Lazy Annotation Revisited

MSR-TR-2014-65

Kenneth L. McMillan
Micrsoft Research

Abstract

Lazy Annotation is a method of software model checking that performs a backtracking search for a symbolic counterexample. When the search backtracks, the program is annotated with a learned fact that constrains future search. In this sense, the method is closely analogous to conflict-driven clause learning in SAT solvers.

In this paper, we develop several improvements to the basic Lazy Annotation approach. The resulting algorithm is compared both conceptually and experimentally to two approaches based on similar principles but using different learning strategies: unfolding-based Bounded Model Checking and Property-Driven Reachability.

1 Introduction

Lazy Annotation is a method of software model checking motivated by conflict-driven clause learning (CDCL) in Boolean satisfiability (SAT) solvers. It performs a backtracking search for a symbolic execution of a program that violates a safety property. When the search reaches a conflict, it backtracks, annotating the program with a learned fact that constrains future search. As in CDCL, the learned fact is derived as a Craig interpolant.

In this paper, we develop several improvements to the basic Lazy Annotation approach. Among other things, we adapt Lazy Abstraction to large-block encodings [6], allowing us to exploit the power of modern satisfiability modulo theories (SMT) solvers. We compare the resulting algorithm to unfolding-based Bounded Model Checking and to Property-Driven Reachability. These methods share the general approach of conflict-driven learning, but differ in their search and learning strategies. Our goal will be to clarify these distinctions conceptually, and to test empirically the relative strengths of the different strategies. In particular, we will try to answer two questions:

1. Whether structured or unstructured search is more effective, and
2. What characterizes an effective conflict learning strategy, in terms of reducing bounded search and converging to an unbounded solution.

Related work. The basic idea of Lazy Annotation was introduced by Jaffar *et al.* [17] in the context of Constraint Logic Programming (CLP). A more general approach, handling richer theories and recursive procedures, was introduced by the author [21], along with the name Lazy Annotation. Here, we adopt the latter approach, in particular its use of proof-based interpolants and its method of inferring unbounded proofs from bounded proofs. Following Jaffar *et al.*, we work within the CLP framework, but generalize from simple linear arithmetic constraints to constraints in full first-order logic, exploiting the strength of modern SMT solvers. Moreover, we will allow clauses with multiple sub-goals (called the *nonlinear case* in [16]). Thus we can verify, for example, recursive procedural programs.

The IC3 method of Bradley [9] is similar at a high level to Lazy Annotation. Both methods perform bounded verification for increasing bounds, until an inductive invariant can be inferred from the bounded proof. Both methods generate symbolic goal states that are known to reach an error. These are refuted in a bounded sense by computing interpolants (in a sense we will define). Apart from various optimizations, the primary difference is in the particular strategy for computing these interpolants, an issue we will study in detail.

The class of algorithms based on IC3 has been called Property-Driven Reachability [12]. Although PDR originally applied only to propositional logic and the linear case, the approach was later extended to the non-linear case and richer logics [16] making it suitable for software model checking. Cimatti and Griggio give a hybrid approach applying PDR to software model checking [10]. Propositional PDR can also be applied to software via predicate abstraction [16, 11].

Another related approach was introduced recently by Bayless *et al.* [5] for the propositional linear case. In addition, there are various other CDCL-like methods [15, 23] to which the conclusions of this study may be relevant.

2 Informal discussion of Lazy Annotation

To give an intuitive explanation of the basic search and learning strategy of Lazy Annotation (LA in the sequel) we first consider the special case of transition systems (equivalently, imperative programs with a single loop). This will allow us to use the familiar vocabulary of transition systems, and to compare approaches more easily.

We model a transition system using the following *constrained Horn clauses*:

$$I(\bar{x}) \Rightarrow R(\bar{x}) \quad (1)$$

$$R(\bar{x}) \wedge T(\bar{x}, \bar{x}') \Rightarrow R(\bar{x}') \quad (2)$$

Here, $\bar{x}$ is a vector of variables representing the program state. The free variables are considered to be universally quantified in these clauses (a convention we will use in the remainder of the paper). The predicate I is a fixed set of initial states, while T is a fixed transition relation. Predicate R represents an *unknown* inductive invariant for which we wish to solve. The solution must satisfy the *query* formula $R(\bar{x}) \Rightarrow S(\bar{x})$, where S is a fixed set of safe states. Any such R constitutes a proof that our transition system is safe (that is, no initial state can reach a non-safe state via any sequence of transitions).

Our strategy for finding a solution for R will be to search for a refutation. A refutation takes the form of path in the transition system from an initial state to a non-safe state. As we will see, such a path corresponds to a ground derivation of a contradiction using our clauses.

As in bounded model checking, we search for a refutation path by *unwinding* the system k steps. This gives the following set of clauses:

$$\begin{aligned} I(\bar{x}) &\Rightarrow R_0(\bar{x}) \\ R_0(\bar{x}) \wedge T(\bar{x}, \bar{x}') &\Rightarrow R_1(\bar{x}') \\ &\dots \\ R_{k-1}(\bar{x}) \wedge T(\bar{x}, \bar{x}') &\Rightarrow R_k(\bar{x}') \\ R_k(\bar{x}) &\Rightarrow S(\bar{x}) \end{aligned}$$

A refutation for this acyclic (or non-recursive) set of clauses is a transition sequence that violates safety after exactly k steps. Correspondingly, a solution for $R_0 \dots R_k$ represents a proof that there is no such path. If we find a solution of the k-step unwound system, we can then attempt to derive from it a solution for the original cyclic system.

During our search for a refutation path, we maintain a candidate solution for $R_0 \dots R_k$ called the *annotation*. The annotation of R_i is an over-approximation of the set of reachable states of the system after i steps. We require that the

annotation be inductive. That is, it must satisfy all the Horn clauses, but not necessarily the query (unlike in PDR, we don't require that the annotation be an expanding chain).

Our search takes the unsafe states as an initial goal and symbolically executes the system backward from this goal. At each step we narrow the search by making a *decision*. A decision is simply an arbitrary constraint on our symbolic path. The search reaches a *conflict* when the goal (the symbolic path) becomes unsatisfiable. In this case, we backtrack, undoing the most recent decision. In the process we *learn* an annotation that prevents us from making the same decision in the future. The learned annotation is computed as an interpolant.

A *search goal* is a conjunction of facts to be derived and constraints to be satisfied. Our initial goal is $R_k(\bar{x}_k) \wedge \neg S(\bar{x}_k)$. That is, we wish to derive an unsafe state reachable in k steps. A backward step in the search corresponds to *resolution* of the goal with a clause. Thus, in the first step, we resolve the goal with the clause $R_{k-1}(\bar{x}) \wedge T(\bar{x}, \bar{x}') \Rightarrow R_k(\bar{x}')$ to obtain the new goal $R_{k-1}(\bar{x}_{k-1}) \wedge T(\bar{x}_{k-1}, \bar{x}_k) \wedge \neg S(\bar{x}_k)$. This goal represents a state reachable in $k-1$ steps that can reach an unsafe state in one step. As we perform resolution steps, our goal represents execution paths of increasing length.

Now suppose a goal is satisfiable in the current annotation. That is, when we substitute the annotation for R_i into the goal, the resulting formula is satisfiable. This means we can reach the error from R_i. We then make a decision, adding an arbitrary constraint to the goal. Decisions prevent the goal from becoming overly complex as we perform resolution steps. A decision typically constrains the most recent execution step. Thus, if our goal is $R_i(\bar{x}_i) \wedge T(\bar{x}_i, \bar{x}_{i+1}) \wedge \cdots \neg S(\bar{x}_k)$, it becomes $R_i(\bar{x}_i) \wedge D_i(\bar{x}_i, \bar{x}_{i+1}) \wedge T(\bar{x}_i, \bar{x}_{i+1}) \wedge \cdots \neg S(\bar{x}_k)$. Here D_i is a fixed predicate representing the i-th decision. The choice is D_i is arbitrary, but we require that the goal remain satisfiable.

Because decisions constrain the search, we may find after making a resolution step that the goal is unsatisfiable. At this point we are in conflict, and must backtrack to the previous goal, undoing one resolution step and one decision. In the process, we strengthen the annotation so that the last decision becomes infeasible. Our strengthening must be a value of R_{i+1} such that the resolved clause is true, that is,

$$R_i(\bar{x}_i) \wedge T(\bar{x}_i, \bar{x}_{i+1}) \Rightarrow R_{i+1}(\bar{x}_{i+1}) \tag{3}$$

and such that the prior decision is infeasible, that is, such that

$$R_{i+1}(\bar{x}_{i+1}) \wedge D_{i+1}(\bar{x}_{i+1}, \bar{x}_{i+2}) \wedge T(\bar{x}_{i+1}, \bar{x}_{i+2}) \wedge \cdots \neg S(\bar{x}_k) \tag{4}$$

is unsatisfiable. The reader may recognize that the formula $R_{i+1}(\bar{x}_{i+1})$ is an *interpolant* between two parts of the infeasible goal. We can compute such an interpolant from a proof of unsatisfiability of the goal, provided the proof system admits feasible interpolation [20]. The new annotation forces a different decision after backtracking.

LA can terminate in one of two ways. After resolving on R_0, the goal contains no facts to derive. In this case, it is a feasible BMC formula representing a path

from initial to unsafe states. On the other hand, if the initial goal becomes unsatisfiable (that is, if R_k implies S under the annotation) then we have a proof that no unsafe state is reachable in k steps.

We can then use the annotation of the bounded unwinding as a hint in constructing a solution of the original cyclic problem. We will will refer to this as the *convergence phase*. In [21], the convergence approach was to start with all of the conjuncts of annotations R_i of the bounded problem and apply the Houdini algorithm [13] to reduce these to their maximal inductive subset. If this yields a solution, we are done, otherwise we increase the unfolding depth k. The convergence phase in PDR is similar: annotations are propagated forward until a fixed point is reached. There are many other possibilities, however, including Lazy Abstraction with Interpolants (LAWI) and predicate abstraction. Here, we will focus on solving the bounded problem and leave aside the largely orthogonal question of convergence.

3 Formal description of Lazy Annotation

We will now formalize LA as an algorithm, extending it from simple transition systems to the general case of constrained Horn clauses.

We use standard first-order logic over a signature Σ of function and predicate symbols of defined arities. We use ϕ, ψ for formulas, P, Q, R for predicate symbols, x, y, z for individual variables t, u for terms and $\bar{x}$ for a vector of variables. Truth of a formula is relative to a background theory $\mathcal{T}$. A subset of the signature $\Sigma_I \subseteq \Sigma$ is *interpreted* by the theory. We assume the symbol $=$ has the usual interpretation. We assume theory $\mathcal{T}$ is *complete* in that it has at most one model. Thus, every sentence over Σ_I has a defined truth value. This assumption can be removed, in which case many of the definitions that follow become relative to a choice of theory model. Unless otherwise stated, we assume that $\mathcal{T}$ is decidable.

The *vocabulary* of ϕ, denoted $L(\phi)$, consists of its free variables and the subset of $\Sigma \setminus \Sigma_I$ occurring in ϕ. An *interpolant* for $A \wedge B$ is a formula I such that $A \Rightarrow I$ and $B \Rightarrow \neg I$ are valid, and $L(I) \subseteq L(A) \cap L(B)$. We will write $\phi[X]$ for a formula with free variables in X. A P-fact is a formula of the form $P(t_1, \ldots, t_n)$. A formula or term is *ground* if it contains no variables. When a set of formulas appears as a formula, it represents the conjunction of the set. If ϕ is a formula and σ a symbol substitution, $\phi\sigma$ is the result of performing substitution σ on ϕ.

Definition 1 *Relative to a vocabulary of predicate symbols $\mathcal{R}$, we say*

1. *A* fact *is a P-fact for some $P \in \mathcal{R}$,*
2. *A* constraint *is a formula ϕ s.t. $L(\phi) \cap \mathcal{R} = \emptyset$,*
3. *A* goal *is a set of facts and constraints.*

4. *A* rule *is a sentence of the form* $\forall X.B[X] \Rightarrow H[X]$ *where the* body $B[X]$ *is a goal and the* head $H[X]$ *is a fact.*

We will also write goals in the form $F[X] \mid C[X]$ *and rules in the form* $F[X] \Rightarrow H[X] \mid C[X]$*, where* $F[X]$ *is the set of facts (the* subgoals*) and* $C[X]$ *is the set of constraints.*

Definition 2 *A* ground instance *of a rule* $F[X] \Rightarrow H[X] \mid C[X]$ *(respectively a goal* $F[X] \mid C[X]$*) is* $F\sigma \Rightarrow H\sigma$ *(respectively* $F\sigma$*) for any ground substitution* σ *on* X *such that* $C\sigma$ *is true in* $\mathcal{T}$*.*

Definition 3 *A* ground derivation *from a set of rules* $\mathcal{C}$ *is a sequence of ground instances of rules in* $\mathcal{C}$ *in which each subgoal is the head of a preceding clause. A* ground derivation *of a goal* G *is a ground derivation of all the subgoals of some ground instance of* G*.*

Definition 4 *A Horn reachability problem is a triple* $(\mathcal{R}, \mathcal{C}, G)$ *where* $\mathcal{R}$ *is a vocabulary of predicate symbols,* $\mathcal{C}$ *is a set of rules over* $\mathcal{R}$ *and* G *is a goal over* $\mathcal{R}$*. It is* satisfiable *if there is a ground derivation of* G *from* $\mathcal{C}$*. A* dual solution *of the problem is a model of* $\mathcal{C}$ *in which* G *is unsatisfiable.*

Definition 5 *A Horn reachability problem* $(\mathcal{R}, \mathcal{C}, G)$ *is* acyclic *if there is a total order* $<$ *on* $\mathcal{R}$ *such that for all rules* $F \Rightarrow P(\bar{x}) \mid C$ *in* $\mathcal{C}$ *and all subgoals* $Q(\bar{y})$ *in* F*,* $Q < P$*.*

A set of Horn clauses has a least model, which is the set of derivable facts. Thus, a Horn reachability problem has a dual solution *iff* it is unsatisfiable.

We assume that the predicates in $\mathcal{R}$ occur only in the form $P(\bar{x})$ where $\bar{x}$ is a vector of distinct variables. We can enforce this by introducing new variables and equalities, for example rewriting $B[X] \Rightarrow P(f(x))$ to $B[X] \wedge y = f(x) \Rightarrow P(y)$. We represent the interpretation of a predicate $P(\bar{x})$ symbolically by a characteristic formula $\phi[\bar{x}]$. We write $\alpha(P) = \lambda\bar{x}.\ \phi[\bar{x}]$ to mean that, in the interpretation α, $P(\bar{x})$ holds iff ϕ holds. Given relations P and Q, we write $P \wedge Q$ for the intersection of P and Q and $P \vee Q$ for their union. Further, we will write $\top$ for $\lambda\bar{x}.$ True and $\bot$ for $\lambda\bar{x}.$ False (where the arity of $\bar{x}$ is understood from context).

The procedure maintains a model α of the rules $\mathcal{C}$ called the *annotation*. This model over-approximates the derivable facts (*i.e.* reachable program states). Initially, $\alpha(P) = \top$ for all $P \in \mathcal{R}$. The LA procedure is shown in Figure 1. As Jaffar *et al.* observe [17], it is essentially Prolog execution with a form of tabling using interpolants. We assume a procedure $\mathrm{Itp}(A, B)$ that takes two inconsistent formulas A and B and returns an interpolant for $A \wedge B$.

The main procedure is Search, which takes a goal G and searches for a ground derivation of it. It assumes that the goal is satisfiable in the current annotation, that is, $G\alpha$ is satisfiable. If not, the problem is trivially unsatisfiable. If there are no subgoals in G, the problem is trivially *satisfiable*, and we return Sat. Else we arbitrarily choose a subgoal $P(\bar{x})$ to derive. We then loop over

all rules C with head matching $P(\bar{x})$. For each such C, we call the procedure RSTEP to continue the search using C to derive $P(\bar{x})$ (procedure REN renames the variables in C to avoid clashes with the goal). If the search succeeds, we return SAT. Else RSTEP returns a value for P that contains all facts derivable using C and rules out the goal. After the loop, the disjunction of the returned values over-approximates P. Thus, we strengthen $\alpha(P)$ by this disjunction, maintaining α as a model of $\mathcal{C}$ and making $G\alpha$ unsatisfiable.

Procedure RSTEP takes a goal G, a subgoal $P(\bar{x})$ of G to be satisfied, and a rule C of the form $B \Rightarrow P(\bar{y})$ to be used to derive the subgoal (where the free variables of the goal G and clause C are distinct). First we resolve the rule with the goal. Since $\bar{y}$ is a vector of variables, the most general unifier is trivial: we just map $\bar{y}$ to $\bar{x}$. We produce the *prefix* by applying the unifier to the body of the rule and the *suffix* by removing the subgoal from G. The *resolvent* G' is the union of the prefix and suffix. This is our new goal.

Now we test whether the new goal G' is satisfiable in the current annotation, using decision procedure for theory $\mathcal{T}$ (for example, an SMT solver). If it is unsatisfiable, we compute an interpolant $\phi[\bar{x}]$ between the prefix and suffix. Because $\phi[\bar{x}]$ is implied by the prefix, we know that the assignment $P = \lambda\bar{x}.\ \phi$ satisfies rule C in the current annotation. Moreover, since $\phi[\bar{x}]$ is inconsistent with the suffix, we know that this interpretation is inconsistent with the original goal G. We therefore return the symbolic relation $\lambda\bar{x}.\ \phi$ as an over-approximation of P showing that the subgoal cannot be derived using rule C.

On the other hand, suppose the new goal is satisfiable in the current annotation. We now make a *decision*. The decision is chosen from a finite language $\mathcal{L}_D(G')$ using only the variables of the new goal G'. Though the decision is arbitrary, it must at least be consistent with G' so that the resulting goal is satisfiable. This means that the disjunction of formulas in language $\mathcal{L}_D(G')$ must be valid. In our implementation, our decision language is the set of truth assignments to the atoms of G'. Thus, we can construct a decision consistent with G' by using the satisfying assignment returned by our decision procedure. Having made a decision, we now have a satisfiable goal, which we attempt to solve by calling the main procedure SEARCH recursively.

Theorem 1 (Total correctness) *Given an acyclic Horn reachability problem* $\Pi = (\mathcal{R}, \mathcal{C}, G)$*, such that* G *is satisfiable,* SEARCH(G) *terminates and:*

- *if it returns* SAT*,* Π *is satisfiable*
- *if it returns* UNSAT*,* Π *is unsatisfiable and* α *is a dual solution of* Π*.*

The proof can be found in Appendix D. We can also extend this procedure to *generalized Horn clauses* as in [19]. In this case, the body of a rule can be a combination of subgoals and constraints using both conjunction and disjunction. This extension, along with some discussion of implementation issues, can be found in Appendix A.

```
    Procedure RSTEP(G, P(x̄), C = (B ⇒ P(ȳ)))
    Input: goal G, subgoal P(x̄), rule C
    Output: SAT, or a bound on P refuting the goal
1       Let pref = B⟨x̄/ȳ⟩ and suff = G \ {P(x̄)}
2       Let G' = pref ∪ suff
3       While TRUE do:
4           if G'α is unsatisfiable, return λx̄. Itp(pref, suff)
5           choose D in L_D(G') s.t. (D ∧ G')α is satisfiable
6           if SEARCH(G' ∪ {D}) = SAT return SAT
7       Done.

    Procedure SEARCH(G = (F | C))
    Input: goal G s.t. Gα satisfiable
    Output: SAT or UNSAT and Gα unsatisfiable
1       If F (the subgoal set) is empty, return SAT
2       Choose a subgoal P(x̄) in F
3       Let R = ∅
3       For each rule C = (B ⇒ P(ȳ)) in C do:
4           Let R_C = RSTEP(G, P(x̄), REN(C))
5           If R_C = SAT return SAT
6           Let R = R ∪ {R_C}
7       Done
8       Let α(P) = α(P) ∧ (∨R)
9       Return UNSAT.
```

Figure 1: Basic unwinding algorithm.

3.1 Comparison to BMC

We will use BMC to refer to Bounded Model Checking by unfolding and applying a decision procedure, as in [7]. In the transition system case, a completed search goal in LA is precisely a k-step BMC formula. If we make no decisions, LA reduces to BMC followed by interpolation. The difference between BMC and LA is therefore in the decision and learning strategies. Decision-making in BMC is *unstructured* in the sense that the decision procedure may make decisions on any variables in the goal formula in any order. In LA, decision making is *structured*. That is, we alternate resolution (unfolding) and decision steps. Similarly, learning in BMC is unstructured. The decision procedure may learn clauses that span the entire BMC formula. In LA, learning is structured. Each learned annotation is a set of facts describing a single program state.

It is not clear *a priori* what the most effective strategy is. The unstructured approach allows greater flexibility and more opportunities for heuristic optimization. On the other hand, a more structured approach may guide us to learn more general facts, reducing the search space more rapidly. In section 5.2 we will try to resolve this question empirically.

3.2 Comparison to PDR

Like LA, PDR has structured search and learning strategies. The fundamental differences are in the form of the goals and the interpolation approach.

The variable elimination trade-off. In PDR, the goal is restricted to the form $R(\bar{x}) \wedge C(\bar{x})$ where $R(\bar{x})$ is an atom to be derived, and the constraint $C(\bar{x})$ is a (quantifier-free) conjunction of literals. Thus, after resolving we must approximate the goal in some way that uses only the variables $\bar{x}$. For example, suppose we resolve the goal $R_k(\bar{x}_k) \wedge C_k(\bar{x_k})$ with the clause $R_{k-1}(\bar{x}_{k-1}) \wedge T(\bar{x}_{k-1}, \bar{x}_k) \Rightarrow R_k(\bar{x}_k)$ to obtain $R_{k-1}(\bar{x}_{k-1}) \wedge T(\bar{x}_{k-1}, \bar{x}_k) \wedge C_k(\bar{x}_k)$. To obtain a new goal, we must somehow eliminate $\bar{x}_k$. The weakest such goal would be equivalent to $\exists \bar{x}_k.\ R_{k-1}(\bar{x}_{k-1}) \wedge T(\bar{x}_{k-1}, \bar{x}_k) \wedge C_k(\bar{x}_k)$. However, it may not be possible or desirable to eliminate this quantifier precisely.

Instead, we may compute a *stronger* goal in the right vocabulary. This is, in effect, decision making. As an example, if we constrain each unwanted variable to have a specific concrete value, then eliminating those variables becomes trivial. However, this may result in too-specific goal, and hence weak or irrelevant annotations. An alternative would be to use quantifier elimination for those variables for which it is inexpensive, and to use concrete values otherwise. In any event, there is an inherent trade-off to be made between the generality of a goal and its cost. We will refer to this as the *variable elimination trade-off*. LA avoids this trade-off by simply not eliminating variables from the goals. A disadvantage of this as that the goals grow syntactically larger as the search deepens.

Interpolation in PDR. The most important difference between PDR and LA is in the computation of interpolants. For simplicity, consider the transition system case. In LA, the interpolant is computed by dividing the goal into two parts (Equations 3 and 4). The prefix is a single step from R_i to R_{i+1}, while the suffix is a path from R_{i+1} to a safety violation at R_k. By contrast, in PDR the interpolant is between a *prefix path* from R_0 to R_{i+1} and a cube $C_{i+1}[\bar{x}_{i+1}]$ (a conjunction of literals). In other words, the clause learned by PDR is an interpolant for $A \wedge B$, where

$$\begin{aligned} A &\equiv I(\bar{x}_k) \wedge \bigwedge_{k=0\ldots i} T(\bar{x}_k, \bar{x}_{k+1}) \\ B &\equiv C_{i+1}[\bar{x}_{i+1}] \end{aligned}$$

In fact, it is not *any* such interpolant, but an interpolant that is inductive *relative* to the current annotation. This means that if P is the interpolant, we have $R_k(\bar{x}_k) \wedge P(\bar{x}_k) \wedge T(\bar{x}_k, \bar{x}_{k+1}) \Rightarrow P(\bar{x}_{k+1})$, for $k = 0 \ldots i$. Intuitively, a simple relatively inductive interpolant might be likely to participate in an eventual inductive invariant. We may construct such an interpolant as a clause using a subset of (the negations of) the literals in the goal. Bradley gives an approach to finding such a clause that is relative inductive [9]. We may also apply generalization rules specific to theories [16].

This approach to interpolation has advantages and disadvantages. The form of the goal makes it possible to search effectively for a relatively inductive in-

terpolant. On the other hand, because of the variable elimination trade-off, the goal may be more specialized than necessary. The resulting weak or irrelevant annotation may provide little reduction in the search space. This is another question that must be answered empirically, and we will attempt to do this in Section 5.1.

Goal preservation. Implementations of PDR typically carry the refutation of each goal all the way to the depth bound, prioritizing goals by depth. We will call this method "goal preservation". This tends to speed the bounded refutation process and also can produce counterexamples longer than the depth bound [9]. Goal preservation is equally applicable to LA, though we will not use it here.

4 Improvements to Lazy Annotation

In this section, we introduce a number of improvements within the basic LA framework, relative to the implementation described in [21]. In Appendix C.2, we evaluate these improvements experimentally to show their individual effect on performance.

Decision space. In the implementation of [21], a decision is simply a choice of basic block exiting a control location in the control-flow graph. This is effective for simple basic blocks, but in a "large block" encoding [6] the resulting goals are too complex. We require a more fine-grained decision space to sufficiently narrow the search. To achieve this, we use truth assignments to the atomic formulas of the goal (*i.e.*, minterms) as decisions. Such a minterm is easily extracted from a satisfying assignment. In a large-block encoding, a minterm of the transition relation corresponds roughly to an execution path. If the code has disjunctive guards, a minterm also fixes a disjunct that is true. However, we do not fix the values of data variables. This reduces the combinatorial complexity of the problem while allowing a large space of data values.

Back-jumping. In a CDCL SAT solver, many decisions are not actually used in the proof of a conflict. In such a case, we backtrack over the decision to an earlier decision that is actually used, and learn an interpolant at that point. The same situation can occur in LA in the case of multiple sub-goals. It will help to think of a goal in LA as a tree in which the leaves are sub-goals and the interior vertices are constraints. Each resolution step expands one leaf of the tree. Now suppose we expand leaf Q, followed by leaf R on a different branch, and reach a conflict. Suppose further that the proof of unsatisfiability does not use any formulas introduced in the expansion of Q. After backtracking from the expansion of R (strengthening R using an interpolant) we can immediately backtrack from the expansion of Q without any additional annotation (put another way, since this step is not used in the proof, its interpolant is TRUE). We may continue to back-jump in this way until we reach an expansion that is in the proof core of the conflict, eliminating unneeded calls to the decision procedure.

Resolution heuristic. Each time we make a resolution step, we must

choose a sub-goal to expand and a rule to derive it. We can use a heuristic for this choice that is closely related to variable scoring heuristics in CDCL SAT solvers. We maintain a *relevance score* for each rule, initially zero. When we backtrack from a resolution without strengthening the annotation (for example, by back-jumping) we decrement the relevance score of the rule. When choosing a rule to resolve with, we select first the rule with the highest score.

Interpolant generalization. Given a proof of a conflict produced by an SMT solver, we can compute an interpolant for the resolution step using methods of feasible interpolation [20]. However, these interpolants may be both syntactically complex and weaker than necessary, depending on the proof actually obtained by the decision procedure. We can borrow an idea from PDR to improve the result. First, notice the asymmetry between the prefix and suffix in the interpolation. The suffix has fixed truth values assigned to each atom by decision making. The prefix on the other hand contains propositionally complex formulas from the annotation and the transition relation. The interpolant thus tends to be a disjunction at the top level, as most case splitting in the proof will occur on the prefix side. We therefore attempt to strengthen it by greedily dropping disjuncts as long as it remains implied by the prefix. Often this leads to a simple interpolant in clause form. If the interpolant is still syntactically complex, we can use more aggressive interpolation methods as in [2]. A simple approach is to sample prime implicants of the prefix. The interpolant is the disjunction of the interpolants for the prime implicants. Empirically, generalization of interpolants is crucial to performance of the algorithm for large-block encodings (see Appendix C.2).

Eager propagation. In IC3, before increasing the unwinding depth, annotations are propagated forward. That is, we copy annotations from earlier to later predicates in the unwinding while the annotation remains a model of the rules. More frequent propagation is also possible and was found to be effective by Suda [24]. In experiments with LA, propagation after completing a bound was found to strengthen the annotation infrequently. A somewhat more effective approach is to propagate eagerly, during the search. When backtracking to a sub-goal with predicate R_i that is an instance of R in the unwinding, we attempt to propagate annotations of earlier instances of R. If any propagation succeeds in strengthening the annotation of R, we backtrack again, in the hope that the strengthening will rule out an earlier sub-goal.

5 Experiments

We will now consider some experiments comparing the performance of PDR, traditional BMC and our improved LA for large-block encodings. We wish to determine experimentally (1) whether structured or unstructured search is more effective, and (2) which interpolation approach is more effective in reducing the bounded search and in converging to an unbounded solution.

As a representative implementation of PDR, we will use the PDR engine implemented in the Z3 theorem prover [16]. This implementation supports lin-

ear integer arithmetic (LIA) and has limited support for other theories, such as the theory of arrays. Z3/PDR computes interpolants for linear arithmetic using Farkas' lemma and inductive generalization. To represent BMC, we will use Stratified Inlining (SI) in Corral [18]. This tool inlines procedures until either a complete error path is found (using Z3), or the problem becomes unsatisfiable (indicating the program is safe) or until a given recursion bound is reached (in which case the result is inconclusive). Inlining a procedure is equivalent to performing a resolution step on the procedure's summary. The author has implemented LA within Z3. It is used as a non-recursive Horn solver in Duality [19]. Bounded solving is done using LA, while convergence is achieved using LAWI. Duality supports full (quantified) first-order logic with linear integer arithmetic and arrays using an interpolation procedure for proofs in this theory implemented in Z3 [22]. Neither Z3/PDR nor Duality use goal preservation in the experiments. This is the default setting in Z3/PDR, as the authors report the method does not produce a clear benefit [8].

Details of the experimental setup can be found in Appendix B.

5.1 First experiment

Our first experiment uses benchmark problems from the SV-COMP 2013 software model checking competition. We use an encoding of these problems into a Horn clause representation available in the SV-COMP repository, provided by Gurfinkel [14]. These are linear cyclic problems that are obtained by inlining all procedures. For the experiment, two subsets of the benchmark problems were chosen: the "Control Flow and Integer Variables" subset and the "Product Lines" subset. These were chosen because they do not rely on complex pointer reasoning and can be encoded using only the theory of linear arithmetic, allowing them to be handled by PDR. This choice was made in advance and was not expanded after obtaining and analyzing data to avoid the possibility of bias due to "benchmark shopping".

For additional context, we include the tool UFO [1]. This tool was the winner of the SV-COMP 2013 in both of the chosen categories. Fortuitously, the Horn clause versions of the problems were generated by UFO. Thus we can be fairly confident that UFO, Z3/PDR and Duality/LA are using the same logical representation of the problem, giving a direct comparison. On the other hand, since Corral cannot use this problem representation (and thus would require a different language front-end) we omit it from this experiment. An additional difficulty in comparison arises because the competition version of UFO is not a single algorithm, but a portfolio of seven algorithms run in parallel. To make a fair comparison against the other algorithms, we use the two most successful of these seven (those that most frequently had the least run time). The first we will call Boxes. It augments LAWI with a multiple-interval abstract domain. The second we will call CPred. It augments LAWI with a Cartesian predicate abstraction domain.

The four chosen tool configurations were run on a 4-core 64-bit 2.67GHz Intel Xeon CPU with 24GB of main memory. The tools were run on all benchmark

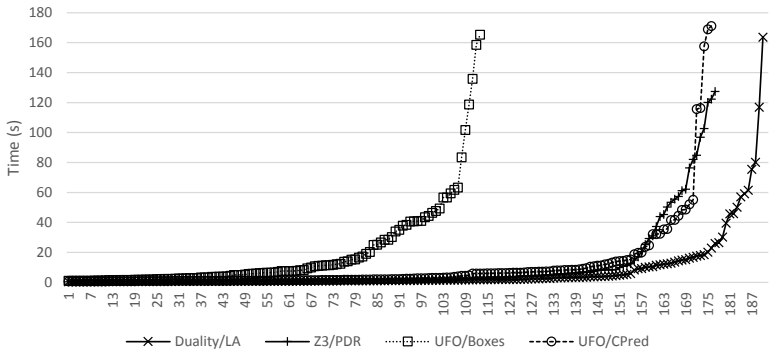

Figure 2: Cactus plot of run times on SV-COMP 2013 problems

problems with a time-out of 180 seconds. Time for compilation and optimization of the C language source code is not included. The benchmark problems completed by all tools in under one second were discarded. The run times for the remaining problems are plotted in Figure 2. Each line shows the run times for all completed benchmarks sorted in increasing order. Thus, a lower line is better. We observe that overall CPred and Z3/PDR are roughly comparable, while Duality solves a larger subset of problems (in fact, all but one).

The are several possible reasons for the difference between LA and PDR. One possibility is that the outer convergence loop is generating different unwindings. To eliminate this possibility, we will focus on the subset of benchmarks in the "ssh" and "ssh-simplified" sub-categories. These benchmarks are simple loops (*i.e.* transition systems) and therefore have only one possible unwinding. For simple loops, we may compare the quality of the annotations generated by the two methods in terms of the number of resolutions and the depth of the unwinding at convergence.

Figure 3 shows scatter plots comparing the unwinding depth at convergence and the number of resolutions steps for Z3/PDR and Duality/LA. Points on the boundary are time-outs. Both measures are substantially lower for Duality/LA, indicating more effective conflict learning. The greater convergence depth for Z3/PDR could be explained by learning many annotations that are true only to a bounded depth. To obtain an inductive invariant in PDR, the search must exceed the depth at which these annotations fail to propagate. The greater search depth may be sufficient to explain the higher number of resolution steps.

From this, it appears that inductive generalization is not able to fully remedy the over-specialization resulting from the variable elimination trade-off in PDR, though it is crucial to the performance of PDR (see Appendix C.1). We may conjecture two possible reasons for the greater convergence depth in PDR: over-specialization producing irrelevant learned clauses or too-aggressive propagation of these clauses. In Appendix C.2 we observe that propagation only weakly affects performance while generalization strongly affects convergence, providing some evidence for the former hypothesis.

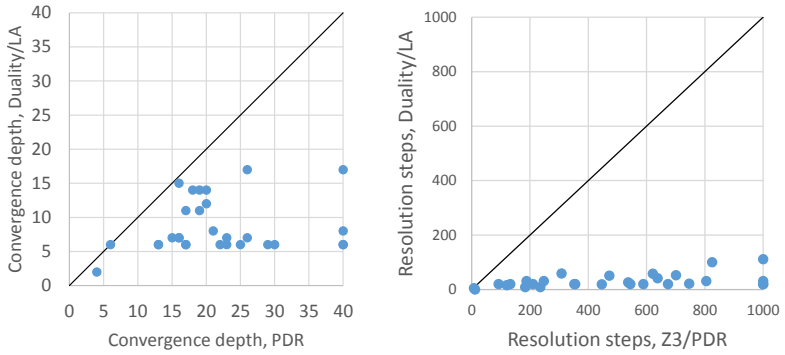

Figure 3: Comparison of Duality/LA and PDR on SV-COMP 2013 ssh benchmarks

5.2 Second experiment

In our second experiment, we consider a broader class of Horn reachability problems. We use procedural programs modeled using unknown relations as procedure summaries. Each procedure is represented by a single rule. For example, procedure P that calls Q twice on its input and then increments the result would be modeled by the clause $Q(x, y) \wedge Q(y, z) \wedge x' = z + 1 \Rightarrow P(x, x')$. Because one procedure may call many procedures, we will have multiple subgoals. Further, we expand the constraint language to include uninterpreted function symbols (UIF's), arrays and quantifiers, and we allow user-specified background axioms.

Our benchmark examples come from the Static Device Driver (SDV) tool [3]. They are safety properties of example device drivers for the Microsoft Windows kernel. SDV translates these problems into the Boogie programming language [4]. Corral then checks the required properties using a field abstraction. Global variables and fields of structures are added to the abstraction on a counter-example driven basis until the property is proved, or a counterexample is successfully concretized. We translate the verification of each abstract model into a Horn reachability problem using the Boogie verification condition generator, after converting program loops into tail-recursive procedures.

We compare the performance of SI with Stratified Inlining (SI). To make a fair comparison, we check only bounded safety properties (that is, we assume each loop is executed up to k times for fixed k). We effect this bound in Duality by simply terminating the unwinding if and when it reaches the recursion bound (as it happens, such termination does not occur, so Duality/LA performs unbounded verification).

The benchmark problems in Boogie use UIF's to model operations on heap addresses. Universal quantifiers occur both in the background axioms that define these functions and in "assume" statements in the program code (assumptions about the initial state of the heap). Since Z3/PDR does not support uninterpreted function symbols and quantifiers in constraints, we are unable to

apply it to this benchmark set. This shows a significant disadvantage of the interpolation strategy of PDR. That is, it is not obvious how to resolve the variable elimination trade-off in this case because UIF's do not admit quantifier elimination. Thus, it would be necessary to fall back on decision making, but in this case the decisions would have to be made on models of the UIF's, which could lead to significant problems of over-specialization. LA's strategy avoids this problem by exploiting feasible interpolation.

Both LA and SI may fail due to the undecidability of the theory. This means that both methods may produce "false alarms" caused by a failure of Z3's quantifier instantiation heuristics. The comparison is fair, however, since the correctness of counterexamples is in both cases determined by Corral, using Z3 for concretization (in no case did either tool produce a counterexample that Corral determined to be incorrect).

A scatter plot comparing the run times of LA and SI is shown in Figure 4. Each point represents the full verification time for one property, including the time for Corral to compute abstraction refinements. We observe that in some cases SI is approximately two times faster. This can be accounted for by the overhead of running Z3 in proof-generating mode in order to produce interpolants. On the other hand, on a significant number problems LA is substantially faster, by up to two orders of magnitude.

We note several important overheads in LA relative to SI. First, the learning phase in LA is orders-of-magnitude more costly than clause learning in an SMT solver. Further, because of backtracking, LA may add and remove a given constraint in the goal many times (while SI never removes constraints). Thus we must attribute the overall better performance of LA on bounded problems to more effective learning. In fact, inspection shows that LA often learns concise and relevant procedure summaries, even in the presence of quantifiers. For example, we see summaries of the form $\forall i.a'[i] = a[i] \vee p(a'[i])$, where p is a simple predicate. This says the procedure preserves elements of array a except where it establishes property p.

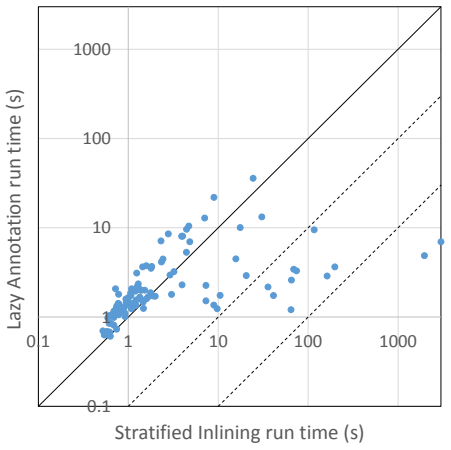

Figure 4: Lazy Annotation *vs* Stratified Inlining

An aspect of these problems that may help to explain better learning performance of LA is the fact that some procedures are called at many sites. Whenever an annotation of such a procedure is learned, it simultaneously strengthens the summaries of all of the instances of the procedure in the current goal. Thus the learned annotations are re-usable in a way that is not possible in an unstructured search.

Finally, we consider the effects of the individual improvements to LA introduced here. To briefly summarize, minterm decision making may help or hurt, but with interpolant generalization it helps significantly. Generalization seems to be needed in the case when convergence depth is an issue. The heuristic for sub-goal choice is not very effective, while back-jumping provides a modest speed-up. Eager propagation is effective in the limited case (simple loops) for which it was implemented. Details of the experiments supporting these conclusions can be found in Appendix C.2.

6 Conclusion

We have observed that traditional BMC, LA and PDR can all be viewed as backtracking search with conflict-driven learning. The methods differ fundamentally in two aspects: search strategy (structured *vs.* unstructured) and interpolation strategy (relative induction *vs.* proof-based). Comparing LA with PDR on software model checking problems, we found that PDR's interpolation strategy as implemented in Z3 produced less effective learned annotations. We conjectured that this is due to over-specialized goals resulting from the *variable elimination trade-off*. This is illustrative of a general tension in CDCL-like methods relating the generality and cost of decisions and interpolants. Comparing LA with BMC, we found that structured conflict learning in LA was more effective than unstructured learning in an SMT solver, even on bounded problems (consistent with the results of [5] in the propositional case). The high overhead of learning in LA was more than compensated by the resulting reduction in search. We found that decision making does in fact lead to improved performance for large-block encodings by reducing the decision problems. However, it requires some form of interpolant generalization to prevent over-specialized goals from producing weak annotations. An interesting remaining question is whether some form of inductive generalization would helpful in LA, or whether the cost outweighs the benefit.

Acknowledgments The author would like to thank Akash Lal for assistance in using SDV and corral.

References

[1] A. Albarghouthi, A. Gurfinkel, Y. Li, S. Chaki, and M. Chechik. UFO: Verification with interpolants and abstract interpretation - (competition contribution). In N. Piterman and S. A. Smolka, editors, *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 637–640. Springer, 2013.

[2] A. Albarghouthi and K. L. McMillan. Beautiful interpolants. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 313–329. Springer, 2013.

[3] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In E. A. Boiten,

J. Derrick, and G. Smith, editors, *IFM*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2004.

[4] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[5] S. Bayless, C. G. Val, T. Ball, H. H. Hoos, and A. J. Hu. Efficient modular SAT solving for IC3. In *FMCAD*, pages 149–156. IEEE, 2013.

[6] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *FMCAD*, pages 25–32. IEEE, 2009.

[7] A. Biere, E. M. Clarke, R. Raimi, and Y. Zhu. Verifiying safety properties of a Power PC microprocessor using symbolic model checking without BDDs. In N. Halbwachs and D. Peled, editors, *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 1999.

[8] N. Bjørner. private communication, 2014.

[9] A. R. Bradley. SAT-based model checking without unrolling. In R. Jhala and D. A. Schmidt, editors, *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.

[10] A. Cimatti and A. Griggio. Software model checking via IC3. In P. Madhusudan and S. A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 277–293. Springer, 2012.

[11] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. IC3 modulo theories via implicit predicate abstraction. In E. Ábrahám and K. Havelund, editors, *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2014.

[12] N. Eén, A. Mishchenko, and R. K. Brayton. Efficient implementation of property directed reachability. In P. Bjesse and A. Slobodová, editors, *FMCAD*, pages 125–134. FMCAD Inc., 2011.

[13] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In J. N. Oliveira and P. Zave, editors, *FME*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2001.

[14] A. Gurfinkel. `https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/LRA/svcomp13/`, 2013.

[15] W. R. Harris, S. Sankaranarayanan, F. Ivancic, and A. Gupta. Program analysis via satisfiability modulo path programs. In M. V. Hermenegildo and J. Palsberg, editors, *POPL*, pages 71–82. ACM, 2010.

[16] K. Hoder and N. Bjørner. Generalized property directed reachability. In A. Cimatti and R. Sebastiani, editors, *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2012.

[17] J. Jaffar, A. E. Santosa, and R. Voicu. An interpolation method for CLP traversal. In I. P. Gent, editor, *CP*, volume 5732 of *Lecture Notes in Computer Science*, pages 454–469. Springer, 2009.

[18] A. Lal, S. Qadeer, and S. K. Lahiri. Corral: A solver for reachability modulo theories. Technical Report MSR-TR-2012-9, Microsoft Research, April 2012.

[19] K. McMillan and A. Rybalchenko. Computing relational fixed points using interpolation. Technical Report MSR-TR-2013-6, Microsoft Research, January 2013.

[20] K. L. McMillan. An interpolating theorem prover. In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2004.

[21] K. L. McMillan. Lazy annotation for program testing and verification. In T. Touili, B. Cook, and P. Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 104–118. Springer, 2010.

[22] K. L. McMillan. Available at `http://z3.codeplex.com`, 2013.

[23] K. L. McMillan, A. Kuehlmann, and M. Sagiv. Generalizing DPLL to richer logics. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 462–476. Springer, 2009.

[24] M. Suda. Triggered clause pushing for IC3. *CoRR*, abs/1307.4966, 2013.

A Extension and implementation issues

A.1 Additional implementation details

Most of the cost of the algorithm is incurred at line 4 of RSTEP, where $G'\alpha$ is tested for satisfiability (using the Z3 SMT solver in our implementation). Successive goals change incrementally, thus it is helpful to use the SMT solver incrementally. The procedure adds and remove some formulas in a resolution step (lines 1–2), and reverts to a previous goal on backtracking. Moreover, new conjuncts are added to $G'\alpha$ when we strengthen the annotation at line 8 of SEARCH. Unfortunately, this means that formulas are not removed in a strictly first-in first-out manner, as required by incremental solving in Z3. In practice, to remove an assertion we "pop" the solver context until it is removed and then reassert any formulas that were incorrectly removed as a side effect. As a result we may lose some useful learned clauses in the solver. There is also some significant inefficiency in restarting the solver, as as much decision-making and propagation may be repeated from the previous run. We do not attempt to remedy this.

When the constraints contain quantifiers, the satisfying assignment may not be sufficient to determine the truth value of quantified subformulas. In this case we do not attempt to decide the truth value of these subformulas, and our decisions are incomplete. If a decision implies a given constraint (which is usually the case, since a minterm is an implicant of the formula) we could in principle remove the constraint from the solver as it is redundant. We do not do this, however, as we wish to retain learned clauses inferred from the constraint.

A.2 Generalized Horn clause extension

It is useful to extend the language of rules to allow disjunctions in the body. In particular, the verification conditions for procedures contain such disjunctions arising from control flow branching.

Definition 6 *A* generalized rule *over a predicate vocabulary $\mathcal{R}$ is a formula of the form $\forall X.\ B[X] \Rightarrow H[X]$, where*

- *The* head $H[X]$ *is a fact, and*
- *The* body $B[X]$ *is a formula of the form* $\exists Y.\ \phi[X,Y]$ *where* $\phi[X,Y]$ *is quantifier-free, and symbols in* $\mathcal{R}$ *occur only positively in* $\phi[X,Y]$.

Because the unknown predicates occur only positively in the body of the rule, we could in principle convert the body to disjunctive normal form, obtaining an equivalent set of rules of exponential size. Instead, we can encode the problem into Horn clauses by adding adding a condition parameter to each predicate. Let F_i stand for the i-th subgoal $P_i(\bar{x}_i)$ in the body B, and let the head H be $Q(\bar{y})$. We substitute a fresh Boolean variable b_i for each subgoal, obtaining a new Horn rule

$$\{P_i(b_i, \bar{x}_i)\} \wedge (c \Rightarrow B\langle b_i/F_i\rangle) \Rightarrow Q(c, \bar{y})$$

where c is a fresh Boolean variable. The new Boolean parameter represents the condition under which a goal is actually required. With this encoding, we need never expand sub-goals whose condition is false in a chosen satisfying assignment. When all remaining sub-goals have false conditions, we can immediately return SAT.

B Details of experiments

B.1 First experiment

The benchmarks in this experiment use the Horn logic specification for SMT-LIB and were generated by Arie Gurfinkel using UFO [14]. These are a subset of the problems in the "Control Flow and Integer Variables" subset and the "Product Lines" from the benchmark set of the SV-COMP 2013 competition. One problem was omitted because the tools disagree on the result (the original C program is unsafe, but the translation to Horn format appears to be safe). In the conversion to Horn format by UFO, integer variables were converted to real type. Since Duality does not support real arithmetic, these variables were converted back to integer type before running the tools.

The commands to run the tools in the four selected configurations are shown in Table 1. The two UFO configurations are run not on the Horn representation, but on the compiled and optimized LLVM bitcode, as generated by the script used in the competition. Time to read and process the bitcode is generally negligible, as we omit problems requiring less than one second to solve. All the tools are run under Ubuntu Linux 2.04 LTS on a 4-core 64-bit 2.67GHz Intel Xeon W3520 CPU with 24GB of main memory.

B.2 Second experiment

The input files for this experiment are in the Boogie language as generated by the Microsoft Static Driver Verifier (SDV) tool. These files are processed by Corral. For each benchmark, Corral generates a sequence of verification problems

Tool	Command
Duality/LA	`z3 fixedpoint.engine=duality`
Z3/PDR	`z3 fixedpoint.engine=pdr`
BOXES	`ufo --lawi --ufo-post=BOXES --ufo-widen-step=17 --ufo-use-ints --ufo-increfine=REF5 --ufo-consrefine=true --ufo-dvo=true --ufo-simplify=false --ufo-false-edges=true`
CPRED	`ufo --lawi --ufo-post=CPRED --ufo-use-ints --ufo-increfine=REF5 --ufo-consrefine=true --ufo-dvo=true --ufo-simplify=false --ufo-false-edges=true`

Table 1: Commands to run tools in the first experiment

Method	Command
LA	`corral /useDuality /k:1 /main:main /useProverEvaluate /catchAll /recursionBound:3 /timeLimit:3000 /sdv /cloops /ann:PruneCounterLoop:15 /memLimit:1000`
SI	`corral /k:1 /main:main /useProverEvaluate /catchAll /recursionBound:3 /timeLimit:3000 /sdv /cloops /ann:PruneCounterLoop:15 /memLimit:1000`

Table 2: Commands to run Corral with LA and SI

using successively refined abstractions. These problems are solved by either Duality/LA or Stratified Inlining as implemented in the Boogie tool. In the former case, verification conditions are generated by Boogie and passed to Duality/LA in the Z3 fixed point format. We report the cumulative time needed to solve all of the successive refinements for each problem (including VC generation). We do not use the Houdini-style inference of invariants implemented in Corral. The specific Corral command lines needed to solve each problem are generated by SDV. Basic command lines needed to run Corral in the two configurations are shown in Table 2. The experiment was run under Microsoft Windows 8.1 on a 3.6 GHz Intel Xeon ES-1620 CPU with four cores and 64GB of RAM.

B.3 Data and software

A package containing the Horn clause representations of the benchmarks and instructions for reproducing the results can be obtained by contacting the author. The version of Z3 used here, including Duality/LA, is avaible in source form at `z3.codeplex.com`. Corral is available at `corral.codeplex.com`.

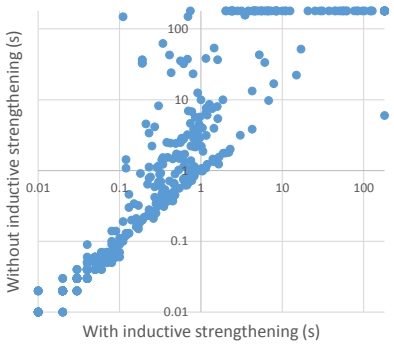

Figure 5: Run times of PDR with and without inductive generalization in the first experiment.

C Additional data

C.1 Effectiveness of inductive generalization

Figure 5 shows a scatter plot comparing run times of Z3/PDR on the full benchmark set of the first experiment, with and without inductive generalization (again points on the boundary are time-outs). Without inductive generalization, the interpolant prefix for PDR is similar to that for LA, though the suffix differs. It can be seen that inductive generalization strongly contributes to the performance of PDR.

C.2 Evaluation of improvements

Figures 6 and 7 show the performance obtained in the two experiments using the LA algorithm, successively adding the features we introduced in this paper. The configurations we test are as follows:

- Mark I: baseline with no improvements. Since there is no decision-making, this version is simply a slightly inefficient version of LAWI that computes sequence or tree interpolants one step at a time.
- Mark II: Add minterm decisions to Mark I.
- Mark III: Add interpolant generalization to Mark II. Note interpolant generalization is not effective in Mark I because the interpolants are not usually disjunctive.
- Mark IV: Add the decision heuristic to Mark III. Scores of clauses are reduced if backtracking does not strengthen the annotation. This is only effective in the non-linear case.
- Mark V: Add back-jumping to Mark IV. This is only effective in the non-linear case.

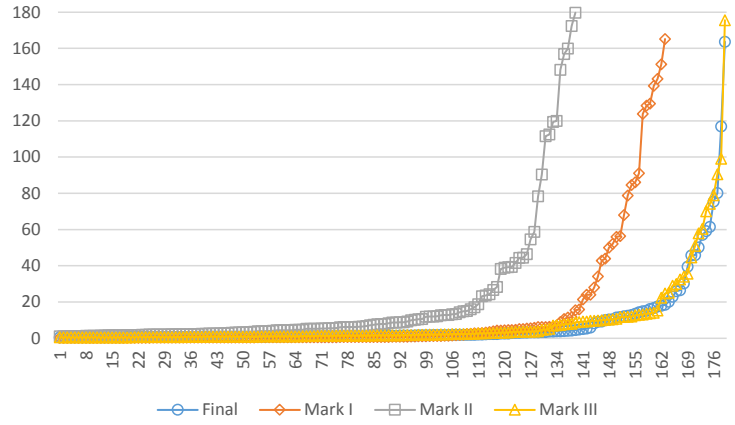

Figure 6: Comparing successive improvements in the first experiment.

- Final: Add eager propagation to Mark V. This is only implemented for simple loops.

Performance is shown for both the first and second experiments. For the second experiment we run all configurations on the same sequence of refinements (thus producing a different counterexample cannot affect the sequence of problems solved). In both figures we leave out the configurations that represent no change from the previous configuration on the given benchmark. Thus, Marks IV and V are not shown in the first experiment since the problems are linear while Final is not shown in the second experiment since there are no simple loops (loops were converted to tail recursions).

Notice that in the first experiment Mark II performs worse than Mark I. That is, by making decisions, we make the satisfiability problems easier, but the goals become more specialized, leading to too-weak interpolants. Adding interpolant generalization in Mark III, however, overcomes this problem to yield a significant improvement over Mark I. There is no visible improvement from adding eager propagation (Final in the first experiment). We note, though, that in the implementation this optimization is applied only to simple loops. In the "ssh" and "ssh-simplified" sub-categories, which are simple loops that require deep unfolding, we do see a benefit as shown in the scatter-plot in Figure 8.

In the second experiment, we see a significant improvement from decision making (Mark II) but not from interpolant generalization. This indicates that generalization may only be useful when convergence depth is a factor (not the case in these bounded problems). The decision heuristic produces no visible improvement, but the addition of back-jumping gives a modest improvement.

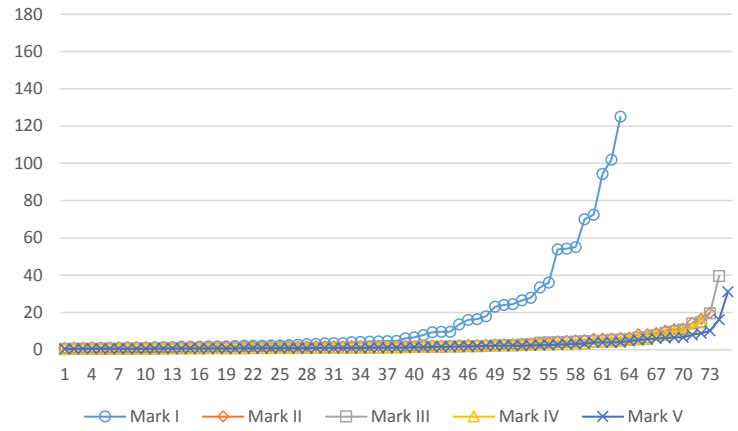

Figure 7: Comparing successive improvements in the second experiment.

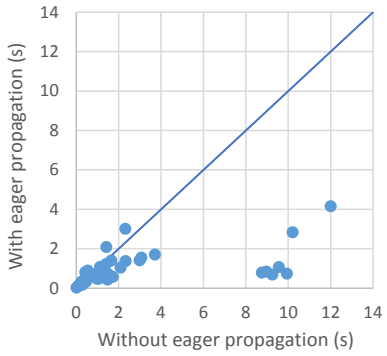

Figure 8: Effect of eager propagation on run time for simple loops.

D Proofs

Theorem 1 (Total correctness) *Given an acyclic Horn reachability problem* $\Pi = (\mathcal{R}, \mathcal{C}, G)$*, such that* G *is satisfiable,* SEARCH(G) *terminates and:*

- *if it returns* SAT*,* Π *is satisfiable*
- *if it returns* UNSAT*,* Π *is unsatisfiable and* α *is a dual solution of* Π*.*

Proof. Sketch. The procedure maintains the invariants that α is a model of $\mathcal{C}$ and if the goal at any level of recursion is satisfiable then the original goal is satisfiable. It follows that if SEARCH returns SAT, goal G is in fact satisfiable. If RSTEP returns an interpretation I for P, the interpolant properties guarantee that α strengthened by I models C. From this it follows that on UNSAT return from SEARCH, α is a model of $\mathcal{C}$. Thus α is always an upper bound on the ground-derivable facts (since it is greater than the least model). Moreover since all the values of P returned by RSTEP are inconsistent with goal G, it follows that α is inconsistent with the goal. Thus if SEARCH returns UNSAT, α is inconsistent with goal G.

Because the language $\mathcal{L}_D(G')$ is finite, RSTEP cannot loop infinitely. Further because the number of rules is finite and the problem is acyclic, the recursion depth and the loop in SEARCH are bounded. Thus the procedure terminates (but note it is assumed that the theory is decidable). □