Generalized Counterexamples to Liveness Properties

Gadi Aleksandrowicz, Jason Baumgartner, Alexander Ivrii, Ziv Nevo   IBM Corporation

***Abstract*—We consider *generalized counterexamples* in the context of liveness property checking. A generalized counterexample comprises only a subset of values necessary to establish the existence of a concrete counterexample. While useful in various ways even for safety properties, the length of a generalized *liveness* counterexample may be exponentially shorter than that of a concrete counterexample, entailing significant potential algorithmic benefits.**

One application of this concept extends the k-LIVENESS proof technique of [1] to enable failure detection. The resulting algorithm is simple, and poses negligible overhead to k-LIVENESS in practice. We additionally propose dedicated algorithms to search for generalized liveness counterexamples, and to manipulate generalized counterexamples to and from concrete ones. Experiments confirm the capability of these techniques to detect failures more efficiently than existing techniques for various benchmarks.

I. Introduction

It is well-known that counterexamples are often redundant, containing many values that are irrelevant to the failure exhibited therein. The process of eliminating unnecessary values from a trace is referred to as *generalization*, and has numerous benefits. For example, the elimination of irrelevant values facilitates manual and automated debugging [2], and improves the effectiveness of counterexample-guided abstraction refinement [3].

This paper focuses upon counterexamples to liveness properties. For finite systems, such counterexamples may efficiently be represented as lasso-shaped traces consisting of a prefix and a loop suffix exhibiting a state repetition which can be infinitely unrolled. The length of a lasso is the sum of the prefix and suffix lengths. A unique benefit of generalizing a liveness counterexample is that it may shorten the lasso length – possibly exponentially so – if the set of state variables comprising a state repetition is reduced during generalization.

Example 1: Let q, x, y be Boolean signals whose initial and next-state behaviors are determined as follows: $q_0 = 1$, $x_0 = 0$, $y_0 = 0$, $q' = (q \wedge x) \vee (\neg q \wedge y)$, $x' = q \wedge y$, $y' = \neg x$. Consider liveness property FGq, specifying that on every trace q must eventually become true forever. A counterexample would illustrate $q = 0$ at least once in its loop suffix, for example $(q, x, y) = (1, 0, 0) \rightarrow (0, 0, 1) \rightarrow (1, 0, 1) \rightarrow (0, 1, 1) \rightarrow (1, 0, 0)$ of length 4. Note that $(q = 1) \wedge (x = 0) \Rightarrow (q' = 0) \wedge (y' = 1)$ and $(q = 0) \wedge (y = 1) \Rightarrow (q' = 1) \wedge (x' = 0)$. This illustrates a generalized counterexample: $(1, 0, \cdot) \rightarrow (0, \cdot, 1) \rightarrow (1, 0, \cdot)$ of length 2.

Example 2: We may modify q' from Example 1 to $q' = (q \wedge x \wedge (cnt = 0)) \vee (\neg q \wedge y)$, where cnt is an n-bit cyclic counter. Now the minimal concrete counterexample has length 2^n, while the generalized counterexample from Example 1 is still valid.

Example 3: One may argue that cnt is sequentially unobservable in Example 2 because $q \wedge x \equiv 0$, hence a transformation-based approach may enable the detection of an adequate short counterexample [4]. We may modify this example to $x' = (q \vee i) \wedge y$, where i is a nondeterministic input. Now cnt becomes observable, precluding a direct application of transformation-based methods. However, the generalized counterexample is still valid since both transitions $(1, 0, \cdot) \rightarrow (0, \cdot, 1)$ and $(0, \cdot, 1) \rightarrow (1, 0, \cdot)$ can be achieved *for some value of the inputs*, here $i = 0$ for transition $(0, \cdot, 1) \rightarrow (1, 0, \cdot)$.

We show that, surprisingly, the traces produced by the underlying safety model checker of k-LIVENESS [1] are often sufficient to witness a counterexample. Furthermore, in many cases the traces which do not exhibit a counterexample may be manipulated using our techniques to yield valid counterexamples.

II. Preliminaries

We represent a finite state system S as a tuple $\langle i, x, I(x), T(i, x, x')\rangle$, which consists of primary inputs i, state variables x, predicate $I(x)$ defining the initial states, and predicate $T(i, x, x')$ defining the transition relation. Next-state variables are denoted as x'. We assume that T is represented as a *netlist*, that is a directed acyclic graph with nodes corresponding to logic gates. Given the values of x and i, the values of x' may thus be uniquely computed by propagation – i.e., using Boolean or three-valued simulation.

State variables and their negations are called *literals*, and disjunctions (conjunctions) of literals are called *clauses* (*cubes*). A *state* is a Boolean assignment to all of x. A *generalized state* is an assignment to a subset of x, representing a set of states. We denote concrete states by s and generalized states by t throughout the paper.

ISBN 978-0-9835678-3-7/13. Copyright owned jointly by the authors and FMCAD Inc.

Definition 1: Given two generalized states t_0 and t_1, we say that t_0 is a *predecessor* of t_1 if for every concrete state $s_0 \in t_0$ there exists a concrete state $s_1 \in t_1$ and an input i_0 such that $(i_0, s_0, s_1') \models T$.

Definition 2: We say that t_0 is a *concrete* predecessor of t_1 if t_0 is a concrete state.

Note that the definition of a predecessor is not symmetric, and it does not require that every state in t_1 is reachable from a state in t_0. For practical purposes we need more restricted notions of a predecessor.

Definition 3: Given two generalized states t_0 and t_1 and input i_0, we say that t_0 is an *implying* predecessor of t_1 with respect to i_0 if $t_0 \wedge i_0 \wedge T \wedge \neg t_1'$ is unsatisfiable.

Definition 4: We say that t_0 is a *propagating* predecessor of t_1 with respect to i_0 if t_0 and i_0 imply t_1' by propagation.

From these definitions, each concrete predecessor is also propagating with respect to some input, and each propagating predecessor with respect to i_0 is also implying with respect to i_0. We omit explicit input references when they are clear from context.

Example 4: Consider $x' = x \wedge (y \oplus i)$, where x and y are state variables and i is an input. Then $x = 1 \wedge y = 1$ is an implying predecessor of $x = 1$ with respect to $i = 0$. Further, $x = 1$ is predecessor of $x = 1$ but cannot be an implying predecessor since there is no value of i which works for all y.

Definition 5: A *concrete trace* is a sequence of concrete states $\langle s_0, \ldots, s_n \rangle$ such that $s_0 \models I$, and for each $0 \leq k < n$, s_k is a concrete predecessor of s_{k+1}.

Definition 6: A *generalized trace* is a sequence of generalized states $\langle t_0, \ldots, t_n \rangle$ such that t_0 contains an initial state, and for each $0 \leq k < n$, t_k is a predecessor of t_{k+1}.

We say that *concretizing* a state t is the process of adding literals to t, and *generalizing* t is the process of removing literals from t.

A. Generalized Counterexamples to Liveness

In the spirit of [1] we consider liveness properties given in the form FGq. More general liveness properties (and fairness constraints) may be reduced to this form using additional logic. Furthermore, since the validity of $FG(Xq)$ is equivalent to the validity of FGq, we can assume that q itself is a state variable.

Definition 7: A *concrete counterexample* to FGq is a concrete trace $\langle s_0, \ldots, s_n \rangle$ and an index m with $0 \leq m < n$, such that **(1)** $s_m = s_n$, and **(2)** $\exists k \in [m..n]$ with $s_k \implies \neg q$.

Thus $s_0, \ldots, s_{m-1}$ corresponds to the lasso prefix, and $s_m, \ldots, s_n$ corresponds to the loop suffix, with cycle s_k exhibiting $\neg q$. Note that $s_m = s_n$ implies that s_n is a concrete predecessor of s_{m+1}, hence the loop can be infinitely unrolled.

Definition 8: A *generalized counterexample* to FGq is a generalized trace $\langle t_0, \ldots, t_n \rangle$ and an index m with $0 \leq m < n$, such that **(1)** $t_m \implies t_n$, and **(2)** $\exists k \in [m..n]$ with $t_k \implies \neg q$.

Note that we do not require that $t_m = t_n$, but rather that t_n is more concrete than t_m.

Examples 1-3 illustrate that the length of a generalized counterexample to FGq may be exponentially shorter (with respect to netlist size) than that of a concrete counterexample. Theorem 1 will demonstrate that the former implies the existence of the latter. Because a generalized counterexample may be exponentially shorter than a concrete one, in cases it may be easier to detect a generalized counterexample, which motivates the algorithms in Sections IV and V.

In practice, a generalized counterexample may actually be more informative and easier to debug since it more clearly illustrates the "essential" reason for the failure. Similarly, it is often undesirable in practice that a liveness counterexample on a reduced netlist (after cone-of-influence, redundancy removal, ...) be extended to a possibly exponentially-longer unreduced trace merely to ensure a state repetition over irrelevant logic.

III. Trace Manipulation Algorithms

A. Trace Concretization

Given a generalized trace, we may fully or partially concretize it using Algorithm 1. $ConcretizeInitial(t_0)$ returns a concretization of t_0 which still contains a state in I, which may be computed with a satisfiability query. $ConcretizeForward(\tilde{t}_k, t_{k+1})$ returns a concretization of t_{k+1} with $\tilde{t}_k$ as its predecessor. If $\tilde{t}_k$ is a propagating predecessor of t_{k+1}, we can use three-valued simulation to implement $ConcretizeForward$, using an unknown X value for any state variable not in $\tilde{t}_k$ and assessing which state variables attain fixed values in t_{k+1}. Alternatively, we can use a satisfiability query: if $\tilde{t}_k$ is an implying predecessor of t_{k+1} with respect to some i_k, for each state variable x not in t_{k+1} we can consider the query $\tilde{t}_k \wedge i_k \wedge T \wedge x'$. If this query is unsatisfiable, $x = 0$ can be added to t_{k+1}. Similarly, if $\tilde{t}_k \wedge i_k \wedge T \wedge \neg x'$ is unsatisfiable, $x = 1$ can be added to t_{k+1}.

Theorem 1: Any generalized counterexample c to FGq may be extended to a concrete counterexample $\tilde{c}$.

Proof: Consider a generalized counterexample c to FGq with its lasso state repeating at times m and $n > m$. By the discussion above, we can find a concrete trace $\tilde{c}$ which agrees with valuations in c, though the states at times m and n in $\tilde{c}$ may not be identical. However, since

Algorithm 1 Trace Concretization

Input: A trace $\langle t_0, \ldots, t_n \rangle$
Output: A trace $\langle \tilde{t}_0, \ldots, \tilde{t}_n \rangle$ with $t_k \implies \tilde{t}_k$ for all k.

$\tilde{t}_0 \leftarrow ConcretizeInitial(t_0)$
for $k = 0, \ldots, n-1$ **do**
  $\tilde{t}_{k+1} \leftarrow ConcretizeForward(\tilde{t}_k, t_{k+1})$

Algorithm 2 Trace Generalization

Input: A trace $\langle t_0, \ldots, t_n \rangle$
Output: A trace $\langle \tilde{t}_0, \ldots, \tilde{t}_n \rangle$ with $\tilde{t}_k \implies t_k$ for all k.

$\tilde{t}_n \leftarrow GeneralizeFinal(t_n)$
for $k = n-1, \ldots, 0$ **do**
  $\tilde{t}_k \leftarrow GeneralizeBackward(t_k, \tilde{t}_{k+1})$

the loop of c can be infinitely unrolled, assuming a finite system, eventually a state in $\tilde{c}$ will repeat, thus yielding a concrete counterexample. ■

Theorem 1 demonstrates that a generalized liveness counterexample may be mapped to a concrete one using simulation, implying a scalable algorithm.

B. Trace Generalization

Given a trace, we may use Algorithm 2 to generalize it. $GeneralizeFinal(t_n)$ returns a generalization of t_n. For example, if the trace witnesses a number of failures of q and $t_n \implies \neg q$, this corresponds to removing some of the other variables from t_n. $GeneralizeBackward(t_k, \tilde{t}_{k+1})$ returns a generalization of t_k which still forms a predecessor of $\tilde{t}_{k+1}$. If t_k is a propagating predecessor of t_{k+1}, then we can generalize t_k using ternary simulation: if replacing the value of a state variable in t_k by X does not influence any of the variables in t'_{k+1}, then this variable can be removed from t_k. More generally, when t_k is an implying predecessor of t_{k+1} with respect to some i_k, we can consider the unsatisfiability of $t_k \wedge i_k \wedge T \wedge \neg t'_{k+1}$ and generalize from t_k variables unnecessary in the unsatisfiable core returned by the SAT solver.

C. Modifying Traces with Tentative Loops

The following example demonstrates that the processes of concretizing and generalizing a trace are both capable of creating or destroying the validity of that trace as a counterexample.

Example 5: Let q, x, y be state variables with initial values $q_0 = 1$, $x_0 = 0$, $y_0 = 0$ and next-state values $q' = q \wedge x$, $x' = x$, $y' = \neg y$. The concrete trace $(1,0,0) \rightarrow (0,0,1) \rightarrow (0,0,0)$ does not exhibit a counterexample to FGq. A partially-generalized trace $(1,0,\cdot) \rightarrow (0,0,\cdot) \rightarrow (0,0,\cdot)$ does exhibit a counterexample. A futher-generalized trace $(1,0,\cdot) \rightarrow (0,0,\cdot) \rightarrow (0,\cdot,\cdot)$ again does not exhibit a counterexample.

Consider a generalized trace $\langle t_0, \ldots, t_n \rangle$ with a pair of indices $i < j$ such that $t_i \wedge t_j \neq \bot$ and $\exists k \in [i..j]$ with $t_k \implies \neg q$. The condition $t_i \wedge t_j \neq \bot$ means that there is no state variable present in opposite polarities in t_i vs t_j. We call $\langle t_i, \ldots, t_j \rangle$ a *tentative* loop. We propose the following technique, referred to as $ConcretizeTentative(i, j)$: starting from t_i, concretize the trace forward by conjuncting the states $t_{i+1}, \ldots, t_j$ with the values forced by propagation. In this way, the concretized state t_j might now become more concrete than t_i yielding a counterexample. One may further tailor the concretization process to yield a repeating state when possible via an appropriate SAT query.

Design	k generalized	k concrete	k modified
cubak	20	20	20
cujc128f	5	1	1
cutf2	9	12	5
cutq2	16	16	12
lmcs06dme2p0	4	5	4

TABLE I
VALUES OF k YIELDING VALID COUNTEREXAMPLES

Design	k-LIVENESS	BMC
cubak	295s	12084s
cuhanoi10	5s	3492s

TABLE II
k-LIVENESS WITH INTERNAL IC3 TRACE VS. BMC

IV. COUNTEREXAMPLES VIA k-LIVENESS

The k-LIVENESS algorithm of [1] proves properties of form FGq by bounding the number of times that q can become false: if there are no traces with more than k occurrences of $\neg q$, then on every trace q must eventually become true forever. The algorithm works by gradually increasing k until a proof is obtained.

When FGq does not hold, it is noted in [1] that a bounded counterexample trace for some k may be analyzed to see if it is a valid unbounded counterexample: given a finite system and large-enough k, there must be a trace with a repeated state. Though for a realistic system, it is stipulated that k would likely need to be impractically large.

Surprisingly, we find that the opposite is true: on 44 of the HWMCC'12 benchmarks with failing liveness properties, the traces returned by the underlying safety model checker exhibit a counterexample with reasonably-small values of k. Additionally, on most of these one may detect a counterexample for even smaller values of k by manipulating traces with $ConcretizeTentative$. A few selected results are presented in Table I.

As in [1], we have implemented k-LIVENESS on top of IC3/PDR. PDR minimizes proof obligations using ternary simulation [5], and thus directly yields generalized counterexamples for bounded property failures. Column 2 corresponds to the smallest value of k for which this generalized trace kept internally by IC3 exhibits a **generalized** counterexample. Column 3 corresponds to the smallest k for which the concretization of the trace from Column 2 using Algorithm 1 exhibits a **concrete** counterexample. The final column uses $ConcretizeTentative(i, j)$ on the trace of Column 2, for each tentative loop $\langle t_i, \ldots, t_j \rangle$ therein.

On cutf2 and lmcs06dme2p0, considering generalized traces detects counterexamples earlier due to removal of irrelevant state variables. On cujc128f, removing state variables from later timesteps precludes the detection of counterexamples. And on cutf2, partial concretization of the generalized trace yields a counterexample earlier than the other two methods.

Regarding impact on verification resources: on most of the *failing liveness* HWMCC'12 testcases, direct bounded model checking (BMC) often yields a counterexample with significantly lesser resources than k-LIVENESS augmented with our techniques. We note that the set of public liveness testcases is unfortunately quite small. Nonetheless, our techniques in cases are substantially faster than existing method such as BMC: see Table II. This offers a some evidence of the practical utility of our techniques on classes of complex problems.

V. Searching for Generalized Counterexamples

In this section we present an algorithm which directly searches for a minimal *propagating* generalized counterexample. This algorithm uses bounded model checking applied to a ternary-valued encoding of the netlist. This algorithm incrementally increases the unfolding depth n every time it proves that no generalized counterexample of length $\leq n$ exists.

For a given n, we seek a sequence $t_0, \ldots, t_n$ of generalized states and a sequence $i_0, \ldots, i_n$ of inputs so that the following conditions are satisfied:

1) t_0 contains an initial state;
2) for each $k \in [0..n-1]$ the assignments to t_k and to i_k alone imply t_{k+1};
3) $\exists m \in [0..n-1]$ such that $t_m \implies t_n$;
4) $\exists k \in [m..n-1]$ such that $t_k \implies \neg q$.

Note that every concrete lasso-shaped counterexample satisfies these conditions, thus if there are concrete counterexamples of length n, the suggested scheme will succeed with the value n or less.

Unfortunately, on the limited set of failing HWMCC'12 benchmarks, the minimal length of a propagating counterexample is the same as the minimal length of a concrete counterexample, and so the proposed scheme does not help. On the other hand, on contrived Examples 1-3, this algorithm detects generalized counterexamples of length 2 for any size of cnt, which not surprisingly may outperform by a large degree other techniques which search for a concrete counterexample.

VI. Related Work

The concept of minimizing counterexample traces has been explored extensively for a variety of purposes such as enhanced debugging, e.g. [2]. A related concept of generalizing a predecessor of a given state either by ternary simulation, via a SAT solver, or using quantifier elimination has also been widely explored, e.g. [6]. However, a significant distinction is that we we consider generalized counterexamples to *liveness* properties which can be significantly shorter than concrete counterexamples, and as such dedicated algorithms which search for *generalized* counterexamples may be developed.

The work of [4] addresses the topic of netlist transformations which preserve the existence of a liveness counterexample. For example, the cone-of-influence reduction combined with other netlist rewriting techniques can remove various signals from the netlist, thus possibly shortening lengths of counterexamples. However, netlist transformations apply to *all* time-frames and all possible traces, which does not offer the granularity of state-specific reductions enabled by our technique.

Cycle-dependent abstractions do allow the granularity of abstracting variables irrelevant *at a particular timestep*, though are typically only applicable as embedded in specific proof techniques (e.g., [7]). However, in general the existence of a counterexample on an abstracted model does not imply the existence of a counterexample on the concrete model. Additionally, this prior work does not address shortening of liveness counterexamples.

References

[1] K. Claessen and N. Sörensson, "A liveness checking algorithm that counts," in *FMCAD*, 2012.
[2] K.-H. Chang, V. Bertacco, and I. Markov, "Simulation-based bug trace minimization with BMC-based refinement," in *ICCAD*, 2005.
[3] Dong Wang et al., "Formal property verification by abstraction refinement with formal, simulation and hybrid engines," in *DAC*, 2001.
[4] J. Baumgartner and H. Mony, "Scalable liveness checking via property-preserving transformations," in *DATE*, 2009.
[5] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient implementation of property directed reachability," in *FMCAD*, 2011.
[6] P. Chauhan, E. M. Clarke, and D. Kroening, "Using SAT based image computation for reachability analysis," 2003.
[7] L. Zhang, M. Prasad, and M. Hsiao, "Interleaved invariant checking with dynamic abstraction," in *CHARME*, 2005.