Formula Slicing: Inductive Invariants from Preconditions

Egor George Karpenkov and David Monniaux

Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble, France
CNRS, VERIMAG, F-38000 Grenoble, France

Abstract. We propose a "formula slicing" method for finding inductive invariants. It is based on the observation that many loops in the program affect only a small part of the memory, and many invariants which were valid before a loop are still valid after.
Given a precondition of the loop, obtained from the preceding program fragment, we weaken it until it becomes inductive. The weakening procedure is guided by counterexamples-to-induction given by an SMT solver. Our algorithm applies to programs with arbitrary loop structure, and it computes the strongest invariant in an abstract domain of weakenings of preconditions. We call this algorithm "formula slicing", as it effectively performs "slicing" on formulas derived from symbolic execution.
We evaluate our algorithm on the device driver benchmarks from the International Competition on Software Verification (SV-COMP), and we show that it is competitive with the state-of-the-art verification techniques.

1 Introduction

In automated program verification, one crucial task is establishing *inductive invariants* for loops: properties that hold initially, and also by induction for any number of execution steps.

Abstract-interpretation-based approaches restrict the class of expressible invariants to a predefined *abstract domain*, such as intervals, octagons, or convex polyhedra (all of which can only express convex properties). Any *candidate invariants* which can not be expressed in the chosen abstract domain get over-approximated. Traditionally, this restriction applies at all program locations, but approaches such as *path focusing* [1] limit the precision loss only to loop heads, representing program executions between the loop-heads *precisely* using first-order formulas.

This is still a severe restriction: if a property flows from the beginning of the program to a loop head, and holds inductively after, but is not representable within the chosen abstract domain, it is discarded. In contrast, our idea exploits the insight that many loops in the program affect only a small part of the memory, and many invariants which were valid before the loop are still valid.

Consider finding an inductive invariant for the motivating example in Fig. 1. Symbolic execution up to the loop-head can precisely express all reachable states:

$$i = 0 \land (p \neq 0 \implies x \geq 0) \land (p = 0 \implies x < 0) \tag{1}$$

```
int x = input(), p = input();
if (p)
    assume(x >= 0);
else
    assume(x < 0);
for (int i=0; i < input(); i++) x *= 2;
```

Fig. 1: Motivating Example for Finding Inductive Weakenings.

Yet abstraction in a numeric convex domain at the loop head yields $i = 0$, completely losing the information that x is positive iff $p \neq 0$. Observe that this information loss is not *necessary*, as the sign of x stays invariant under the multiplication by a positive constant (assuming mathematical integers for the simplicity of exposition). To avoid this loss of precision, we develop a "formula slicing" algorithm which computes inductive *weakenings* of propagated formulas, allowing to propagate the formulas representing inductive invariants *across* loop heads. In the motivating example, formula slicing computes an inductive weakening of the initial condition in Eq. 1), which is $(p \neq 0 \implies x \geq 0) \land (p = 0 \implies x < 0)$, and is thus true at every iteration of the loop. The computation of inductive weakenings is performed by iteratively filtering out conjuncts falsified by *counterexamples-to-induction*, derived using an SMT solver. In the motivating example, transition $i = 1$ from $i = 0$ falsifies the constraint $i = 0$, and the rest of the conjuncts are inductive.

The formula slicing fixpoint computation algorithm is based on performing abstract interpretation on the lattice of conjunctions over a finite set of predicates. The computation starts with a seed invariant which *necessarily* holds at the given location on the first time the control reaches it, and during the computation it is iteratively weakened until inductiveness. The algorithm terminates within a polynomial number of SMT calls with the *smallest* invariant which can be expressed in the chosen lattice.

Contributions We present a novel insight for generating inductive invariants, and a method for creating a lattice of weakenings from an arbitrary formula describing the loop precondition using a *relaxed conjunctive normal form* (Def. 2) and best-effort quantifier elimination (Sec. 4).

We evaluate (Sec. 7) our implementation of the formula slicing algorithm on the "Device Drivers" benchmarks from the International Competition on Software Verification [2], and we demonstrate that it can successfully verify large, real-world programs which can not be handled with traditional numeric abstract interpretation, and that it is competitive with state of the art techniques.

Related Work The *Houdini* [3] algorithm mines the program for a set of predicates, and then finds the largest inductive subset, dropping the candidate non-inductive lemmas until the overall inductiveness is achieved. The optimality proof for *Houdini* is present in the companion paper [4]. A very similar algorithm is used by Bradley et Al. [5] to generate the inductive invariants from negations of the counter-examples to induction.

Inductive weakening based on counterexamples-to-induction can be seen as an algorithm for performing predicate abstraction [6]. Generalizing inductive weakening to *best abstract postcondition computation* Reps et al. [7] use the weakening approach for computing the best abstract transformer for any finite-height domain, which we also perform in Sec. 3.1.

Generating inductive invariants from a number of heuristically generated lemmas is a recurrent theme in the verification field. In *automatic abstraction* [8] a set of predicates is found for the simplified program with a capped number of loop iterations, and is filtered until the remaining invariants are inductive for the original, unmodified program. A similar approach is used for synthesizing bit-precise invariants by Gurfinkel et Al. [9].

The complexity of the inductive weakening and that of the related template abstraction problem are analyzed by Lahiri and Qadeer [10].

Overview We introduce the necessary background in Sec. 2 and the weakening algorithm in Sec. 3. We define the space of all used weakenings in Sec. 4. We develop the formula slicing algorithm for applying inductive weakening to real programs in Sec. 5, we describe our implementation and the required optimizations and improvements in Sec. 6, and we conclude with the empirical evaluation on the SV-COMP dataset in Sec. 7.

2 Background

2.1 Logic Preliminaries

We operate over first-order, existentially quantified logic formulas within an efficiently decidable theory. A set of all such formulas over free variables in X is denoted by $\mathcal{F}(X)$. Checking such formulas for satisfiability is NP-hard, but with modern SMT (*satisfiability modulo theories*) solvers these checks can often be performed very fast.

A formula is said to be an *atom* if it does not contain logical connectives (e.g. it is a comparison $x \leq y$ between integer variables), a *literal* if it is an atom or its negation, and a *clause* if it is a disjunction of literals. A formula is in *negation normal form* (NNF) if negations are applied only to atoms, and it is in *conjunctive normal form* (CNF) if it is a conjunction of clauses. For a set of variables X, we denote by X' a set where the prime symbol was added to all the elements of X. With $\phi[a_1/a_2]$ we denote the formula ϕ after all free occurrences of the variable a_1 have been replaced by a_2. This notation is extended to sets of variables: $\phi[X/X']$ denotes the formula ϕ after all occurrences of the free variables from X were replaced with corresponding free variables from X'. For brevity, a formula $\phi[X/X']$ may be denoted by ϕ'. We use the brackets notation to indicate what free variables can occur in a formula: e.g. $\phi(X)$ can only contain free variables in X. The brackets can be dropped if the context is obvious.

A formula $\phi(X)$, representing a set of program states, is said to be *inductive* with respect to a formula $\tau(X \cup X')$, representing a *transition*, if Eq. 2 is valid:

$$\phi(X) \wedge \tau(X \cup X') \implies \phi'(X') \tag{2}$$

That is, all transitions originating in ϕ end up in ϕ'. We can query an SMT solver for the inductiveness of $\phi(X)$ with respect to $\tau(X \cup X')$ using the constraint in Eq. 3, which is unsatisfiable iff $\phi(X)$ is inductive.

$$\phi(X) \wedge \tau(X \cup X') \wedge \neg\phi'(X') \tag{3}$$

For a quantifier-free formula ϕ inductiveness checking is co-NP-complete. However, if ϕ is existentially quantified, the problem becomes Π_2^p-complete. For efficiency, we shall thus restrict inductiveness checks to quantifier-free formulas.

2.2 Program Semantics and Verification Task

Definition 1 (CFA). A control flow automaton is a tuple $(\mathit{nodes}, \mathit{edges}, n_0, X)$, where *nodes* is a set of program control states, modelling the program counter, $n_0 \in \mathit{nodes}$ is a program starting point, and X is a set of program variables. Each edge $e \in \mathit{edges}$ is a tuple $(a, \tau(X \cup X'), b)$, modelling a possible transition, where $\{a, b\} \subseteq \mathit{nodes}$, and $\tau(X \cup X')$ is a formula defining the semantics of a transition over the sets of input variables X and output variables X'.

A non-recursive program in a C-like programming language can be trivially converted to a CFA by inlining functions, replacing loops and conditionals with guarded `goto`s, and converting guards and assignments to constraints over input variables X and output variables X'.

A *concrete data state* m of a CFA is a variable assignment $X \to \mathbb{Z}$ which assigns each variable an integral value.[1] The set of all concrete data states is denoted by $\mathcal{C}$. A set $r \subseteq \mathcal{C}$ is called a *region*. A formula $\phi(X)$ defines a region S of all states which it models ($S \equiv \{c \mid c \models \phi\}$). A set of all formulas over X is denoted by $\mathcal{F}(X)$. A *concrete state* c is a tuple (m, n) where m is a concrete data state, and $n \in \mathit{nodes}$ is a control state. A *program path* is a sequence of concrete states $\langle c_0, \ldots, c_n \rangle$ such that for any two consecutive states $c_i = (m_i, n_i)$ and $c_{i+1} = (m_{i+1}, n_{i+1})$ there exists an edge (n_i, τ, n_{i+1}) such that $m_i(X) \cup m_{i+1}(X') \models \tau(X \cup X')$. A concrete state $s_i = (m, n)$, and the contained node n, are both called *reachable* iff there exists a program path which contains s_i.

A *verification task* is a pair (P, n_e) where P is a CFA and $n_e \in \mathit{nodes}$ is an *error node*. A verification task is *safe* if n_e is not reachable. Safety is traditionally decided by finding a *separating* inductive invariant: a mapping from program locations to regions which is closed under the transition relation and does not contain the error state.

2.3 Invariant and Inductive Invariant

A set of concrete states is called a *state-space*, and is defined using a mapping from nodes to regions. A mapping $I : \mathit{nodes} \to \mathcal{F}(X)$ is an *invariant* if it contains

[1] The restriction to integers is for the simplicity of exposition, and is not present in the implementation.

all reachable states, and an *inductive invariant* if it is closed under the transition relation: that is, it satisfies the conditions for *initiation* and *consecution*:

$$\begin{aligned}\text{Initiation: } & I(n_0) = \top \\ \text{Consecution: } & \text{for all edges } (a, \tau, b) \in \mathit{edges}, \text{ for all } X, X' \\ & I(a)(X) \wedge \tau(X \cup X') \implies (I(b))'(X') \end{aligned} \tag{4}$$

Intuitively, the initiation condition dictates that the initial program state at n_0 (arbitrary contents of memory) is covered by I, and the consecution condition dictates that under all transitions I should map into itself. Similarly to Eq. 3, the consecution condition in Eq. 4 can be verified by checking one constraint for unsatisfiability using SMT for each edge in a CFA. This constraint is given in Eq. 5, which is unsatisfiable for each edge $(a, \tau, b) \in \mathit{edges}$ iff the consecution condition holds for I.

$$I(a)(X) \wedge \tau(X \cup X') \wedge \neg(I(b))'(X') \tag{5}$$

2.4 Abstract Interpretation Over Formulas

Program analysis by abstract interpretation [11] searches for inductive invariants in a given *abstract domain*: the class of properties considered by the analysis (e.g. upper and lower bounds on each numeric variable). The run of abstract interpretation effectively *interprets* the program in the given *abstract domain*, performing operations on the elements of an abstract domain instead of concrete values (e.g. the interval $x \in [1, 2]$ under the transition `x += 1` becomes $x \in [2, 3]$).

We define the abstract domain $\mathcal{D} \equiv 2^{\mathcal{L}} \cup \{\bot\}$ to be a powerset of the set of formulas $\mathcal{L} \subseteq \mathcal{F}(X)$ with an extra element $\bot$ attached. A *concretization* of an element $d \in \mathcal{D}$ is a conjunction over all elements of d, or a formula *false* for $\bot$.

Observe that $\mathcal{D}$ forms a complete lattice by using set operations of intersection and union as meet and join operators respectively, and using *syntactical* equality for comparing individual formulas. The syntactic comparison is an over-approximation as it does not take the formula semantics into account. However, this comparison generates a complete lattice of height $\|\mathcal{L}\| + 2$.

2.5 Large Block Encoding

The approach of large block encoding [12] for model checking, and the approach of path focusing [1] for abstract interpretation are based on the observation that by *compacting* a control flow and reducing a number of abstraction points, analysis precision and sometimes even analysis performance can be greatly improved. Both approaches utilize SMT solvers for performing abstraction afterwards.

A simplified version of compaction is possible by applying the following two rules to a CFA until a fixed point is reached:

– Two consecutive edges (a, s_1, b) and (b, s_2, c) with no other existing edge entering or leaving b get replaced by a new edge $(a, \exists \hat{X}.\, s_1[X'/\hat{X}] \wedge s_2[X/\hat{X}], c)$.

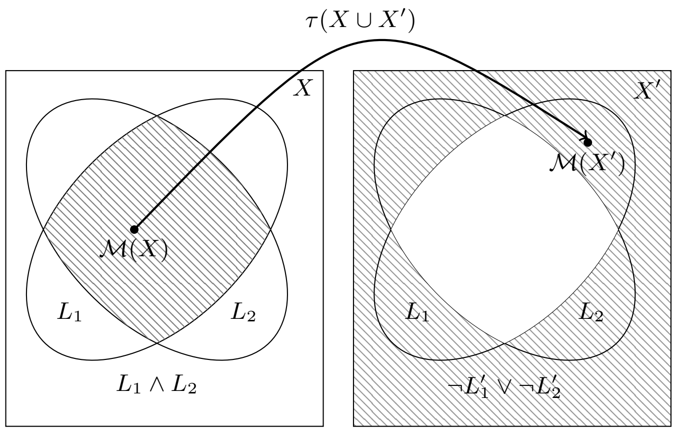


Fig. 2: Formula $\phi(X) \equiv L_1(X) \wedge L_2(X)$ is tested for inductiveness under $\tau(X \cup X')$. Model $\mathcal{M}$ identifies a counter-example to induction. From $\mathcal{M} \models \neg L_2'(X')$ we know that the lemma L_2 has to be dropped. As weakening progresses, the shaded region in the left box is growing, while the shaded region in the right box is shrinking, until there are no more counterexamples to induction.

– Two parallel edges (a, s_1, b) and (a, s_2, b) get replaced by $(a, s_1 \vee s_2, c)$.

In our approach, this pre-processing is used on the CFA obtained from the analyzed program.

3 Counterexample-to-Induction Weakening Algorithm

The approaches [3,5,8,9] mentioned in Sec. 1 are all based on using counterexamples to induction for filtering the input set of candidate lemmas. For completeness, we restate this approach in Alg. 1.

In order to perform the weakening without syntactically modifying ϕ during the intermediate queries, we perform *selector variables* annotation: we replace each lemma $l_i \in \phi$ with a disjunction $s_i \vee l_i$, using a fresh boolean variable s_i. Observe that if all selector variables are assumed to be false the annotated formula $\phi_{\text{annotated}}$ is equivalent to ϕ, and that assuming any individual selector s_i is equivalent to removing (replacing with $\top$) the corresponding lemma l_i from ϕ. Such an annotation allows us to make use of *incrementality* support by SMT solvers, by using the *solving with assumptions* feature.

Alg. 1 iteratively checks input formula ϕ for inductiveness using Eq. 3 (line 13). The solver will either report that the constraint is unsatisfiable, in which case ϕ is inductive, or provide a counterexample-to-induction represented by a model $\mathcal{M}(X \cup X')$ (line 14). The counterexample-driven algorithm uses $\mathcal{M}$ to find the set of lemmas which should be removed from ϕ, by removing the lemmas modelled by $\mathcal{M}$ in $\neg\phi'$ (line 20). The visualization of such a filtering step for a formula ϕ consisting of two lemmas is given in Fig. 2.

As shown in related literature [4], Alg. 1 terminates with the *strongest* possible weakening within the linear number of SMT calls with respect to $\|\phi_{\text{annotated}}\|$.

Algorithm 1 Counterexample-Driven Weakening.

```
1: Input: Formula $\phi(X)$ to weaken in RCNF, transition relation $\tau(X \cup X')$
2: Output: Inductive $\hat{\phi} \subseteq \phi$
3: ▷ Annotate lemmas with selectors, $S$ is a mapping from selectors to lemmas they annotate.
4: $S, \phi_{\text{annotated}} \leftarrow$ ANNOTATE($\phi$)
5: $T \leftarrow$ SMT solver instance
6: $query \leftarrow \phi_{\text{annotated}} \land \tau \land \neg\phi'_{\text{annotated}}$
7: Add $query$ to constraints in $T$
8: $assumptions \leftarrow \emptyset$
9: $removed \leftarrow \emptyset$
10: ▷ In the beginning, all of the lemmas are present
11: for all $(selector, lemma) \in S$ do
12:     $assumptions \leftarrow assumptions \cup \{\neg selector\}$
13: while $T$ is satisfiable with $assumptions$ do
14:     $\mathcal{M} \leftarrow$ model of $T$
15:     $assumptions \leftarrow \emptyset$
16:     for all $(selector, lemma) \in S$ do
17:         if $\mathcal{M} \models \neg lemma'$ or $lemma'$ is irrelevant to satisfiability then
18:             ▷ $lemma$ has to be removed.
19:             $assumptions \leftarrow assumptions \cup \{selector\}$
20:             $removed \leftarrow removed \cup \{lemma\}$
21:         else
22:             $assumptions \leftarrow assumptions \cup \{\neg selector\}$
23: ▷ Remove all lemmas which were filtered out
24: return $\phi[removed/\top]$
```

3.1 From Weakenings to Abstract Postconditions

As shown by Reps et Al. [7], the inductive weakening algorithm can be generalized for the abstract postcondition computation for any finite-height lattice.

For given formulas $\psi(X)$, $\tau(X \cup X')$, and $\phi(X)$ consider the problem of finding a weakening $\hat{\phi} \subseteq \phi$, such that all feasible transitions from ψ through τ end up in $\hat{\phi}$. This is an abstract postcondition of ψ under τ in the lattice of all weakenings of ϕ (Sec. 2.4). The problem of finding it is very similar to the problem of finding an inductive weakening, as similarly to Eq. 3, we can check whether a given weakening of ϕ is a postcondition of ψ under τ using Eq. 6,

$$\psi(X) \land \tau(X \cup X') \land \neg\phi'_{\text{annotated}}(X') \tag{6}$$

Alg. 1 can be adapted for finding the *strongest* postcondition in the abstract domain of weakenings of the input formula with very minor modifications. The required changes are accepting an extra parameter ψ, and changing the queried constraint (line 6) to Eq. 6. The found postcondition is indeed strongest [7].

4 The Space of All Possible Weakenings

We wish to find a *weakening* of a set of states represented by $\phi(X)$, such that it is inductive under a given transition $\tau(X \cup X')$. For a single-node CFA defined by initial condition ϕ and a loop transition τ such a weakening would constitute an *inductive invariant* as by definition of weakening it satisfies the initial condition and is inductive.

We start with an observation that for a formula in NNF replacing any subset of literals with $\top$ results in an over-approximation, as both conjunction and disjunction are monotone operators. E.g. for a formula $\phi \equiv (l_a \wedge l_b) \vee l_c$ such possible weakenings are $\top$, $l_b \vee l_c$, and $l_a \vee l_c$.

The set of weakenings defined in the previous paragraph is redundant, as it does not take the formula structure into account — e.g. in the given example if l_c is replaced with $\top$ it is irrelevant what other literals are replaced, as the entire formula simplifies to $\top$. The most obvious way to address this redundancy is to convert ϕ to CNF and to define the set of all possible weakenings as conjunctions over the subsets of clauses in ϕ_{CNF}. E.g. for the formula $\phi \equiv l_a \wedge l_b \wedge l_c$ possible weakenings are $l_a \wedge l_b$, $l_b \wedge l_c$, and $l_a \wedge l_c$. This method is appealing due to the fact that for a set of lemmas the *strongest* (implying all other possible inductive weakenings) inductive subset can be found using a linear number of SMT checks [5]. However (Sec. 2.1) polynomial-sized CNF conversion (e.g. Tseitin encoding) requires introducing existentially quantified boolean variables which make inductiveness checking Π_2^p-hard.

The arising complexity of finding inductive weakenings is inherent to the problem: in fact, the problem of finding *any* non-trivial ($\neq \top$) weakening within the search space described above is Σ_2^p-hard (see proof in Appendix A).

Thus instead we use an over-approximating set of weakenings, defined by all possible subsets of lemmas present in ϕ after the conversion to *relaxed conjunctive normal form*.

Definition 2 (Relaxed Conjunctive Normal Form (RCNF)). A formula $\phi(X)$ is in relaxed conjunctive normal form if it is a conjunction of quantifier-free formulas (lemmas).

For example, the formula $\phi \equiv l_a \wedge (l_b \vee (l_c \wedge l_d))$ is in RCNF. The over-approximation comes from the fact that non-atomic parts of the formula are grouped together: the only possible non-trivial weakenings for ϕ are l_a and $l_b \vee (l_c \wedge l_d)$, and it is impossible to express $l_a \wedge (l_b \vee l_c)$ within the search space.

We may abuse the notation by treating ϕ in RCNF as a set of its conjuncts, and writing $l \in \phi$ for a lemma l which is an argument of the parent conjunction of ϕ, or $\phi_1 \subseteq \phi_2$ to indicate that all lemmas in ϕ_1 are contained in ϕ_2, or $\|\phi\|$ for the number of lemmas in ϕ. For ϕ in RCNF we define a set of all possible *weakenings* as conjunctions over all sets of lemmas contained in ϕ. We use an existing, optimal counter-example based algorithm in order to find the *strongest* weakening of ϕ with respect to τ in the next section.

A trivially correct conversion to a relaxed conjunctive normal is to convert an input formula ϕ to a conjunction $\bigwedge\{\phi\}$. However, this conversion is not very interesting, as it gives rise to a very small set of weakenings: ϕ and $\top$. Consequently, with such a conversion, if ϕ is not inductive with respect to the transition of interest, no non-trivial weakening can be found. On the other extreme, ϕ can be converted to CNF explicitly using associativity and distributivity laws, giving rise to a very large set of possible weakenings. Yet the output of such a conversion is exponentially large.

We present an algorithm which converts ϕ into a polynomially-sized conjunction of lemmas. The following rules are applied recursively until a fixpoint is reached:

Flattening All nested conjunctions are flattened. E.g. $a \land (b \land c) \mapsto a \land b \land c$.

Factorization When processing a disjunction over multiple conjunctions we find and extract a common factor. E.g. $(a \land b) \lor (b \land c) \mapsto b \land (a \lor c)$.

Explicit expansion with size limit A disjunction $\bigvee L$, where each $l \in L$ is a conjunction, can be converted to a conjunction over disjunctions over all elements in the cross product over L. E.g. $(a \land b) \lor (c \land d)$ can be converted $(a \lor c) \land (a \lor d) \land (b \lor c) \land (b \lor d)$.

Applying such an expansion results in an exponential blow-up, but we only perform it if the resulting formula size is smaller than a fixed constant, and we limit the expansion depth to one.

Eliminating Existentially Quantified Variables The formulas resulting form large block encoding (Sec. 2.5) may have intermediate (neither input nor output), existentially bound variables. In general, existential quantifier elimination (with e.g. Fourier-Motzkin) is exponential. However, for many cases such as simple deterministic assignments, existential quantifier elimination is easy: e.g. $\exists t.\, x' = t + 3 \land t = x + 2$ can be trivially replaced by $x' = x + 5$ using substitution.

We use a two-step method to remove the quantified variables: we run a best-effort pattern-matching approach, removing the bound variables which can be eliminated in polynomial time, and in the second step we drop all the lemmas which still contain the existentially bound variables. The resulting formula is an over-approximation of the original one.

5 Formula Slicing: Overall Algorithm

We develop the *formula slicing* algorithm in order to apply the inductive weakening approach for generating inductive invariants in large, potentially non-reducible programs with nested loops.

"Classical" Houdini-based algorithms consist of two steps: *candidate* lemmas generation, followed by counterexample-to-induction-based filtering. However, in our case candidate lemmas representing postconditions depend on previous filtering steps, and careful consideration is required in order to generate *unique* candidate lemmas which do not depend on the chosen iteration order.

Abstract Reachability Tree In order to solve this problem we use abstract reachability tree [13] (ART) as a main datastructure for our algorithm. For the simplicity of notation we introduce the projection function π_i, which projects the i^{th} element of the tuple. An ART describes the current invariant candidate processed by the analysis for a fixed CFA $(\mathit{nodes}, \mathit{edges}, n_0, X)$, and is defined by a set of nodes T. Each node $t \in T$ is a triple, consisting of a CFA node $n \in \mathit{nodes}$, defining which location t corresponds to, an abstract domain element $d \in \mathcal{D}$, defining the reachable state space at t, and an optional backpointer $b \in (T \cup \{\emptyset\})$, defining the tree structure. The tree topology has to be consistent with the structure of the underlying CFA: node $a \in T$ can have a backpointer to the node $b \in T$ only if there exists an edge $(\pi_1(a), \_, \pi_1(b))$ in the CFA. The starting tree node t_0 is $(n_0, \top, \emptyset)$.

An ART is *sound* if the output of each transition over-approximates the strongest postcondition: that is, for each node $t \in T$ with non-empty backpointer $b = \pi_3(t)$, an edge $e = (\pi_1(b), \tau, \pi_1(t))$ must exist in *edges*, and the abstract domain element associated with t must over-approximate the strongest post-condition of b under τ. Formally, the following must hold: $\exists X.\, [\![\pi_2(b)]\!] \wedge \tau \implies [\![\pi_2(t)]\!]'$ (recall that priming is a renaming operation $[X/X']$). A node $b \in T$ is *fully expanded* if for all edges $(\pi_1(t), \tau, n) \subseteq \mathit{edges}$ there exists a node $t \in T$, where $\pi_1(t) = n$, and $\pi_2(t)$ over-approximates the strongest post-condition of $\pi_2(b)$ under τ. A node $(a, d_1, \_)$ *covers* another node $(a, d_2, \_)$ iff $[\![d_2]\!] \implies [\![d_1]\!]$. A sound labelled ART where all nodes are either fully expanded or covered represents an inductive invariant.

The transfer relation for the formula slicing is given in Alg. 3. In order to generate a successor for an element (n_a, d, b), and an edge (n_a, τ, n_b) we first traverse the chain of backpointers up the tree. If we can find a "sibling" element s where $\pi_1(s) = n_a$[2] by following the backpointers, we weaken s until inductiveness (line 4) relative to the new incoming transition τ, and return that as a postcondition. Such an operation effectively performs widening [11] to enforce convergence. Alternatively, if no such sibling exists, we convert $\exists X. \wedge \tau$ to RCNF form (line 6), and this becomes a new element of the abstract domain.

The main fixpoint loop performs the following calculation: for every leaf in the tree which is not yet expanded or covered, all successors are found using the transfer relation defined in Alg. 3, and for each newly created element, coverage relation is checked against all elements in the same partition. A simplified version of this standard fixpoint iteration on ART is given in Alg. 2.

Observe that our algorithm has a number of positive features. Firstly, because our main datastructure is an ART, in case of a counterexample we get a *path* to a property violation (though due to abstraction used, not all taken transitions are necessarily feasible, similarly to the *leaping counterexamples* of LoopFrog [14]). Secondly, our approach for generating initial candidate invariants ensures uniqueness, even in the case of a non-reducible CFA.

As a downside, tree representation may lead to the exponential state-space explosion (as a single node in a CFA may correspond to many nodes in an ART). However, from our experience in the evaluation (Sec. 7), with a good iteration order (stabilizing inner components first [15]) this problem does not occur in practice.

[2] In the implementation, the *sibling* is defined by a combination of callstack, CFA node and loopstack.

Algorithm 2 Formula Slicing: Overall Algorithm

```
Input: CFA $(nodes, edges, n_0, X)$
▷ Expanded.
$E \leftarrow \emptyset$
▷ Covered.
$C \leftarrow \emptyset$
$t_0 \leftarrow (n_0, \top, \emptyset)$
$T \leftarrow \{t_0\}$
while $\exists t \in (T \setminus E \setminus C)$ do
    ▷ Expand.
    for all edge $e \in edges$ where $\pi_1(e) = \pi_1(t)$ do
        $T \leftarrow T \cup \{$ TransferRelation$(e, t)$ $\}$
    $E \leftarrow E \cup \{t\}$
    ▷ Check Coverage.
    for all $t_1 \in (T \setminus C)$ where $\pi_1(t_1) = \pi_1(t)$ do
        if $[\![\pi_2(t_1)]\!] \implies [\![\pi_2(t)]\!]$ then
            $C \leftarrow C \cup \{t_1\}$
        if $[\![\pi_2(t)]\!] \implies [\![\pi_2(t_1)]\!]$ then
            $C \leftarrow C \cup \{t\}$
```

Algorithm 3 Formula Slicing: Postcondition Computation.

```
function TransferRelation(edge $e \equiv (n_a, \tau, n_b)$, state $t \equiv (n_a, d, b)$)
    sibling $s \leftarrow$ FindSibling$(b, n_0)$
    if $s \neq \emptyset$ then
        ▷ Abstract postcondition of $d$ under $\tau$ in weakenings of $s$ (Sec. 3.1).
        $e \leftarrow$ Weaken$(d,\ \tau \wedge n_b,\ s)$
    else
        ▷ Convert the current invariant candidate to RCNF.
        $e \leftarrow$ ToRCNF$([\![d]\!] \wedge \tau)$
    return $(n_b, e, t)$
function FindSibling(state $b$, CFA node $n$)
    if $\pi_1(b) = n$ then
        return $b$
    else if $\pi_3(b) = \emptyset$ then
        return $\emptyset$
    else
        return FindSibling$(\pi_3(b), n)$
```

5.1 Example Formula Slicing Run

Consider running formula slicing on the program in Fig. 3, which contains two nested loops. The corresponding edge encoding is given in Eq. 7:

```
int p, c, s=nondet(), x = 0, y = 0;
p = s ? 1 : 2;
while (nondet()) { // A(X)
    x++;
    c = 100;
    while (nondet()) { // B(X)
        if (p != 1 && p != 2) {
            c = 0;
        }
        y++;
    }
    assert(c == 100);
}
assert((s && p == 1) || (!s && p == 2));
```

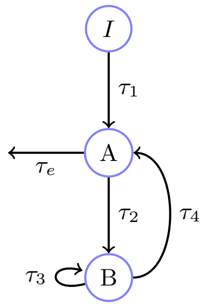


Fig. 3: Example Program with Nested Loops: Listing and CFA.

$$
\begin{aligned}
\tau_1 \equiv & x' = 0 \land y' = 0 \land (p' = 1 \land s' \lor p' = 2 \land \neg s') \\
\tau_2 \equiv & x' = x + 1 \land c' = 100 \\
\tau_3 \equiv & (\neg(p \neq 1 \land p \neq 2) \lor (p \neq 1 \land p \neq 2 \land c' = 0)) \\
& \land y' = y + 1 \land p' = p \\
\tau_4 \equiv & x' = x \land y' = y \land p' = p \land c' = c
\end{aligned} \tag{7}
$$

Similarly to Eq. 3, we can check candidate invariants $A(X), B(X)$ for inductiveness by posing an SMT query shown in Eq. 8. The constraint in Eq. 8 is unsatisfiable iff $\{A : A(X), B : B(X)\}$ is an inductive invariant (Sec. 2.3).

$$
\exists X \cup X' \bigvee \begin{array}{r}
\tau_1(X') \land \neg A(X') \\
A(X) \land \tau_2(X \cup X') \land \neg B(X') \\
B(X) \land \tau_3(X \cup X') \land \neg B(X') \\
B(X) \land \tau_4(X \cup X') \land \neg A(X')
\end{array} \tag{8}
$$

Eq. 8 is unsatisfiable iff *all* of the disjunction arguments are unsatisfiable, and hence the checking can be split into multiple steps, one per analyzed edge. Each postcondition computation (Alg. 3) either generates an initial seed invariant candidate, or picks one argument of Eq. 8, and weakens the right hand side until the constraint becomes unsatisfiable. Run of the formula slicing algorithm on the example is given below:

- Traversing τ_1, we get the initial candidate invariant $I(A) \leftarrow \bigwedge\{x = 0, y = 0, p = 1 \lor p = 2, s \implies p = 1\}$.
- Traversing τ_2, the candidate invariant for B becomes $I(B) \leftarrow \bigwedge\{x = 1, y = 0, p = 1 \lor p = 2, s \implies p = 1, c = 100\}$.
- After traversing τ_3, we weaken the candidate invariant $I(B)$ by dropping the lemma $y = 0$ which gives rise to the counterexample to induction (y gets incremented). The result is $\bigwedge\{x = 1, p = 1 \lor p = 2, s \implies p = 1, c = 100\}$, which is inductive under τ_3.

– The edge τ_4 is an identity, and the postcondition computation results in lemmas $x = 0$ and $y = 0$ dropped from $I(A)$, resulting in $\bigwedge\{y = 0, p = 1 \vee p = 2, s \implies p = 1\}$.
– After traversing τ_2, we obtain the weakening of $I(A)$ by dropping the lemma $x = 1$ from $I(B)$, resulting in $\bigwedge\{p = 1 \vee p = 2, s \implies p = 1, c = 100\}$.
– Finally, the iteration converges, as all further postconditions are already covered by existing invariant candidates. Observe that the computed invariant is sufficient for proving the asserted property.

6 Implementation

We have developed the Slicer tool, which runs the formula slicing algorithm on an input C program. Slicer performs inductive weakenings using the Z3 [16] SMT solver, and best-effort quantifier elimination using the `qe-light` Z3 tactic. The source code is integrated inside the open-source verification framework CPAchecker [17], and the usage details are available at `http://slicer.metaworld.me`. Our tool can analyze a verification task (Sec. 2.2) by finding an inductive invariant and reporting `true` if the found invariant *separates* the initial state from the error property, and `unknown` otherwise.

We have implemented the following optimizations:

Live Variables We precompute live variables, and the candidate lemmas generated during RCNF conversion (Alg. 3, line 6) which do not contain live variables are discarded.

Non-Nested Loops When performing the inductive weakening (Alg. 3, line 4) on the edge (N, τ, N) we annotate and weaken the candidate invariants on both sides (without modifications described in Sec. 3.1), and we cache the fact that the resulting weakening is inductive under τ.

CFA Reduction We pre-process the input CFA and we remove all nodes from which there exists no path to an error state.

6.1 Syntactic Weakening Algorithm

A syntactic-based approach is possible as a faster and less precise alternative which does not require SMT queries. For an input formula $\phi(X)$ in RCNF, and a transition $\tau(X \cup X')$, syntactic weakening returns a subset of lemmas in ϕ, which are not *syntactically modified* by τ: that is, none of the variables are modified or have their address taken. For example, the lemma $x > 0$ is not syntactically modified by the transition $y' = y + 1 \wedge x \geq 1$, but it is modified by $x' = x + 1$.

7 Experiments and Evaluation

We have evaluated the formula slicing algorithm on the "Device Drivers" category from the International Competition on Software Verification (SV-COMP) [2]. The dataset consists of 2120 verification tasks, of which 1857 are designated as *correct* (the error property is unreachable), and the rest admit a counter-example. All the experiments were performed on Intel Xeon E5-2650 at 2.00 GHz, and limits of 8GB RAM, 2 cores, and 600 seconds CPU time per program. We compare the following three approaches:

Slicer-CEX (rev `21098`) Formula slicing algorithm running counterexample-based weakening (Sec. 3).

Slicer-Syntactic Same, with syntactic weakening (Sec. 6.1).

Predicate Analysis (rev `21098`) Predicate abstraction with interpolants [18], as implemented inside CPAchecker [19]. We have chosen this approach for comparison as it represents state-of-the-art in model checking, and was found especially suitable for analyzing device drivers.

PAGAI [20] (git hash `e44910`) Abstract interpretation-based tool, which implements the path focusing [1] approach.

Unabridged experimental results are available at `http://slicer.metaworld.me`.

In Tab. 1 we show overall precision and performance of the four compared approaches. As formula slicing is over-approximating, it is not capable of finding counterexamples, and we only compare the number of produced safety proofs.

From the data in the table we can see that predicate analysis produces the most correct proofs. This is expected since it can generate new predicates, and it is *driven* by the target property. However, formula slicing and abstract interpretation have much less timeouts, and they do not require target property annotation, making them more suitable for use in domains where a single error property is not available (advanced compiler optimizations, multi-property verification, and boosting another analysis by providing an inductive invariant). The programs verified by different approaches are also different, and formula slicing verifies 22 programs predicate analysis could not.

The performance of the four analyzed approaches is shown in the quantile plot in Fig. 4a. The plot shows that predicate analysis is considerably more time consuming than other analyzed approaches. Initially, PAGAI is much faster than other tools, but around 15 seconds it gets overtaken by both slicing approaches. Though the graph seems to indicate that PAGAI overtakes slicing again around 100 seconds, in fact the bend is due to out of memory errors.

The quantile plot also shows that the time taken to perform inductive weakening does not dominate the overall analysis time for formula slicing. This can be seen from the small timing difference between the syntactic and counterexample-based approaches, as the syntactic approach does not require querying the SMT solver in order to produce a weakening.

Finally, we present data on the number of SMT calls required for computing inductive weakenings in Fig. 4b. The distribution shows that the overwhelming majority of weakenings can be found within just a few SMT queries.

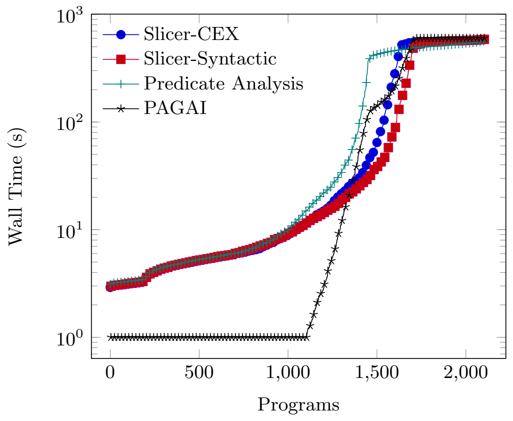
8 Conclusion and Future Work

We have proposed a "formula slicing" algorithm for efficiently finding potentially disjunctive inductive invariants in programs, which performs abstract interpretation in the space of weakenings over the formulas representing the "initial" state. We have demonstrated that it could verify many programs other approaches could not, and that the algorithm can be run on real programs.

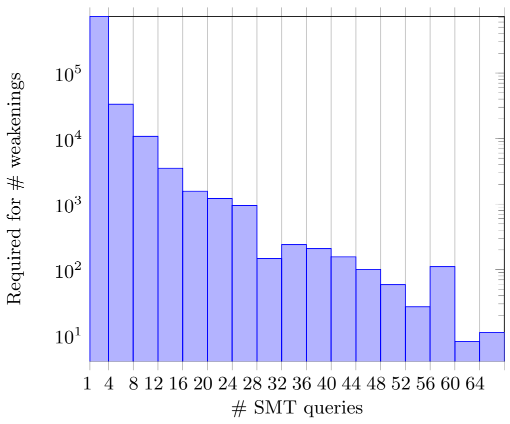
The motivation for our approach is addressing the limitation of abstract interpretation which forces it to perform abstraction after each analysis step,

Tool	# proofs	# incorrect	# timeouts	# memory outs
Slicer-CEX	1253	0	475	0
Slicer-Syntactic	1166	0	407	0
Predicate Analysis	1301	0	657	0
PAGAI	1214	3	409	240

Table 1: Evaluation results. The "# incorrect" column shows the number of safety proofs the tool has produced where the analyzed program admitted a counterexample.



(a) Quantile plot showing performance of the compared approaches. Shows analysis time for each benchmark, where the data series are sorted by time separately for each tool. For readability, the dot is drawn for every 20th program, and the time is rounded up to one second.



(b) Distribution of the number of iterations of inductive weakening (Sec. 3) required for convergence across all benchmarks. Horizontal axis represents the number of SMT calls required for convergence of each weakening, and vertical axis represents the count of the number of such weakenings.

which often results in a very rough over-approximation. Thus we believe our method is well-suited for augmenting numeric abstract interpretation.

As with any new inductive invariant generation technique, a possible future work is investigating whether formula slicing can be used for increasing the performance and precision of other program analysis techniques, such as k-induction, predicate abstraction or property-directed reachability. An obvious approach would be feeding the invariants generated by formula slicing to a convex analysis running abstract interpretation or policy iteration [21].

Furthermore, the inductive weakening approach could also be used for the generalization of the k-induction algorithm over multiple properties. If we check a set of properties P for inductiveness under the loop transition τ, and $\bigwedge P$ is not inductive, the weakening can find the largest inductive subset.

References

1. D. Monniaux and L. Gonnord, “Using bounded model checking to focus fixpoint iterations,” in *SAS*. Springer, 2011.
2. D. Beyer, “Reliable and reproducible competition results with benchexec and witnesses (Report on SV-COMP 2016),” in *TACAS*. Springer, 2016.
3. C. Flanagan and K. R. M. Leino, “Houdini, an annotation assistant for ESC/Java,” in *FME*, 2001, pp. 500–517.
4. C. Flanagan, R. Joshi, and K. R. M. Leino, “Annotation inference for modular checkers,” *Information Processing Letters*, 2001.
5. A. R. Bradley and Z. Manna, “Checking safety by inductive generalization of counterexamples to induction,” in *FMCAD*, 2007, pp. 173–180.
6. S. Graf and H. Saïdi, “Construction of abstract state graphs with PVS,” in *CAV*, 1997, pp. 72–83.
7. T. Reps, M. Sagiv, and G. Yorsh, “Symbolic implementation of the best transformer,” in *VMCAI*, 2004.
8. A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke, “Automatic abstraction in SMT-based unbounded software model checking,” in *CAV*, 2013, pp. 846–862.
9. A. Gurfinkel, A. Belov, and J. Marques-Silva, “Synthesizing safe bit-precise invariants,” in *TACAS*, 2014, pp. 93–108.
10. S. K. Lahiri and S. Qadeer, “Complexity and algorithms for monomial and clausal predicate abstraction,” in *CADE*, 2009, pp. 214–229.
11. P. Cousot and R. Cousot, “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints,” in *POPL*, 1977, pp. 238–252.
12. D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani, “Software model checking via large-block encoding,” in *FMCAD*, 2009, pp. 25–32.
13. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, “The software model checker Blast,” *STTT*, vol. 9, no. 5-6, pp. 505–525, 2007.
14. D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and C. M. Wintersteiger, “Loop summarization using abstract transformers,” in *ATVA*, 2008, pp. 111–125.
15. F. Bourdoncle, “Efficient chaotic iteration strategies with widenings,” in *Formal Methods in Programming and Their Applications*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1993, vol. 735, pp. 128–141.
16. L. M. de Moura and N. Bjørner, “Z3: an efficient SMT solver,” in *TACAS*, 2008, pp. 337–340.
17. D. Beyer and M. E. Keremoglu, “CPAchecker: A tool for configurable software verification,” in *CAV*, 2011, pp. 184–190.
18. K. L. McMillan, “Lazy abstraction with interpolants,” in *CAV*, 2006, pp. 123–136.
19. D. Beyer, M. E. Keremoglu, and P. Wendler, “Predicate abstraction with adjustable-block encoding,” in *FMCAD*, 2010, pp. 189–197.
20. J. Henry, D. Monniaux, and M. Moy, “PAGAI: A path sensitive static analyser,” *Electr. Notes Theor. Comput. Sci.*, vol. 289, pp. 15–25, 2012.
21. E. G. Karpenkov, D. Monniaux, and P. Wendler, “Program analysis with local policy iteration,” in *VMCAI*. Springer, 2016, pp. 127–146.
22. L. J. Stockmeyer, “The polynomial-time hierarchy,” *Theoretical Computer Science*, vol. 3, no. 1, pp. 1–22, 1976.

A Complexity of Finding a Non-Trivial Inductive Weakening Over Literals

As we have mentioned in Sec. 4, a more expressive space of weakenings over formulas is to consider replacing any subset of literals with $\top$ after a NNF conversion. In this appendix we show that it leads to a number of undesirable properties, including the absence of *strongest* inductive weakening (Ex. 1), and Σ_2^p complexity for finding any non-trivial inductive weakening (Thm. 1).

Example 1 (No Strongest Inductive Weakening). Consider a program over four Boolean variables a, b, c, d and the transition relation $\tau \equiv a \wedge b \wedge c \wedge d \wedge \neg a' \wedge b' \wedge \neg c' \wedge d'$ (the only possible transition is from $a \wedge b \wedge c \wedge d$ to $\neg a \wedge b \wedge \neg c \wedge d$). Consider finding the weakening of $\phi \equiv (a \wedge b) \vee (c \wedge d)$, Both the $\{a\}$-weakening $(b \vee (c \wedge d))$ and the $\{c\}$-weakening $((a \wedge b) \vee d)$ are inductive, but their intersection $(a \wedge b) \vee (b \wedge d) \vee (c \wedge d)$ (obviously inductive) is not a weakening of ϕ and there is no inductive weakening stronger than either of these.

Theorem 1 (Σ_2^p-completeness). *The problem of deciding, given quantifier-free SMT formulas $\phi(X)$ and $\tau(X \cup X')$, whether there exists a non-trivial ($\not\equiv \top$) weakening of ϕ that is inductive with respect to τ is Σ_2^p-complete.*

Proof (Belonging to Σ_2^p). Let S be some subset of literals of ϕ. Let $\hat{\phi}$ be the weakening of ϕ where all literals in S are replaced with $\top$. Checking that $\hat{\phi}$ is inductive with respect to τ is in co-NP, therefore the problem of finding a non-trivial $\hat{\phi}$ is in Σ_2^p

We show completeness by constructing from an arbitrary closed $\exists^*\forall^*$ formula ψ a loop τ and a precondition I such that the existence of a non-trivial ($\not\equiv \top$) weakening of the precondition is equivalent to the truth of ψ. Without loss of generality, let ψ have m Boolean variables $x_0, \dots, x_{m-1}$ bound by the existential quantifier and n Boolean variables $y_0, \dots, y_{n-1}$ bound by the universal one:

$$\begin{aligned} \psi \equiv & \exists x_0, \dots, x_{m-1}. \\ & \forall y_0, \dots, y_{n-1}. G(x_0, \dots, x_{m-1}, y_0, \dots, y_{n-1}) \end{aligned} \tag{9}$$

Let us denote the bitvector $(x_0, \dots, x_{m-1})$ as X and the bitvector $(y_0, \dots, y_{n-1})$ as Y. Let $enc : \mathbb{B}^m \to [0, 2^m - 1]$ denote the function for standard integer encoding of the X bitvector, x_0 being the lowest-order bit and x_{m-1} the highest-order one. Let $succ : \mathbb{B}^m \setminus \{\top^m\} \to \mathbb{B}^m$ be the successor function such that $enc(succ(X)) = 1 + enc(succ(X))$, which is only defined for non-overflowing values.

Now we define the transition system over the set of boolean variables X and the overflow bit o. Let the initial state $I(X, o)$ be $X = \bot \wedge o = \bot$, and let the transition relation $\tau(X, X', o, o')$ to be:

$$\begin{aligned} & \big(\neg(\forall Y. G(X, Y)) \wedge \\ & ((X \neq \top \wedge X' = succ(X) \wedge o' = o) \vee (X = \top \wedge o' = \top))\big) \\ & \bigvee \big(X' = X \wedge o' = o\big) \end{aligned} \tag{10}$$

```
bitvector X = ⊥;
boolean o = ⊥;
while(nondet()) {
    // Non-deterministic choice.
    bitvector Y = nondet();
    if (not G(X,Y)) {
        if (X == ⊤) {
            // Set the overflow
            // bit.
            o = ⊤;
            X = nondet();
        } else {
            // Increment a given
            // bitvector.
            X = succ(X);
        }
    }
}
```

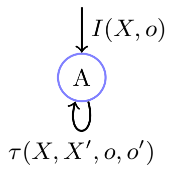


Fig. 4: Counter Program and Transition System

In plain terms, the transition relation may increment X as long as it is not overflowing and the guard can be falsified for some Y, and X is forced to stay constant on overflow or when it reaches some $\hat{X}$ such that $\forall Y.G(\hat{X}, Y)$. Initialization and transition relation for the transition system, and the corresponding program are shown in Fig. 4.

Lemma 1. *There exists a non-trivial ($\not\equiv \top$) inductive invariant for the program in Fig. 4 if and only if ψ (Eq. 9) is satisfiable.*

Observe that τ can be satisfied for all possible values of X by a suitable choice of X'. Let $f(X)$ be the largest (under *enc*) possible value of X' which satisfies $\tau(X, X', o, o')$.

Proof. Sufficient Condition. Assume ψ is satisfiable for some $\hat{X}$. Then $\hat{X}$ is a fixed point under f (as it satisfies G for all possible values of Y). Consider the set of values defined by $R \equiv \neg o \wedge enc(X) \leq \hat{X}\}$. It is inductive, since the largest value in R set maps to itself under f, and all other values map to the "next" (under *enc*) value in R. It is also non-trivial, since the bit o is defined not to be $\top$.

Proof. Necessary Condition. Assume there exists a non-trivial inductive invariant for the program in Fig. 4. At every transition, X either stays constant or is incremented by 1. Since we have assumed the existence of a non-trivial inductive invariant, there exists $\hat{X}$ such that it is a fixpoint under f and $enc(\hat{X}) \leq 2^m - 1$ (otherwise the entire state space is reachable, and the only possible inductive invariant is $\top$). This is only possible if $\forall Y.G(\hat{X}, Y)$ (otherwise $\hat{x}$ may be incremented). But this is exactly the condition for ψ being satisfiable.

Corollary 1. *For every non-trivial inductive invariant of the program in Fig. 4 there exists some $\hat{X}$ such that $\{X \mid enc(X) < enc(\hat{X})\}$ is inductive. Furthermore, the reachable state space is exactly all X smaller (under enc) than $\hat{X}$, and $\{X \mid X \neq \hat{X}\}$ is inductive (as the states larger than $\hat{X}$ are not reachable).*

Now consider finding inductive (with respect to τ Fig. 4) weakenings of the following formula ϕ:

$$\phi \equiv \bigvee (x_i \wedge \neg x_i) \tag{11}$$

Each x_i represents i'th bit of X. Observe that for any $\hat{X} \in [0, 2^m - 1]$, we can weaken ϕ to be equivalent to $X \neq \hat{X}$, by making a suitable weakening choice for every i'th bit of $\hat{X}$ (if the i-th bit in $\hat{X}$ is $\bot$ we replace $\neg x_i$ by $\top$, if it is $\top$ we replace x_i by $\top$).

From Corollary 1 we know that for every non-trivial inductive invariant there exists $\hat{X}$, s.t. the set of all X not equal to $\hat{X}$ is inductive. Thus if a non-trivial inductive invariant exists, there exists a non-trivial inductive weakening of ϕ. In Lemma 1 we have shown that deciding the existence of a non-trivial inductive invariant is as hard as deciding the satisfiability of an arbitrary $\exists^*\forall^*$ formula ψ, thus deciding an existence of a non-trivial inductive weakening is as hard as well.

Proof (Σ_2^p-completeness). Membership in Σ_2^p is proved in Lemma 1. Reduction from the Σ_2^p-complete problem is done from deciding the truth of $\exists^*\forall^*$ propositional formulas [22, Th. 4.1]. Transforming G into τ can be done within a logarithmic working space.

Relationship to Template Abstraction Complexity Lahiri and Qadeer [10] consider the problem of *template abstraction*: given a precondition, a post-condition, a transition relation and a formula $\phi(C, X)$, C and X being sets of Boolean variables, check whether an appropriate choice of C makes ϕ an inductive invariant. They show this problem to be Σ_2^p-complete as well. Our class of problems is a strict subset of theirs (our weakening problems can be immediately translated into template abstraction problems, but not all template abstraction problems correspond to weakenings), but we still show completeness.