Finding State Solutions to Temporal Logic Queries

Mihaela Gheorghiu, Arie Gurfinkel, and Marsha Chechik

Department of Computer Science, University of Toronto,
Toronto, ON M5S 3G4, Canada.
Email: {mg,arie,chechik}@cs.toronto.edu

Abstract. Different analysis problems for state-transition models can be uniformly treated as instances of temporal logic query-checking, where only states are sought as solutions to the queries. In this paper, we propose a symbolic query-checking algorithm that finds exactly the state solutions to any query. We show that our approach generalizes previous ad-hoc techniques, and this generality allows us to find new and interesting applications, such as finding stable states. Our algorithm is linear in the size of the state space and in the cost of model checking, and has been implemented on top of the model checker NuSMV, using the latter as a black box. We show the effectiveness of our approach by comparing it, on a gene network example, to the naive algorithm in which all possible state solutions are checked separately.

1 Introduction

In the analysis of state-transition models, many problems reduce to questions of the type: "What are all the states that satisfy a property φ?". Symbolic model checking can answer some of these questions, provided that the property φ can be formulated in an appropriate temporal logic. For example, suppose the erroneous states of a program are characterized by the program counter (pc) being at a line labeled ERROR. Then the states that may lead to error can be discovered by model checking the property $EF\ (\texttt{pc} = \texttt{ERROR})$, formalized in the branching temporal logic CTL [10].

There are many interesting questions which are not readily expressed in temporal logic and require specialized algorithms. One example is finding the reachable states, which is often needed in a pre-analysis step to restrict further analysis only to those states. These states are typically found by computing a forward transitive closure of the transition relation [8]. Another example is the computation of "procedure summaries". A *procedure summary* is a relation between states, representing the input/output behavior of a procedure. The summary answers the question of which inputs lead to which outputs as a result of executing the procedure. They are computed in the form of "summary edges" in the control-flow graphs of programs [21, 2]. Yet another example is the algorithm for finding *dominators/postdominators* in program analysis, proposed in [1]. A state t is a postdominator of a state s if all paths from s eventually reach t, and t is a dominator of s if all paths to s pass through t.

Although these problems are similar, their solutions are quite different. Unifying them into a common framework allows reuse of specific techniques proposed for each problem, and opens a way for creating efficient implementations to other problems of a similar kind. We see all these problems as instances of *model exploration*, where properties of a model are *discovered*, rather than checked. A common framework for model exploration has been proposed under the name of *query checking* [5].

Query checking finds *which* formulas hold in a model. For instance, a query $EF\ ?$ is intended to find all propositional formulas that hold in the reachable states. In general, a CTL query is a CTL formula with a missing propositional subformula, designated by a placeholder ("?"). A *solution* to the query is any propositional formula that, when substituted for the placeholder, makes a CTL formula that holds in the model. The general query checking problem is: given a CTL query on a model, find all of its propositional solutions. For example, consider the model in Figure 1(a), where each state is labeled by the atomic propositions that hold in it. Here, some solutions to $EF\ ?$ are $p \wedge \neg q \wedge r$, representing the reachable state s_0, and $q \vee r$, representing the set of states $\{s_1, s_2\}$. On the other hand, $\neg r$ is not a solution: $EF\ \neg r$ does not hold, since no states where r is false are reachable. Query checking can be solved by repeatedly substituting each possible propositional formula for the placeholder, and returning those for which the resulting CTL formula holds. In the worst case, this approach is exponential in the size of the state space and linear in the cost of CTL model checking.

Each of the analysis questions described above can be formulated as a query. Reachable states are solutions to $EF\ ?$. Procedure summaries can be obtained by solving $EF\ ((\mathtt{pc} = \mathtt{PROC\_END}) \wedge ?)$, where $\mathtt{pc} = \mathtt{PROC\_END}$ holds in the return statement of the procedure. Dominators/postdominators are solutions to the query $AF\ ?$ (*i.e.*, what propositional formulas eventually hold on all paths). This insight gives us a uniform formulation of these problems and allows for easy creation of solutions to other, similar, problems. For example, a problem reported in genetics research [4, 12] called for finding *stable states* of a model, that are those states which, once reached, are never left by the system. This is easily formulated as $EFAG\ ?$, meaning "what are the reachable states in which the system will remain forever?".

These analysis problems further require that solutions to their queries be states of the model. For example, a query $AF\ ?$ on the model in Figure 1(a) has solutions $p \wedge \neg q \wedge r$ and $q \wedge r$. The first corresponds to the state s_0 and is a state solution. The second corresponds to a set of states $\{s_1, s_2\}$ but neither s_1 nor s_2 is a solution by itself. When only state solutions are needed, we can formulate a restricted *state query-checking* problem by constraining the solutions to be single states, rather than arbitrary propositional formulas (that represent *sets* of states). A naive state query checking algorithm is to repeatedly substitute each state of the model for the placeholder, and return those for which the resulting CTL formula holds. This approach is linear in the size of the state space and in the cost of CTL model checking. While of significantly more efficient than general query checking, this approach is not "fully" symbolic, since it requires many runs of a model-checker.

While several approaches have been proposed to solve general query checking, none are effective for solving the state query-checking problem. The original algorithm of Chan [5] was very efficient (same cost as CTL model checking), but was restricted to *valid* queries, *i.e.*, queries whose solutions can be characterized by a single propositional formula. This is too restrictive for our purposes. For example, neither of the queries $EF\ ?$, $AF\ ?$, nor the stable states query $EF\ AG\ ?$ are valid. Bruns and Godefroid [3] generalized query checking to all CTL queries by proposing an automata-based CTL model checking algorithm over a lattice of sets of all possible solutions. This algorithm is exponential in the size of the state space. Gurfinkel and Chechik [15] have also provided a symbolic algorithm for general query checking. The algorithm is based on reducing query checking to multi-valued model checking and is implemented in a tool TLQSolver [7]. While empirically faster than the corresponding naive approach of substituting every propositional formula for the placeholder, this algorithm still has the same worst-case complexity as that in [3], and remains applicable only to modest-sized query-checking problems. An algorithm proposed by Hornus and Schnoebelen [17] finds solutions to any query, one by one, with increasing complexity: a first solution is found in time linear in the size of the state space, a second, in quadratic time, and so on. However, since the search for solutions is not controlled by their shape, finding all state solutions can still take exponential time. Other query-checking work is not directly applicable to our state query-checking problem, as it is exclusively concerned either with syntactic characterizations of queries, or with extensions, rather than restrictions, of query checking [23, 25].

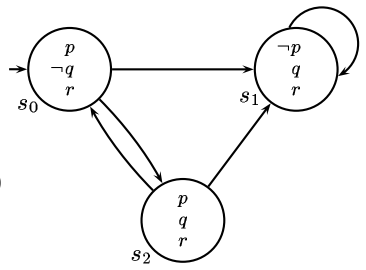
In this paper, we provide a symbolic algorithm for solving the state query-checking problem, and describe an implementation using the state-of-the-art model-checker NuSMV [8]. The algorithm is formulated as model checking over a lattice of sets of states, but its implementation is done by modifying only the interface of NuSMV. Manipulation of the lattice sets is done directly by NuSMV. While the running time of this approach is the same as in the corresponding naive approach, we show empirical evidence that our implementation can perform better than the naive, using a case study from genetics [12].

The algorithms proposed for the program analysis problems described above are special cases of ours, that solve only EF ? and AF ? queries, whereas our algorithm solves any CTL query. We prove our algorithm correct by showing that it *approximates* general query checking, in the sense that it computes exactly those solutions, among all given by general query checking, that are states. We also generalize our results to an approximation framework that can potentially apply to other extensions of model checking, *e.g.*, vacuity detection, and point to further applications of our technique, *e.g.*, to querying XML documents.

There is a also a very close connection between query-checking and sanity checks such as vacuity and coverage [19]. Both problems require checking several "mutants" of the property to obtain the final solution. In fact, the algorithm for solving state-queries presented in this paper bears many similarities to the coverage algorithms described in [19]. Since query-checking is a more general approach, we believe it can provide a uniform framework for studying all these problems.

The rest of the paper is organized as follows. Section 2 provides the model checking background. Section 3 describes the general query-checking algorithm. We formally define the state query-checking problem and describe our implementation in Section 4. Section 5 presents the general approximation technique for model checking over lattices of sets. We present our case study in Section 6, and conclude in Section 7.

(a)



(b)

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

Fig. 1. (a) A simple Kripke structure; (b) CTL semantics.

2 Background

In this section, we review some notions of lattice theory, minterms, CTL model checking, and multi-valued model checking.

Lattice theory. A *finite lattice* is a pair $(\mathcal{L}, \sqsubseteq)$, where $\mathcal{L}$ is a finite set and $\sqsubseteq$ is a partial order on $\mathcal{L}$, such that every finite subset $B \subseteq \mathcal{L}$ has a least upper bound (called *join* and written $\sqcup B$) and a greatest lower bound (called *meet* and written $\sqcap B$). Since the lattice is finite, there exist $\top = \sqcup\mathcal{L}$ and $\bot = \sqcap\mathcal{L}$, that are the maximum and respectively minimum elements in the lattice. When the ordering $\sqsubseteq$ is clear from the context, we simply refer to the lattice as $\mathcal{L}$. A lattice if *distributive* if meet and join distribute over each other. In this paper, we work with lattices of propositional formulas. For a set of atomic propositions P, let $\mathcal{F}(P)$ be the set of propositional formulas over P. For example, $\mathcal{F}(\{p\}) = \{\mathsf{true}, \mathsf{false}, p, \neg p\}$. This set forms a finite lattice ordered by implication (see Figure 2(a)). Since $p \Rightarrow \mathsf{true}$, p is under true in this lattice. Meet and join in this lattice correspond to logical operators $\wedge$ and $\vee$, respectively.

A subset $B \subseteq \mathcal{L}$ is called *upward closed* or an *upset*, if for any $a, b \in \mathcal{L}$, if $b \in B$ and $b \sqsubseteq a$, then $a \in B$. In that case, B can be identified by the set M of its minimal elements ($b \in B$ is minimal if $\forall a \in B,\ \neg(a \sqsubseteq b)$), and we write $B = {\uparrow}M$. For example, for the lattice $(\mathcal{F}(\{p\}), \Rightarrow)$ shown in Figure 2(a), ${\uparrow}\{p, \neg p\} = \{p, \neg p, \mathsf{true}\}$. The set $\{p, \neg p\}$ is not an upset, whereas $\{p, \neg p, \mathsf{true}\}$ is. For singletons, we write ${\uparrow}a$ for ${\uparrow}\{a\}$. We write $\mathcal{U}(\mathcal{L})$ for the set of all upsets of $\mathcal{L}$, *i.e.*, $A \subseteq \mathcal{L}$ iff ${\uparrow}A \in \mathcal{U}(\mathcal{L})$. $\mathcal{U}(\mathcal{L})$ is closed under union and intersection, and therefore forms a lattice ordered by set inclusion. We call $(\mathcal{U}(\mathcal{L}), \subseteq)$ the *upset lattice* of $\mathcal{L}$. The upset lattice of $\mathcal{F}(\{p\})$ is shown in Figure 2(b).

An element j in a lattice $\mathcal{L}$ is *join-irreducible* if $j \neq \bot$ and j cannot be decomposed as the join of other lattice elements, *i.e.*, for any x and y in $\mathcal{L}$, $j = x \sqcup y$ implies $j = x$ or $j = y$ [11]. For example, the join-irreducible elements of the lattice in Figure 2(a) are p and $\neg p$, and of the one in Figure 2(b) — ${\uparrow}\mathsf{true}$, ${\uparrow}p$, ${\uparrow}\neg p$, and ${\uparrow}\mathsf{false}$.

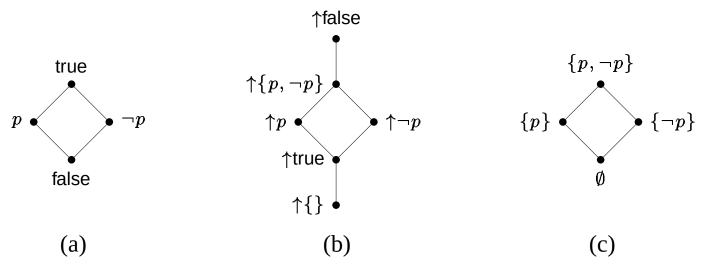


Fig. 2. Lattices for $P = \{p\}$: (a) $(\mathcal{F}(P), \Rightarrow)$; (b) $(\mathcal{U}(\mathcal{F}(P)), \subseteq)$; (c) $(2^{\mathcal{M}(P)}, \subseteq)$.

Minterms. In the lattice of propositional formulas $\mathcal{F}(P)$, a join-irreducible element is a conjunction in which every atomic proposition of P appears, positive or negated. Such conjunctions are called *minterms* and we denote their set by $\mathcal{M}(P)$. For example,

$$\mathcal{M}(\{p, q\}) = \{p \wedge q, p \wedge \neg q, \neg p \wedge q, \neg p \wedge \neg q\}.$$

CTL Model Checking. CTL model checking is an automatic technique for verifying temporal properties of systems expressed in a propositional branching-time temporal logic called *Computation Tree Logic* (CTL) [9]. A system model is a *Kripke structure* $\mathcal{K} = (S, R, s_0, A, I)$, where S is a set of states, $R \subseteq S \times S$ is a (left-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 that are true in each state. CTL formulas are evaluated in the states of $\mathcal{K}$. Their semantics can be described in terms of infinite execution paths of the model. For instance, a formula $AG\ \varphi$ holds in a state s if φ holds in every state, on every infinite execution path $s, s_1, s_2, \ldots$ starting at s; $AF\ \varphi$ ($EF\ \varphi$) holds in s if φ holds in some state, on every (some) infinite execution path $s, s_1, s_2, \ldots$. The formal semantics of CTL is given in Figure 1(b). Without loss of generality we consider only CTL formulas in *negation normal form*, where negation is applied only to atomic propositions [9]. In Figure 1(b), the function $[\![\varphi]\!] : S \to \{\mathsf{true}, \mathsf{false}\}$ indicates the result of checking a formula φ in state s; the set of successors for a state s is $R(s) \triangleq \{s' | (s, s') \in R\}$; $\mu Z.f(Z)$ and $\nu Z.f(Z)$ are least and greatest fixpoints of f, respectively, where $\mu Z.f(Z) = \bigvee_{i>0} f^i(\mathsf{false})$ and $\nu Z.f(Z) = \bigwedge_{i>0} f^i(\mathsf{true})$. Other temporal operators are derived from the given ones, for example: $EF\ \varphi = E[\mathsf{true}\ U\ \varphi]$, $AF\ \varphi = A[\mathsf{true}\ U\ \varphi]$. The operators in pairs $(AX, EX), (AG, EF), (AF, EG), \ldots$ are duals of each other.

A formula φ holds in a Kripke structure $\mathcal{K}$, written $\mathcal{K} \models \varphi$, if it holds in the initial state, *i.e.*, $[\![\varphi]\!](s_0) = \mathsf{true}$. For example, on the model in Figure 1(a), where $A = \{p, q, r\}$, properties $AG\ (p \vee q)$ and $AF\ q$ are true, whereas $AX\ p$ is not. The complexity of model-checking a CTL formula φ on a Kripke structure $\mathcal{K}$ is $O(|\mathcal{K}| \times |\varphi|)$, where $|\mathcal{K}| = |S| + |R|$.

Multi-valued model checking. *Multi-valued* CTL model checking [6] is a generalization of model checking 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 is an involution ($\neg\neg\ell = \ell$) and satisfies De Morgan laws. Conjunction and disjunction are the meet and join operations of $(\mathcal{L}, \sqsubseteq)$, respectively. When the ordering and the negation operation of an algebra $(\mathcal{L}, \sqsubseteq, \neg)$ are clear from the context, we refer to it as $\mathcal{L}$. In this paper, we only use a version of multi-valued model checking where the model remains classical, *i.e.*, both the transition relation and the atomic propositions are two-valued, but properties are specified in a multi-valued extension of CTL over a given De Morgan algebra $\mathcal{L}$, called χCTL($\mathcal{L}$). The logic χCTL($\mathcal{L}$) has the same syntax as CTL, except that the allowed constants are all $\ell \in \mathcal{L}$. Boolean values true and false are replaced by the $\top$ and $\bot$ of $\mathcal{L}$, respectively. The semantics of χCTL($\mathcal{L}$) is the same as of CTL, except $[\![\varphi]\!]$ is extended to $[\![\varphi]\!] : S \to \mathcal{L}$ and the interpretation of constants is: for all $\ell \in \mathcal{L}$, $[\![\ell]\!](s) \triangleq \ell$. The other operations are defined as their CTL counterparts (see Figure 1(b)), where $\vee$ and $\wedge$ are interpreted as lattice operators $\sqcup$ and $\sqcap$, respectively. The complexity of model checking a χCTL($\mathcal{L}$) formula φ on a Kripke structure $\mathcal{K}$ is still $O(|\mathcal{K}| \times |\varphi|)$, *provided that* meet, join, and quantification can be computed in constant time [6], which depends on the lattice.

3 Query Checking

In this section, we review the query-checking problem and a symbolic method for solving it.

Background. Let $\mathcal{K}$ be a Kripke structure with a set A of atomic propositions. A CTL query, denoted by $\varphi[?]$, is a CTL formula containing a *placeholder* "?" for a propositional subformula (over the atomic propositions in A). The CTL formula obtained by substituting the placeholder in $\varphi[?]$ by a formula $\alpha \in \mathcal{F}(A)$ is denoted by $\varphi[\alpha]$. A formula α is a *solution* to a query if its substitution into the query results in a CTL formula that holds on $\mathcal{K}$, *i.e.*, if $\mathcal{K} \models \varphi[\alpha]$. For example, $p \wedge \neg q \wedge r$ and $q \vee r$ are among the solutions to the query AF ? on the model of Figure 1(a), whereas $\neg r$ is not.

In this paper, we consider queries in *negation normal form* where negation is applied only to the atomic propositions, or to the placeholder. We further restrict our attention to queries with a single placeholder, although perhaps with multiple occurrences. For a query $\varphi[?]$, a substitution $\varphi[\alpha]$ means that all occurrences of the placeholder are replaced by α. For example, if $\varphi[?] = EF\ (? \wedge AX\ ?)$, then $\varphi[p \vee q] = EF\ ((p \vee q) \wedge AX\ (p \vee q))$. We assume that occurrences of the placeholder are either non-negated everywhere, or negated everywhere, *i.e.*, the query is either *positive* or *negative*, respectively. Here, we limit our presentation to positive queries; see Section 5 for the treatment of negative queries.

The general CTL query-checking problem is: given a CTL query on a model, find all its propositional solutions. For instance, the answer to the query AF ? on the model in Figure 1(a) is the set consisting of $p \wedge \neg q \wedge r$, $q \wedge r$ and every other formula implied by these, including p, $q \vee r$, and true. If α is a solution to a query, then any β such that $\alpha \Rightarrow \beta$ (*i.e.*, any weaker β) is also a solution, due to the monotonicity of positive queries [5]. Thus, the set of all possible solutions is an upset; it is sufficient for the query-checker to output the strongest solutions, since the rest can be inferred from them.

One can restrict a query to a subset $P \subseteq A$ [3]. We then denote the query by $\varphi[?P]$, and its solutions become formulas in $\mathcal{F}(P)$. For instance, checking $AF\ ?\{p, q\}$ on the model of Figure 1(a) should result in $p \wedge \neg q$ and q as the strongest solutions, together with all those implied by them. We write $\varphi[?]$ for $\varphi[?A]$.

If P consists of n atomic propositions, there are 2^{2^n} possible distinct solutions to $\varphi[?P]$. A "naive" method for finding all solutions would model check $\varphi[\alpha]$ for every possible propositional formula α over P, and collect all those α's for which $\varphi[\alpha]$ holds in the model. The complexity of this naive approach is 2^{2^n} times that of usual model-checking.

Symbolic Algorithm. A symbolic algorithm for solving the general query-checking problem was described in [15] and has been implemented in the TLQSolver tool [7]. We review this approach below.

Since an answer to $\varphi[?P]$ is an upset, the upset lattice $\mathcal{U}(\mathcal{F}(P))$ is the space of all possible answers [3]. For instance, the lattice for $AF\ ?\{p\}$ is shown in Figure 2(b). In the model in Figure 1(a), the answer to this query is $\{p, \mathsf{true}\}$, encoded as $\uparrow\{p\}$, since p is the strongest solution.

Symbolic query checking is implemented by model checking over the upset lattice. The algorithm is based on a state semantics of the placeholder. Suppose query $?\{p\}$ is evaluated in a state s. Either p holds in s, in which case the answer to the query should be $\uparrow p$, or $\neg p$ holds, in which case the answer is $\uparrow\neg p$. Thus we have:

$$[\![?\{p\}]\!](s) = \begin{cases} \uparrow p & \text{if } p \in I(s), \\ \uparrow\neg p & \text{if } p \notin I(s). \end{cases}$$

This case analysis can be logically encoded by the formula $(p \wedge \uparrow p) \vee (\neg p \vee \uparrow\neg p)$.

Let us now consider a general query $?P$ in a state s (where ? ranges over a set of atomic propositions P). We note that the case analysis corresponding to the one above can be given in terms of minterms. Minterms are the strongest formulas that may hold in a state; they also are mutually exclusive and complete — exactly one minterm j holds in any state s, and then $\uparrow j$ is the answer to $?P$ at s. This semantics is encoded in the following translation of the placeholder:

$$\mathcal{T}(?P) = \bigvee_{j \in \mathcal{M}(P)} (j \wedge \uparrow j).$$

The symbolic algorithm is defined as follows: given a query $\varphi[?P]$, first obtain $\varphi[\mathcal{T}(?P)]$, which is a χCTL formula (over the lattice $\mathcal{U}(\mathcal{F}(P))$), and then model check this formula. The semantics of the formula is given by a function from S to $\mathcal{U}(\mathcal{F}(P))$, as described in Section 2. Thus model checking this formula results in a value from $\mathcal{U}(\mathcal{F}(P))$. That value was shown in [15] to represent all propositional solutions to $\varphi[?P]$. For example, the query AF ? on the model of Figure 1(a) becomes

$$\begin{aligned} AF\ (& (p \wedge q \wedge r \wedge \uparrow(p \wedge q \wedge r)) \vee \\ & (p \wedge q \wedge \neg r \wedge \uparrow(p \wedge q \wedge \neg r)) \vee \\ & (p \wedge \neg q \wedge r \wedge \uparrow(p \wedge \neg q \wedge r)) \vee \\ & (p \wedge \neg q \wedge \neg r \wedge \uparrow(p \wedge \neg q \wedge \neg r)) \vee \\ & \ldots). \end{aligned}$$

The result of model-checking this formula is $\uparrow\{p \wedge \neg q \wedge r, q \wedge r\}$.

The complexity of this algorithm is the same as in the naive approach. In practice, however, TLQSolver was shown to perform better than the naive algorithm [15, 7].

4 State Solutions to Queries

Let $\mathcal{K}$ be a Kripke structure with a set A of atomic propositions. In general query checking, solutions to queries are arbitrary propositional formulas. On the other hand, in *state query checking,* solutions are restricted to be single states. To represent a single state, a propositional formula needs to be a minterm over A. In *symbolic* model checking, any state s of $\mathcal{K}$ is uniquely represented by the minterm that holds in s. For example, in the model of Figure 1(a), state s_0 is represented by $p \wedge \neg q \wedge r$, s_2 by $p \wedge q \wedge r$, etc. Thus, for state query checking, an answer to a query is a set of minterms, rather than an upset of propositional formulas. For instance, for the query $AF\ ?$, on the model of Figure 1(a), the state query-checking answer is $\{p \wedge \neg q \wedge r\}$, whereas the general query checking one is $\uparrow\{r \wedge q, p \wedge \neg q \wedge r\}$. While it is still true that if j is a solution, everything in $\uparrow j$ is also a solution, we no longer view answers as upsets, since we are interested only in minterms, and j is the only minterm in the set $\uparrow j$ (minterms are incomparable by implication). We can thus formulate state query checking as *minterm query checking*: given a CTL query on a model, find all its minterm solutions. We show how to solve this for any query $\varphi[?P]$, and any subset $P \subseteq A$. When $P = A$, the minterms obtained are the state solutions.

Given a query $\varphi[?P]$, a naive algorithm would model check $\varphi[\alpha]$ for every minterm α. If n is the number of atomic propositions in P, there are 2^n possible minterms, and this algorithm has complexity 2^n times that of model-checking. Minterm query checking is thus much easier to solve than general query checking.

Of course, any algorithm solving general query checking, such as the symbolic approach described in Section 3, solves minterm query checking as well: from all solutions, we can extract only those which are minterms. This approach, however, is much more expensive than needed. Below, we propose a method that is tailored to solve just minterm query checking, while remaining symbolic.

4.1 Solving minterm query checking

Since an answer to minterm query checking is a set of minterms, the space of all answers is the powerset $2^{\mathcal{M}(P)}$ that forms a lattice ordered by set inclusion. For example, the lattice $2^{\mathcal{M}(\{p\})}$ is shown in Figure 2(c). Our symbolic algorithm evaluates queries over this lattice. We first adjust the semantics of the placeholder to minterms. Suppose we evaluate $?\{p\}$ in a state s. Either p holds in s, and then the answer should be $\{p\}$, or $\neg p$ holds, and then the answer is $\{\neg p\}$. Thus, we have

$$[\![?\{p\}]\!](s) = \begin{cases} \{p\} & \text{if } p \in I(s), \\ \{\neg p\} & \text{if } p \notin I(s). \end{cases}$$

This is encoded by the formula $(p \wedge \{p\}) \vee (\neg p \wedge \{\neg p\})$. In general, for a query $?P$, exactly one minterm j holds in s, and in that case $\{j\}$ is the answer to the query. This gives the following translation of placeholder:

$$\mathcal{A}_m(?P) \triangleq \bigvee\nolimits_{j \in \mathcal{M}(P)} (j \wedge \{j\}).$$

Our minterm query-checking algorithm is now defined as follows: given a query $\varphi[?P]$ on a model $\mathcal{K}$, compute $\varphi[\mathcal{A}_m(?P)]$, and then model check this over $2^{\mathcal{M}(P)}$.

For example, for AF ?, on the model of Figure 1(a), we model check

$$\begin{aligned} AF\ (&(p \wedge q \wedge r \wedge \{p \wedge q \wedge r\})\vee \\ &(p \wedge q \wedge \neg r \wedge \{p \wedge q \wedge \neg r\})\vee \\ &(p \wedge \neg q \wedge r \wedge \{p \wedge \neg q \wedge r\})\vee \\ &(p \wedge \neg q \wedge \neg r \wedge \{p \wedge \neg q \wedge \neg r\})\vee \\ &\ldots), \end{aligned}$$

and obtain the answer $\{p \wedge \neg q \wedge r\}$, that is indeed the only minterm solution for this model.

To prove our algorithm correct, we need to show that its answer is the set of all minterm solutions. We prove this claim by relating our algorithm to the general algorithm in Section 3. We show that, while the general algorithm computes the set $B \in \mathcal{U}(\mathcal{F}(P))$ of all solutions, ours results in the subset $M \subseteq B$ that consists of only the minterms from B. We first establish an "approximation" mapping from $\mathcal{U}(\mathcal{F}(P))$ to $2^{\mathcal{M}(P)}$ that, for any upset $B \in \mathcal{U}(\mathcal{F}(P))$, returns the subset $M \subseteq B$ of minterms.

Definition 1 (Minterm approximation). *Let P be a set of atomic propositions.* Minterm approximation $f_m : \mathcal{U}(\mathcal{F}(P)) \rightarrow 2^{\mathcal{M}(P)}$ *is* $f_m(B) \triangleq B \cap \mathcal{M}(P)$, *for any* $B \in \mathcal{U}(\mathcal{F}(P))$.

With this definition, $\mathcal{A}_m(?P)$ is obtained from $\mathcal{T}(?P)$ by replacing $\uparrow j$ with $f_m(\uparrow j) = \{j\}$. The minterm approximation preserves set operations; this can be proven using the fact that any set of propositional formulas can be partitioned into minterms and non-minterms.

Proposition 1. *The minterm approximation $f_m : \mathcal{U}(\mathcal{F}(P)) \rightarrow 2^{\mathcal{M}(P)}$ is a lattice homomorphism, i.e., it preserves the set operations: for any $B, B' \in \mathcal{U}(\mathcal{F}(P))$, $f_m(B) \cup f_m(B') = f_m(B \cup B')$ and $f_m(B) \cap f_m(B') = f_m(B \cap B')$.*

By Proposition 1, and since model checking is performed using only set operations, we can show that the approximation preserves model-checking results. Model checking $\varphi[\mathcal{A}_m(?P)]$ is the minterm approximation of checking $\varphi[\mathcal{T}(?P)]$. In other words, our algorithm results in set of all minterm solutions, which concludes the correctness argument.

Theorem 1 (Correctness of minterm approximation). *For any state s of $\mathcal{K}$,*

$$f_m([\![\varphi[\mathcal{T}(?P)]]\!](s)) = [\![\varphi[\mathcal{A}_m(?P)]]\!](s).$$

In summary, for $P = A$, we have the following correct symbolic state query-checking algorithm : given a query $\varphi[?]$ on a model $\mathcal{K}$, translate it to $\varphi[\mathcal{A}_m(?A)]$, and then model check this over $2^{\mathcal{M}(A)}$.

The worst-case complexity of our algorithm is the same as that of the naive approach. With an efficient encoding of the approximate lattice, however, our approach can outperform the naive one in practice, as we show in Section 6.

4.2 Implementation

Although our minterm query-checking algorithm is defined as model checking over a lattice, we can implement it using a classical symbolic model checker. This in done by encoding the lattice elements in $2^{\mathcal{M}(P)}$ such that lattice operations are already implemented by a symbolic model checker. The key observation is that the lattice $(2^{\mathcal{M}(P)}, \subseteq)$ is isomorphic to the lattice of propositional formulas $(\mathcal{F}(P), \Rightarrow)$. This can be seen, for instance, by comparing the lattices in Figures 2(a) and 2(c). Thus, the elements of $2^{\mathcal{M}(P)}$ can be encoded as propositional formulas, and the operations become propositional disjunction and conjunction. A symbolic model checker, such as NuSMV [8], which we used in our implementation, already has data structures for representing propositional formulas and algorithms to compute their disjunction and conjunction — BDDs [24]. The only modifications we made to NuSMV were parsing the input and reporting the result.

While parsing the queries, we implemented the translation $\mathcal{A}_m$ defined in Section 4.1. In this translation, for every minterm j, we give a propositional encoding to $\{j\}$. We cannot simply use j to encode $\{j\}$. The lattice elements need to be *constants* with respect to the model, and j is not a constant — it is a propositional formula that contains model variables. We can, however, obtain an encoding for $\{j\}$, by renaming j to a similar propositional formula over fresh variables. For instance, we encode $\{p \wedge \neg q \wedge r\}$ as $x \wedge \neg y \wedge z$. Thus, our query translation results in a CTL formula with double the number of propositional variables compared to the model. For example, the translation of $AF\ ?\{p, q\}$ is

$$\begin{aligned} AF\ (&(p \wedge q \wedge x \wedge y)\vee \\ &(p \wedge \neg q \wedge x \wedge \neg y)\vee \\ &(\neg p \wedge q \wedge \neg x \wedge y)\vee \\ &(\neg p \wedge \neg q \wedge \neg x \wedge \neg y)). \end{aligned}$$

We input this formula into NuSMV, and obtain the set of minterm solutions as a propositional formula over the encoding variables $x, y, \ldots$. For $AF\ ?\{p, q\}$, on the model in Figure 1(a), we obtain the result $x \wedge \neg y$, corresponding to the only minterm solution $p \wedge \neg q$.

4.3 Exactness of minterm approximation

In this section, we address the applicability of minterm query checking to general query checking. When the minterm solutions are the strongest solutions to a query, minterm query checking solves the general query-checking problem as well, as all solutions to that query can be inferred from the minterms. In that case, we say that the minterm approximation is *exact*. We would like to identify those CTL queries that admit exact minterm approximations, independently of the model. The following can be proven using the fact that any propositional formula is a disjunction of minterms.

Proposition 2. *A positive query $\varphi[?P]$ has an exact minterm approximation in any model iff $\varphi[?P]$ is distributive over disjunction, i.e., $\varphi[\alpha \vee \beta] = \varphi[\alpha] \vee \varphi[\beta]$.*

An example of a query that admits an exact approximation is $EF\ ?$; its strongest solutions are always minterms, representing the reachable states. In [5], Chan showed that deciding whether a query is distributive over *conjunction* is EXPTIME-complete. We obtain a similar result by duality.

Theorem 2. *Deciding whether a CTL query is distributive over disjunction is EXPTIME-complete.*

Since the decision problem is hard, it would be useful to have a grammar that is guaranteed to generate queries which distribute over disjunction. Chan defined a grammar for queries distributive over conjunction, that was later corrected by Samer and Veith [22]. We can obtain a grammar for queries distributive over disjunction, from the grammar in [22], by duality.

5 Approximations

The efficiency of model checking over a lattice is determined by the size of the lattice. In the case of query checking, by restricting the problem and approximating answers, we have obtained a more manageable lattice. In this section, we show that our minterm approximation is an instance of a more general approximation framework for reasoning over any lattice of sets. Having a more general framework makes it easier to accommodate other approximations that may be needed in query checking. For example, we use it to derive an approximation to *negative* queries. This framework may also apply to other analysis problems that involve model checking over lattices of sets, such as vacuity detection [14].

We first define general approximations that map larger lattices into smaller ones. Let U be any finite set. Its powerset lattice is $(2^U, \subseteq)$. Let $(\mathcal{L}, \subseteq)$ be any sublattice of the powerset lattice, *i.e.*, $\mathcal{L} \subseteq 2^U$.

Definition 2 (Approximation). *A function $f : \mathcal{L} \to 2^U$ is an* approximation *if:*

1. *it satisfies $f(B) \subseteq B$ for any $B \in \mathcal{L}$ (i.e., $f(B)$ is an under-approximation of B), and*
2. *it is a lattice homomorphism, i.e., it respects the lattice operations: $f(B \cap C) = f(B) \cap f(C)$, and $f(B \cup C) = f(B) \cup f(C)$.*

From the definition of f, the image $f(\mathcal{L})$ of $\mathcal{L}$ through f is a sublattice of 2^U, having $f(\top)$ and $f(\bot)$ as its maximum and minimum elements, respectively.

We consider an approximation to be correct if it is preserved by model checking: reasoning over the smaller lattice is the approximation of reasoning over the larger one. Let φ be a χCTL($\mathcal{L}$) formula. We define its translation $\mathcal{A}(\varphi)$ into $f(\mathcal{L})$ to be the χCTL($f(\mathcal{L})$) formula obtained from φ by replacing any constant $B \in \mathcal{L}$ occurring in φ by $f(B)$. The following theorem simply states that the result of model checking $\mathcal{A}(\varphi)$ is the approximation of the result of model checking φ. Its proof follows by structural induction from the semantics of χCTL, and uses the fact that approximations are homomorphisms. [18] proves a similar result, albeit in a somewhat different context.

Theorem 3 (Correctness of approximations). *Let $\mathcal{K}$ be a classical Kripke structure, $\mathcal{L}$ be a De Morgan algebra of sets, f be an approximation function on $\mathcal{L}$, and φ be a χCTL($\mathcal{L}$) formula. Let $\mathcal{A}(\varphi)$ be the translation of φ into $f(\mathcal{L})$. Then for any state s of $\mathcal{K}$,*

$$f([\![\varphi]\!](s)) = [\![\mathcal{A}(\varphi)]\!](s).$$

Theorem 1 is a corollary to Theorem 3. Our minterm approximation satisfies condition (1) of Definition 2, since $f_m(B) = B \cap \mathcal{M}(P) \subseteq B$, and it also satisfies condition (2) by Proposition 1. Thus, f_m is an approximation to which Theorem 3 applies, yielding Theorem 1.

The minterm approximation defined in Section 4.1 was restricted to positive queries. The general approximation framework defined above makes it easy to derive a minterm approximation for negative queries. We denote a negative query by $\varphi[\neg ?P]$. To obtain the minterm solutions to $\varphi[\neg ?P]$, we can check $\varphi[?P]$, that is, ignore the negation and treat the query as positive. For example, to check the negative query $AF\ \neg ?\{p, q\}$, we check $AF\ ?\{p, q\}$ instead. The minterm solutions to the original negative query are the duals of the *maxterm* solutions to $\varphi[?P]$. A maxterm is a *disjunction* where all the atomic propositions are, positive or negated. We denote by $\mathcal{X}(P)$ the set of maxterms over a set P of atomic propositions. For example, $\mathcal{X}(\{p, q\}) = \{p \vee q, p \vee \neg q, \neg p \vee q, \neg p \vee \neg q\}$. A minterm j is a solution to $\varphi[\neg ?P]$ iff its negation $\neg j$ is a maxterm solution to $\varphi[?P]$. We thus need to define a *maxterm approximation* $f_x : \mathcal{U}(\mathcal{F}(P)) \rightarrow 2^{\mathcal{X}(P)}$ for positive queries. We define f_x such that, for any upset B, it returns the subset of maxterms in that set, *i.e.*, $f_x(B) = B \cap \mathcal{X}(P)$. According to Definition 2, f_x is an approximation: (1) holds by f_x's definition, and (1) follows from the fact that any set of propositional formulas can be partitioned into maxterms and non-maxterms. We define the translation:

$$\mathcal{A}_x(?P) \triangleq \bigvee_{j \in \mathcal{M}(P)} (j \wedge f_x(\uparrow j)).$$

Then, by Theorem 3, model-checking $\varphi[\mathcal{A}_x(?P)]$ results in all the maxterm solutions to $\varphi[?P]$. By negating every resulting maxterm, we obtain all minterm solutions to $\varphi[\neg ?P]$. For example, maxterm solutions to $AF\ ?\{p, q\}$ for the model of Figure 1(a) is the set $\mathcal{X}(\{p, q\})$; thus, the minterm solutions to $AF\ \neg ?\{p, q\}$ are the entire set $\mathcal{M}(\{p, q\})$.

In summary, we have shown that minterm approximations can be generalized to an approximation framework over any lattices of sets, which is applicable, for instance, to finding minterm solutions to negative queries.

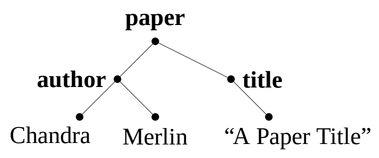
6 Case Study

In this section, we study the problem of finding stable states of a model, and evaluate the performance of our implementation by comparing it to the naive approach to state query checking.

In a study published in plant research, a model of gene interaction has been proposed to compute the "stable states" of a system of genes [12]. This work defined stable

			Algorithms	
	Model	**Query**	**Ours**	**Naive**
1	original	$EF\ AG\ ?$	117	145
2	mutant 1	$EF\ AG\ ?$	116	144
3	mutant 2	$EF\ AG\ ?$	117	145
4	mutant 3	$EF\ AG\ ?$	117	146
5	original	$AG\ ?$	116	145
6	original	$EF\ ?$	118	146
7	original	$AF\ ?$	117	145

(a)



(b)

Fig. 3. (a) Experimental results; (b) An XML example (adapted from [13]).

states as reachable gene configurations that no longer change, and used discrete dynamical systems to find such states. A different publication, [4], advocated the use of Kripke structures as appropriate models of biological systems, where model checking can answer some of the relevant questions about their behaviour. [4] also noted that query-checking might be useful as well, but did not report any applications of this technique. Motivated by [4], we repeated the study of [12] using our state query-checking approach.

The model of [12] consists of 15 genes, each with a "level of expression" that is either boolean (0 or 1), or ternary (0,1, or 2). The laws of interaction among genes have been established experimentally and are presented as logical tables. The model was translated into a NuSMV model with 15 variables, one per gene, of which 8 are boolean and the rest are ternary, turning the laws into NuSMV next-state relations. The model has 559,872 states and was submitted for review separately with our paper.

The problem of finding all stable states of the model and the initial states leading to them is formulated as the minterm query checking of $EFAG?$, where $?$ ranges over all variables. Performance of our symbolic algorithm (Section 4) and the naive state query-checking algorithm for this query is summarized in the top row of the table in Figure 3(a), where the times are reported in minutes. Our algorithm was implemented using NuSMV as described in Section 4.2. The naive algorithm was also implemented using NuSMV by generating all possible minterms over the model variables, replacing each for the placeholder in $EFAG?$ and calling NuSMV to check the resulting formulas. Both algorithms were run on a Pentium 4 processor with 2.8GHz and 1 GB of RAM. Our algorithm gave an answer in under two hours, being about 20% faster than the naive.

To have a larger basis of comparison between the two algorithms, we varied the model (see rows 2-4), and the checked queries (see rows 5-7). Each "mutant" was obtained by permanently switching a different gene off, as indicated in [12]. The performance gain of our algorithm remained unchanged.

Discussion. Performance improvements observed in our case study may not be attainable for every model. If the model is sufficiently small, our algorithm is likely to be faster. As models grow, however, the naive algorithm, which uses fewer BDD variables, will be more scalable. For more challenging models, a combination of the two approaches may yield the best results.

A potentially more scalable alternative can be obtained by an iterative approach. Suppose we are interested in checking a query $AF\ ?$ with two propositions, a and b. We first check $AF\ ?\{a\}$ and $AF\ ?\{b\}$. If no value is found for a proposition, then the query has no minterm solutions. Otherwise, the results correspond to the values each proposition has in all minterm solutions. For example, suppose we obtain $a = \mathsf{false}$, whereas b can be either true or false. We proceed by checking a query for each pair of propositions, using for the placeholder replacement only those values found in the previous step. For example, we check $AF?\{a, b\}$, replacing $?$ by $(\neg a \wedge b \wedge \{\neg a \wedge b\}) \vee (\neg a \wedge \neg b \wedge \{\neg a \wedge \neg b\})$. We continue with checking triples of propositions using the valued obtained for pairs, and so on, until the query is checked on all atomic propositions, or it has been established that no answer exists. In this iterative process, there is place for heuristics that would switch between checking queries by our alogrithm or the naive one, based on the resources available (time vs. memory). We plan to evaluate this approach in future work.

7 Conclusions

We have identified and formalized the state query-checking problem, which is of practical interest and can be solved more efficiently than general query checking. We have presented a symbolic algorithm that solves this problem, described a simple implementation using the NuSMV model checker, and showed its effectiveness on a realistic case study. We proved our algorithm correct by introducing the notion of approximation, which we have extended to reasoning over any lattice of sets. Our state query-checking algorithm generalizes techniques previously proposed for computing procedure summaries [2] and postdominators [1]. In essence, we generalized these algorithms, specialized for $EF\ ?$ and $AF\ ?$ queries, respectively, to arbitrary CTL queries. Our algorithm solves general state-based queries by computing fixpoints over *pre*-image computations, *i.e.*, iterating over EX and AX. While some of these queries can be solved by fixpoints over *post*-image computations, such as the query $EF\ ?$ for discovering the reachable states, not every state-based CTL query can be solved that way, and this impossibility result follows from the work in [16].

We have also presented the application of state query checking to finding stable states in gene networks. In the rest of this section we present another possible application, that we plan to investigate further in the future.

State query checking can be applied to querying XML documents, which are modelled as trees. A simple example, of a fragment from a document containing information about research papers and adapted from [13], is shown in Figure 3(b). An example query is "what are the titles of all papers authored by Chandra?". Viewing tree nodes as states and edges as transitions yields a state-transition model, on which CTL properties can be evaluated [20]. Unfortunately, our example, like many other XML queries, needs to refer to both past and future, and is expressed as a CTL+Past formula as follows [13]:

$$EX^{\text{past}}\ (\text{title} \wedge EX^{\text{past}}\ (\text{paper} \wedge EX\ (\text{author} \wedge EX\ \text{Chandra}))).$$

Such formulas cannot be evaluated without modifying the internals of standard model-checkers. Formulating this question as a query yields

$$\text{paper} \wedge EX\ (\text{title} \wedge EX\ ?) \wedge EX\ (\text{author} \wedge EX\ \text{Chandra}),$$

whose desired solutions are states (here, the node labeled "A Paper Title"), and which avoids the use of the past, and can be solved by our approach, without modifying existing model checkers.

References

1. B. Aminof, T. Ball, and O. Kupferman. "Reasoning About Systems with Transition Fairness". In *Proc. of LPAR'04*, volume 3452 of *LNCS*, pages 194–208, 2005.
2. T. Ball and S. Rajamani. "Bebop: A Symbolic Model Checker for Boolean Programs". In *Proc. of SPIN'00*, volume 1885 of *LNCS*, pages 113–130, 2000.
3. G. Bruns and P. Godefroid. "Temporal Logic Query-Checking". In *Proc. of LICS'01*, pages 409–417, 2001.
4. N. Chabrier-Rivier, M. Chiaverini, V. Danos, F. Fages, and V. Schachter. "Modeling and Querying Biomolecular Interaction Networks". *Theor. Comp. Sci.*, 325(1):25–44, 2004.
5. W. Chan. "Temporal-Logic Queries". In *Proc. of CAV'00*, volume 1855 of *LNCS*, pages 450–463, 2000.
6. M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. "Multi-Valued Symbolic Model-Checking". *ACM Trans. on Soft. Eng. and Methodology*, 12(4):1–38, 2003.
7. M. Chechik and A. Gurfinkel. "TLQSolver: A Temporal Logic Query Checker". In *Proc. of CAV'03*, volume 2725 of *LNCS*, pages 210–214, 2003.
8. A. Cimatti, E.M. Clarke, , E. Giunchilia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. "NuSMV Version 2: An Open Source Tool for Symbolic Model Checking". In *Proceedings of 14th Conference on Computer-Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.
9. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. 1999.
10. E.M. Clarke, E.A. Emerson, and A.P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". *ACM Trans. on Prog. Lang. and Systems*, 8(2):244–263, 1986.
11. B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. 1990.
12. C. Espinosa-Soto, P. Padilla-Longoria, and E. R. Alvarez-Buylla. "A Gene Regulatory Network Model for Cell-Fate Determination during Arabidopsis thaliana Flower Development That Is Robust and Recovers Experimental Gene Expression Profiles". *The Plant Cell*, 16:2923–2939, 2004.
13. G. Gottlob and C. Koch. "Monadic Queries over Tree-Structures Data". In *Proc. of LICS'02*, pages 189–202, 2002.
14. A. Gurfinkel and M. Chechik. "How Vacuous Is Vacuous?". In *Proc. of TACAS'04*, volume 2988 of *LNCS*, pages 451–466, 2004.
15. A. Gurfinkel, M. Chechik, and B. Devereux. "Temporal Logic Query Checking: A Tool for Model Exploration". *IEEE Trans. on Soft. Engineering*, 29(10):898–914, 2003.
16. T. A. Henzinger, O. Kupferman, and S. Qadeer. "From Pre-Historic to Post-Modern Symbolic Model Checking". *Form. Methods Syst. Des.*, 23(3):303–327, 2003.
17. S. Hornus and P. Schnoebelen. "On Solving Temporal Logic Queries". In *Proc. of AMAST'02*, volume 2422 of *LNCS*, pages 163–177, 2002.
18. B. Konikowska and W. Penczek. "Reducing Model Checking from Multi-Valued CTL* to CTL*". In *Proc.of CONCUR'02*, LNCS, 2002.
19. O. Kupferman. "Sanity Checks in Formal Verification". In *Proc. of CONCUR'06*, LNCS, 2006.

20. G. Miklau and D. Suciu. ”Containment and Equivalence for an XPath fragment”. In *Proc. of PODS’02*, pages 65–76, 2002.
21. T. W. Reps, S. Horwitz, and M. Sagiv. “Precise Interprocedural Dataflow Analysis via Graph Reachability”. In *Proc. of POPL’95*, pages 49–61, 1995.
22. M. Samer and H. Veith. “Validity of CTL Queries Revisited”. In *Proc. of CSL’03*, volume 2803 of *LNCS*, pages 470–483, 2003.
23. M. Samer and H. Veith. “Parameterized Vacuity”. In *Proc. of FMCAD’04*, volume 3312 of *LNCS*, pages 322–336, 2004.
24. F. Somenzi. “Binary Decision Diagrams”. In *Calculational System Design*, volume 173 of *NATO Science Series F: Computer and Systems Sciences*, pages 303–366. 1999.
25. D. Zhang and Rance Cleaveland. “Efficient Temporal-Logic Query Checking for Presburger Systems”. In *Proc. of ASE’05*, pages 24–33, 2005.