An Abstraction-Refinement Framework for Trigger Querying

Guy Avni and Orna Kupferman

School of Computer Science and Engineering, Hebrew University, Jerusalem 91904, Israel

Abstract. *Trigger querying* is the problem of finding, given a system M and an LTL formula φ, the set of *scenarios* that trigger φ in M; that is, the language L of finite computations of M such that all infinite computations that have a prefix in L continue with a suffix that satisfies φ. For example, the trigger query $M \models? \mapsto Ferr$ asks for the set of scenarios after which err aught to eventually happen. Trigger querying thus significantly extends query checking, which seeks propositional solutions, and is an extremely useful methodology for system exploration and understanding. The weakness of trigger querying lies in the fact that the size of the solution is linear in the size of the system. For trigger querying to become feasible in practice, we must offer solutions to cope with systems of big, and possibly infinite, state spaces.

In this paper we describe an abstraction-refinement framework for trigger querying. The general idea is to replace the reasoning about M by reasoning about an abstraction M_A of M, and return to the user two languages, L_l and L_u, that under- and over-approximate L, respectively. We consider predicate abstraction, and the languages L_l and L_u are defined with respect to the set of predicates. The challenge in defining the approximating languages is that trigger querying does not have a clear polarity, and the definition of L_l and L_u has to combine the upper- and over-approximations of M. We describe an automata-theoretic approach for refining and reducing $L_u \setminus L_l$. While refinement for model checking is *lengthwise*, in the sense that it is based on counterexamples, here we suggest both lengthwise and *widthwise* refinement, where the latter is based on cuts in an automaton for $L_u \setminus L_l$ and thus can symbolically handle batches of counterexamples. We show that our framework is robust and can be applied also for classical query checking as well as variants and extensions of trigger querying.

1 Introduction

The field of formal verification developed from the need to verify that a system satisfies its specification. In practice, formal-verification methodologies are used not only for ensuring correctness of systems but also for understanding systems and exploring their models [20]. In [6], 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 sub-formula is the place-holder "?". A solution to the query is a propositional assertion that, when replaces the place-holder, results in a formula that is satisfied in M. For example, if the query is $AG(? \rightarrow ack)$, then the set of solutions includes all assertions θ for which $M \models AG(\theta \rightarrow ack)$. 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 [5, 8, 9, 25].

A serious shortcoming of query checking is the fact that the solutions are propositional assertions. Thus, query checking is restricted to questions regarding one point in time, whereas most interesting questions about systems involve scenarios that develop over time. For example, solutions to the query $AG(? \rightarrow ack)$ are propositional assertions that imply ack in all the states of the system. Such assertions are clearly less interesting than scenarios after which ack is valid. As another example, consider a programmer trying to understand the code of some software. 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 [21], the authors 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, scenarios that are feasible in M and for which if a computation of M has a prefix that follows the scenario, then its suffix satisfies φ.[2]

Kupferman and Lustig formalized trigger querying using the temporal operator $\mapsto$ (triggers). The trigger operator was introduced in SUGAR (the precursor of PSL [4], called suffix implication there), and it plays an important role also in popular industrial specification formalisms like ForSpec [1] and System Verilog Assertions (SVA) [26]. Consider a system M with a set P of atomic propositions. A word w over the alphabet 2^P triggers an LTL formula φ in the system M, denoted $M \models w \mapsto \varphi$, if at least one computation of M has w as a prefix, and 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 then the set of words w that trigger φ in M. Since, as shown in [21], the solution is regular, trigger-querying algorithms return the solution by means of a regular expression or an automaton on finite words. The weakness of trigger querying lies in the fact that the size of the solution is linear in the size of the system. For trigger querying to become feasible in practice, we must offer solutions to cope with systems of big, and possibly infinite, state spaces.

In this paper we describe an *abstraction-refinement framework for trigger querying*. Abstraction is a well known approach for coping with the huge, and possibly infinite, state space of systems [2, 13]. [3] In particular, in the context of model checking, the counterexample guided abstraction-refinement (CEGAR) method has proven to be very effective [11, 12, 22]. Recall that the solution to the trigger query $M \models ? \mapsto \varphi$ is a reg-

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

[2] The definition in [21] is a bit different and allows also vacuous triggers: scenarios that are not feasible in M are considered solutions too.

[3] A different approach to cope with the complexity of trigger querying is studied in [24]. There, the user approximates the trigger by a statistical analysis of traces in the system.

ular language L over the alphabet 2^P. The general idea of our framework is to replace the reasoning about M by reasoning about an abstraction M_A of M, and return to the user two languages, L_l and L_u, that under- and over-approximate L. In more detail, we consider *predicate abstraction*, where the state space of M_A is 2^{Φ}, for a set Φ of propositional assertions on P. The abstraction M_A is a *modal transition system* [23], and has two types of transitions: *may transitions*, which over-approximate the transitions of M, and *must transitions*, which under-approximate them. The approximating languages L_l and L_u are over the alphabet 2^{Φ}, and they satisfy $L_l \subseteq L \subseteq L_u$, with an appropriate adjustment of $\subseteq$ to the different alphabets.

While L_l and L_u under- and over-approximate L, finding them combines both the under- and over-approximations of M. Intuitively, it follows from the fact that the solution to a trigger query does not have a clear polarity: it is not hard to see that $M \models w \mapsto \varphi$ if the set of the states of M that are reachable by tracing w is not empty and all the states in it satisfy $A\varphi$. When we consider an abstraction that under-approximates the transitions of M, two contradicting things happen: (1) we make it harder for words to make it into the solution, as fewer computations can trace w, and (2) we make it easier for words to make it into the solution, as more states satisfy $A\varphi$. Similar and dual difficulties arise when we try to work with an abstraction that over-approximates M or when we search for L_u.[4]

The smaller is the gap between L_l and L_u is, the more informative our approximating trigger languages are. Given a set of predicates Φ, refinement amounts to extending Φ so that $L_u \setminus L_l$ is reduced. We suggest two approaches for refinement. Both approaches are based on a deterministic automaton $\mathcal{C}$ for $L_u \setminus L_l$. As we show, the construction of $\mathcal{C}$ is based on an analysis of the reasons for lack of information, and avoids the complementation of L_l. The first approach, *lengthwise refinement*, is similar to the one performed in CEGAR, and is based on clinging to a single word $\tau \in (2^{\Phi})^*$ accepted by $\mathcal{C}$, and refining both the transitions that $\mathcal{C}$ traverses in its accepting run on τ as well as the accepting state that the run on τ reaches. The second approach, *widthwise refinement*, is possible thanks to the fact $\mathcal{C}$ accepts *all* the words in $L_u \setminus L_l$. Widthwise refinement iteratively reduces the language of $\mathcal{C}$ by clinging to a cut in its underlying graph. As we elaborate in the paper, $\mathcal{C}$ has cuts of special interest – these that correspond to may transitions that are not must transitions in M_A. An advantage of widthwise refinement is that it manipulates sets of states and thus has a simple symbolic implementation. We also suggest a hybrid approach that combines lengthwise and widthwise refinements, by basing the refinement on a sub-language of $L_u \setminus L_l$, and is also symbolic. All the three approaches are complete, in the sense that, unless we start with an infinite-state system, repeated application of them results in $L_l = L = L_u$.

We show that our framework is robust and can handle variants and extensions of trigger querying: classical query checking (in fact, the abstraction-refinement framework there is much simpler), constrained trigger querying (where the input also includes a regular expression, which restricts the range of solutions), and necessary conditions (where the goal is to characterize necessary conditions on the triggers; this problem is only NLOGSPACE-complete in the system).

[4] A different combination of may and must transitions, whose goal is to combine verification and falsification, is suggested also in [17].

Beyond making trigger-querying and its variants feasible in practice, we find the framework interesting from a theoretical point of view as it involves several conceptual differences from CEGAR, and thus involves new general ideas about abstraction and refinement. As we elaborate in Section 6, we believe that these ideas are useful also in other abstraction-refinement methodologies.

Due to the lack of space, some proofs and examples are omitted from this version and can be found in the full version, in the authors' homepages.

2 Preliminaries

We model systems over a set P of atomic propositions by a Kripke structure $M = \langle P, S, S_0, R\rangle$, where $S = 2^P$ is the set of states, $S_0 \subseteq S$ is a set of initial states, and $R \subseteq S \times S$ is a total transition relation. Since we take the set of states to be 2^P, we do specify a labeling function: the set of atomic propositions that hold in state $s \in 2^P$ is simply s. Note that our Kripke structures are deterministic (see Remark 2). A computation of M is a (finite or infinite) sequence of states $\pi = s_1, s_2, \ldots$ such that $s_1 \in S_0$ and $R(s_i, s_{i+1})$ for every $i \geq 1$. For an index $i \geq 1$, we use $\pi[1..i]$ to denote the prefix $s_1, \ldots, s_i$ of π and use π^i to denote the suffix $s_i, s_{i+1}, \ldots$ of π. Note that a word over the alphabet 2^P corresponds to at most one computation in M. The language of M, denoted $L(M)$, is the set of all computations of M.

For a Kripke structure M, a set of states S, and an LTL formula φ, we use $(M, S) \models \varphi$ to indicate that 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 $(M, s) \models \varphi$. We denote by $[\![\varphi]\!]$ the set of states that satisfy φ. I.e., for every $s \in 2^P$ we have that $s \in [\![\varphi]\!]$ iff $(M, s) \models \varphi$.

2.1 Trigger querying

A word $w \in (2^P)^*$ triggers an LTL formula φ in a Kripke structure M, denoted $w \mapsto \varphi$, if w is a computation of M and for every infinite computation $\pi \in L(M)$, if w is a prefix of π, then the suffix of π from position $|w|$ satisfies φ. Formally, $M \models w \mapsto \varphi$ iff for every computation π of M, if $\pi[1..|w|] = w$ then $\pi^{|w|} \models \varphi$. Note that there is an "overlap" and the $|w|$-th position in π participates in both the prefix w and the suffix satisfying φ. Trigger querying was introduced and studied in [21]. The solution of the trigger query $M \models ? \mapsto \varphi$ is the language of all words triggering φ in M.

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. These scenarios are 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 $M \models ? \mapsto F$*err*.

Remark 1. Our definition is a variant of the one defined in [21]. There, $M \models w \mapsto \varphi$ iff for every infinite computation $\pi \in L(M)$ if $\pi[1 \ldots |w|] = w$ then $\pi^{|w|} \models \varphi$. Thus,

all words not in $L(M)$ vacuously trigger all LTL formulas. As discussed in [21], the variants are technically similar. We find the variant with no vacuous solutions more appealing in practice.

The definition of trigger querying refers to computations, rather than states, in M. It is more convenient to work with an equivalent definition, which is state based:

Lemma 1. *[21] For a Kripke structure M, an LTL formula φ, and a finite word $w = s_1, \ldots, s_n$, it holds that $M \models w \mapsto \varphi$ iff $w \in L(M)$ and $s_n \in [\![\varphi]\!]$.*

By Lemma 1, triggering a formula is the same as triggering the set of states that satisfy this formula. Accordingly, we are going to use the notation $M \models w \mapsto T$, for a set $T \subseteq S$. Also, by Lemma 1, the language of triggers is simply the language of M when viewed as an automaton with $[\![\varphi]\!]$ being the set of accepting states. As also follows from Lemma 1, our framework can be extended to additional, more expressive universal formalisms, such as $\forall$CTL*.

2.2 Predicate abstraction

Consider a concrete Kripke structure $M_C = \langle P, 2^P, S_{0_C}, R_C \rangle$ and a set of predicates $\Phi = \{\theta_1, \ldots, \theta_m\}$ such that θ_i is a Boolean formula over P. Given M_C and Φ, we construct an abstract Kripke structure M_A by merging concrete states that agree on the predicates in φ into a single abstract state. Thus, the set of states of M_A is 2^Φ and a concrete state is mapped to an abstract state if it satisfies exactly all the predicates associated with the abstract state.

For a concrete state $c \in 2^P$ and an abstract state $a \in 2^\Phi$ we say that $c \models a$ iff c satisfies exactly all the predicates in a. Formally, $c \models \bigwedge_{\theta \in a} \theta \ \wedge \ \bigwedge_{\theta \notin a} \neg\theta$. We then also say that $c \in a$. Thus, for convenience, we are going to view an abstract state both as a set of predicates (and use the notation $\theta \in a$, for $\theta \in \Phi$) and as a set of concrete states (and use $c \in a$). Note that the predicates in Φ need not be independent. Thus, some subsets of Φ may be inconsistent, in which case no concrete state corresponds to them.

Typically, Φ is much smaller than P. Consequently, moving to the state space 2^Φ involves loss of information and calls for an approximation of M's transition relation. The abstract structure M_A (also known as a *modal transition system* [23]) has two types of transitions: *may transitions*, which over-approximate these of M, and *must transitions*, which under-approximate them. Formally, $M_A = \langle P, 2^\Phi, S_{0_A}, \rightarrow_{may}, \rightarrow_{must} \rangle$, where S_{0_A}, $\rightarrow_{may}$, and $\rightarrow_{must}$ are defined as follows.

- $a \in S_{0_A}$ iff there exists $c \in a \cap S_{0_C}$,
- $a \rightarrow_{may} a'$ iff there exists $c \in a$ and $c' \in a'$ such that $R_C(c, c')$, and
- $a \rightarrow_{must} a'$ iff for all $c \in a$ there exists $c' \in a'$ with $R_C(c, c')$.

A *may computation* is a sequence of states of M_A in which every two consecutive states have a may transition between them. Formally, $\pi = a_0, a_1, \ldots$ is a may computation if for every $i \geq 0$ it holds that $a_i \rightarrow_{may} a_{i+1}$. A *must computation* is defined

in a similar way, with $a_i \rightarrow_{must} a_{i+1}$. Note that every must computation is a may computation, but not vise versa.

Consider an LTL formula φ over Φ and an abstract state a. We distinguish between *may satisfaction* and *must satisfaction*. We say that $(M_A, a) \models_{may} \varphi$ if for every infinite may computation π that starts in a, we have $\pi \models \varphi$. Must satisfaction is defined similarly. Since may computations over-approximate the set of concrete computations, and must computations under-approximate them, and since the more computations there are, the less likely it is to satisfy an LTL formula, we have the following.

Lemma 2. *[16] Consider an LTL formula φ over Φ.*

1. *If $(M_A, a) \models_{may} \varphi$ then for every $c \in a$ it holds that $(M_C, c) \models \varphi$.*
2. *If $(M_A, a) \not\models_{must} \varphi$ then for every $c \in a$ it holds that $(M_C, c) \not\models \varphi$.*

For a concrete Kripke structure M_C and a set Φ of predicates, we denote by $M_C(\Phi)$ the abstract Kripke structure with state space 2^Φ that is induced by M_C.

Remark 2. Recall that our Kripke structures are deterministic. It is possible to define trigger querying also for nondeterministic systems – this is also the setting in [21], which make the trigger-querying problem PSPACE-complete in the size of the system. As in LTL model checking, abstraction is essential also when the complexity is only NLOGSPACE in the system. We will discuss the nondeterministic setting further in Remark 3.

2.3 Relating concrete and abstract languages

We relate words over predicates with words over atomic propositions. We define two functions: $abs : 2^P \rightarrow 2^\Phi$ and $conc : 2^\Phi \rightarrow 2^{2^P}$. For $c \in 2^P$, we define $abs(c) = \{\theta \in \Phi : c \models \theta\}$. For $a \in 2^\Phi$ we define $conc(a) = \{c \in 2^P : c \models a\}$. Thus, $abs(c)$ is the abstract state to which c belongs, and $conc(a)$ is the set of concrete states that belong to a.

We extend the definition of $conc$ and abs to words. For a finite or infinite word $w = w_1, w_2, \ldots$ over 2^P we define $abs(w)$ to be the word $\tau = \tau_1, \tau_2, \ldots$ over 2^Φ of the same length as w such that for every $i \geq 1$ it holds that $abs(w_i) = \tau_i$. For a word $\tau = \tau_1, \tau_2, \ldots$ over 2^Φ we define the language $conc(\tau)$ as the set of words w such that $\tau = abs(w)$. That is, for every $w = w_1, w_2, \ldots \in conc(\tau)$ and $i \geq 1$ it holds that $w_i \in conc(\tau_i)$.

For an abstract structure $M_C(\Phi)$ and an abstract may or must computation $\tau = \tau_1, \tau_2, ..$ of M_A, we say that τ has a *matching* concrete computation iff there is a computation $w \in conc(\tau) \cap L(M_C)$. Note that if τ is a must computation then it always has a matching concrete computation, but if τ is a may computation, it need not have a matching concrete computation.

We relate languages over predicates with languages over atomic propositions. For languages $L \subseteq (2^P)^*$ and $T \subseteq (2^\Phi)^*$ we say that $L \subseteq T$ iff for every $w \in L$ it holds that $abs(w) \in T$. Defining language containment in the other direction is more involved, as $conc(\tau)$, for $\tau \in (2^\Phi)^*$, contains many concrete computations and not all of them may be of interest. Therefore, in addition to the usual $T \subseteq L$ relation, we

define a variant of containment that has a concrete Kripke structure M_C as an additional parameter. For a single word $\tau \in (2^{\Phi})^*$ and a language $L \subseteq (2^P)^*$, we say that τ *is in* L *with respect to* M_C, denoted $\tau \in_{M_C} L$, if $conc(\tau) \cap L(M_C) \subseteq L$ and $conc(\tau) \cap L(M_C) \neq \emptyset$. That is, for every concrete word $w \in conc(\tau)$, if $w \in L(M_C)$ then w in L, and there is a word $w \in conc(\tau)$ that satisfies this condition non-vacuously. Now, for languages $T \subseteq (2^{\Phi})^*$ and $L \subseteq (2^P)^*$, we say that $T \subseteq_{M_C} L$ if for every $\tau \in T$, it holds that $\tau \in_{M_C} L$.

3 Approximating Triggers

Let M_C be a concrete Kripke structure, Φ a set of predicates, and φ an LTL formula over Φ. Also, let $M_A = M_C(\Phi)$ and $L_c \subseteq (2^P)^*$ be the solution to the trigger query $M_C \models ? \mapsto \varphi$. That is, for a word w over 2^P, it holds that $w \in L_c$ iff $M_C \models w \mapsto \varphi$. As discussed above, a nondeterministic automaton for L_c is exponential in the size of M_C, and our goal is to replace it by approximating languages by reasoning about M_A. Thus, given M_C, Φ, and φ, our goal is to find two languages $L_l, L_u \subseteq (2^{\Phi})^*$ such that $L_l \subseteq_{M_C} L_c \subseteq L_u$.

We distinguish between *may-triggering* and *must-triggering*. Consider an abstract Kripke structure M_A. For a word $\tau = a_1, \ldots, a_n$ over 2^{Φ} and a set of states $S \subseteq 2^{\Phi}$ we say that $\tau \mapsto_{may} S$ iff τ is a may computation of M_A and $a_n \in S$. Similarly, $\tau \mapsto_{must} S$ iff τ is a must computation of M_A and $a_n \in S$.

We use $M_A \models \tau \mapsto_{\alpha} [\![\varphi]\!]_{\beta}$, where $\alpha, \beta \in \{may, must\}$, to denote that τ α-triggers the set of states that β-satisfy φ.

We define the two languages $L_l, L_u \subseteq (2^{\Phi})^*$ as follows:

- $L_l = \{\tau \in (2^{\Phi})^* : M_A \models \tau \mapsto_{must} [\![\varphi]\!]_{may}\}$.
- $L_u = \{\tau \in (2^{\Phi})^* : M_A \models \tau \mapsto_{may} [\![\varphi]\!]_{must}\}$.

Note that L_l and L_u are defined by M_A when viewed as a deterministic automaton. For L_l, the automaton follows the must transitions and its accepting states are $[\![\varphi]\!]_{may}$. For L_u, it follows may transitions and its accepting states are $[\![\varphi]\!]_{must}$. Thus, the complexity is still linear in the system, but now it is the abstract system, which is considerably smaller.

Recall that $L_c = \{w \in (2^P)^* : M_C \models w \mapsto [\![\varphi]\!]\}$. Intuitively, L_l under-approximates L_c as words τ in L_l should pass two criteria that are more difficult to pass than these that words in L_c should: First, the word has to be a must (rather than concrete) computation. Second, the last state in the path should may satisfy (rather than satisfy) φ. Likewise, L_u over-approximates L_c as words τ in L_u should pass two criteria that are less difficult to pass than these that words in L_c should: the word has to be a may computation and the last state in the path should must satisfy φ. We now prove this intuition formally.

Theorem 1. $L_l \subseteq_{M_C} L_c$.

Proof: Let $\tau = a_1, \ldots, a_n \in L_l$. We show that $\tau \in_{M_C} L_c$. Thus, $L(M_C) \cap conc(\tau)$ is not empty and for every $w \in L(M_C) \cap conc(\tau)$ it holds that $M_C \models w \mapsto \varphi$. We first

prove that $L(M_C) \cap conc(\tau)$ is not empty. Recall that τ is a must computation and so there must be a valid concrete computation $w = c_1, \ldots, c_n$ such that for all $1 \leq i < n$ it holds that $c_i \in a_i$. Clearly, $w \in conc(\tau)$ and $w \in L(M_C)$.

Next we prove that for every $w \in L(M_C) \cap conc(\tau)$ it holds that $M_C \models w \mapsto \varphi$. Consider a word $w = c_1, \ldots, c_n \in L(M_C) \cap conc(\tau)$. We show that $(M_C, c_n) \models \varphi$. Since $\tau \in L_l$, we know that τ is a must computation in M_A. By definition, for $1 \leq i \leq n$ it holds that $c_i \in a_i$. By Lemma 2, $a_n \in [\![\varphi]\!]_{may}$ implies that for every $c \in a_n$ it holds that $(M_C, c) \models \varphi$, in particular $(M_C, c_n) \models \varphi$, and we are done. □

Theorem 2. $L_c \subseteq L_u$.

Proof: Consider a word $w \in L_c$. We prove that $abs(w) \in L_u$. That is, for the word $\tau = abs(w)$, it holds that $\tau \mapsto_{may} [\![\varphi]\!]_{must}$. By definition, $w \in L_c$ implies that $w \in L(M_C)$. Let $w = c_1, \ldots, c_n$. Consider the sequence $\tau = a_1, \ldots, a_n$ of M_A, where for every $1 \leq i \leq |w|$ it holds that $a_i = abs(c_i)$. Since for every $1 \leq i < n$ it holds that $R_C(c_i, c_{i+1})$, then $a_i \rightarrow_{may} a_{i+1}$. By definition, $w \in L_c$ also implies that $c_n \models \varphi$. By Lemma 2, this implies that $a_n \in [\![\varphi]\!]_{must}$. We conclude that $\tau \mapsto_{may} [\![\varphi]\!]_{must}$, and we are done. □

For a word $\tau \in (2^{\Phi})^*$, we say that Φ is *informative* for τ if $\tau \notin L_u$ or $\tau \in L_l$. Thus, refining M_A with respect to Φ is sufficient in order to know whether the words in $conc(\tau) \cap L(M_C)$ trigger φ in M_C: either $\tau \notin L_u$, in which case they all do not, or $\tau \in L_l$, in which case they do. In Example 1 and 2 we show words that are uninformative.

Remark 3. The proofs of Theorems 1 and 2 depend on M_C being deterministic. As we demonstrate in the full version, the theorems are not valid in the nondeterministic setting. In the full version we also suggest an alternative definition for L_l and L_u, with which the theorems are valid. Since the added technical difficulties are orthogonal to the ones addressed in this paper, we prefer to focus on the deterministic setting. We still describe below the alternative definitions. First, for the lower bound, we define L_l^{nd} to be $\{\tau \in (2^{\Phi})^* : \tau \mapsto_{may} [\![\varphi]\!]_{may}\} \cap L_{must}(M_A)$. Informally, τ is in L_l^{nd} iff every may computation that induces τ ends in a state that may-satisfies φ, and there is a must computation that induces τ. Now, for the upper bound, we define L_u^{nd} to be all the words $\tau \in (2^{\Phi})^*$ such that there is a may computation $a_1, ..., a_n$ that induces τ and $(M_A, a_n) \models_{must} \varphi$. Note that when applied to deterministic systems, we have that L_l^{nd} and L_u^{nd} coincide with L_l and L_u, respectively.

4 Refinement

The search for approximated triggers starts with a set of predicates Φ and two languages L_l and L_u. During the refinement process we iteratively close the gap between L_l and L_u by adding predicates to Φ. In this section we analyze the reasons why Φ need not be informative with respect to some words, and describe an automata-theoretic approach for characterizing the non-informative words and refinement.

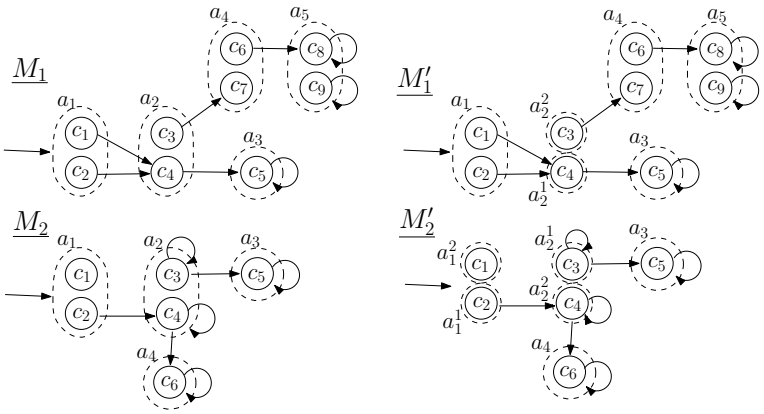

Fig. 1. Information loss in L_u and L_l.

4.1 Between the over and under approximations

We start with examples explaining four different types of "misses" in the approximating languages.

Example 1. In this example we demonstrate the case where the word τ triggers φ in M_C but is not in the under-approximation. Formally, $\tau \in_{M_C} L_c$ and $\tau \notin L_l$.

Consider the Kripke structure M_1 appearing in Figure 1. Let M_1^A be its abstraction with state space $\{a_1, \ldots, a_5\}$. Consider the formula $\varphi_1 = FGa_3 \wedge G\neg a_5$ and the word $\tau_1 = a_1a_2a_3$. Note that $conc(\tau_1) \cap L(M_C) = \{c_1c_4c_5, c_2c_4c_5\}$, and that both computations trigger φ_1. Indeed, they both end in c_5 and the only computation that starts in c_5 satisfies φ_1. Hence, $\tau_1 \in_{M_1} L_c$. On the other hand, $\tau_1 \notin L_l$, as τ_1 is a may computation that is not a must computation in M_1^A.

Consider now the word $\tau_2 = a_1a_2$. Note that $conc(\tau_2) \cap L(M_1) = \{c_1c_4, c_2c_4\}$, and that both computations trigger φ_1. Indeed, they both end in c_4 and the only computation that starts in c_4 satisfies φ_1. Hence, $\tau_2 \in_{M_1} L_c$. On the other hand, $\tau_2 \notin L_l$. While it is a must computation in M_1^A, it ends in the state a_2, which does not may satisfy φ_1.

Example 2. In this example we demonstrate the case where the word τ does not trigger φ in M_C but is in the over-approximation. Formally, $\tau \in L_u$ and $\tau \notin_{M_C} L_c$.

Consider the Kripke structure M_2 appearing in Figure 1. Let M_2^A be its abstraction with state space $\{a_1, \ldots, a_4\}$. Consider the formula $\varphi_2 = Ga_2 \vee Ga_3$ and the word $\tau_3 = a_1a_2a_3$. Since τ_3 is a may computation in M_2^A that ends in a_3, which must satisfies Ga_3, we have that $\tau_3 \in L_u$. Nevertheless, there is no concrete computation that matches τ_3, so $\tau_3 \notin_{M_2} L_c$.

Consider now the word $\tau_4 = a_1a_2$. Again, since τ_4 is a may computation and a_2 must satisfies φ_2, we have that $\tau_4 \in L_u$. Note that $(M_2, c_4) \not\models \varphi_2$ because the concrete computation $c_4c_6^\omega$ does not satisfy φ_2. Since $c_2c_4 \in conc(\tau_4) \cap L(M_2)$ we have $\tau_4 \notin_{M_2} L_c$.

Note that, for technical convenience, in both examples we use $c_1, c_2, \ldots$ and $a_1, a_2, \ldots$. Nevertheless, the structures can be generated using "real" propositions and predicates. For example, consider the following assignment of propositions to the variables in M_2. Let $P = \{a, b, c, d\}$, $c_1 = \{a, c\}$, $c_2 = \{a, c, d\}$, $c_3 = \{a, b\}$, $c_4 = \{a, b, d\}$, $c_5 = \{a, d\}$, and $c_6 = \{d\}$. By setting $\Phi = \{d \wedge \neg a, a \wedge c, a \wedge b\}$ we get: $a_1 = \{a \wedge c\}$, $a_2 = \{a \wedge b\}$, $a_3 = \emptyset$, and $a_4 = \{d \wedge \neg a\}$.

Our refinement procedure is based on a deterministic automaton $\mathcal{C}$ over the alphabet 2^Φ that accepts exactly all the words with respect to which Φ is not informative. In other words, $L(\mathcal{C}) = L_u \backslash L_l$. Rather than constructing $\mathcal{C}$ by taking the product of the automata for L_u and $\overline{L_l}$, we construct it according to an analysis of words with respect to which Φ is not informative. While the examples above demonstrate four possibilities for a word $\tau = a_1, \ldots, a_n \in (2^\Phi)^*$ to be in $L_u \setminus L_l$, we shall prove that we can group them into two types: Either τ is a may computation that is not a must computation in M_A and $a_n \models_{must} \varphi$, or τ is a must computation in M_A and $a_n \models_{must} \varphi$ but $a_n \not\models_{may} \varphi$.

Accordingly, $\mathcal{C}$ maintains two copies of M_A. In the first copy, $\mathcal{C}$ follows the must transitions of M_A, and it accepts words that end in a state that must satisfies but does not may satisfy φ. The automaton $\mathcal{C}$ moves from the first copy to the second one when it follows a may transition that is not a must transition. In the second copy, $\mathcal{C}$ follows may transitions, and it accepts words that end in a state that must satisfies φ. Formally, $\mathcal{C} = \langle 2^\Phi, (2^\Phi \times \{1, 2\}) \cup \{a_{init}\}, \delta_C, \{a_{init}\}, F_C \rangle$, where δ_C and F_C are defined as follows (when the condition does not hold, there is no transition and the run gets stuck):

- $\delta_C(a_{init}, a') = \langle a', 1 \rangle$, if $a' \in S_{0_A}$.
- $\delta_C(\langle a, 1 \rangle, a') = \langle a', 1 \rangle$, if $a \rightarrow_{must} a'$. Note that this implies that $a \rightarrow_{may} a'$ too.
- $\delta_C(\langle a, 1 \rangle, a') = \langle a', 2 \rangle$, if $a \rightarrow_{may} a'$ and $a \not\rightarrow_{must} a'$.
- $\delta_C(\langle a, 2 \rangle, a') = \langle a', 2 \rangle$, if $a \rightarrow_{may} a'$.
- $F_C = ((\llbracket \varphi \rrbracket_{must} \setminus \llbracket \varphi \rrbracket_{may}) \times \{1\}) \cup (\llbracket \varphi \rrbracket_{must} \times \{2\})$.

Lemma 3. $L(\mathcal{C}) = L_u \setminus L_l$.

4.2 The refinement algorithm

Before we turn to describe how we use $\mathcal{C}$ in the process of refinement, let us review the classical counterexample guided abstract refinement (CEGAR) methodology for verification of LTL properties (see [11]). The methodology is based on the fact that if an abstraction that over-approximates the concrete structure M_C satisfies an LTL formula, then so does M_C, and if the abstraction does not satisfy the LTL formula, then a counterexample for the satisfaction can be used for refining the abstraction. Formally, in CEGAR we model check M_A with may transition only. If $M_A \models \varphi$, then we are guaranteed that $M_C \models \varphi$, and we are done. If $M_A \not\models \varphi$, then we get a computation π in $L(M_A)$ such that $\pi \not\models \varphi$ and check whether π corresponds to a concrete computation. If it does, we conclude that $M_C \not\models \varphi$ and we are done. Otherwise, the abstract computation π is spurious and we use it in order to refine M_A to a new abstract structure M'_A that no longer has π as a computation. In the case of predicate abstraction, the refinement is done by adding predicates.

Consider the over and under approximations L_u and L_l of L_c. Let us use the notations $L_u(\Phi)$, $L_l(\Phi)$, and $\mathcal{C}(\Phi)$ in order to indicate that L_u, L_l, and $\mathcal{C}$ have been defined

with respect to the set Φ of predicates. The objective of the refinement algorithms is to tighten the gap between L_u and L_l. That is, we start with an initial set of predicates Φ and we refine the set to Φ' so that $L_u(\Phi') \setminus L_l(\Phi')$ is smaller than (in fact, strictly contained in) $L_u(\Phi) \setminus L_l(\Phi)$. In the case of CEGAR, refinement is *lengthwise*, in the sense that it is based on one counterexample that forms a path in the graph of M_A. In our case, we introduce, in addition to lengthwise refinement, also *widthwise* refinement. This is possible thanks to the automaton $\mathcal{C}$, which maintains all the words in $L_u \setminus L_l$, and thus constitutes a compact presentation of all "counterexamples". We also suggest a hybrid approach that combines lengthwise and widthwise refinements. Below we describe the three approaches in detail.

Describing the approaches, we use the *split* operator, defined below. Consider two sets of predicates Φ and Φ' such that $\Phi \subseteq \Phi'$. That is, Φ' extends Φ. For a state $a \in 2^\Phi$ we denote by $split(a, \Phi')$ the refinement of a with respect to Φ'. Formally $split(a, \Phi') = \{a' \in 2^{\Phi'} : a' \cap \Phi = a\}$. Thus, all the sets in $split(a, \Phi')$ agree with a on the predicates in Φ and differ on the predicates in $\Phi' \setminus \Phi$. We extend the definition of the split operator to words. For $\tau = a_1, \ldots, a_n \in (2^\Phi)^*$ we define $split(\tau, \Phi') = \{a'_1, \ldots, a'_n \in (2^{\Phi'})^n : a'_i \in split(a_i, \Phi)$ for all $0 \leq i \leq n\}$.

Lengthwise refinement The lengthwise refinement procedure, `refineWord`, gets as input a concrete Kripke structure M_C, a set of predicates Φ, and a word $\tau \in (2^\Phi)^*$ such that Φ is not informative with respect to τ. It then refines $M_C(\Phi)$ according to τ. Thus, the output is a set $\Phi' \supset \Phi$ such that Φ' is informative with respect to all computations $\tau' \in split(\tau, \Phi')$. We note that the procedure can get as input also a concrete word $w \in (2^P)^*$. It then executes `refineWord` with respect to $abs(w)$.

Consider a word $\tau \in L_u \setminus L_l$. Thus, $\tau \in L(C)$. The procedure `refineWord` proceeds in two steps. In the first step, we extend Φ to $\hat{\Phi}$ such that no computation in $split(\tau, \hat{\Phi})$ gets to the second copy of $\mathcal{C}$. In the second step we extend $\hat{\Phi}$ to Φ' so that the accepting states in the first copy of $\mathcal{C}$ do not include states that are reachable by computations in $split(\tau, \Phi')$.

For the first step, we initialize $\hat{\Phi}$ to Φ and extend it iteratively as follows. Let a_{init}, $\langle a_1, b_1\rangle$, ..., $\langle a_n, b_n\rangle$ be the accepting run of $\mathcal{C}(\Phi)$ on τ. If $b_n = 2$, then τ is a may computation that is not a must computation. We then find the first index $1 \leq i < n$ for which $b_{i+1} = 2$. Note that $a_i \rightarrow_{may} a_{i+1}$ but $a_i \not\rightarrow_{must} a_{i+1}$. We add to $\hat{\Phi}$ a predicate ρ that splits the abstract state a_i into two abstract states: a_i^1 consists of the concrete states that have outgoing edges into concrete states in a_{i+1}, and a_i^2 consists of the states that do not have outgoing edges into the concrete states in a_{i+1}. Thus, after refining, $a_i^1 \rightarrow_{must} a_{i+1}$ and $a_i^2 \not\rightarrow_{may} a_{i+1}$. Note that taking $\rho = \bigvee_{c \in a_i:\ \exists c' \in a_{i+1} \text{ with } R_C(c,c')} c$ achieves this goal. We continue with this step as long as there is a word in $split(\tau, \hat{\Phi})$ that $\mathcal{C}(\hat{\Phi})$ accepts with a run that ends in the second copy.

We start the second step with $\hat{\Phi}$, so the runs of $\mathcal{C}(\hat{\Phi})$ on all words in $split(\tau, \hat{\Phi})$ end in the first copy. Recall that the set of accepting states in this copy is $(\llbracket\varphi\rrbracket_{must} \setminus \llbracket\varphi\rrbracket_{may}) \times \{1\}$. In order to remove a state $\langle a, 1\rangle$ from F_C we use standard CEGAR, which studies counterexamples to the may-satisfaction of φ in a. As in CEGAR, if the counterexample is spurious, we use it to refine M_A so that may-satisfaction is challenged. Unlike standard CEGAR, here the procedure does not terminate when we detect a counterexample

that is not spurious. Instead, such a counterexample witnesses that the must-satisfaction of φ in a is due to under-approximation, and we use it in order to refine M_A so that must-satisfaction is challenged.

Example 3. As a first example, consider the Kripke structure M_1 in Figure 1, its abstraction M_1^A, the formula $\varphi_1 = FGa_3 \wedge G\neg a_5$, and the computation $\tau_1 = a_1a_2$. As shown in Example 1, $\tau_1 \in L_u \setminus L_l$. Since τ_1 is a must computation, the accepting run of $\mathcal{C}$ on τ_1 ends in the state $\langle a_2, 1\rangle$. Accordingly, we do not perform iterations in the first step of `refineWord` and continue to the second step, where CEGAR methods return the computation $\pi_1 = a_2a_4a_5^\omega$. We then find the state a_2 as a failure state and return the predicate $\rho_1 = c_3$ (note that only c_3 has an edge to states in a_4). We split the state a_2 (see M_1' on the right side of Figure 1). Note that all the computations starting at a_2^1 are concrete computations. It follows that a_2^1 is no longer an accepting state and we terminate the procedure. Note also that after the refinement, the word $a_1a_2^1$, which is the only word that is both in $\mathit{split}(\tau_1, \{a_1, a_2^1, a_2^2, a_3, a_4, a_5\})$ and a computation in the final abstract structure, is in L_l as required.

As a second example, consider the Kripke structure M_2 in Figure 1, its abstraction M_2^A, the formula $\varphi_2 = Ga_2 \vee Ga_3$, and the word $\tau_2 = a_1a_2a_3$. As shown in Example 2, $\tau_2 \in L_u \setminus L_l$. Since τ_2 is a may computation that is not a must computation, the accepting run of $\mathcal{C}$ on τ_2 ends in the state $\langle a_3, 2\rangle$. We perform an iteration of the first step of `refineWord`. The failure state is a_1 and we add the predicate c_2, which splits a_1 into a_1^1 and a_1^2. We continue to another iteration of the first step and find the word $a_1^1a_2a_3$. The run on it ends in the state $\langle a_3, 2\rangle$. The failure state is a_2 and we split it by adding the predicate c_3. We construct the abstract structure $M_A(\{a_1^1, a_1^2, a_2^1, a_2^2, a_3, a_4\})$ (see M_2' on the right side of Figure 1). Since all the edges are now concrete edges, $L_l = L_u$, and we skip the second step of `refineWord`.

Widthwise refinement For two sets of abstract states $S, T \subseteq 2^\Phi$, we say that the pair $\langle S, T\rangle$ induces an *interesting frontier* in $\mathcal{C}$ if (1) all the states in $S \times \{1\}$ are reachable in $\mathcal{C}$ from s_{init}, and (2) all the states in $T \times \{2\}$ can reach an accepting state in $\mathcal{C}$. Interesting frontiers are interesting indeed: if there are two states $a \in S$ and $a' \in T$ such that $a \rightarrow_{may} a'$ but $a \not\rightarrow_{must} a'$, then the transition from $\langle a, 1\rangle$ to $\langle a', 2\rangle$ participates in an accepting run of $\mathcal{C}$. We refer to a pair $\langle a, a'\rangle$ as above as a *bridge* in $\langle S, T\rangle$.

Widthwise refinement is based on a calculation of interesting frontiers and elimination of their bridges. The refinement procedure `refineCut` calculates frontiers that are not only interesting but also constitute a cut in the graph of $\mathcal{C}$: every accepting run of $\mathcal{C}$ that ends in the second copy must contain a bridge. Thus, as $\mathcal{C}$ is deterministic, elimination of bridges necessarily reduces the language of $\mathcal{C}$.

Consider a set of abstract states $P \subseteq 2^\Phi$. We define $post_C^1(P)$ as the set of states in the first copy of $\mathcal{C}$ that have incoming edges from states in P. Formally, $post_P^1(S) = \{a' : \text{ there exists } a \in P \text{ such that } \langle a', 1\rangle \in \delta_C(\langle a, 1\rangle, a')\}$. We define $pre_C^2(P)$ as the set of states in the second copy of $\mathcal{C}$ that have outgoing edges into states in P. Formally, $pre_C^2(P) = \{a : \text{ there exists } a' \in P \text{ such that } \langle a', 2\rangle \in \delta_C(\langle a, 2\rangle, a'\}$. The procedure `refineCut`, described in Figure 2, starts with the interesting frontier $\langle S_{0A}, [\![\varphi]\!]_{must}\rangle$ (note that indeed, all states in $S_{0A} \times \{1\}$ are reachable from the initial state of $\mathcal{C}$, and all the states in $[\![\varphi]\!]_{must} \times \{2\}$ are accepting in $\mathcal{C}$), and iteratively apply

$post^1_C$ and pre^2_C on the sets found reachable so far. The sets can be maintained by BDDs and their update is symbolic. The termination of `refineCut` is determined by the user. In the description below we guard the iterations by a Boolean flag $cont$ that the user may update in the **update**$(cont)$ procedure. Several updates are possible: the procedure can run for a bounded number of iterations (which may be a parameter to the procedure), until a fixed-point is reached (which guarantees that $\mathcal{C}(\Phi')$ accepts only must computations, and is therefore typically too good), or until a desired number of bridges is accumulated. The procedure also uses the procedure **refine**, which, as described above, splits the states in the sources of bridges so that they are no longer bridges.

Procedure `refineCut`;
Input: a set of predicates Φ
Output: a set of predicates Φ'
$\Phi' \leftarrow \Phi$; $S \leftarrow S_{0_A}$; $T \leftarrow [\![\varphi]\!]_{must}$;
while $cont$ **do**
    $S \leftarrow post^1_C(S) \cup S$;
    $T \leftarrow pre^2_C(T) \cup T$;
    update$(cont)$;
end
$B \leftarrow (S \times T) \cap (\rightarrow_{may} \setminus \rightarrow_{must})$;
$\Phi' \leftarrow$**refine**(B, Φ);

Fig. 2. The symbolic `refineCut` procedure.

Note that `refineCut` is similar to step one of `refineWord` in that it only refines paths that correspond to words in $L(\mathcal{C})$. It does not refine the accepting states like step two of `refineWord` (that is, such states may be refined as a result of moving to Φ', but they do not play a role in deciding which predicates to add).

Hybrid refinement Recall that lengthwise refinement clings to the transitions $\mathcal{C}$ traverses when a single word is read. Dually, widthwise refinement clings to a cut in $\mathcal{C}$ that contains a single transition in a run of many accepted words. Hybrid refinement combines the two approaches by clinging to a *language* of words.

Hybrid refinement gets from the user a regular expression r over 2^Φ of words he wants the approximating languages to be informative about. As with lengthwise refinement, the input can also be given as an expression over 2^P, in which case we replace $c \in 2^P$ by $abs(c)$. The procedure `refineLanguage` then constructs a nondeterministic automaton $\mathcal{A}$ for $L(r)$ and runs `refineCut` on the product of $\mathcal{C}$ with $\mathcal{A}$. Accordingly, the frontier and bridges are limited to words accepted by both $\mathcal{C}$ and $\mathcal{A}$.

5 Variants of Trigger Querying

In this section we consider several variants of trigger querying and show that our framework is robust and can handle them too. We start with the classical (non-triggered) query-checking problem, where an abstraction-refinement framework is quite straightforward.

5.1 Query checking

The input to the LTL *query-checking* problem is a model M over a set P of atomic propositions and a query φ, where a query is an LTL formula in which some subformula is the place-holder ? (e.g., $AG?$). The solution to the query-checking problem, denoted $QC(M, \varphi)$, is the set of strongest propositional assertions over P that, when replace the place-holder, result in a formula that is satisfied by M. We use $\varphi[? \leftarrow \theta]$ to denote that formula obtained from φ by replacing ? by θ. So, $\theta \in QC(M, \varphi)$ iff $M \models \varphi[? \leftarrow \theta]$ and for all propositional assertions ξ over P, if $\xi \rightarrow \theta$ then $M \not\models \varphi[? \leftarrow \xi]$.

Note that we consider here queries in LTL. Adjusting the framework to branching temporal logic is possible; it is more complicated, as it combines may and must transitions, but the expected thing works, and we leave it out of the scope of our contribution. In particular, for branching temporal logic, researchers have already found methods to cope with the complexity of query checking [5, 19]. On the other hand, known algorithms for solving LTL query checking do not do much better than checking all possible solutions. A nice exception, based on integer linear programming, is presented in [10], but it works only on a subclass of queries.

Abstraction for query checking For two sets of propositional assertions Γ_1 and Γ_2, we say that $\Gamma_1 \tilde{\subseteq} \Gamma_2$ if for every $\theta \in \Gamma_1$, there exists $\xi \in \Gamma_2$ such that $\xi \rightarrow \theta$ (possibly $\xi = \theta$). Thus, all the propositional formulas in Γ_1 are implied by these in Γ_2. In the *abstract query-checking* problem, we are given a concrete Kripke structure M_C, a set of predicates Φ, and an LTL query φ over Φ. The goal is to find two sets, Γ_l and Γ_u, of propositional assertions over Φ that under- and over-approximate the set of solutions. Formally, $\Gamma_l \tilde{\subseteq} QC(M_C, \varphi) \tilde{\subseteq} \Gamma_u$.

As we show below, the straightforward thing to do, namely to reason about the over- and under-approximations of M_C, work. Formally, let $M_A^{may} = \langle \Phi, 2^\Phi, S_{0_A}, \rightarrow_{may} \rangle$ and $M_A^{must} = \langle \Phi, 2^\Phi, S_{0_A}, \rightarrow_{must} \rangle$. Then,

Theorem 3. $QC(M_A^{may}, \varphi) \tilde{\subseteq} QC(M_C, \varphi) \tilde{\subseteq} QC(M_A^{must}, \varphi)$.

Refinement for query checking Let $\Gamma_l = QC(M_A^{may}, \varphi)$ and $\Gamma_u = QC(M_A^{must}, \varphi)$. As is the case with refinement for trigger querying, the goal of refinement is to decrease $\Gamma_u \setminus \Gamma_l$. The refinement is based on a propositional assertion θ over Φ such that $\theta \in \Gamma_u \setminus \Gamma_l$. We can choose θ arbitrarily, but typically the user provides assertions he finds interesting.

Given a formula $\theta \in \Gamma_u \setminus \Gamma_l$, there is $\xi \in \Gamma_u$ such that $\xi \rightarrow \theta$. Since $\xi \notin \Gamma_l$, it follows that $M_A^{may} \not\models \varphi[? \leftarrow \xi]$. Accordingly, refinement is similar to the one in CEGAR, which examines the counterexample for the satisfaction of $\varphi[? \leftarrow \xi]$ in M_A^{may}.

Unlike CEGAR, here we refine even when the counterexample is not spurious. Indeed, predicates need to be added in order to split states along the counterexample so that the corresponding concrete computation would match a must computation in the abstraction. The process can continue until $\Gamma_l = QC(M_C, \varphi) = \Gamma_u$, but is typically terminated earlier, when the gap between Γ_l and Γ_u is of less interest.

5.2 Constrained trigger querying

In this variant, the input to trigger querying contains also a regular expression r over 2^P, and the set of solutions is restricted to ones in $L(r)$. Let $C_c = L_c \cap L(r)$ be the solution to the trigger querying with respect to the concrete structure. Given a set Φ of predicates, our goal is to return two sets of abstract computations that approximate C_c from below and above. Let $abs(r) = \{abs(w) : w \in L(r)\}$ and $\overline{abs}(r) = \{abs(w) : w \notin L(r)\}$. The lower and upper bounds can now be obtained by restricting L_l and L_u according to r. Formally, let $C_l = L_l \setminus \overline{abs}(r)$ and $C_u = L_u \cap abs(r)$.

Theorem 4. $C_l \subseteq_{M_C} C_c \subseteq C_u$.

We start the refinement by refining L_l and L_u. We use the algorithms described in the previous sections. In particular, we suggest to use `refineLanguage` with the input language $L(r)$. Note that it is possible for $\tau \in L_l$ to have $w, w' \in conc(\tau)$ with $w \in L(r)$ but $w' \notin L(r)$. Such computations τ are in $C_u \setminus C_l$. Let $\tau = a_1, \ldots, a_n \in L_l \cap (C_u \setminus C_l)$. We continue the refinement according to the regular expression r. Since τ is a must computation, there must be at least two concrete computations $w, w' \in conc(\tau)$ such that $w \in L(r)$ and $w' \notin L(r)$. We find the first index $1 \leq i$ such that $w_i \neq w'_i$. We add a predicate that splits the abstract state a_i so that w_i and w'_i are mapped to different abstract states. We continue until $split(\tau, \Phi') \in C_l$.

5.3 Necessary Conditions

Trigger querying study sufficient conditions for φ to be triggered: if $M \models w \mapsto \varphi$, then after executing w, the suffix must satisfy φ. A dual problem is the one of finding necessary conditions for φ to hold. For a Kripke structure M and an LTL formula φ, the necessary condition for φ in M, denoted $NC(M, \varphi)$, is a set of finite computations such that for every $\pi \in L(M)$, if $\pi^n \models \varphi$ then $\pi[1..n] \in NC(M, \varphi)$. We require $NC(M, \varphi)$ to be minimal. Thus, if $w \in NC(M, \varphi)$ then there is a computation $\pi \in L(M)$ such that $\pi^{|w|} \models \varphi$ and $\pi[1, \ldots, |w|] = w$.

It is shown in [21] that the problem of finding $NC(M, \varphi)$ can be solved in nondeterministic logarithmic space. Still, as in LTL model checking, abstraction would be of great help in coping with large state spaces, and as with trigger querying, we are looking for languages that approximate $NC(M, \varphi)$ from above and below. As we show below, such languages can be obtained by reasoning about the may and must abstraction of M.

Theorem 5. $NC(M_A^{must}, \varphi) \subseteq_{M_C} NC(M_C, \varphi) \subseteq NC(M_A^{may}, \varphi)$.

The refinement algorithm for necessary conditions is similar to the one used in query checking: given $\tau \in N_u \setminus N_l$, the algorithm refines both the computation τ (in case it is a may but not must computation) and the last state in it (in case it must-satisfies but does not may-satisfy φ).

6 Discussion

We described an abstraction-refinement framework for trigger querying. Beyond making trigger-querying and its variants feasible in practice, we find the framework interesting from a theoretical point of view as it involves several conceptual differences from CEGAR, and thus involves new general ideas about abstraction and refinement:

1. In CEGAR, the goal is to find a solution to a binary query: does the system satisfy the specification. Here, we sought a solution to a query that is not binary: we searched for the language L_c, and we approximated L_c from both above and below. Consequently, the lack of information in the abstraction is reflected in the distance between the approximating languages, and this distance can serve the user in the refinement process. Furthermore, termination of the procedure is determined by the user, when he finds the approximation satisfying.
2. In CEGAR, one needs to over-approximate the transitions of the system in order to reason about universal properties and to under-approximate them in order to reason about existential ones. For specification formalisms with both universal and existential path quantifiers, CEGAR needs both may and must transitions [23], and it is common to use a three-valued semantics in such cases [16]. Trigger querying does not have a universal or existential polarity, and both types of approximations are needed. However, the three-value semantics is refined to a precise measure of the lack of information, by means of $|L_u \setminus L_l|$.
3. In CEGAR, we have to use the model-checking algorithm in order to generate counterexamples, some of which may be spurious. The set of spurious counterexamples in CEGAR corresponds to the set $L_u \setminus L_l$ in our setting. Unlike the case of CEGAR, here it was possible to model this set easily by means of the automaton $\mathcal{C}$, and it was therefore possible to base the refinement process on $\mathcal{C}$. In particular, it enabled both lengthwise and widthwise refinement, and the fact the set of "counterexamples" is regular enabled a symbolic refinement procedure.

These ideas are relevant and could be helpful in several variants of CEGAR: in model checking of quantitative specifications, where the query is not binary [7], in a CEGAR method for μ-calculus, where formulas need not have a universal or existential polarity [18], in attempts to refine the three-valued semantics [3], and in algorithms that gather batches of counterexamples before refining [14, 15].

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 196–211. Springer, 2002.
2. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S.K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys*, 2006.
3. T. Ball, O. Kupferman, and G. Yorsh. Abstraction for falsification. In *Proc. 17th CAV*, LNCS 3576, pages 67–81. Springer, 2005.

4. 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.
5. G. Bruns and P. Godefroid. Temporal logic query checking. In *Proc. 16th LICS*, pages 409–420. IEEE Computer Society, 2001.
6. W. Chan. Temporal-logic queries. In *Proc. 12th CAV*, LNCS 1855, pages 450–463. Springer, 2000.
7. K. Chatterjee, L. Doyen, and T. Henzinger. Quantative languages. In *Proc. 17th CSL*, pages 385–400, 2008.
8. M. Chechik, M. Gheorghiu, and A. Gurfinkel. Finding state solutions to temporal queries. In *Proc. Integrated Formal Methods*, pages 273-292, 2007.
9. M. Chechik and A. Gurfinkel. TLQSolver: A temporal logic query checker. In *Proc. 15th CAV*, LNCS 2725, pages 210–214. Springer, 2003.
10. H. Chockler, A. Gurfinkel, and O. Strichman. Variants of ltl query checking. In *Haifa Verification Conference*, LNCS 6504, pages 76-92, 2010.
11. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.
12. E.M. Clarke, A. Gupta, and O. Strichman. Sat-based counterexample-guided abstraction refinement. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(7):1113–1123, 2004.
13. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th POPL*, pages 238–252. ACM, 1977.
14. L. de Alfaro and P. Roy. Solving games via three-valued abstraction refinement. *Inf. Comput.*, 208(6):666–676, 2010.
15. M. Glusman, G. Kamhi, S. Mador-Haim, R. Fraer, and M.Y. Vardi. Multiple-counterexample guided iterative abstraction refinement: An industrial evaluation. In *Proc. 9th TACAS*, LNCS 2619, pages 176–191, 2003.
16. P. Godefroid and R. Jagadeesan. Automatic abstraction using generalized model checking. In *Proc. 14th CAV*, LNCS 2404, pages 137–150, 2002.
17. P. Godefroid, A.V. Nori, S.K. Rajamani, and S. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Proc. 37th POPL*, pages 43–56, 2010.
18. O. Grumberg, M. Lange, M. Leucker, and S. Shoham. Don't know in the μ-calculus. In *Proc. 6th VMCAI*, LNCS 3385, pages 233–249. Springer, 2005.
19. A. Gurfinkel, M. Chechik, and B. Devereux. Temporal logic query checking: A tool for model exploration. *IEEE Trans. Software Eng.*, 29(10):898–914, 2003.
20. U. Kühne, D. Große, and R. Drechsler. Property analysis and design understanding. In *DATE*, pages 1246–1249, 2009.
21. O. Kupferman and Y. Lustig. What triggers a behavior? In *Proc. 7th Int. Conf. on Formal Methods in Computer-Aided Design*, pages 146–153. IEEE Computer Society, 2007.
22. R.P. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton Univ. Press, 1994.
23. K.G. Larsen and G.B. Thomsen. A modal process logic. In *Proc. 3rd LICS*, 1988.
24. D. Lo and S. Maoz. Mining scenario-based triggers and effects. In *Proc. 23rd ASE*, pages 109–118, 2008.
25. M. Samer and H. Veith. Validity of ctl queries revisited. In *Proc. 12th CSL*, LNCS 2803, pages 470–483. Springer, 2003.
26. S. Vijayaraghavan and M. Ramanathan. *A Practical Guide for SystemVerilog Assertions*. Springer, 2005.