proof of correctness of $T_\uparrow$.

We start by showing that a value of cut of a χCTL($\mathcal{L}$) propositional formula $T_f(?_1)$ can be computed directly, using a classical propositional formula. We prove this result for T_f instead of just for $T_\uparrow$ because we intend to use it in Section VII.

Lemma 1: Let $?_1$ be a temporal logic query, $\mathcal{L}$ a lattice, $\ell \in \mathcal{J}(\mathcal{L})$, $f : \mathcal{J}(PF(?_1)) \to \mathcal{L}$, and T_f as defined above. Then,

$$T_f(?_1) \Uparrow \ell \quad = \quad \bigvee_{\{j_i \in \mathcal{J}(PF(?_1)) \mid f(j_i) \sqsupseteq \ell\}} j_i$$

Proof:

$$
\begin{array}{ll}
 & T_f(?_1) \Uparrow \ell \quad \text{(Theorem 2)} \\
= & T_f(?_1) \sqsupseteq \ell \quad \text{(defn. of } T_f) \\
= & \left(\bigvee_{\{j_i \in \mathcal{J}(PF(?_1))\}} j_i \wedge f(j_i)\right) \sqsupseteq \ell \quad (\ell \text{ is join-irreducible}) \\
= & \bigvee_{\{j_i \in \mathcal{J}(PF(?_1))\}} (j_i \sqsupseteq \ell \wedge f(j_i) \sqsupseteq \ell) \quad (j_i \text{ is boolean}) \\
= & \bigvee_{\{j_i \in \mathcal{J}(PF(?_1))\}} (j_i \wedge f(j_i) \sqsupseteq \ell) \quad (\sqsupseteq \text{ is boolean}) \\
= & \bigvee_{\{j_i \in \mathcal{J}(PF(?_1)) \mid f(j_i) \sqsupseteq \ell\}} j_i
\end{array}
$$

□

We now continue to define specific properties of $T_\uparrow$. We show that a cut of a χCTL($\mathcal{L}$) formula $T_\uparrow(\varphi[?_1])$ with respect to a join-irreducible $\uparrow\ell \in \mathcal{J}(U(PF(?_1)))$ is equivalent to a CTL formula $\varphi[\ell]$ obtained from $\varphi[?_1]$ by substituting ℓ for the placeholder.

Theorem 5: Let $\varphi[?_1]$ be a positive temporal logic query, $\ell \in PF(?_1)$. Then,

$$(T_\uparrow(\varphi[?_1]) \Uparrow \uparrow\ell) \quad = \quad \varphi[\ell]$$

Proof:
By Theorem 2, it is sufficient to show that $(T_\uparrow(?_1) \Uparrow \uparrow\ell) = \ell$.

$$
\begin{array}{ll}
 & T_\uparrow(?_1) \Uparrow \uparrow\ell \quad \text{(Theorem 2)} \\
= & T_\uparrow(?_1) \sqsupseteq \uparrow\ell \quad \text{(Lemma 1)} \\
= & \bigvee_{\{j_i \in \mathcal{J}(PF(?_1)) \mid \uparrow j_i \sqsupseteq \uparrow\ell\}} j_i \quad \text{(anti-monotonicity of } \uparrow) \\
= & \bigvee_{\{j_i \in \mathcal{J}(PF(?_1)) \mid j_i \sqsubseteq \ell\}} j_i \quad \text{(Theorem 1)} \\
= & \ell
\end{array}
$$

□

The next theorem establishes the correctness of our transformation from positive temporal logic queries with one placeholder to χCTL formulas over $U(PF(?_1))$.

Theorem 6: Let $\varphi[?_1]$ be a temporal logic query, and $T_\uparrow$ be as defined above. Then,

$$[\![\varphi[?_1]]\!] \quad = \quad [\![T_\uparrow(\varphi[?_1])]\!]$$

Proof:

$$
\begin{array}{ll}
 & [\![T_\uparrow(\varphi[?_1])]\!](s) \quad \text{(Theorem 3, Proposition 2)} \\
= & \bigsqcup_{j \in PF(?_1)} (\uparrow j \wedge [\![T_\uparrow(\varphi[?_1]) \Uparrow \uparrow j]\!](s)) \quad \text{(Theorem 5)} \\
= & \bigsqcup_{j \in PF(?_1)} (\uparrow j \wedge [\![\varphi[j]]\!](s)) \quad \text{(defn. of } \sqcup) \\
= & \{j \in PF(?_1) \mid [\![\varphi[j]]\!](s)\} \quad \text{(Definition 10)} \\
= & [\![\varphi[?_1]]\!](s)
\end{array}
$$

□

VII. Deciding More Complex Queries

In this section, we describe how to solve non-positive queries. We start with queries with multiple placeholders, then consider negative queries, both with a single and with multiple placeholders, and finally deal with mixed queries.

A. Queries with Multiple Placeholders

1) Reduction to χCTL : Let φ be a query with two placeholders, $?_1$ and $?_2$, where only one placeholder is used, e.g., $\varphi[?_1, ?_2] = ?_1$. Let $\uparrow j \subseteq PF(?_1)$ be the solution to $[\![?_1]\!](s)$ on a state s as a query with a single placeholder. Then a substitution $(a, b) \in PF(?_1) \times PF(?_2)$ satisfies φ in state s if and only if $a \in \uparrow j$, and, therefore, the solution to $[\![\varphi[?_1, ?_2]]\!](s)$ is $\uparrow j \times PF(?_2) = \uparrow(j, \bot)$. Thus, if $\varphi[?_1, ?_2]$ uses only one placeholder, say $?_1$, its solution is obtained by embedding the solution for $?_1$ into the lattice $U(PF(?_1) \times PF(?_2))$.

In general, given a query $\varphi[?_1, \ldots, ?_n]$ on n placeholders, with $PF(?_i)$ being the lattice of propositional formulas for the ith placeholder, the set of all possible substitutions is the cross product $L = PF(?_1) \times \cdots \times PF(?_n)$. The set L forms a lattice under the implication ordering lifted pointwise to the elements of L. Also, the set of all solutions to a positive query in TLQ^n is given by $U(L) = U(PF(?_1) \times \cdots \times PF(?_n))$. Thus, for any $?_i \in$ TLQ^n, with $1 \leq i \leq n$, the desired embedding is given by the function $f_i : \mathcal{J}(PF(?_i)) \to U(L)$ defined as:

$$
\begin{array}{lll}
f_i(j) & \triangleq & PF(?_1) \times \cdots PF(?_{i-1}) \times \uparrow j \times \cdots \\
 & = & \uparrow(\bot_1, \ldots, \bot_{i-1}, j, \ldots, \bot_n) \quad \text{(defn. of } f_i)
\end{array}
$$

where $\bot_i$ is the least element of $PF(?_i)$. That is, to translate a query $?_i \in$ TLQ^n, we use the translation function T_f, defined in Section VI-A, with the above-defined f. Finally, we transform the query on n placeholders by replacing each placeholder in turn (going from $?_i$ to the result of T_{f_i}):

$$T^n(\varphi[?_1, \ldots, ?_n]) \triangleq \varphi[T_{f_1}(?_1), \ldots, T_{f_n}(?_n)] \quad \text{(defn. of } T^n)$$

2) Example: We now give an example of computing a solution to query with two placeholders. We evaluate the query $?_1\{p, q\} \wedge EX?_2\{p, q\}$ in state s_0 of the model in Figure 1. From the example in Section VI with one placeholder,

$$
\begin{array}{lll}
[\![?_1]\!](s_0) & = & \uparrow\{p \wedge \neg q\} \\
[\![EX?_2]\!](s_0) & = & \uparrow\{p \wedge q, \neg p \wedge q\}
\end{array}
$$

Further, note that $\uparrow\mathsf{false} = PF(\{p, q\})$. Lifting the solution to the lattice $PF(\{p, q\}) \times PF(\{p, q\})$, we obtain

$$
\begin{array}{lll}
[\![?_1]\!](s_0) & = & f_1(\uparrow\{p \wedge \neg q\}) = \uparrow\{p \wedge \neg q\} \times \uparrow\{\mathsf{false}\} \\
[\![EX?_2]\!](s_0) & = & f_2(\uparrow\{p \wedge q, \neg p \wedge q\}) \\
 & = & \uparrow\{\mathsf{false}\} \times \uparrow\{p \wedge q, \neg p \wedge q\}
\end{array}
$$

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) Proof of Correctness: As in Section VI-B, we do so by reducing multi-valued model checking into classical model checking [12]. In this case, the model-checking lattice is $U(L)$, and the set of its join-irreducible elements is $\{\uparrow j \mid j \in L\}$.

Given a De Morgan algebra L, we now show that a cut of a χCTL(L) formula obtained by the translation T^n from a query $\varphi[?_1, \ldots, ?_n]$ with respect to a join-irreducible $\uparrow j$ is equivalent to a CTL formula resulting from substituting j into φ. The correctness of this approach results from the following theorem.

Theorem 7: Let $\varphi[?_1, \ldots, ?_n] \in$ TLQ^n be a positive temporal logic query with at most n placeholders, T^n : TLQ$^n \to \chi$CTL($\mathcal{L}$) be as defined above, and $(\ell_1, \ldots, \ell_n) \in PF(?_1) \times \cdots \times PF(?_n)$. Then,

$$(T^n(\varphi[?_1, \ldots, ?_n]) \Uparrow \uparrow(\ell_1, \ldots, \ell_n)) \quad = \quad \varphi[\ell_1, \ldots, \ell_n]$$

Proof:
As in Theorem 5, it is sufficient to show that

$$(T^n(?_i) \Uparrow \uparrow(\ell_1, \ldots, \ell_n)) = \ell_i$$

$$
\begin{array}{ll}
 & T^n(?_i) \Uparrow \uparrow(\ell_1,\ldots,\ell_n) \\
 & \quad \text{Theorem 2} \\
= & T^n(?_i) \sqsupseteq \uparrow(\ell_1,\ldots,\ell_n) \\
 & \quad \text{defn. of } T^n \\
= & T_{f_i}(?_i) \sqsupseteq \uparrow(\ell_1,\ldots,\ell_n) \\
 & \quad \text{Lemma 1} \\
= & \bigvee_{\{j_k \in \mathcal{J}(PF(?_i)) \mid f_i(j_k) \sqsupseteq \uparrow(\ell_1,\ldots,\ell_n)\}} j_k \\
 & \quad \text{defn. of } f_i\text{, upset law} \\
= & \bigvee_{\{j_k \in \mathcal{J}(PF(?_i)) \mid (\bot_1,\ldots,j_k,\ldots,\bot_n) \sqsubseteq (\ell_1,\ldots,\ell_n)\}} j_k \\
 & \quad \text{defn. of } \sqsubseteq \text{ and } \bot_i \\
= & \bigvee_{\{j_k \in \mathcal{J}(PF(?_i)) \mid j_k \sqsubseteq \ell_i\}} j_k \\
 & \quad \text{join-irreducible law} \\
= & \ell_i
\end{array}
$$

□

Theorem 8: Let $\varphi[?_1,\ldots,?_n] \in \text{TLQ}^n$ be a positive query with at most n placeholders, s be a state of a Kripke structure, and T^n be as defined above. Then,

$$[\![\varphi[?_1,\ldots,?_n]]\!](s) \quad = \quad [\![T^n(\varphi[?_1,\ldots,?_n])]\!](s)$$

Proof:

The proof is similar to the proof of Theorem 6. □

B. Deciding Negative Queries

We now show how non-positive queries can be solved by transforming them into positive queries, query checking, and then post-processing the result.

1) Negative Queries with Single Placeholder: We start by considering a query with a single negated placeholder: $\varphi[?_1]$. We can transform it into a positive query by removing all negations appearing in front of $?_1$, denoting this transformation by N. For example, if $\varphi[?_1] = EX\neg?_1$, then $N(\varphi[?_1]) = EX?_1$. The following proposition formalizes the connection between negative and positive queries.

Proposition 3: Let $\varphi[?_1]$ be a negative query. Then

$$N(\varphi)[\neg?_1] \quad = \quad \varphi[?_1]$$

Proof:

The proofs of this and other results shown later in this section are omitted due to space limitations. □

Thus, we can find a solution to $\varphi[?_1]$ by first solving the positive query $N(\varphi[?_1])$ and then negating the resulting substitutions:

Theorem 9: Let $\varphi[?_1]$ be a negative query. Then,

$$[\![\varphi[?_1]]\!](s) \quad = \quad \{\neg\ell \mid \ell \in [\![N(\varphi)[?_1]]\!](s)\}$$

Alternatively, query-checking for negative queries can be reduced directly to multi-valued model-checking over the lattice $D(PF(?_1))$ — the lattice of downward-closed sets over $PF(?_1)$. In this case, we first transform the negative query $\varphi[?_1]$ into the corresponding positive query $N(\varphi[?_1])$. Then we apply the transformation T_f from Section VI-A, where $f : PF(?_1) \rightarrow D(PF(?_1))$ defined as $f(j) \triangleq \downarrow\neg j$. Combining the two transformations yields a transformation $T^- = T_f \circ N$. For example, consider the query $\varphi[?_1] = \neg?_1\{p\}$ in a state s_0 of a Kripke structure in Figure 1. First, we obtain a positive query: $N(\varphi)[?_1] = ?_1\{p\}$. Next, we apply the transformation T_f and get $T_f(?_1\{p\}) = p \wedge \downarrow\neg p \vee \neg p \wedge \downarrow p$. The solution of this χCTL($\mathcal{L}$) formula in state s_0 of the model is $\downarrow\neg p$, which is also a solution to $[\![\varphi]\!](s_0)$, as required.

Theorem 10: Let $\varphi[?_1]$ be a negative query, s be a state of a Kripke structure, and T^- be as defined above. Then,

$$[\![\varphi[?_1]]\!](s) \quad = \quad [\![T^-(\varphi[?_1])]\!](s)$$

2) Monotone Queries with Multiple Placeholders: The same technique can be applied to queries with multiple negated placeholders. Given a query $\varphi[?_1,\ldots,?_n]$, negative in $?_i$, let N_i be the transformation that removes all negations in front of $?_i$. For example, if $\varphi[?_1,?_2] = \neg?_1 \wedge EX\neg?_2$ then $N_1(\varphi) = ?_1 \wedge EX\neg?_2$, and $N_2(\varphi) = \neg?_1 \wedge EX?_2$.

Proposition 4: Let $\varphi[?_1,\ldots,?_n]$ be a query negative in $?_i$. Then

$$N_i(\varphi)[?_1,\ldots,\neg?_i,\ldots,?_n] \quad = \quad \varphi[?_1,\ldots,?_n]$$

Thus, to find a solution to a query $\varphi[?_1,\ldots,?_n]$ negative in $?_i$, we first solve $N_i(\varphi)$, and then transform its solution as we did in the case of a single placeholder.

Theorem 11: Let $\varphi[?_1,\ldots,?_n]$ be a query with at most n placeholders negative in $?_i$, then

$$\begin{array}{l}[\![\varphi[?_1,\ldots,?_n]]\!](s) = \\ \quad \{(a_1,\ldots,\neg a_i,\ldots,a_n) \mid (a_1,\ldots,a_i,\ldots,a_n) \in [\![N_i(\varphi)]\!](s)\}\end{array}$$

C. Deciding Mixed Queries

To solve a mixed query, we first transform it into a monotone query, apply the query checking algorithm to it, and then post-process the solution.

A mixed query $\varphi[?_1]$ can be transformed into a monotone query $\varphi[?_{1+},?_{1-}]$ that is positive in $?_{1+}$ and negative in $?_{1-}$ by replacing all positive occurrences of $?_1$ with $?_{1+}$, and all negative occurrences by $?_{1-}$. In what follows, we denote this transformation by H. For example, let $\varphi[?_1] = ?_1 \wedge EX\neg?_1$, then $H(\varphi[?_1]) = ?_{1+} \wedge EX\neg?_{1-}$. Note that H does not change the restrictions placed on the placeholders, that is, $PF(?_1) = PF(?_{1+}) = PF(?_{1-})$.

Note that $a \in PF(?_1)$ is a substitution satisfying a mixed query $[\![\varphi[?_1]]\!](s)$ if and only if (a,a) is a substitution satisfying $[\![H(\varphi)[?_{1+},?_{1-}]]\!](s)$. This yields the following theorem.

Theorem 12: Let $\varphi[?_1]$ be a query mixed in $?_1$. Then,

$$[\![\varphi[?_1]]\!](s) \quad = \quad \{a \mid (a,a) \in [\![H(\varphi)[?_{1+},?_{1-}]]\!](s)\}$$

As it is the case with negative queries, the technique for solving a query with a single mixed placeholder can be easily extended to queries with multiple mixed placeholders.

VIII. Running Time

In this section, we study complexity of our symbolic approach to query checking. The direct approach to query checking (take all $2^{2^{|var(?_1)|}}$ possible solutions, substitute them for each placeholder and check results using classical model checking) has the complexity that is double-exponential in $|var(?_1)|$ [3]. Complexity of deciding valid queries is exponential in $|var(?_1)|$ [3]. Yet, despite these bad worst-case complexities, query checking seems to perform well in practice, as illustrated in Table IV).

Symbolic implementation of query checking allows us to analyze its running time at a finer granularity. In particular, in Section VIII-A, we show that the running time of query checking depends on the type of query and on the complexity of performing operations on terminal nodes of decision diagrams. In Section VIII-B, we discuss a few heuristics aimed at improving the running time of query checkers. Section VIII-C gives a few performance results, aimed to illustrate our discussion.

A. Complexity and Symbolic Encoding

In Section VI, we showed that the solution to $\varphi[?_1]$ can be obtained by evaluating a χCTL($\mathcal{L}$) formula $T_\uparrow(\varphi[?_1])$ on K. From Section II-C, model checking of a χCTL($\mathcal{L}$) formula $T_\uparrow(\varphi[?_1])$ requires at most $|S| \times |T_\uparrow(\varphi[?_1])|$ iterations. Since the size of a query is defined to be the number of temporal logic operators occurring in it, and since the transformation $T_\uparrow$ does not introduce any new temporal operators, $|T_\uparrow(\varphi[?_1])| = |\varphi[?_1]|$.

Each iteration of the model checking algorithm requires computing unions, intersections, and quantification operations. As described in Section IV, a query $\varphi[?_1]$ is solved using a symbolic χCTL model checker [9] implemented using Algebraic Decision Diagrams (ADDs). To simplify the analysis, we express the worst case complexity of performing any operation on terminal nodes of an ADD as $CT(n)$, where n stands for the number of propositions restricting the placeholder.

Unions and intersections. The complexity of performing an ADD operation is linear in the size of each the argument, and linear in $CT(n)$ – the complexity of performing $\wedge$ and $\vee$ operations on terminal nodes [31].

Existential quantification. The cost of performing existential quantification using BDDs is exponential in $|A|$. It is easy to show that the size of the *largest* ADD representing a function on n variables is only twice the size of the *largest* BDD on n variables. Thus, the cost of existential quantification using ADDs is also exponential in $|A|$. Moreover, since quantification is implemented as a series of $\vee$ operations, its complexity is also linear in $CT(n)$.

Combining the above arguments, we get:

Lemma 2: The worst case complexity Q of solving a query $\varphi[?_1]$ with a *single* temporal operator on a Kripke structure $K = (S, R, s_0, A, I)$ is linear in $|S|$ and $CT(|var(?_1)|)$, and is exponential in $|A|$.

Theorem 13: The worst case complexity of solving an arbitrary query $\varphi[?_1]$ on a Kripke structure $K = (S, R, s_0, A, I)$ is $O(|\varphi[?_1]| \times Q)$.

To analyze the complexity of queries with multiple placeholders, we note that the lattice $PF(?_1) \times \cdots \times PF(?_n)$ is isomorphic to the lattice of propositional formulas over $\Sigma_{i=1}^{n}|var(?_i)|$ atomic propositions. Thus, the worst case complexity of solving a query with multiple placeholders can be expressed relative to the worst case complexity of solving a query with a single placeholder.

Theorem 14: The worst case complexity of solving a query $\varphi[?_1, \ldots, ?_n]$ with n placeholders on a Kripke structure $K = (S, R, s_0, A, I)$ is linear in $|\varphi[?_1, \ldots, ?_n]|$, $|S|$ and $CT(\Sigma_{i=1}^{n}|var(?_i)|)$, and is exponential in $|A|$.

Note that the worst case complexity of query checking is dominated by $CT(|var(?_1)|)$, which in turn depends on the number of solutions to a query, and in the worst case is double-exponential in $|var(?_1)|$. A possible implementation of a query checker can represent the elements of $U(PF(?_1))$ by their minimal elements [4] so that the complexity of meet and join operations is quadratic in the number of minimal elements in their operands. Note that the set of minimal elements in the solution to a query is exactly the set of maximally strong solutions to it. Thus, a query with a few maximally strong solutions is easier to solve than a query with a large number of maximally strong solutions. Queries about states, such as Chan's *valid* queries [3] and the ones used in our case study in Section V, have at most $2^{|var(?_1)|} \leq |S|$ maximally strong solutions. On the other hand, queries about paths, such as $EG?_1$, may contain up to $O(2^{2^{|var(?_1)|}})$ maximally strong solutions [7] and can therefore be infeasible even for small models. Note that this does not mean that all queries about paths are always infeasible, as illustrated by the query $EFEG?_1\{\texttt{CC}\}$ in the last row of Table IV, but rather that there exists a model and a query with the worst case complexity.

B. Improving Complexity Bounds

We can refine the upper bound on complexity given in Theorem 13 by noticing that usually a query is built by combining a CTL fragment with a fragment containing placeholders. For example, to solve the query $AG(?_1 \Rightarrow AFa)$, we first apply CTL model checking to AFa and then use the result to evaluate the query.

Theorem 15: The worst case complexity of solving a query $\varphi[?_1]$, where the placeholder occurs in the scope of $V \leq |\varphi[?_1]|$ temporal operators is $V \times Q + (|\varphi[?_1]| - V) \times M$, where Q and M are the worst case complexities of query checking and model checking a formula with one temporal operator.

Complexity of queries with multiple placeholders is also reflected by the above theorem.

We also note that simple additional heuristics can help improve the expected time and space complexities of query checking. For example, constructing decision diagrams for $T(?_1)$ is quite expensive. Yet, it can be avoided for most queries. Further, in the majority of operations performed during query checking, one of the operands, namely, the transition relation, is a boolean function. The complexity of performing $\wedge$ and $\vee$ on terminal nodes with at least one boolean operand is $O(1)$ and thus does not depend on CT.

C. Experimental Results

In Section VIII-A, we have shown that for a large class of queries, the worst case complexity of using symbolic χCTL model checking for solving queries is comparable to the worst case complexity of symbolic CTL model checking. We now illustrate these observations by verifying several CTL formulas and related queries on the Cruise Control System, summarizing the results in Table VII.

CTL formulas are checked using χChek with classical logic as the De Morgan algebra. The query in the second row is restricted to three boolean atomic propositions required to encode the enumerated type for CC. Note that the running time of this query is less than double of the running time of the corresponding CTL formula (row 1). A similar picture

	Query	Time
1	$EF(\texttt{CC} = \texttt{Override})$	$0.18s$
2	$EF?_1\{\texttt{CC}\}$	$0.22s$
3	$EF(\texttt{CC} = \texttt{Cruise} \wedge \texttt{Throttle} = \texttt{tAccel})$	$0.17s$
4	$EF(\texttt{CC} = \texttt{Cruise} \wedge ?_1\{\texttt{Throttle}\})$	$0.23s$
5	$EF(\texttt{CC} = \texttt{Cruise} \wedge ?_1\{\texttt{Throttle}, \texttt{Running}, \texttt{Toofast}\})$	$0.37s$
6	$EF(\texttt{CC} = \texttt{Cruise} \wedge ?_1\{\texttt{Throttle}, \texttt{Running}, \texttt{Toofast}, \texttt{Ignition}\})$	$0.41s$
7	$AG(\texttt{CC} = \texttt{Inactive} \Rightarrow \texttt{Running})$	$0.14s$
8	$AG(\texttt{CC} = \texttt{Inactive} \Rightarrow ?_1\{\texttt{Ignition}, \texttt{Running}\})$	$0.17s$
9	$AG(?_1\{\texttt{CC}\} \Rightarrow ?_2\{\texttt{Ignition}, \texttt{Running}\})$	$0.33s$
10	$AG(?_1\{\texttt{CC}\} \Rightarrow ?_2\{\texttt{Ignition}, \texttt{Running}, \texttt{Toofast}\}$	$0.42s$
11	$AG(?_1\{\texttt{CC}\} \Rightarrow ?_2\{\texttt{Ignition}, \texttt{Running}, \texttt{Toofast}, \texttt{Brake}\}$	$0.69s$
12	$EF(\texttt{CC} = \texttt{Cruise} \wedge EX\texttt{CC} = \texttt{Override})$	$0.17s$
13	$EF(\texttt{CC} = \texttt{Cruise} \wedge EX?_1\{\texttt{CC}\})$	$0.19s$
14	$EF(?_1\{\texttt{CC}\} \wedge EX\texttt{CC} = \texttt{Override})$	$0.19s$
15	$EF(?_1\{\texttt{CC}\} \wedge EX?_2\{\texttt{CC}\})$	$0.33s$
16	$EF(?_1\{\texttt{CC}\} \wedge EX?_2\{\texttt{CC}, \texttt{Ignition}\})$	$0.35s$
17	$EF(?_1\{\texttt{CC}\} \wedge EX?_2\{\texttt{CC}, \texttt{Ignition}, \texttt{Running}\})$	$0.36s$
18	$EF(?_1\{\texttt{CC}\} \wedge EX(?_2\{\texttt{CC}\} \wedge ?_3\{\texttt{Ignition}, \texttt{Running}\}))$	$0.36s$
19	$EF(?_1\{\texttt{CC}\} \wedge EX(?_2\{\texttt{CC}\} \wedge EX(?_3\{\texttt{CC}\})))$	$0.43s$

TABLE VII
QUERY CHECKING EXPERIMENTS.

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, 5 and 6 of the table, we see that the observed slowdown is only marginal. Experiments in rows 7–11 and 12–19 paint a similar picture, while also illustrating how the performance of TLQSolver changes as the number of placeholders goes up.

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 can become a feasible tool for a variety of model exploration applications.

IX. CONCLUSION

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

A. Summary and Discussion

In this paper, we have demonstrated the applicability of query checking for model exploration, using the Cruise Control system as a running example. Using query checking for reachability analysis and discovering guards and invariants depends on being able to decide queries with multiple placeholders, while guided simulation and test case generation also require the ability to produce witnesses for temporal logic queries. Further, we are 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.

Recognizing that query checking can be reduced to multi-valued model-checking [8], we provided an algorithm for this encoding and built a tool TLQSolver on top of our existing symbolic multi-valued model-checker χChek. Our implementation allowed us not only to generate solutions to temporal logic queries, but also to provide witnesses explaining answers to existential queries. Symbolic representation also allows us to study complexity of the query-checking problem from the decision diagram point of view, and to suggest a number of optimizations. Our experience applying TLQSolver suggests that it can be used effectively for a variety of software engineering tasks. Yet, we hope that the work reported in this paper will open further venues for engineering query-checking implementations to ensure that they can handle larger problems, so that practical query checking can become a useful tool for model exploration and analysis.

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, the query checker implemented by William Chan [3] does not implement fairness. Further, the presentation in this paper used CTL as our temporal logic. Yet, multi-valued model-checking for other temporal logics, such as LTL and CTL*, have also been studied in literature (e.g., [32]), and thus our framework can be adopted to deciding queries defined over these logics. Finally, the underlying framework of χChek is based on μ-calculus, so we can easily extend our implementation to handle μ-calculus queries. For example, the query $\nu Z.EXEF(Z \wedge ?)$ – which propositional formulas hold infinitely often – can be very useful in discovering fairness conditions of a system under analysis.

B. Future Work

In the future, we plan to implement some of the heuristics suggested in Section VIII and develop further ones. We will also experiment with different decision diagram implementations.

Further case studies are essential for assessing the feasibility of query checking on realistic systems. In addition, success of query checking in practice cannot be achieved without an effective methodology. We intend to use our case studies to guide us in the development of such a methodology.

X. Acknowledgements

We thank the program committee of FSE and anonymous TSE referees for their feedback on earlier versions of this paper. Financial support for this research has been provided by NSERC and CITO.

References

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

[2] 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*, vol. 8, no. 2, pp. 244–263, April 1986.

[3] W. Chan, "Temporal-Logic Queries," in *Proceedings of the 12th Conference on Computer Aided Verification (CAV'00)*, ser. Lecture Notes in Computer Science, vol. 1855. Chicago, IL, USA: Springer, July 2000, pp. 450–463.

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

[5] K. McMillan, *Symbolic Model Checking*. Kluwer Academic, 1993.

[6] O. Kupferman, M. Vardi, and P. Wolper, "An Automata-Theoretic Approach to Branching-Time Model Checking," *Journal of the ACM*, vol. 27, no. 2, pp. 312–360, March 2000.

[7] S. Hornus and P. Schnoebelen, "On Solving Temporal Logic Queries," in *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology (AMAST'2002)*, ser. Lecture Notes in Computer Science, vol. 2422. Springer, 2002, pp. 163–177.

[8] M. Chechik, S. Easterbrook, and V. Petrovykh, "Model-Checking Over Multi-Valued Logics," in *Proceedings of Formal Methods Europe (FME'01)*, ser. Lecture Notes in Computer Science, vol. 2021. Springer, March 2001, CSRG Technical Report 420, pp. 72–98.

[9] 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)*, ser. Lecture Notes in Computer Science, vol. 2404, July 2002.

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

[11] M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel, "Multi-Valued Symbolic Model-Checking," *ACM Transactions on Software Engineering and Methodology*, January 2003, (Accepted for publication.).

[12] A. Gurfinkel and M. Chechik, "Multi-Valued Model-Checking via Classical Model-Checking," in *Proceedings of 14th International Conference on Concurrency Theory (CONCUR'03)*, Marseille, France, September 2003.

[13] A. Gurfinkel, "Multi-Valued Symbolic Model-Checking: Fairness, Counterexamples, Running Time," Master's thesis, University of Toronto, Department of Computer Science, October 2002, available from http://www.cs.toronto.edu/~chechik/pubs/gurfinkelMSThesis.ps.

[14] A. Gurfinkel and M. Chechik, "Proof-like Counterexamples," in *Proceedings of 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, ser. Lecture Notes in Computer Science, vol. 2619, April 2003, pp. 160–175.

[15] F. Somenzi, "CUDD: CU Decision Diagram Package Release," 2001. [Online]. Available: http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html

[16] M. Fujita, P. C. McGeer, and J. C.-Y. Yang, "Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation," *Formal Methods in System Design: An International Journal*, vol. 10, no. 2/3, pp. 149–169, April 1997.

[17] M. Fröhlich and M. Werner, "The Graph Visualization System daVinci — A user interface for applications." Department of Computer Science, Bremen University, Tech. Rep. 5/94, 1994. [Online]. Available: citeseer.nj.nec.com/fr94graph.html

[18] Y. Dong, C. Ramakrishnan, and S. A. Smolka, "Model Checking and Evidence Exploration", in *Proceedings of the 10th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'03)*, Alabama, USA, April 2003, pp. 214–223.

[19] P. Stevens and C. Stirling, "Practical Model-Checking using Games," in *Proceedings of 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, ser. Lecture Notes in Computer Science, B. Steffen, Ed., vol. 1384. New York, NY, USA: Springer-Verlag, 1998, pp. 85–101. [Online]. Available: citeseer.nj.nec.com/stevens98practical.html

[20] J. Kirby, "Example NRL/SCR Software Requirements for an Automobile Cruise Control and Monitoring System," Wang Institute of Graduate Studies, Tech. Rep., October 1988, revision 2.0 by Skip Osborne and Aaron Temin.

[21] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated Consistency Checking of Requirements Specifications," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 3, pp. 231–261, July 1996.

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

[23] 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)*. Orlando, FL: ACM, November 1998, pp. 56–69.

[24] ——, "An Algorithm for Strengthening State Invariants Generated from Requirements Specifications," in *Proceedings of 5th IEEE International Symposium on Requirements Engineering (RE'01)*. Toronto, Canada: IEEE Computer Society, August 2001, pp. 182–191.

[25] J. Atlee, "Automated Analysis of Software Requirements," Ph.D. dissertation, University of Maryland, College Park, Maryland, December 1992.

[26] 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)*. Washington, DC, USA: IEEE Computer Society, April 2001, pp. 83–93.

[27] 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)*. Toulouse, France: ACM Press, September 1999, pp. 146–162.

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

[29] H. Hong, S. Cha, I. Lee, O. Sokolsky, and H. Ural, "Data Flow Testing as Model Checking," in *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, Portland, Oregon, USA, May 2003.

[30] M. Chechik, B. Devereux, S. Easterbrook, A. Lai, and V. Petrovykh, "Efficient Multiple-Valued Model-Checking Using Lattice Representations," in *Proceedings of 12th International Conference on Concurrency Theory (CONCUR'01)*, ser. Lecture Notes in Computer Science, vol. 2154, Aalborg, Denmark, August 2001, pp. 451–465.

[31] I. Wegener, *Branching Programs and Binary Decision Diagrams: Theory and Applications*, ser. Monographs on Discrete Mathematics and Applications. SIAM, 2000.

[32] G. Bruns and P. Godefroid, "Generalized Model Checking: Reasoning about Partial State Spaces," in *Proceedings of 11th International Conference on Concurrency Theory (CONCUR'00)*, ser. Lecture Notes in Computer Science, vol. 1877. Springer, August 2000, pp. 168–182.

Arie Gurfinkel received the B.Sc. and M.Sc. degrees in computer science from the University of Toronto, in 2000 and 2003, respectively, and is currently working toward his Ph.D. degree. His research interests lie in the intersection of formal methods and software engineering, with an emphasis on automated techniques for program verification.

Marsha Chechik received her Ph.D. from the University of Maryland in 1996 and joined the Computer Science department at the University of Toronto, where she is currently an associate professor. Prof. Chechik's main research interests are in applying formal methods to improve quality of software. She has authored over 35 papers in formal methods, software specification and verification, model-checking and requirements engineering. She is a member of the IEEE Computer Society.

Benet Devereux received his B.Sc. from the University of Western Ontario in 1999 and his M.Sc. from the University of Toronto in 2001, and is currently working toward his Ph.D. His principal research interest is the application of logic to requirements engineering and software engineering,