Seventh International Conference on Formal Methods in Computer-Aided Design

What Triggers a Behavior?

Orna Kupferman and Yoad Lustig
School of Engineering and Computer Science
Hebrew University, Jerusalem, 91904, Israel
Email: {orna,yoadl}@cs.huji.ac.il

***Abstract*—We introduce and study *trigger querying*. Given a model M and a temporal behavior φ, trigger querying is the problem of finding the set of scenarios that trigger φ in M. That is, if a computation of M has a prefix that follows the scenario, then its suffix satisfies φ. Trigger querying enables one to find, for example, given a program with a function f, the scenarios that lead to calling f with some parameter value, or to find, given a hardware design with signal *err*, the scenarios after which the signal *err* ought to be eventually raised.**

We formalize trigger querying using the temporal operator $\mapsto$ (triggers), which is the most useful operator in modern industrial specification languages. A regular expression r triggers an LTL formula φ in a system M, denoted $M \models r \mapsto \varphi$, if for every computation π of M and index $i \geq 0$, if the prefix of π up to position i is a word in the language of r, then the suffix of π from position i satisfies φ. The solution to the trigger query $M \models ? \mapsto \varphi$ is the maximal regular expression that triggers φ in M. Trigger querying is useful for studying systems, and it significantly extends the practicality of traditional query checking [6]. Indeed, in traditional query checking, solutions are restricted to propositional assertions about states of the systems, whereas in our setting the solutions are temporal scenarios.

We show that the solution to a trigger query $M \models ? \mapsto \varphi$ is regular, and can be computed in polynomial space. Unfortunately, the polynomial-space complexity is in the size of M. Consequently, we also study *partial trigger querying*, which returns a (non empty) subset of the solution, and is more feasible. Other extensions we study are *observable trigger querying*, where the partial solution has to refer only to a subset of the atomic propositions, *constrained trigger querying*, where in addition to M and φ, the user provides a regular constraint c and the solution is the set of scenarios respecting c that trigger φ in M, and *relevant trigger querying*, which excludes vacuous triggers — scenarios that are not induced by a prefix of a computation of M. Trigger querying can be viewed as the problem of finding sufficient conditions for a behavior φ in M. We also consider the dual problem, of finding necessary conditions to φ, and show that it can be solved in space complexity that is only logarithmic in M.

I. Introduction

The field of formal verification developed from the need to verify that a system satisfies its specification. Since its conception, the field has enjoyed great progress in the development of practical tools and better understanding of the problems and models related to formal verification. One of the concepts that has emerged in the context of formal verification is that of *model exploration*. The idea, as first noted by Chan in [6], is that, in practice, model checking is often used for understanding the system rather than for verifying its correctness.

Chan suggested to formalize model exploration by means of *query checking*. The input to the query-checking problem is a model M and a query φ, where a query is a temporal-logic formula in which some proposition is replaced by the place-holder "?" (e.g., $AG?$). A solution to the query is a propositional assertion that, when it replaces the place-holder, results in a formula that is satisfied in M. For example, if the query is $AG?$, then the set of solutions include all assertions ψ for which $M \models AG\psi$. A query checker should return the strongest solutions to the query (strongest in the sense that they are not implied by other solutions).[1] The work of Chan was followed by further work on query checking, studying its complexity, cases in which only a single strongest solution exists, the case of multiple (possibly related) place-holders, and more [4], [8], [15], [7].

We believe that model exploration, and in particular query checking, is a very natural and interesting task. Query checking suffers, however, from a serious shortcoming: The result of a query check is a propositional assertion. Thus, query checking is restricted to questions regarding one point in time, whereas most interesting questions about systems involve scenarios that develop over time.

Consider, for example, a programmer trying to understand the code of some computer program. In particular, the programmer is interested in situations in which some function is called with some parameter value. The actual state in which the function is called is by far less interesting than the scenario that has lead to it. Query checking does not enable us to reveal such scenarios.

In this work we introduce and study *trigger querying*, which addresses the shortcoming described above. Given a model M and a temporal behavior φ, trigger querying is the problem of finding the set of scenarios that trigger φ in M. That is, the set of scenarios such that if a computation of M has a prefix that follows a scenario in the set, then its suffix satisfies φ.

We formalize trigger querying using the temporal operator $\mapsto$ (triggers). The trigger operator was introduced in SUGAR (the precursor of PSL [3], called suffix implication there). We use the name trigger suggested in ForSpec [1] as it is more indicative of the operator meaning. System Verilog Assertions (SVA) [17] is another popular industrial specification formalism in which the operator triggers plays an important role. Consider a system M with a set AP of atomic propositions. A word w over the alphabet 2^{AP} triggers an LTL formula φ in

[1]Note that a query may not only have several solutions, but may also have several strongest solutions.

the system M, denoted $M \models w \mapsto \varphi$, if for every computation π of M, if w is a prefix of π, then the suffix of π from position $|w|$ satisfies φ (note that there is an "overlap" and the $|w|$-th letter of π participates both in the prefix w and in the suffix satisfying φ.) The solution to the trigger query $M \models ? \mapsto \varphi$ is the set of words w that trigger φ in M. Since, as we show, the solution is regular, trigger-querying algorithms return the solution by means of a regular expression or an automaton on finite words.

Let us consider an example. Assume that M models a hardware design with a signal *err* that is raised whenever an error occurs. We might be interested in characterizing the scenarios after which the signal *err* is raised. This is, exactly the set of scenarios that trigger *err* — the solution to the trigger query $M \models ? \mapsto$ *err*. It may also be the case that we are really interested in characterizing the scenarios after which *err* aught to be raised. The difference is that now we are interested in "crossing the point of no return"; that is, the point from which *err* would eventually (possibly in the distant future) be raised. The set of such scenarios are the solution to the trigger query $M \models ? \mapsto Ferr$.

Another way to see the importance of the extension of query checking from a propositional to a temporal setting is to go back to the context of model checking. It is widely acknowledged that if a bug is found, it should be reported with a temporal counter example. Indeed, counter examples allow the user to see the bug in context and to understand what has caused the bug and how to fix it. This corresponds to the model explorer need to see full scenarios rather than states. Getting from a query checker the propositional assertions that are the solutions to the query $? \rightarrow Ferr$ (or even to $G(? \rightarrow Ferr)$) is much less informative than getting the full scenarios that lead to *err*. [2]

We solve trigger querying and show that the problem is tight for polynomial space. Unlike LTL model-checking, whose complexity is also polynomial space, here the polynomial-space complexity is not only in the length of the specification but also in the size of the system. Consequently, we consider a more feasible version of trigger querying. The idea is that when the user cannot get a complete characterization of the scenarios triggering a behavior, he may still be interested in getting examples of words triggering the behavior. In *partial trigger querying*, the algorithm returns a subset of the solution to the trigger query (unless the complete solution is empty, the subset should not be empty). The complexity demands of partial trigger querying are indeed lower than these of trigger querying. Specifically, the complexity in the system is nondeterministic polynomial time rather than polynomial space. Beyond the lower complexity, partial trigger querying can be implemented symbolically, and we describe BDD-based and SAT-based algorithms for solving trigger querying.

In addition to trigger querying as presented above, we introduce and study several natural variants of the problem. First, suppose that a finite word w cannot be generated by the system M (i.e., it is not a prefix of a computation of M). Then, w satisfies the query $M \models ? \mapsto \varphi$ in a vacuous way, as indeed, every computation of M that has w as a prefix continues to a suffix satisfying φ. A user, however, is rarely interested in seeing such vacuous triggers. In *relevant trigger querying*, we exclude vacuous triggers, and the solution to a relevant trigger query is restricted to words generated by the system.

The next variant we consider is *constrained trigger querying*. Model exploration is usually not a specific question to which there is a definite answer but rather an open-ended activity. Accordingly, trigger querying does not consist of a single query but rather it is an interactive dialog between the user and the trigger-query tool. A natural course of events is one in which the user refines the trigger queries in order to find scenarios that not only trigger the behavior in question, but also satisfy some constraints. For example, the user may search for scenarios that trigger $Ferr$ and in which the signal *ack* is never raised. In a constrained trigger query, the user provides, in addition to the system M and the behavior φ, also a regular expression c serving as a constraint for the possible solutions. In the above example, $c = (\neg ack)^*$. The solution for a constrained trigger query is the set of words that trigger φ in M and satisfy the constraint c.

Another variant is that of *observable trigger querying*. In many cases, the user would like to get a solution that depends only on a subset of the atomic propositions. For example, the user may wonder whether the environment can control the input signal *req* in a way that triggers the signal *err*, and if so, how. Technically, this corresponds to asking whether there is a word w over the alphabet $2^{\{req\}}$ such that all words over 2^{AP} that agree with w on the assignment to *req* trigger *err*. Thus, in addition to M and φ, the input to dominant trigger querying contains a set $O \subseteq AP$ of observable atomic propositions, and the solution is a set of words over 2^O.

The last variant of trigger querying we consider (in fact, it is more dual than variant) is the problem of finding *necessary conditions*. Recall that a word w triggers a behavior φ in M if all the computations of M with prefix w continue to a suffix that satisfies φ. Thus, a word triggering φ can be viewed as a sufficient condition for φ to happen in M, and trigger querying can be viewed as the problem of finding the set of sufficient conditions. Dually, a set of necessary conditions for φ to happen in M is a set $N \subseteq (2^{AP})^*$ such that for every computation π of M and position $i > 0$, if the suffix of π from position i satisfies φ, then the prefix of π up to position i is in N. As with traditional query checking, φ may have several sets of necessary conditions, and we are interesting in the strongest one, where strongest here means minimal in the language-containment partial order. Unlike traditional query checking, we show that a unique strongest necessary condition always exists. We also show that finding necessary conditions is computationally easier than finding sufficient solutions (i.e., trigger querying), and is only polynomial in the size of M.

[2]Note that temporal querying cannot be reduced to a search for counter examples. For example, the solution to $M \models ? \mapsto Xerr$ is the set of words w such that all the computations of M that start with w would reach *err* in their next cycle. On the other hand, the counterexamples to $M \models G\neg Xerr$ are words w such that there is a computation of M that starts with w and reaches *err* in its next cycle; such words w do not necessarily trigger $Xerr$.

II. Preliminaries

A word over an alphabet Σ is a sequence of letters from Σ. A word may be finite or infinite. We denote by Σ^* (Σ^ω) the set of finite (resp. infinite) words over Σ. Also, $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. For an infinite word $w = w_0 w_1 w_2 \ldots$ and natural numbers $i \leq j$ we denote by $w[i..j]$ the finite word $w_i w_{i+1} \ldots w_j$ and denote by w^i the infinite word $w_i w_{i+1} \ldots$.

A language is a set of words. For a language $L \subseteq \Sigma^\infty$, we denote by *pref*$(L)$ the set of prefixes of words in L. That is, *pref*$(L) = \{u \in \Sigma^* \mid \exists v \in \Sigma^\infty$ such that $uv \in L\}$. For a finite word $w \in \Sigma^*$ and a regular expression r over Σ, we use $w \models r$ to indicate that $w \in L(r)$.

A Kripke structure is a quintuple $M = \langle AP, Q, Q_0, R, L\rangle$, where AP is a set of atomic propositions, Q is a set of states, $Q_0 \subseteq Q$ is a set of initial states, $R \subseteq Q \times Q$ is a transition relation, and $L : Q \to 2^{AP}$ is a labelling function that labels each state with the set of atomic propositions that hold in it. We assume that the transition relation is total; i.e., for every state q there exists at least one state q' such that $R(q, q')$. A sequence of states $q_0, q_1 \ldots$ is a *computation* of M if $q_0 \in Q_0$ and for every $i \geq 0$, we have $R(q_i, q_{i+1})$. Unless we note otherwise, the computation is infinite. Each computation $q_0, q_1 \ldots$ induces the word $L(q_0)L(q_1)\ldots$ over the alphabet 2^{AP}. The set of words induced by computations of M is called the *language of* M and is denoted by $L(M)$. Note that $L(M) \subseteq (2^{AP})^\omega$.

For an LTL formula φ, a Kripke structure M, and a set of states S, we use $M, S \models \varphi$ to indicate that all the states in S satisfy φ. That is, all the computations that start in states in S satisfy φ. When $S = S_0$, we write $M \models \varphi$. Also, when $S = \{s\}$ is a singleton, we write $M, s \models \varphi$.

The specification language PSL [9] introduces the *suffix-implication* operator, denoted $\mapsto$. Similar operators exist in other modern specification formalisms such as the operator TRIGGERS in ForSpec [1] and in SVA (in fact, in SVA the trigger operator is the only temporal operator) [17]. The syntax of suffix implication is as follows. Let AP be the set of atomic propositions. For a regular expression r over the alphabet 2^{AP} and an LTL formula φ over AP, the expression $r \mapsto \varphi$ is in PSL.[3] The semantics of suffix implication is that an infinite word $w \in (2^{AP})^\omega$ satisfies $r \mapsto \varphi$ iff for every $i \geq 0$, if $w[0..i] \models r$ then $w^i \models \varphi$. Note the overlap in the i-th letter, which appears both in $w[0..i]$ and in w^i. Note also that the semantics ignores empty prefixes of w. For a Kripke structure M, a regular expression r, and an LTL formula φ, we say that r triggers φ in M, denoted $M \models r \mapsto \varphi$, if all words in $L(M)$ satisfy $r \mapsto \varphi$.

III. Trigger Querying

For a Kripke structure M and an LTL formula φ, trigger querying deals with question of the type "for which words $w \in \Sigma^*$, it holds that $M \models w \mapsto \varphi$". We denote this instance of trigger querying by $M \models ? \mapsto \varphi$. The *solution* of the trigger query $M \models ? \mapsto \varphi$ is the language of words that trigger φ in M; i.e. $L = \{w \in \Sigma^* \mid M \models w \mapsto \varphi\}$. While the solution language L may be infinite, the finiteness of M implies that L is always regular. We therefore restrict our attention to regular expressions (or finite automata) as representations of the solution language. Thus, a solution to the trigger query $M \models ? \mapsto \varphi$ is a regular expression r for which $L(r) = \{w \in \Sigma^* \mid M \models w \mapsto \varphi\}$. Note that the solution r is the maximal (in terms of language containment) regular expression that satisfies $M \models r \mapsto \varphi$.

Remark 1: The definition of trigger querying is not as useful as it first seems because of a technical subtlety: A word $w \in \Sigma^*$ that is not a finite computation of M is a *vacuous solution* to the trigger query $M \models ? \mapsto \varphi$ for any formula φ. Vacuous solutions are rarely interesting to users and are still members in the solution we define. In Section V-A, we define *relevant trigger querying*, which excludes such solutions. As discussed there, relevant trigger querying is technically very similar to trigger querying, in the sense that the algorithms and results about one variant carry on to the other one with minor changes. ■

As a first step toward solving trigger querying, we provide an alternative characterization for the set of words w that satisfy $M \models w \mapsto \varphi$. Consider a word $w = w_0 \ldots w_n \in \Sigma^*$, and a finite computation $s = s_0 \ldots, s_n$ of M. We say that s *induces* w iff for every $i \in \{0, \ldots, n\}$, it holds that $L(s_i) = w_i$. Note that a word w may be induced by several finite computations. We denote the set of finite computations that induce w by $induce(w)$. Also, we denote by $\delta(w)$ the set of states s for which there exists a finite computation ending in s that induces w. Formally, $\delta(w) = \{s_n \in Q \mid \exists s_0 \ldots s_n \in induce(w)\}$.

Lemma 2: For a Kripke structure M and an LTL formula φ, it holds that $M \models w \mapsto \varphi$ iff $M, \delta(w) \models \varphi$.

Proof: Assume first that $M, \delta(w) \models \varphi$. Thus, every computation starting in a state in $\delta(w)$ satisfies φ. Therefore, since every prefix of a computation of M that induces w ends in $\delta(w)$, we get that $M \models w \mapsto \varphi$.

For the other direction, assume that $M \models w \mapsto \varphi$. If $\delta(w) = \emptyset$ then $M, \emptyset \models \varphi$ vacuously. Otherwise, consider a state $s \in \delta(w)$. We show that $M, s \models \varphi$. Since $s \in \delta(w)$, there exists a finite computation $\pi = s_0 s_1 \ldots s_n$ of M such that $s = s_n$ and π induces w. Assume, by way of contradiction, that $M, s \not\models \varphi$. Then, there exists a computation $s, s^1 s^2 \ldots$ that does not satisfy φ. Consider the computation $c = s_0 s_1 \ldots s_{n-1} s s^1 s^2 \ldots$ of M. By the above, $c \not\models w \mapsto \varphi$, contradicting the assumption that $M \models w \overset{r}{\mapsto} \varphi$. ■

The alternative characterization allows us to reduce trigger querying to global model checking of φ on M.

Theorem 3: Trigger querying can be solved in polynomial space, and is PSPACE-hard.[4]

Proof: Let $M = \langle AP, Q, Q_0, R, L\rangle$ and let $[\![\varphi]\!]_M = \{s \in Q \mid M, s \models \varphi\}$ denote the set of states s for which $M, s \models \varphi$. Computing $[\![\varphi]\!]_M$ is the global model-checking problem for LTL, and is known to be in PSPACE [16].

[3]In PSL, the regular expression r is not defined directly over the alphabet 2^{AP}, but rather over the alphabet of Boolean expressions over 2^{AP}. In our setting, r would be the output of trigger querying, and it is natural to return it as a regular expression over the alphabet 2^{AP}.

[4]Trigger querying is not a decision problem and therefore cannot be PSPACE-complete.

By Lemma 2, the solution to the trigger query $M \models ? \mapsto \varphi$ is the language $L = \{w \in \Sigma^* \mid \delta(w) \subseteq [\![\varphi]\!]_M\}$. We construct a deterministic finite automaton $\mathcal{A}$ such that $L(\mathcal{A}) = L$. Finding a regular expression equivalent to $\mathcal{A}$ can be done using standard methods.

Intuitively, we would like to "transform M into an automaton" and then apply the subset construction setting the accepting states to be nonempty subsets of $[\![\varphi]\!]_M$. In an automaton, the alphabet is on the transitions, whereas in a Kripke structure, it is on the states. We move the label of a state to the transitions into the state, which makes it easier to deal with the overlap between the prefix and the suffix in the definition of the trigger operator.

We define $\mathcal{A} = \langle \Sigma, 2^Q \cup \{q_{in}\}, \{q_{in}\}, \rho, F\rangle$, where q_{in} is a new state, and ρ and F are defined as follows:

- The transition relation ρ is defined for every $\sigma \in 2^{AP}$ as follows. First, for the state q_{in}, we define $\rho(q_{in}, \sigma) = \{s \mid s \in Q_0 \text{ and } L(s) = \sigma\}$. Then, for a state $S \in 2^Q$, we define $\rho(S, \sigma) = \bigcup_{q \in S}\{s \mid R(q, s) \text{ and } L(s) = \sigma\}$.
- The set F of accepting states is the collection of subsets of $[\![\varphi]\!]_M$. Thus, $F = \{S \subseteq Q \mid S \subseteq [\![\varphi]\!]_M\}$.

It is not hard to prove by an induction on the length of w that $L(\mathcal{A})$ contains only words w such that $\delta(w) \subseteq [\![\varphi]\!]_M$. Hence, by Lemma 2, $L(\mathcal{A})$ is the solution to $M \models ? \mapsto \varphi$.

We turn now to the lower bound. PSPACE-hardness follows from the PSPACE hardness of LTL model checking. Indeed, for every Kripke structure M and LTL formula φ, we have that $M \models \varphi$ iff the solution to the trigger query $M \models ? \mapsto \varphi$ contains 2^{AP} (that is, all words of length 1). Unfortunately, the situation for trigger querying is worse as the complexity of the upper bound above is polynomial space in the size of M and not only in the length of φ. Accordingly, we now prove that the *structure complexity* of trigger querying, that is, the complexity in terms of the Kripke structure, assuming the formula is of a fixed length, is PSPACE-hard.

The proof is by a reduction from universality of nondeterministic automata on finite words (NFA, for short), which is known to be PSPACE-hard [14]. Given an NFA $\mathcal{A}$, we construct a Kripke structure M and a formula φ of a fixed length, such that for every word $w \in \Sigma^*$, it holds that $M \models w \mapsto \varphi$ iff $w \notin L(\mathcal{A})$. Thus, a solution to the trigger query $M \models ? \mapsto \varphi$ is an automaton that complements the language of $\mathcal{A}$. Since the length of φ is fixed, the PSPACE-hardness that follows is indeed for the structure complexity. Let $\mathcal{A} = \langle 2^{AP}, Q, Q_0, \rho, F\rangle$. To avoid vacuous solutions, we assume that $\mathcal{A}$ has a run on every finite word (if this is not the case, we can add a rejecting sink). Intuitively, we introduce a new atomic proposition a, and construct M such that for every word $w \in \Sigma^*$, the word $w \cdot a^\omega \in L(M)$ iff $w \in L(\mathcal{A})$. Thus, $M \models w \mapsto X\neg a$ iff $w \notin L(\mathcal{A})$.

In the NFA $\mathcal{A}$, the alphabet is on the transitions, whereas in the Kripke structure M, the alphabet is on the states. We construct M such that each state of it is associated with a state of $\mathcal{A}$ and the letter $\mathcal{A}$ is about to read. Thus, a run $q_0q_1 \ldots q_n$ of $\mathcal{A}$ on $w_1w_2 \ldots w_n$ corresponds to the computation $\langle q_0, w_1\rangle\langle q_1, w_2\rangle \ldots \langle q_{n-1}, w_n\rangle$ of M. In addition to the states associated with $\mathcal{A}$'s states and letters, M contains a special state q_a, labelled $\{a\}$, corresponding to acceptance in $\mathcal{A}$. Formally, $M = \langle AP \cup \{a\}, S, S_0, R, L\rangle$, where

- $S = (Q \times \Sigma) \cup \{q_a\}$ where q_a is a new state.
- $S_0 = Q_0 \times \Sigma$.
- $R = \{(\langle q, \sigma\rangle, \langle q', \sigma'\rangle) \mid q' \in \rho(q, \sigma)\} \cup \{(\langle q, \sigma\rangle, q_a) \mid \rho(q, \sigma) \cap F \neq \emptyset\} \cup \{\ (q_a, q_a)\}$.
- $L(\langle q, \sigma\rangle) = \sigma$ and $L(q_a) = \{a\}$.

For $n > 0$, it is not hard to prove, by an induction on n, that $\langle q_0, w_1\rangle\langle q_1, w_2\rangle \ldots \langle q_{n-1}, w_n\rangle \in \mathit{pref}(L(M))$ iff $q_0q_1 \ldots q_{n-1}$ is a run of $\mathcal{A}$ on $w_1 \ldots w_{n-1}$. Therefore, $w \cdot a^\omega \in L(M)$ iff $w \in L(\mathcal{A})$. It follows that $M \models w \mapsto X\neg a$ iff $w \notin L(\mathcal{A})$ as desired. ■

Remark 4: A note for readers familiar with alternating automata: Modulo the technical issue of the alphabet being on the transitions vs. the states, the automaton $\mathcal{A}$ is simply M when viewed as a universal automaton with $[\![\varphi]\!]_M$ being the set of accepting states. The construction described in the proof translates this automaton to a deterministic one. ■

IV. Partial solutions for trigger querying

As shown in Section III, the complexity of solving trigger querying is polynomial space in the size of the system. Unfortunately, such complexity might prove infeasible for many practical systems. Therefore, practical considerations lead us to search for partial yet more efficient solutions.

A reasonable approach is to search for subsets of the solution to a trigger query. The motivation for such an approach is that a user that is unable to get a complete characterisation of the words that trigger a behavior is usually still interested in specific scenarios that trigger the behavior.

A partial solution to a trigger query $M \models ? \mapsto \varphi$ is a subset of the solution to the trigger query. We allow the subset to be empty only if the complete solution is empty. Also, as in the case of complete solutions, we restrict attention to regular subsets. Formally, a partial solution to the trigger query $M \models ? \mapsto \varphi$ is a regular expression r such that $L(r) \subseteq \{w \in \Sigma^* \mid M \models w \mapsto \varphi\}$ and $L(r) = \emptyset$ iff $\{w \in \Sigma^* \mid M \models w \mapsto \varphi\} = \emptyset$.

The search of partial solutions may take various forms. A natural possibility is to search for a single word as a partial solution to a trigger query. It follows from the proof of Theorem 3, however, that the system complexity of deciding whether the solution to a trigger query is not empty is already PSPACE-hard. We therefore move to the next natural possibility, which is to search for a single word of a given bounded length. If such a bound is given in binary, however, the structure complexity of the problem remains PSPACE-hard.[5] An algorithm that searches for a single word in the solution is interesting if it also outputs the word. It seems natural, therefore, to consider the case in which the length bound is given in unary. Accordingly, partial trigger querying gets as input a structure M, a fixed formula φ, and a length

[5]This follows from the hardness proof in Theorem 3. There, given an NFA $\mathcal{A}$, we can reduce the problem of deciding the universality of $\mathcal{A}$ to trigger querying. It is not hard to see that a $\mathcal{A}$ is universal iff it accepts all words of length exponential in its size. Therefore, if we allow exponential bounds (whose binary encoding is polynomial), the hardness proof carries over to the partial-solution case.

bound n, given in unary, and decides whether there exists a word w of length at most n such that $M \models w \mapsto \varphi$.

Theorem 5: Partial trigger querying is NP-complete.

Proof: For the upper bound, we show that we can check, given a word w, whether $M \models w \mapsto \varphi$ in time polynomial in M and the length of w (although not necessarily polynomial in φ). Recall that $M \models w \mapsto \varphi$ iff $\delta(w) \subseteq [\![\varphi]\!]_M$. Computing both $[\![\varphi]\!]_M$ and $\delta(w)$ can be done in time polynomial in M and the length of w. Thus, the decision problem is in NP.

We proceed to prove the lower bound by a reduction from the following NP-complete problem [12]: given a directed graph $G = (V, E)$ and two vertices $s, t \in V$, are there two vertex-disjoint paths, one from s to t and one from t to s (the only vertices that appear in both paths are s and t).

For clarity of the presentation, we first present the reduction to relevant trigger querying, in which the solution contains only non-vacuous triggers (see Remark 1). We will later comment on the technical adjustment to the general case.

Consider a graph $G = (V, E)$, and two vertices $s, t \in V$. Let $n = |V|$. Note that if two vertex-disjoint paths from s to t and back exist, then two such paths of length at most n exist. We therefore restrict our attention to paths of length at most n. To simplify things further, we add an edge from t to itself (if it does not exist). Then, we can also assume that the paths are of the same length, as otherwise we can pad the shorter path with t's. Finally, we assume that each state in G has a successor (otherwise, we can add a sink to which all dead-ends go).

We intend to construct a Kripke structure M and a formula φ such that the solution to $M \models ? \mapsto \varphi$ contains a word of length at most n if there exist two vertex-disjoint paths from s to t and back, and is empty otherwise. Let $\Sigma = (V \times V) \cup \{p\}$. For simplicity, we assume that the atomic propositions of M encode letters in Σ. A finite computation of M that does not reach a state labelled p encode two sequences of vertices in G. For example, the word $(x_1, y_1) \ldots (x_k, y_k)$ encodes the two sequences of vertices $x_1 \ldots, x_k$ and $y_1, \ldots, y_k$. We define the transitions of M so that the projection of a finite computation on the first element encodes a path from s in G, while the projection on the second element encodes a path from t. For both elements the path may be followed by a p^* suffix. In addition, the following would hold.

1) If $w \in \Sigma^*$ encodes two vertex-disjoint paths from s to t and back, then for all infinite suffixes v for which $wv \in L(M)$, the first letter of v is p.
2) If $w \in \Sigma^*$ does not encode two vertex-disjoint paths from s to t and back, then there exists an infinite suffix v whose first letter is not p and $wv \in L(M)$.

If we succeed in constructing M as above, then for every word $w \in \Sigma^*$, we have that w is a non-vacuous solution to $M \models ? \mapsto Xp$ iff w encodes two vertex-disjoint paths from s to t and back. Thus, setting $\varphi = Xp$, we are done.

We now proceed to define M in detail. In order to know whether the two paths that correspond to a finite computation of M are vertex-disjoint, M chooses nondeterministically one vertex from each path and records it. If the first path reaches t and the second path reaches s, then M compares the recorded vertices. If the recorded vertices differ, M enters a sink state labelled p. If, on the other hand, the recorded vertices are the same, M continues to visit states that are not labelled by p. Note that when the paths are vertex-disjoint, the recorded vertices must be different. On the other hand, when the two paths are not vertex-disjoint, and share a vertex v, then there is a computation of M that generates these paths and chooses to record the vertex v for both paths.

The states of M are tuples in $V \times V \times (V \cup \{\perp\}) \times (V \cup \{\perp\})$ (as well as the special sink state s_p). In the state $\langle v_1, v_2, x_1, x_2 \rangle$, the values v_1 and v_2 stand for the current vertices in the first and second paths, and the values x_1 and x_2 stand for the recorded vertices from these paths. The symbol $\perp$ stands for "no vertex is recorded yet". A state $\langle v_1, v_2, x_1, x_2 \rangle$ is labelled by atomic propositions encoding the letter $\langle v_1, v_2 \rangle$. The state s_p is used to generate the p^ω suffixes and is labelled by p. The state $\langle s, t, \perp, \perp \rangle$ is the single initial state. The transition relation makes sure that a computation of M generates only sequences of pairs that correspond to paths in G. In addition, if the third (resp. fourth) element of a state is $\perp$, then a nondeterministic choice is made whether to record the current vertex v of the first (resp. second) path, which is done by replacing $\perp$ with v. If the third (resp. forth) element is not $\perp$, then it retains its value. Finally, if a vertex of the type $\langle t, s, x_1, x_2 \rangle$ is reached and $x_1 \neq x_2$ (or x_1 equals x_2 but both equal s, t, or $\perp$) then a transition to s_p is taken.

Formally, $M = \langle AP, S, S_0, R, L \rangle$, where AP, S, S_0, and L are defined above. Before defining R, we introduce the following notation. For $v \in V$ and $x \in V \cup \{\perp\}$, we denote by $rec(v, x)$ the set $\{v, \perp\}$ if $x = \perp$ and the set $\{x\}$ if $x \neq \perp$. We proceed to define R (we define it as a function $R : S \to 2^S$):

- $R(s_p) = \{s_p\}$.
- For states of the type $\langle t, s, x_1, x_2 \rangle$ in which $x_1 \neq x_2$, or $x_1 = x_2$ but $x_1 \in \{s, t, \perp\}$, set $R(\langle t, s, x_1, x_2 \rangle) = \{s_p\}$.
- For all other states, $R(\langle v_1, v_2, x_1, x_2 \rangle) = \{ \langle v_1', v_2', x_1', x_2' \rangle \mid E(v_1, v_1'), E(v_2, v_2'),\ x_1' \in rec(v_1, x_1),\ x_2' \in rec(v_2, x_2) \}$.

It is not hard to prove that for every word $w \in \Sigma^*$, we have that w is a non-vacuous solution to $M \models ? \mapsto Xp$ iff w encodes two vertex-disjoint paths from s to t and back. Note that the reduction is polynomial in the size of G, and that φ is fixed.

Since words w in the solution to $M \models ? \mapsto Xp$ may be vacuous solutions, the reduction shows that relevant trigger query is NP-hard. In order to prove NP-hardness for trigger querying, we modify M so that $M \models ? \mapsto Xp$ would not have vacuous solutions. For that, we have to modify M to a Kripke structure M' such that *pref*$(M') = \Sigma^*$. We should make sure that the non-vacuous solutions to $M \models ? \mapsto Xp$ continue to trigger Xp in M'. Thus, reading such a solution, we must move to s_p. We do this by defining M' to subsume M in such a way that for every word $w \in \Sigma^*$, if $w \in L(M)$, then the set of computations that induce w in M' is equal to the set of computations that induce it in M. Thus, M' only generates new words but does not add ways to generate words that are already in $L(M)$. It is not hard to define M' by adding to M a component that generate Σ^* and to which computations get whenever they get stuck in M. ■

A. *Practical considerations*

We suggest two symbolic methods for searching for partial solutions to a trigger query. Both methods consider a bound $n > 0$ on the length of words in the solution. First, a BDD-based method that computes all words of length n in the solution. Second, a SAT-based method that searches for a single word of length n in the solution.[6] For reasons explained below, we recommend the BDD-based method.

Let $\Sigma = 2^{AP}$. The main task of the BDD procedure is to compute the set $\{\langle w_1, \ldots, w_n, s\rangle \in \Sigma^n \times S \mid s \in \delta(w_1 \ldots w_n)\}$. For this purpose, we need BDD variables to represent letters and states. [7] We encode the transition relation as a set $R \subseteq S \times \Sigma \times S$ where $\langle s, \sigma, s'\rangle \in R$ if s' is a successor of s and $L(s') = \sigma$.

We use two vectors of BDD variables $\vec{s}$ and $\vec{s'}$, encoding states. Intuitively, $\vec{s}$ encodes current states and $\vec{s'}$ successor states. In addition, we use n vectors of BDD variables $\vec{w_1}, \ldots, \vec{w_n}$ encoding letters. We also use another vector of BDD variables $\vec{\sigma}$ encoding letters. In fact, the variables in $\vec{\sigma}$ are not necessary but the presentation is clearer with $\vec{\sigma}$. We assume that the algorithm has access to BDDs for the set $[\![\varphi]\!]_M(\vec{s})$, the initial set $S_0(\vec{s})$, and the transition relation $R(\vec{s}, \vec{\sigma}, \vec{s'})$. We also need a BDD $B(\vec{s}, \vec{\sigma})$ for the set $\{\langle s, L(s)\rangle \mid s \in S\}$.

The function UNTAG gets a BDD with $\vec{s'}$ variables and no $\vec{s}$ variables and replaces all the $\vec{s'}$ variables with the corresponding $\vec{s}$ variables. Similarly, for each $i \in \{1, \ldots, n\}$, the function i-TAG gets a BDD with with $\vec{\sigma}$ variables and no $\vec{w_i}$ variables and replaces all the $\vec{\sigma}$ variables with the corresponding $\vec{w_i}$ variables.

We describe the algorithm in Figure IV-A below.

Algorithm 1: BDD based algorithm

```
1 $X \leftarrow S_0$;
2 $X \leftarrow X \cap 1\text{-TAG}(B)$;
3 for $i = 2$ to $n$ do
4     $X \leftarrow \text{UNTAG}(\exists s\ X \cap i\text{-TAG}(R))$;
5 end
6 $Y \leftarrow \exists s\ (X \cap \neg[\![\varphi]\!]_M)$;
7 $Z \leftarrow \neg Y$;
8 return $Z$;
```

Intuitively, in lines 1 – 5, the algorithm computes, in the BDD X, the set $\{\langle \vec{w_1}, \ldots, \vec{w_i}, \vec{s}\rangle \mid \vec{s} \in \delta(\vec{w_1}, \ldots, \vec{w_i})\}$, for the i's between 1 to n. Thus, after line 5, the BDD X contains exactly all tuples $\{\langle w_1, \ldots, w_n, s\rangle \mid s \in \delta(w_1 \ldots w_n)\}$.

Accordingly, in line 6, the algorithm computes all the words $w_1 \ldots w_n$ for which $\delta(w_1 \ldots w_n) \not\subseteq [\![\varphi]\!]_M$, namely words that do not trigger φ. Finally, in line 7, the latter set is complemented resulting in the set of all words that do trigger φ, i.e, the solution to $M \models ? \mapsto \varphi$.

As the NP-complete complexity for partial trigger querying suggests, it is also possible to apply a SAT solver in order to find partial solutions. The formula to be considered can be built along the following lines: Let $\vec{X}$ be a vector of variables representing a *set* of states of M. Let $\vec{X'}$ be another such vector, and let $\vec{\sigma}$ be a vector of variables representing a letter. We denote by $\psi_R(\vec{X}, \vec{\sigma}, \vec{X'})$ a formula that is true iff $\vec{X'}$ represents the set of states that are successors of a state in $\vec{X}$ and whose labelling is $\vec{\sigma}$. Formally, $\vec{X'} = \{q' \in S \mid \exists q \in \vec{X}$ such that $R(q, q')$ and $L(q') = \vec{\sigma}\}$. Let $\psi_\varphi(\vec{X})$ be a formula that is true iff $\vec{X}$ represents a set that is contained in $[\![\varphi]\!]_M$. Finally, let $\psi_I(\vec{X})$ be a formula that is true iff $\vec{X}$ represents the set S_0. The formula to be fed into the SAT solver is $\psi_I(\vec{X_0}) \wedge \bigwedge_{i=1}^{n} \psi_R(\vec{X_{i-1}}, \vec{w_i}, \vec{X_i}) \wedge \psi_\varphi(\vec{X_n})$, where $\vec{X_0}, \ldots, \vec{X_n}$ and $\vec{w_1}, \ldots, \vec{w_n}$ are (vectors of) free variables.

A satisfying assignment assigns values to the (vectors of) variables $\vec{w_1} \ldots \vec{w_n}$ and $\vec{X_0}, \ldots, \vec{X_n}$. It is not hard to see that the values assigned to $\vec{w_1} \ldots \vec{w_n}$ encode a word that is partial solution for the trigger query, and that the values assigned to $\vec{X_0}, \ldots \vec{X_n}$ encode the sets $\delta(\vec{w_1}), \delta(\vec{w_1}\vec{w_2}), \ldots \delta(\vec{w_1} \ldots \vec{w_n})$.

Note that unlike the case in bounded model checking, the suggested algorithm uses as many variables as are states in the structure M (rather than in the symbolic representation of M). The technical need for so many variables arise from the need to consider *all* the elements of a set of states (encoded in the $\vec{X_i}$'s), and it occurs in other (already well challenged) contexts of bounded model checking, e.g., when evaluating the diameter of a model.

V. Variants of trigger querying

In this section we present several natural variants of trigger querying.

A. *Relevant trigger querying*

As noted in Remark 1, the definition of trigger querying allows the solution to contain vacuous solutions, namely words that are not induced by finite computations of M. Vacuous solutions are rarely interesting to users. In this section we define *relevant trigger querying*, which excludes vacuous solutions. We show that the algorithms and results we describe for trigger querying apply, with minor modifications, to the relevant case.

For a Kripke structure M, a word $w \in \Sigma^*$, and an LTL formula φ, the word w *relevantly triggers* φ in M, denoted $M \models w \overset{r}{\mapsto} \varphi$, if w triggers φ in M and w is induced by a finite computation of M. The solution to the *relevant trigger query* $M \models ? \overset{r}{\mapsto} \varphi$ is the set of words that relevantly trigger φ in M (i.e., $\{w \in \Sigma^* \mid M \models w \overset{r}{\mapsto} \varphi\}$).

Remark 6: Note that our notion of vacuity does not coincide with the notion of vacuity in the context of model checking of trigger formulas [5]: when $[\![\varphi]\!]_M = Q$ (for example, when $\varphi =$ **true**), the regular expression r does not affect the satisfaction of $r \mapsto \varphi$ in M. In such cases, all finite computations of M are non-vacuous solutions according to our definition. It is easy to adjust our solutions to a definition that would cause all solutions to be vacuous in such cases. ■

Solving relevant trigger querying is very similar to solving trigger querying. It is not hard to see that a word $w \in \Sigma^*$ is

[6] It is not hard to adapt the algorithms to words of length at most n. We present the versions for length exactly n since they are technically simpler.

[7] In real applications, both states and letters are subsets of the atomic propositions (an assignment to the atomic propositions induces a state, labelled by the letter that corresponds to the observable atomic propositions that are valid in the state). Our solution is general and does not assume such a relation between letters and states.

induced by a finite computation of a Kripke structure M, iff $\delta(w) \neq \emptyset$. Thus, the "relevant counterpart" of Lemma 2 is as follows.

Lemma 7: For a Kripke structure M and an LTL formula φ, it holds that $M \models w \overset{r}{\mapsto} \varphi$ iff $\delta(w) \neq \emptyset$ and $M, \delta(w) \models \varphi$.

It follows that the construction of $\mathcal{A}$ in the proof of Theorem 3 is valid also for the case of relevant trigger querying, except that we have to remove the empty set from F. The lower bound proofs are also easy to adjust for the relevant case. Hence, we can conclude with the following.

Theorem 8: Trigger querying can be solved in polynomial space, and is PSPACE-hard.

B. Constrained trigger querying

Trigger querying typically does not consist of a single query but rather it is an interactive dialog between the user and the trigger-query tool. A natural course of events is one in which the user refines the trigger queries in order to find scenarios that not only trigger the behavior in question, but also satisfy some constraints. For example, the user may search for scenarios that trigger $Ferr$ and in which the signal *ack* is never raised. In a *constrained trigger query*, the user provides, in addition to the system M and the behavior φ, also a regular expression c serving as a mask for the possible solutions. In the above example, $c = (\neg ack)^*$. The solution for a constrained trigger query is the set of words that trigger φ in M and satisfy the constraint c.

Trigger querying can be viewed as a special case of constrained trigger querying with $c = \textbf{true}$. Also, solving a constrained trigger query can proceed by solving the trigger query and intersecting the solution with the constraint language $L(c)$ (the intersection can be implemented as intersection of finite automata). Hence, we have the following.

Theorem 9: Constrained trigger querying can be solved in polynomial space and is PSPACE-hard.

Constrained trigger querying is of special interest when combined with partial trigger querying. Note that in the unconstrained case, it is possible to solve the trigger query and only then intersect the solution with the constraint. In partial trigger querying, such a course of action may lead to an empty set of partial solutions although the set of solutions that satisfy the constraint is not empty. Therefore, the constraint must be taken into account during the search for partial solutions (rather than after it). In practice, it is not hard to modify the algorithms suggested in Subsection IV-A to take the constraint c into account while searching for a partial solution.

Note that relevant trigger querying can be viewed as a special case of constrained trigger querying — one in which the constraint is the set of words induced by a finite computation of the Kripke structure. Nevertheless, the direct algorithms for relevant trigger querying are simpler than these that follow from this view.

C. Observable trigger querying

In many cases, the user would like to get a solution to a trigger query that depends only on a subset of the atomic propositions. For example, the user may wonder whether the environment can control the input signal *req* in a way that triggers the signal *err*, and if so, how. Technically, this corresponds to asking whether there is a word w over the alphabet $2^{\{req\}}$ such that all words over 2^{AP} that agree with w on the assignment to *req* trigger *err*.

Formally, for a word w over 2^{AP} and a set $O \subseteq AP$, let $w_{|O}$ be the word over 2^O obtained from w by projecting its letters on O. Then, for a word $w' \in 2^O$, let $wide(w', AP \setminus O) = \{w : w_{|O} = w'\}$ be the set of words over 2^{AP} obtained from w' by extending its letters to AP. The input to *observable trigger querying* contains, in addition to M and φ, also a set $O \subseteq AP$ of observable atomic propositions, The solution to the observable trigger query $M \models ? \mapsto \varphi$ with a set O of observable atomic propositions is the set $\{w' : M \models w \mapsto \varphi$ for all $w \in wide(w', AP \setminus O)\}$.

Note that the observable atomic propositions are "observable to the query". Thus, unlike the standard interpretation of observable and non-observable atomic propositions, here the idea is not to hide internal signals, but rather to restrict attention to the atomic propositions in terms of which we want the solution to the trigger query to be expressed. Often, these propositions would be related to implementation details or other internal signals that are considered non-observable in the standard context.

Theorem 10: Observable trigger querying can be solved in polynomial space and is PSPACE-hard.

Proof: For the upper bound, we modify the construction of the NFA $\mathcal{A}$ described in the proof of Theorem 3 as follows. The modification is only in the definition of ρ, which now assumes the alphabet 2^O. For a letter $\sigma \in 2^O$, we define $\rho(q_{in}, \sigma) = \{s \mid s \in Q_0$ and $L(s) \cap O = \sigma\}$, and for a state $S \in 2^Q$, we define $\rho(S, \sigma) = \bigcup_{q \in S}\{s \mid R(q,s)$ and $L(s) \cap O = \sigma\}$. Thus, the states in the successor states have to agree with σ on the atomic propositions in O, and all other atomic propositions are ignored. It is not hard to prove that $w \in 2^O$ is accepted by the modified NFA iff all the words in $wide(w, AP \setminus O)$ are accepted by $\mathcal{A}$.

Since trigger querying can be viewed as a special case of observable trigger querying (with $O = AP$), PSPACE-hardness follows from PSPACE-hardness of trigger querying. ■

Remark 11: A note for readers familiar with alternating automata and Remark 4. Recall that the automaton $\mathcal{A}$ constructed in the proof of Theorem 3 is M when viewed as a universal automaton. For nondeterministic automata, the existential projection of an automaton over the alphabet $\Sigma_1 \times \Sigma_2$ to an automaton over the alphabet Σ_1 can be easily done by ignoring the Σ_2 component in the transitions. For universal automata, existential projection involves an exponential blow up, but universal projection is easy. This is why the transition to observable trigger querying, which corresponds to a universal projection of the solution, does not make the problem more complex. ■

D. Necessary conditions

Recall that a word w triggers a behavior φ in M if all the computations of M with prefix w continue to a suffix that satisfies φ. Thus, a word triggering φ can be viewed as a sufficient condition for φ to happen in M, and trigger querying can be viewed as the problem of finding the set of sufficient conditions. In this section, we study the dual problem, namely finding the set of necessary conditions for φ to happen in M.

A *necessary condition* to φ in M is a regular expression r such that for every computation π of M and for every $i \geq 0$, if $\pi^i \models \varphi$, then $\pi[0..i] \models r$. It is easy to see that φ may have several necessary conditions in M (in fact, Σ^* is alway a necessary condition). The necessary conditions, however, are (partially) ordered by language containment. We say that a necessary condition r is *stronger* than a necessary condition r', if $L(r) \subseteq L(r')$.

We show that there always exists a unique strongest interesting necessary condition. Let $G \subseteq S$ denote the set of states from which there is a computation that satisfies φ. Formally, $G = \{s \in S \mid M, s \not\models \neg\varphi\}$. Let r be a regular expression for the set of words $w \in \Sigma^*$ for which there exists a finite computation of M that induces w and ends in G. Formally, $L(r) = \{w \in \Sigma^* \mid \delta(w) \cap G \neq \emptyset\}$.

For every computation π of M and for every $i \geq 0$, if $\pi^i \models \varphi$, then, by the definition of G, it must be that $\pi[0..i] \in L(r)$. Hence, r is a necessary condition. We prove that for every necessary condition r', we have $L(r) \subseteq L(r')$, thus r is the strongest necessary condition. Let r' be a necessary condition. Consider a word $w \in L(r)$. Let π and i be such that w is induced by $\pi[0..i]$. Let s be the last state in $\pi[0..i]$. Since s is in G, there exists some computation π_s of M that starts in s and satisfies φ. The concatenation of $\pi[0..i-1]$ and π_s is a computation of M whose suffix from position i satisfies φ. Since r' is a necessary condition, it must be that $\pi[0..i] \in L(r')$.

Theorem 12: Finding the strongest necessary condition is PSPACE-complete. The structure complexity of the problem is nondeterministic logarithmic space.

Proof: We start with the upper bounds. The set G above can be computed by solving the global LTL model-checking problem ($G = Q \setminus [\![\neg\varphi]\!]_M$). An NFA for the strongest necessary condition can be obtained from M be moving the labels from the states to the transitions into the state (a new initial state should be added), and defining G as the set of accepting states. Since global LTL model-checking is in PSPACE and its structure complexity is NLOGSPACE, we are done. The lower bound also follows from LTL model checking: Checking whether $M \models \varphi$ can be reduced to checking whether there is an initial state in the set $\{s \in S \mid M, s \not\models \varphi\}$, thus, it is reducible to finding the strongest necessary condition for $\neg\varphi$, in a Kripke structure that is obtained from M by marking the initial states with a special atomic proposition. ■

VI. Discussion

We introduced and studied trigger querying — a model-exploration problem in which one searches for the set of scenarios that trigger a behavior in a system.

Algorithms and modelling techniques originally developed for formal verification have turned out to be useful in other areas. This includes, for example, modelling and reasoning about biological systems [11], [10], business processes [13], and AI plans [2]. We believe that the application of trigger querying in these areas is very natural. Indeed, the type of questions one cares about in these areas are of the form "which scenarios trigger an action of a particular cell / an activation of some item in a contract / an action of the robot's arm."

Finally, our work here can be viewed as a first step towards a general temporal-query checking methodology, in which the "?" place-holder may be replaced by a temporal behavior rather than a propositional assertion. It looks like the most appropriate replacement for "?" are temporal events of a bounded duration, and the most convenient way to specify them are regular expressions, as done here. Also, triggers seem to capture a large fraction of the natural model-exploration questions interesting in practice. This, together with the popularity of the triggers operator in industrial setting has convinced us to restrict attention to trigger querying. At any rate, the ideas introduced here for trigger querying are useful in the general case.

References

[1] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. The ForSpec temporal logic: A new temporal property-specification logic. In *Proc. 8th TACAS*, LNCS 2280, pages 296–211. Springer, 2002.

[2] F. Bacchus and F. Kabanza. Planning for temporally extended goals. *Ann. of Mathematics and Artificial Intelligence*, 22:5–27, 1998.

[3] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic Sugar. In *Proc 13th CAV*, LNCS 2102, pages 363–367. Springer, 2001.

[4] G. Bruns and P. Godefroid. Temporal logic query checking. In *Proc. 16th LICS*, pages 409–420. IEEE Computer Society, 2001.

[5] D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M.Y. Vardi. Regular vacuity. In *Proc. 13th Conf. on Correct Hardware Design and Verification Methods*, LNCS 3725, pages 191–206. Springer, 2005.

[6] W. Chan. Temporal-logic queries. In *Proc 12th CAV*, LNCS 1855, pages 450–463. Springer, 2000.

[7] M. Chechik, M. Gheorghiu, and A. Gurfinkel. Finding state solutions to temporal queries. In *Proc. Integrated Formal Methods*, 2007. To appear.

[8] M. Chechik and A. Gurfinkel. TLQSolver: A temporal logic query checker. In *Proc 15th*, LNCS 2725, pages 210–214. Springer, 2003.

[9] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Springer, 2006.

[10] J. Fisher, N. Piterman, A. Hajnal, and T.A. Henzinger. Predictive modeling of signaling crosstalk during c. elegans vulval development. *PLoS Computational Biology*, 3(5):e92, May 2007.

[11] J. Fisher, N. Piterman, E.J.A. Hubbard, M.J. Stern, and D. Harel. Computational insights into *C. elegans* vulval development. *Proceedings of the National Academy of Sciences*, 102(6):1951–1956, February 2005.

[12] S. J. Fortune, J. Hopcroft, and J. Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science*, 10:11–121, 1980.

[13] J. Koehler, G. Tirenni, and S. Kumaran. From business process model to consistent implementation: A case for formal verification methods. In *EDOC*, pages 96–106. IEEE Computer Society, 2002.

[14] A.R. Meyer and L.J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential time. In *Proc. 13th IEEE Symp. on Switching and Automata Theory*, pages 125–129. 1972.

[15] M. Samer and H. Veith. Validity of CTL queries revisited. In *Proc. 17th CSL*, LNCS 2803, pages 470–483. Springer, 2003.

[16] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal of the ACM*, 32:733–749, 1985.

[17] S. Vijayaraghavan and M. Ramanathan. *A Practical Guide for SystemVerilog Assertions*. springer, 2005.