

Program Verification as Probabilistic Inference

Sumit Gulwani

Microsoft Research, Redmond
sumitg@microsoft.com

Nebojsa Jovic

Microsoft Research, Redmond
jovic@microsoft.com

Abstract

In this paper, we propose a new algorithm for proving the validity or invalidity of a pre/postcondition pair for a program. The algorithm is motivated by the success of the algorithms for probabilistic inference developed in the machine learning community for reasoning in graphical models. The validity or invalidity proof consists of providing an invariant at each program point that can be locally verified. The algorithm works by iteratively randomly selecting a program point and updating the current abstract state representation to make it more locally consistent (with respect to the abstractions at the neighboring points). We show that this simple algorithm has some interesting aspects: (a) It brings together the complementary powers of forward and backward analyses; (b) The algorithm has the ability to recover itself from excessive under-approximation or over-approximation that it may make. (Because the algorithm does not distinguish between the forward and backward information, the information could get both under-approximated and over-approximated at any step.) (c) The randomness in the algorithm ensures that the correct choice of updates is eventually made as there is no single deterministic strategy that would provably work for any interesting class of programs. In our experiments we use this algorithm to produce the proof of correctness of a small (but non-trivial) example. In addition, we empirically illustrate several important properties of the algorithm.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; G.3 [Mathematics of Computing]: Probability and Statistics; I.2.6 [Computing Methodologies]: Artificial Intelligence

General Terms Algorithms, Theory, Verification

Keywords Program Verification, Forward and Backward Analysis, Over and Under Approximation, Automated Recovery, Machine Learning, Belief Networks, Factor Graphs, Probabilistic Inference, Markov Chain Monte Carlo, Gibbs Sampling

1. Introduction

The program verification problem is to verify the Hoare triple, $\langle \phi_{\text{pre}}, P, \phi_{\text{post}} \rangle$, where ϕ_{pre} and ϕ_{post} are the precondition and postcondition respectively of program P . The Hoare triple is said to be valid if for all program states satisfying ϕ_{pre} , whenever the

program P terminates, it does so in a state that satisfies ϕ_{post} . A proof of validity of the Hoare triple $\langle \phi_{\text{pre}}, P, \phi_{\text{post}} \rangle$ can be in the form of an invariant at each program point such that the invariants can be easily verified locally. A proof of invalidity of the Hoare triple $\langle \phi_{\text{pre}}, P, \phi_{\text{post}} \rangle$ can be in the form of the proof of validity of the Hoare triple $\langle \phi'_{\text{pre}}, P, \neg \phi_{\text{post}} \rangle$ for some ϕ'_{pre} that is consistent with ϕ_{pre} . (In this formalism for proof of invalidity, we assume that the end of program P is reached during all executions of the program.)

In this paper, we describe how probabilistic inference techniques, heavily studied in the machine learning community, can be used to discover the invariants at each program point that constitute either a proof of validity or a proof of invalidity of a given Hoare triple. The algorithm works by running in parallel the search for validity proof and invalidity proof of the Hoare triple. The proof search routine starts by initializing the state abstractions (potential invariants) at all program points to anything (e.g., \perp). It then iteratively chooses a random program point π whose abstract state representation is locally inconsistent, and updates it to make it less locally inconsistent. To be more precise, the state abstraction at π is chosen randomly from the abstract domain, with probability inversely proportional to its local inconsistency. The local inconsistency of the state abstraction at a program point is a monotonic measure of the set of program states that are not consistent with the state abstraction at the neighboring program points. The proof search routine stops when these abstractions are all locally consistent in which case they constitute program invariants that prove either validity or invalidity of the Hoare triple (depending on what definition of consistency was used for the program's entry and exit points). This algorithm is described formally in Section 3.

The above proof discovery algorithm, though simple, has some interesting aspects to it.

- The algorithm combines the complementary powers of both forward and backward analysis by bringing information from both forward and backward directions at each program point. We define the local inconsistency of an abstract representation of possible program states at a program point as a measure of the set of program states that are not consistent with the state abstractions at *neighboring* program points (as opposed to considering only the immediate *predecessor* program points, which would constitute a forward analysis, or as opposed to considering only the immediate *successor* program points, which would result in only a backward flow of information). Result of this is that effectively, the forward analysis uses the information from the backward analysis to guide its over-approximation, while the backward analysis uses the information from the forward analysis to guide its under-approximation.
- Even though bringing information from both forward and backward directions yields a more refined choice of abstractions, these abstractions may not be the right ones. However, our algorithm places much less burden on the optimality of these

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'07 January 17–19, 2007, Nice, France.

Copyright © 2007 ACM 1-59593-575-4/07/0001...\$5.00.

choices since it offers chances of recovery at each step. This is because it does not distinguish between the information brought in by the forward analysis or the backward analysis, and as a result, the information can get both over-approximated or under-approximated in any step. Contrast this with a traditional forward analysis in which the information at any program point gets weakened in successive updates. Hence, once the information computed at a program point overapproximates the invariants required at that point to establish the postcondition, the postcondition cannot be established.¹

- The inference procedure is based on smooth real-valued cost functions, rather than binary costs (where invariants are either considered to be in agreement or not). This could be thought of as giving the algorithm partial credit for its guesses, and thus providing it with more information about which direction to take in order to further improve the invariants. However, among the possible choices that have approximately equal benefits, the algorithm chooses randomly. Thus, our algorithm falls into the category of sampling techniques studied extensively in [20].

The above aspects are discussed in more detail in Section 5 with examples of parts of the proof search that we saw in our experiments.

The algorithm we introduce here is a form of a Gibbs sampling technique, which is one of the probabilistic inference techniques used in machine learning [20]. Many machine learning problems are concerned with the discovery of hidden quantities at nodes of a graph that satisfy given constraints, and so the machine learning community has developed a number of such inference algorithms, which can both be deterministic and randomized. In this paper, we define program verification in a way that allows most of these techniques to be applied to this problem, as well. For this purpose, we introduce the novel notion of an *inconsistency measure* for any abstract domain \mathcal{A} that is equipped with a partial order \Rightarrow . An inconsistency measure \mathcal{M} for \mathcal{A} is any function from ordered pairs (ϕ, ϕ') of elements from \mathcal{A} to $[0, 1]$ that is monotonically increasing in its first argument ϕ , monotonically decreasing in its second argument ϕ' , and 0 iff $\phi \Rightarrow \phi'$. We use this measure to define the *local inconsistency* of a program point's set of reachable states (which belongs to \mathcal{A}) with respect to the sets of reachable states at neighboring program points. Given these local measures, we can pose the problem of program invariant inference as inference in probabilistic graphical models. The particular inference algorithm we use in this paper uses the inconsistency measure to give preference to those potential invariants that minimize the local inconsistency. A large class of new algorithms for program verification can benefit from a real-valued measure of inconsistency, as it can be used as a measure of the progress of the algorithm.

We implemented our algorithm for programs that have linear assignments and whose conditional predicates have the form of difference constraints. We chose the abstract domain for representing invariants to be the set of boolean formulas over difference constraints. In the spirit of Occam's razor principle, the formulas are of bounded size, thus biasing the search towards simpler proofs. We tested the algorithm on a small non-trivial example program with pre/postcondition pair. We show that our tool can consistently solve this problem in finite time, whereas the brute force search would be infeasible.

¹ Similarly, in a traditional backward analysis the information at any program point gets strengthened in successive updates. Hence, once the obligation computed at a program point underapproximates the invariants that are true at that program point under the precondition, the validity of the pre/postcondition pair cannot be established.

2. Notation

Let \mathcal{A} be some abstract domain, whose elements represent sets of program states. Let $\gamma : \mathcal{A} \rightarrow 2^\Sigma$ be the concretization function that relates the abstract domain and the concrete domain, i.e., for any element $\phi \in \mathcal{A}$, $\gamma(\phi)$ gives the set of all program states represented by ϕ . Let \perp and \top represent the usual bottom and top elements in the abstract domain, i.e., $\gamma(\perp) = \emptyset$ and $\gamma(\top) = \Sigma$. We say that an abstract element ϕ is *stronger than* ϕ' (denoted by $\phi \Rightarrow \phi'$) if $\gamma(\phi) \subseteq \gamma(\phi')$. We also say that ϕ *holds at program point* π to denote that $\gamma(\phi)$ is an over-approximation of the set of program states that are possible at π .

We use the notation π_{entry} and π_{exit} to denote the program's entry and exit point. For simplicity, we assume (without loss of any generality) that there are no self loops on any program node. We use the notation ϕ_i to denote the latest choice of the abstract element (made by our algorithm) at program point π_i .

Given ϕ_i for all program points π_i , and any program point π_k other than π_{entry} , let $\text{Post}(\pi_k)$ denote the set of (strongest) abstract elements such that for any element $\phi \in \text{Post}(\pi_k)$, we have the following property: If for all immediate predecessors π_j of π_k , ϕ_j holds at π_j , then ϕ holds at π_k . Similarly, given ϕ_i for all program points π_i , and any program point π_k other than π_{exit} , let $\text{Pre}(\pi_k)$ denote the set of (weakest) abstract elements such that for any element $\phi \in \text{Pre}(\pi_k)$, we have the following property: If ϕ holds at π_k , then for all immediate successors π_j of π_k , ϕ_j holds at π_j . For notational convenience, we say that:

$$\text{Post}(\pi_{entry}) \stackrel{\text{def}}{=} \perp \quad \text{and} \quad \text{Pre}(\pi_{exit}) \stackrel{\text{def}}{=} \top$$

Since there are no self-loops on any program node, any program point π_k is not its own immediate successor or predecessor. Hence, the definitions of $\text{Post}(\pi_k)$ and $\text{Pre}(\pi_k)$ are well-formed. We use the notation $\text{Post}(\pi_k) \Rightarrow \phi$ to denote that $\phi' \Rightarrow \phi$ for some $\phi' \in \text{Post}(\pi_k)$. Similarly, we sometimes use the notation $\phi \Rightarrow \text{Pre}(\pi_k)$ to denote that $\phi \Rightarrow \phi'$ for some $\phi' \in \text{Pre}(\pi_k)$.

An *inconsistency measure* on the abstract domain \mathcal{A} is any function that maps ordered pairs of elements from \mathcal{A} to $[0, 1]$ and has the following properties for any elements $\phi, \phi', \phi'' \in \mathcal{A}$ that satisfy $\phi \Rightarrow \phi'$.

- $\mathcal{M}(\phi, \phi') = 0$.
- $\mathcal{M}(\phi, \phi'') \leq \mathcal{M}(\phi', \phi'')$.
- $\mathcal{M}(\phi'', \phi') \leq \mathcal{M}(\phi'', \phi)$.

Observe that an inconsistency measure \mathcal{M} (which satisfies the above properties) provides a natural measure of how much inconsistent is the relationship $\phi \Rightarrow \phi'$ for any ordered pair (ϕ, ϕ') .

Given ϕ_i for all program points π_i , and an inconsistency measure \mathcal{M} , we define the *local inconsistency* of ϕ at π_k to be the following quantity:

$$L(\phi, \pi_k) = \text{Min} \{ \mathcal{M}(\phi', \phi) \mid \phi' \in \text{Post}(\pi_k) \} + \text{Min} \{ \mathcal{M}(\phi, \phi') \mid \phi' \in \text{Pre}(\pi_k) \}$$

Given ϕ_i for all program points π_i , we say that ϕ is *locally consistent* at π_k when

$$\text{Post}(\pi_k) \Rightarrow \phi \quad \text{and} \quad \phi \Rightarrow \text{Pre}(\pi_k)$$

Observe that given ϕ_i for all program points π_i , and for any inconsistency measure \mathcal{M} , the following property holds: ϕ is locally consistent at π_k iff $L(\phi, \pi_k) = 0$.

3. Algorithm

In this section, we describe the learning based semi-algorithm for producing a proof of validity or invalidity of the Hoare triple

```

FindProof(P, β) =
1 For all program points πi, initialize φi := ⊥;
2 While penalty of any program point is non-zero:
3   Choose program point πk ∈ {πi | β(φi, πi) ≠ 0} uniformly at random.
4   Choose φ ∈ A with probability proportional to e-β(φ, πk).
5   Update φk := φ;
6   Output('Proof found.');
```

```

Decide(⟨φpre, P, φpost⟩) =
1 Let βV and βI be the penalty functions as defined in Section 3.1.
2 In parallel execute: [ FindProof(P, βV) || FindProof(P, βI) ];
```

Figure 1. The machine learning based semi-algorithm for producing a proof of validity or invalidity of the Hoare triple $\langle \phi_{\text{pre}}, P, \phi_{\text{post}} \rangle$.

$\langle \phi_{\text{pre}}, P, \phi_{\text{post}} \rangle$. The proof of validity consists of providing an abstract element ϕ_i at each program point π_i such that:

- A1. $\phi_{\text{entry}} = \phi_{\text{pre}}$.
- A2. $\phi_{\text{exit}} = \phi_{\text{post}}$.
- A3. ϕ_i is locally consistent at π_i .

The proof of invalidity consists of providing an abstract element ϕ_i at each program point π_i such that:

- B1. $\gamma(\phi_{\text{entry}}) \cap \gamma(\phi_{\text{pre}}) \neq \emptyset$ (i.e., ϕ_{entry} is consistent with ϕ_{pre}).
- B2. $\phi_{\text{exit}} = \neg \phi_{\text{post}}$.
- B3. ϕ_i is locally consistent at π_i .

Note that in the above formalism for proof of invalidity of the Hoare triple, we make the assumption that the program point π_{exit} is reachable in all program executions. (Alternatively, the proof of invalidity may consist of providing a concrete program state σ such that the execution of the program in state σ reaches program point π_{exit} , and in such a state that satisfies $\neg \phi_{\text{post}}$. Such a counterexample may be produced from ϕ_{entry} by selecting a concrete program state σ from $\gamma(\phi_{\text{entry}})$ and checking whether program point π_{exit} is reached when the program is executed in σ . If not, then the process of finding a new $\sigma \in \gamma(\phi_{\text{entry}})$, or the initial process of finding a new set of invariants at each program point that satisfy Properties B1-B3 is repeated.) Henceforth, we assume that our task is to find a set of invariants at each program point such that either properties A1-A3 are satisfied, or properties B1-B3 are satisfied.

We assume that we are given an abstract domain \mathcal{A} (with γ as the concretization function that relates the abstract domain with the concrete domain) such that all invariants in the proofs are expressible in this abstract domain \mathcal{A} . In particular, $\phi_{\text{pre}} \in \mathcal{A}$ and $\phi_{\text{post}} \in \mathcal{A}$. Our algorithm is also parameterized by some inconsistency measure \mathcal{M} (as defined in Section 2) associated with \mathcal{A} , which gives a numeric measure of how much is the partial order relationship (\Rightarrow) not satisfied between two given elements of \mathcal{A} .

The pseudo-code for the learning algorithm is described in Figure 3. The procedure `Decide`($\langle \phi_{\text{pre}}, P, \phi_{\text{post}} \rangle$) runs the process of finding the proof of validity of the Hoare triple $\langle \phi_{\text{pre}}, P, \phi_{\text{post}} \rangle$ (`FindProof`(P, β_V)) in parallel with the process of finding the proof of its invalidity (`FindProof`(P, β_I)). Each of these processes use the same algorithm `FindProof`, but invoked with different penalty functions β_V and β_I . β_V enforces the constraints described in properties A1-A3 on the invariants, while β_I enforces the constraints described in properties B1-B3 on the invariants.

The `FindProof` algorithm works by initializing the abstract elements at all program points to anything (e.g., \perp). It then iteratively chooses a random program point π_k whose abstract element ϕ_k is locally inconsistent, and updates ϕ_k to make it less locally incon-

sistent. To be more precise, ϕ_k is chosen randomly with probability inversely proportional to the exponent of its *penalty* at π_k . Given ϕ_i for all program points π_i , the penalty of an abstract element ϕ at a program point π_k is a non-negative number, which measures unacceptability of ϕ at π_k . The penalty function is the only deterministic part of the algorithm, and should be carefully designed to guide the learning algorithm towards the solution.

3.1 Penalty Function

The penalty function should have the following properties.

- A1. **Soundness:** When the penalty of abstract elements at all program points has been reduced to 0, then the collection of invariants at those program points constitutes a valid proof.
- A2. **Monotonicity:** The penalty function should assign a greater penalty to abstract elements that are more locally inconsistent.

Property A1 is essential for correctness of the algorithm, while property A2 is important for faster convergence of the algorithm. Hence, for any inconsistency measure \mathcal{M} on \mathcal{A} , we can define a valid penalty function (which satisfies properties A1 and A2) for proving the validity of the Hoare triple $\langle \phi_{\text{pre}}, P, \phi_{\text{post}} \rangle$ as follows.

$$\begin{aligned} \beta_V(\phi, \pi_{\text{entry}}) &= 0, \text{ if } \phi = \phi_{\text{pre}} \\ &= \infty, \text{ otherwise} \end{aligned} \quad (1)$$

$$\begin{aligned} \beta_V(\phi, \pi_{\text{exit}}) &= 0, \text{ if } \phi = \phi_{\text{post}} \\ &= \infty, \text{ otherwise} \end{aligned} \quad (2)$$

$$\beta_V(\phi, \pi_k) = N \times L(\phi, \pi_k)$$

Note that Equation 1 and Equation 2 enforce the constraint that the abstract elements at the program points π_{entry} and π_{exit} must be ϕ_{pre} and ϕ_{post} respectively. N denotes a large constant. It is easy to see that a bigger value of N increases likelihood of selection of abstract elements that minimize penalty, and hence may result in faster convergence. However, a smaller value of N may also result in faster convergence in two ways: (a) by decreasing the time required to get out of local minima (if the algorithm ever gets stuck there), (b) by increasing the gradient of change. (This is equivalent of widening/narrowing in standard abstract interpretation terminology [3].) Hence, the choice of N should ideally be determined by performing experiments. The function L is the local inconsistency measure (as defined in Section 2), which is a function of the inconsistency measure \mathcal{M} associated with the abstract domain \mathcal{A} .

Similarly, we can define a valid penalty function (which satisfies properties A1 and A2) for proving the invalidity of the Hoare triple $\langle \phi_{pre}, P, \phi_{post} \rangle$ (assuming that the end of the P is always reached) as follows:

$$\begin{aligned} \beta_I(\phi, \pi_{entry}) &= \infty, \text{ if } \gamma(\phi) \cap \gamma(\phi_{pre}) = \emptyset \\ &= N \times L(\phi, \pi_{entry}), \text{ otherwise} \end{aligned} \quad (3)$$

$$\begin{aligned} \beta_I(\phi, \pi_{exit}) &= 0, \text{ if } \phi = \neg\phi_{post} \\ &= \infty, \text{ otherwise} \end{aligned} \quad (4)$$

$$\beta_I(\phi, \pi_k) = N \times L(\phi, \pi_k)$$

Equation 3 enforces the constraint that there is flexibility in choosing the abstract element at π_{entry} only if it is consistent with ϕ_{pre} , i.e., the intersection of the sets $\gamma(\phi)$ and $\gamma(\phi_{pre})$ is non-empty. Equation 4 enforces the constraint that the abstract element at π_{exit} must be $\neg\phi_{post}$.

4. Derivation and the properties of the algorithm

In this section, we derive the algorithm described above by first posing the program verification problem as a case of inference in probabilistic graphical models, and then turning to one of the simplest probabilistic inference techniques – Gibbs sampling – to perform inference. Properties of this algorithm have been extensively studied in the literature, e.g., [20].

As discussed above, a program invariant we are searching for is a set $\{\hat{\phi}_k\}$ which satisfies the constraints imposed by the program instructions. Inference of program invariants can therefore be viewed as optimization of the level to which these constraints are satisfied. For example, if the abstraction is expressive enough, and the program is correct (the precondition of the program indeed implies the postcondition of the program), then all the constraints imposed by program instructions will be fully satisfied by the optimal set $\{\hat{\phi}_k\}$.

4.1 Discovery of program invariants through probabilistic inference

Instead of a binary treatment of the constraint satisfaction (satisfied or unsatisfied), we construct a real-valued function $f(\phi_0, \phi_1, \dots, \phi_K)$ which attains a maximum value at the program invariant $\{\hat{\phi}_k\}$, i.e.,

$$f(\hat{\phi}_0, \hat{\phi}_1, \dots, \hat{\phi}_K) = \max f(\phi_0, \phi_1, \dots, \phi_K). \quad (5)$$

Clearly, one such function would assign zero to any combination of expressions that does not satisfy the program constraints, and one to true program invariants (of which there could be several, depending on the abstraction). However, as was indicated in the previous section, and as will be discussed further later, in many optimization techniques, it is important that the combinations $\{\phi_k\}$ which only *partially* satisfy the constraints are assigned a non-zero value corresponding to the level of constraint violation.

We can use the structure of the program to write the satisfaction function f as a product of factors f_i associated with different program nodes. Each of these factors is a function of the appropriate set of variables describing the abstract elements before and after the program node. We will denote by Φ_i the set containing the appropriate pre- and post-execution states for the i -th factor, each associated with the appropriate program node. Then, we define

$$f(\phi_0, \phi_1, \dots, \phi_K) = \prod_i f_i(\Phi_i), \quad (6)$$

and define each of the factors $f_i(\Phi_i)$ so that they reach their maximum when the set of abstract elements in Φ_i are consistent with the node between them. The graph consists of a number of variables (in our case these variables describe abstract elements

ϕ_k for different program points π_k), and factors f_i whose product defines a global function of these variables.

The algorithm of the previous section is using factors f_i which drop exponentially with the constraint violation penalty:

$$f_i(\Phi_i) = e^{-\alpha(\Phi_i)} \quad (7)$$

where $\alpha(\Phi_i)$ is defined to be sum of the inconsistencies of the elements of Φ_i multiplied by N .

For example, Figure 2 shows how to visualize the structure of such functions as a factor graph [18] for the program shown in Figure 3(a). We have $\Phi_8 = \{\phi_5, \phi_7, \phi_8\}$, and $\alpha(\Phi_8) = N \times (\mathcal{M}(\phi_5, \phi_8) + \mathcal{M}(\phi_7, \phi_8))$. (This is because f_8 is a join node, which enforces the constraint that $\phi_5 \Rightarrow \phi_8$ and $\phi_7 \Rightarrow \phi_8$.)

Given this function, we can formulate several interesting tasks within the framework of inference in probabilistic graphical models. First, we can normalize the function f by dividing it by a constant Z (often called a partition function), so that the sum over all possible combinations of expressions ϕ_k within the abstraction of interest is equal to one, i.e.,

$Z = \sum_{\phi_0 \in \mathcal{A}} \sum_{\phi_1 \in \mathcal{A}} \dots \sum_{\phi_K \in \mathcal{A}} f(\phi_0, \phi_1, \dots, \phi_K)$. This leads to a probability distribution function

$$p(\phi_0, \phi_1, \dots, \phi_K) = \frac{1}{Z} f(\phi_0, \phi_1, \dots, \phi_K), \quad (8)$$

whose sum over all possible combinations of expressions ϕ_k is indeed equal to one, and whose maximum is still at the same optimal set $\{\hat{\phi}_k\}$ (or a set of optima if there are more than one such set). We can think of (one or more) optimal expression combinations $\hat{\phi}_0, \dots, \hat{\phi}_K$ as very likely under this probability distribution.

In this paper we are especially interested in the case where some of the program states ϕ_k are given while others are hidden. For example, the given states could be program asserts, including the program pre-condition and post-condition (entry and exit beliefs). We will denote by Φ_G the given states and by Φ_H the hidden states. Some interesting questions that probabilistic inference techniques can be employed to answer include:

- What is the most likely set of unobserved program states Φ_H if we are already given some states Φ_G , e.g., $\Phi_G = \{\phi_{entry}, \phi_{exit}\}$. In other words, what is $\arg \max p(\Phi_H | \Phi_G)$, where $p(\Phi_H | \Phi_G)$ denotes the conditional probability distribution over the abstract representations over program states in Φ_H ?
- What are the other likely combinations of expressions for Φ_H given Φ_G , or in other words, can we draw several samples from $p(\Phi_H | \Phi_G)$?²

Note that answers to these two questions (as well as other probabilistic inference goals) are related. For example, if we can sample from the conditional distribution, then we are likely to come across the most likely states $\{\hat{\phi}_k\}$ sooner rather than later. Thus, a search for the optimal state can be performed by sampling. In fact, if all combinations of expressions $\{\phi_k\}$ that fully satisfy the constraints have the total probability p_s under the distribution p , then the probability of not discovering one of these combinations when a single sample is drawn from p is $(1 - p_s)$, and the chance of missing it in n steps is $(1 - p_s)^n$. Therefore, the probability

²“Drawing a sample from a distribution” refers to any randomized algorithm which can produce many samples $\phi_1^t, \dots, \phi_K^t$, so that the fraction of times a certain combination ϕ_1, \dots, ϕ_K is achieved is expected to be equal to $p(\phi_1, \dots, \phi_K)$, i.e., $\lim_{T \rightarrow \infty} \frac{1}{T} \sum_{t=1}^T \mathbb{1}[\phi_1^t, \dots, \phi_K^t = (\phi_1, \dots, \phi_K)] = p(\phi_1, \dots, \phi_K)$, with $\mathbb{1}[\cdot]$ denoting an indicator function, which is equal to one when the equality is satisfied and zero otherwise. Clearly such an algorithm is going to be more likely to produce samples with high probability under the distribution p .

of missing the solution will drop exponentially as more and more samples are taken, and the speed of the drop will depend on the level of the penalty for constraint violation, as this penalty controls the probability of program invariants satisfying the constraints.

Over the last couple of decades, the machine learning community has been developing general techniques for probabilistic inference that use the structure of the graphical model to optimize the performance. These techniques include, for example, belief propagation, variational techniques, and sampling techniques. Each of these techniques has interesting provable and empirically discovered properties, well documented in, among other places, some of the papers we refer to in this paper. Some of these algorithms are meant to estimate (or maximize) probabilities and not draw samples, and as such they are often deterministic, but may get stuck in a local minimum of the optimization criterion. For a comparison of several inference techniques on a simple visual example, see, for example [9]).

4.2 Gibbs sampling

The algorithm described in the previous section is a form of one of the simplest probabilistic inference techniques, known as Gibbs sampling which, like many other probabilistic inference techniques, has first been developed by physicists to compute the distribution over states of a physical system (see references in [20]). We use this technique to draw samples from a distribution over the program states at different program points $p(\phi_1, \dots, \phi_k)$, under the constraint that the boundary states at program asserts (or just the beginning and the end) are equal to the given expressions in Φ_G . In other words, we sample from the conditional distribution $p(\Phi_H | \Phi_G)$. We stop this process once we reach the program invariant $\{\hat{\phi}_k\}$ which maximizes our satisfaction function f . As discussed above, since the combinations of program states $\{\phi_k\}$ with higher levels of constraint satisfaction f are more likely than the ones with less satisfaction (by construction of p), this process should generally attain the maximum of f much sooner than a brute force search for a program invariant.

The Gibbs sampling algorithm consist of iteratively updating each of the expressions ϕ_k while keeping others fixed. To make the discussion more concrete, and without lack of generality, we assume for the rest of this section that we are given the program's entry state ϕ_{entry} and exit state $\phi_K = \phi_{exit}$, and we need to find the rest of the expressions $\phi_1, \dots, \phi_{K-1}$ that satisfy ϕ_0 and ϕ_K , just as in the algorithm in Figure 1. The process is started from an arbitrary initialization of $\phi_1, \dots, \phi_{K-1}$. Then, a series of updates is performed. In each update, one of the current expressions ϕ_j , $j \in \{2, 3, \dots, K-1\}$ is replaced by a *sample* from the conditional distribution $p(\phi_j | \phi_1, \dots, \phi_{j-1}, \phi_{j+1}, \dots, \phi_K)$. It is easily shown that,

$$p(\phi_j | \phi_1, \dots, \phi_{j-1}, \phi_{j+1}, \dots, \phi_K) = \frac{1}{Z_j} \prod_{i | \phi_j \in \Phi_i} f_i(\Phi_i), \quad (9)$$

where Z_j is a scaling constant that normalizes the product of factors involving the expression ϕ_j . Using the factors of the form Equation 7, we get:

$$\begin{aligned} p(\phi_j | \phi_1, \dots, \phi_{j-1}, \phi_{j+1}, \dots, \phi_K) &= \frac{1}{Z_j} e^{-\sum_{i | \phi_j \in \Phi_i} \alpha(\Phi_i)} \\ &= \frac{1}{Z_j} e^{-\beta(\phi_j, \pi_j)}, \end{aligned} \quad (10)$$

where β is the penalty function described in the previous section. Computing this function given the program states at neighboring points is simple, and we can think of this step as satisfying with high probability the requirement that the belief about the state of the program at a program point should be “sandwiched” between

the appropriate pre- and post-conditions of the neighboring instructions, which only depend on the current belief about the state of the small number of neighboring program points.

One appealing property of Gibbs sampling is that, under the assumption that the current sample $\phi_1^t, \dots, \phi_{K-1}^t$ has been drawn from the desired distribution $p(\phi_1, \dots, \phi_{K-1} | \phi_0, \phi_K)$, replacing one of the expressions ϕ_j^t by a sample ϕ_j^{t+1} drawn from the conditional distribution $p(\phi_j | \phi_1, \dots, \phi_{j-1}, \phi_{j+1}, \dots, \phi_K)$ will also result in the new updated sample $\phi_1^t, \dots, \phi_{j-1}^t, \phi_j^{t+1}, \phi_{j+1}^t, \dots, \phi_{K-1}^t$ drawn from the target distribution $p(\phi_1, \dots, \phi_{K-1} | \phi_0, \phi_K)$.

This means that the desired distribution is a stationary distribution of this process, i.e., if at any given time the process starts producing samples from the desired distribution, it will continue to do so. In addition, it has been shown that as long as the distribution p is non-zero everywhere, this process will indeed reach this fixed distribution [20], regardless of how the states are initialized. After that point, and usually even earlier, the most likely configuration of expressions $\phi_2, \dots, \phi_{K-1}$ is likely to be reached, with the probability of missing the solution dropping exponentially as more and more samples are taken. If the constraint violation penalties are high, the probability of sampling a true program invariant will be higher, and the speed of convergence faster. But, on the other hand, if many combinations of program states are highly penalized, the sampling process may get trapped - the samples can get isolated by zero probability regions around them, making it difficult to sufficiently improve them to cross into the regions of the space of the combinations of program states which satisfy the program constraints even better.

Some of the properties of the Gibbs sampling algorithm studied in the machine learning literature are rather intuitive. For instance, since the updates are incremental, it is beneficial to use smooth target functions (f and p , which are equivalent in terms of their maxima as they differ only by a constant scaling factor). In addition, in order to guarantee that the procedure can reach all possible combinations of program states expressible in the abstraction domain, we should not assign zero probability to any combination of the expressions ϕ_k . This is why the factors f_i should be crafted so that $f_i \in [\epsilon, 1]^3$, and so that the expressions that differ a little have similar values, with the ones better satisfying the constraint having higher values, and the ones fully satisfying it having the highest value of 1. Therefore, it is important to use penalties α, β that are smooth and finite to guarantee that the sampling process will indeed traverse the space of possible program states fast enough and avoid getting stuck far from the solution.

5. Discussion

In this section, we discuss several interesting aspects of the simple learning based algorithm. For this purpose, we consider the program shown in Figure 3 with its pre/postcondition pair, as an example. We first show why several existing techniques (with the exception of a recently proposed technique) fail to work well on this example program. We then discuss the interesting aspects of our algorithm that enable it to reason about this example.

5.1 Limitations of Other Techniques

Reasoning about the validity of the program in Figure 3 requires discovering the following invariants at program point π_2 :

$$(x \geq 50 \vee y = 50) \wedge (x < 50 \vee x = y) \wedge (x \leq 100)$$

A simple forward abstract interpretation [3] based analysis over the polyhedron abstract domain will fail to validate the example

³Equivalently, penalties α and β should be zero or positive, but finite

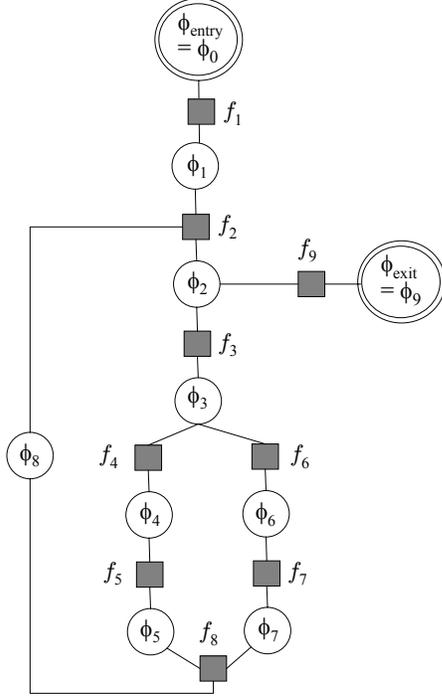


Figure 2. Factor graph of the program shown in Figure 3(a).

program since it only computes invariants that are conjunctions of linear inequalities [6].

Recently, Gulavani and Rajamani have proposed a technique [11] based on counterexample driven refinement for abstract interpretation that can compute disjunctive invariants like the ones required for the example program [11]. The key idea of their technique is to keep track of precision losses during forward fixed-point computation, and do a precise backward propagation from the error to either confirm the error as a true error, or to use refinement techniques (in particular, replacing a widen operation by a disjunction) so as to avoid the false error. The key to discovering the invariant $x < 50 \vee x = y$ is to realize that when the false branch of the if-condition is entered for the first time, then the value of x is 50. However, their refinement technique would only allow for discovering that when the false branch of the if-condition is entered for the first time then $x \geq k$, in the k^{th} stage of their refinement loop (for $k \leq 50$). Hence, their technique will take 50 refinement stages to discover the loop invariant $x \leq 50 \vee x = y$, which is required to prove the correctness (and if this constant 50 were larger, it would take a proportionally larger number of refinement steps). However, interestingly enough, their approach is able to work well on a modification of this example, in which all constants are replaced by symbolic constants.

Predicate abstraction [10] techniques based on counter-example driven refinement (like SLAM [1], BLAST [15], or [2]) are also able to discover such disjunctive invariants like the ones required for the example program. However, the success of these tools on a given problem is contingent on being able to find the right predicates. For the example program, these tools would go into a predicate refinement loop discovering unnecessary predicates $x = 1$, $x = 2$, $x = 3$, and so on, one by one. The number of refinement steps required for this particular example would be 100, (and potentially infinite, if these constants were larger or were symbolic constants).

Recently, Jhala and McMillan have proposed a predicate refinement approach based on interpolants [16], wherein the search of interpolants is restricted to a richer class of languages in successive stages of their algorithm. The choice of the languages L_k that they suggest involves all predicates that contain numeric constants no larger in absolute value than k from the constants that already occur in the program. Since the predicates required to prove the correctness of the example program belong to L_1 , their technique will be able to prove the correctness of the example program.

5.2 Interesting Aspects of the Algorithm

Note that predicate abstraction techniques, for a given set of predicates, compute invariants that are arbitrary boolean combination of the given set of predicates. However, an alternative thing to do would be to restrict the size of the formulas as opposed to restricting the set of predicates. The issue that arises with such an alternative is how to choose between the huge number of solutions that fit in the restricted size. We resolve this issue by performing both forward and backward analysis at the point where the abstract element is to be updated, instead of just performing a forward-only or backward-only analysis. More formally, we compute both weakest precondition $\text{Pre}(\pi)$ and strongest postcondition $\text{Post}(\pi)$ at the point π to be updated. We then choose a random solution ϕ that minimizes the inconsistency of the relationships $\text{Post}(\pi) \Rightarrow \phi$ and $\phi \Rightarrow \text{Pre}(\pi)$ ⁴ (since there may not be one with same level of inconsistency). This simple idea has several interesting aspects to it, which enable it to discover the proof of validity of the example program. We elaborate on these interesting aspects below, by taking as examples parts of proof searches generated by an implementation of our algorithm. In our examples, we consider the abstract domain \mathcal{A} to consist of all boolean formulas that involve at most 3 clauses (conjuncts), with each clause being a disjunction of at most 2 difference constraints.

Combination of Forward and Backward Analyses The generic framework of our learning-based algorithm allows for combining the complementary powers of forward and backward analyses in a simple manner to obtain a strictly more precise analysis. While updating ϕ_k , our algorithm involves computing both $\text{Post}(\pi_k)$ and $\text{Pre}(\pi_k)$. $\text{Post}(\pi_k)$ represents the forward flow of information from the immediate predecessors of π_k , while $\text{Pre}(\pi_k)$ represents the backward flow of information from the immediate successors of π_k . A forward analysis would simply consider the set $\text{Post}(\pi_k)$ and use some heuristic to choose some abstract element $\phi \in \mathcal{A}$ such that $\text{Post}(\pi_k) \Rightarrow \phi$. Our framework allows the forward analysis to guide its choice of ϕ by giving more preference to choices ϕ such that the inconsistency of $\phi \Rightarrow \text{Pre}(\pi_k)$ is minimized. Similarly, a backward analysis, which only considers $\text{Pre}(\pi_k)$, can also use immediate guidance from $\text{Post}(\pi_k)$ in our framework.

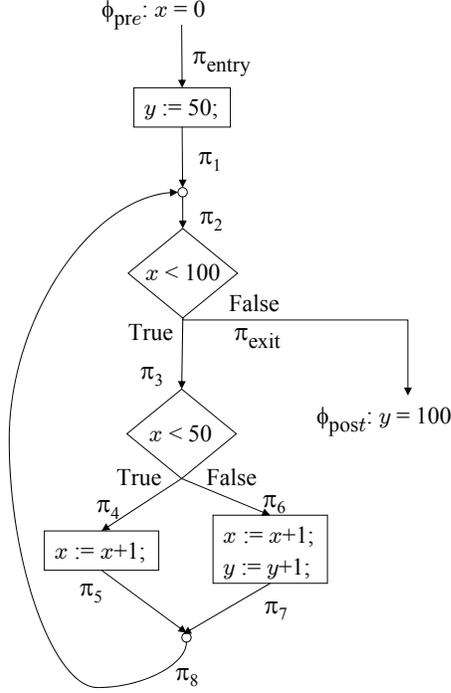
For example, consider the program shown in Figure 3. Suppose ϕ_8 is to be updated, and the abstract elements (which in this case are boolean formulas) at neighboring points of π_8 are as follows:⁵

$$\begin{aligned}\phi_5 &= (x \geq 0) \wedge (x \leq 50) \wedge (y = 50) \\ \phi_7 &= (x \geq 51) \wedge (x \leq 100) \wedge (x = y) \\ \phi_2 &= (x < 100 \vee y = 100)\end{aligned}$$

The above selection of abstract elements at program points π_5 , π_7 , and π_2 would also arise if two rounds of forward analysis and

⁴ More formally, the inconsistency of $\text{Post}(\pi) \Rightarrow \phi$ means the minimum of the inconsistencies of $\phi' \Rightarrow \phi$ (i.e., $\mathcal{M}(\phi', \phi)$) for any $\phi' \in \text{Post}(\pi)$. Similarly, the inconsistency of $\phi \Rightarrow \text{Pre}(\pi)$ means the minimum of the inconsistencies of $\phi \Rightarrow \phi'$ (i.e., $\mathcal{M}(\phi, \phi')$) for any $\phi' \in \text{Pre}(\pi)$.

⁵ For a discussion on how the predicate $x=y$ is discovered by our algorithm, see the discussion under the heading *Random Choices on Page 8*.



(a) Program

Program Point	Invariant
π_0	$x = 0$
π_1	$(y = 50) \wedge (x = 0)$
π_2	$(y = 50 \vee x \geq 50) \wedge (y = x \vee x < 50) \wedge (y = 100 \vee x < 100)$
π_3	$(y = 50 \vee x \geq 50) \wedge (y = x \vee x < 50) \wedge (y = 99 \vee x < 99)$
π_4	$(y = 50) \wedge (x < 50)$
π_5	$(y = 50) \wedge (x < 51)$
π_6	$(x \geq 50) \wedge (y = x \vee x < 50) \wedge (y = 99 \vee x < 99)$
π_7	$(x > 50) \wedge (y = x \vee x < 51) \wedge (y = 100 \vee x < 100)$
π_8	$(y = 50 \vee x \geq 50) \wedge (y = x \vee x < 50) \wedge (y = 100 \vee x < 100)$
π_9	$y = 100$

(b) Proof of validity.

Program Point	Invariant
π_0	$x \geq 100$
π_1	$(x \geq 100) \wedge (y = 50) \wedge (y - x \leq -1)$
π_2	$(x \geq 100) \wedge (y - x \geq 1 \vee y \neq 100)$
π_3	false
π_4	false
π_5	false
π_6	false
π_7	false
π_8	false
π_9	$y \neq 100$

(c) Proof of invalidity when precondition ϕ_{pre} is changed to true.

Figure 3. (a) shows an example program with pre and post conditions. (b) describes the proof of validity, which consists of invariants at each program point such that the invariants can be locally verified. (c) describes the proof of invalidity when precondition ϕ_{pre} is changed to true.

two rounds of backward analysis have been performed around the loop. Note that $\text{Post}(\pi_8)$ and $\text{Pre}(\pi_8)$ can be represented by the following formulas:⁶

$$\begin{aligned} \text{Post}(\pi_8) &: (x \geq 0 \wedge x \leq 50 \wedge y = 50) \vee \\ &\quad (x \geq 51 \wedge x \leq 100 \wedge x = y) \\ \text{Pre}(\pi_8) &: (x < 100 \vee y = 100) \end{aligned}$$

Observe that $\text{Post}(\pi_8)$ is equivalent to the following formula in conjunctive normal form, where each of the clauses is non-redundant.⁷

$$\begin{aligned} (x \leq 100) \wedge (x \leq 50 \vee x = y) \wedge \\ (x \geq 0) \wedge (y = 50 \vee x \geq 51) \end{aligned}$$

Note that we have fixed the abstract domain \mathcal{A} to consist of Boolean formulas involving at most 3 clauses, with each clause being a disjunction of at most 2 difference constraints. Dropping any one of the above 4 clauses yields an optimal over-approximation to $\text{Post}(\pi_8)$ that is an element of \mathcal{A} . However, note that the first

⁶ Technically, $\text{Post}(\pi_8)$ and $\text{Pre}(\pi_8)$ are sets of abstract elements. Hence, more formally, this means that any maximally strong formula that belongs to the abstract domain \mathcal{A} and is implied by the formula corresponding to $\text{Post}(\pi_8)$ belongs to $\text{Post}(\pi_8)$. Similarly, any minimally strong formula that belongs to the abstract domain \mathcal{A} and implies the formula corresponding to $\text{Pre}(\pi_8)$ belongs to $\text{Pre}(\pi_8)$.

⁷ The clause $y = 50 \vee x = y$ is redundant since it is implied by the conjunction of the given 4 clauses.

two clauses $x \leq 100$ and $(x \leq 50 \vee x = y)$ are required to prove $\text{Pre}(\pi_8)$. Hence, taking this guidance from $\text{Pre}(\pi_8)$, the forward analysis should include the first two clauses in its over-approximation of $\text{Post}(\pi_8)$. This is what our algorithm also does.

No distinction between Forward and Backward Information
One way to combine forward and backward analyses is to maintain the following two separate pieces of information at each program point, and use them to guide each other.

- Forward information: Over-approximation of program states that result when the program is executed under precondition. This is computed by the forward analysis.
- Backward information: Under-approximation of program states that ensure that the program will terminate in a state satisfying the postcondition. This is computed by the backward analysis.

The over-approximation process may take guidance from the backward information to ensure that the over-approximation at a program point is not weaker than the under-approximation computed at that point. (Similarly, the under-approximation process may take guidance from the forward information to ensure that the under-approximation computed at a program point is not stronger than the over-approximation computed at that point.) If these constraints cannot be met, they signal the presence of an excessive over-approximation or excessive under-approximation at some program point, which needs to be fixed. By excessive over-approximation, we mean that the invariants computed are weaker than those that

are necessary to prove the postcondition. (Similarly, by excessive under-approximation, we mean that the obligations that need to be established are stronger than what is indeed true of the program under precondition.) Unless the excessive nature of the over-approximation or under-approximation information is fixed, the forward or backward analysis cannot prove the validity of the pre/postcondition pair. The main issue however is to figure out the program points where this happened. We can only design heuristics for this purpose, which may work well for some examples and may not work well for other examples.

Our technique addresses this issue in an interesting manner. Observe that if there is no excessive over-approximation (with respect to precondition) and no excessive under-approximation (with respect to postcondition), then the under-approximation information is a valid over-approximation (assuming that the pre/postcondition pair is valid). Similarly, if there is no excessive under-approximation, then the over-approximation information is a valid under-approximation. To summarize, when no mistake occurs (i.e., both the under-approximation and the over-approximation are not excessive) the under-approximation is an over-approximation, and the over-approximation is an under-approximation. Then, why distinguish the two? Our technique thus maintains one piece of information at each program point, which is its guess for the correct invariant (= unexcessive over-approximation = unexcessive under-approximation) at that program point. Now, this guess may actually be the correct invariant (i.e., it is established by the precondition and is enough to establish the postcondition), or it may be an excessive over-approximation (i.e., weaker than the invariant required to establish the postcondition) or an excessive under-approximation (i.e., it may be stronger than what is true when the precondition holds). The challenge now is that we do not really know which of this is true. The only thing that we know is whether or not these guesses are consistent with the neighboring guesses. However, corrections automatically creep in as the algorithm makes progress trying to make the guesses more consistent with their neighbors, wherein the guesses can get strengthened as well as weakened. Contrast this with the above case where the forward and backward information is kept separate. In that case, inconsistency is in the form of the over-approximation getting weaker than the under-approximation (however, each of the two pieces of information are individually consistent with the corresponding information at the neighboring nodes). But when an inconsistency is detected, it has to be fixed before a proof can be discovered.

It is also interesting to contrast our technique (which maintains one piece of information at each program point) with the backward-only technique (which also maintain only one piece of information at each program point). Note that if in the backward analysis, the under-approximation becomes excessive (i.e., stronger than what is really true about the program at that point given the precondition), validity of pre/postcondition pair cannot be established. Comparatively, if the information computed by our technique is an excessive under-approximation, it will have a chance of recovery (i.e., it has the potential to get strengthened by forward flow of information, which does not happen in a backward-only analysis). A similar comparison holds with the forward-only technique.

As an example, we discuss below how the invariant $x < 50 \vee x = y$ gets discovered at program point π_2 after it is excessively under-approximated to $x = y$ at some stage, in one of the proof searches seen in our experiments. The following is a snapshot of the formulas that were found at some step during the proof search.

$$\begin{aligned}\phi_1 &= (x = 0) \wedge (y = 50) \\ \phi_2 &= (x = y) \wedge (x < 100 \vee y = 100) \\ \phi_3 = \phi_6 &= (x < 99 \vee y = 99) \wedge (x \neq 98 \vee y = 98)\end{aligned}$$

$$\begin{aligned}\phi_4 &= (x < 99 \vee y = 100) \wedge (x \neq 98 \vee y = 99) \\ \phi_5 = \phi_7 = \phi_8 &= (x < 100 \vee y = 100) \wedge (x \neq 99 \vee y = 99)\end{aligned}$$

Snapshot 1

Observe that the above snapshot at program points $\pi_3, \pi_4, \pi_5, \pi_6$, and π_7 makes sense from a backward analysis point of view, wherein, if the postcondition $y = 100$ is pushed backward through the loop two times, we get the above configuration. The above value of ϕ_2 is an under-approximation of $\text{Pre}(\pi_2)$, which is $(x \neq 99 \vee y = 99) \wedge (x \neq 98 \vee y = 98) \wedge (x < 100 \vee y = 100)$. However, it is an excessive under-approximation since $x = y$ is not always true at π_2 . Now, if this were a pure backward analysis, then the validity of the program cannot be established after $\text{Pre}(\pi_2)$ is under-approximated to the above value of ϕ_2 . On the contrary, our algorithm is able to recover from such an excessive under-approximation because the information at a program point gets updated from both forward and backward directions (which results in weakening and strengthening of the information respectively). In this particular case, our algorithm chooses to update $\phi_8, \phi_7, \phi_6, \phi_3$ (but not ϕ_5 and ϕ_4) which changes the above snapshot as follows (This can still be seen as backward propagation of information from ϕ_2).

$$\begin{aligned}\phi_1 &= (x = 0) \wedge (y = 50) \\ \phi_2 &= (x = y) \wedge (x < 100 \vee y = 100) \\ \phi_3 &= (x < 50 \vee x = y) \wedge (x < 99 \vee y = 99) \\ \phi_4 &= (x < 99 \vee y = 100) \wedge (x \neq 98 \vee y = 99) \\ \phi_5 &= (x < 100 \vee y = 100) \wedge (x \neq 99 \vee y = 99) \\ \phi_6 &= (x = y) \wedge (x < 99 \vee y = 99) \\ \phi_7 = \phi_8 &= (x = y) \wedge (x < 100 \vee y = 100)\end{aligned}$$

Snapshot 2

Observe that $\text{Pre}(\pi_2)$ now is $(x \geq 100 \vee x < 50 \vee x = y) \wedge (x < 100 \vee y = 100)$. Now, the algorithm (randomly) decides to update ϕ_2 and detects that it is inconsistent from the forward direction, but it can be made consistent in both directions by updating it to:

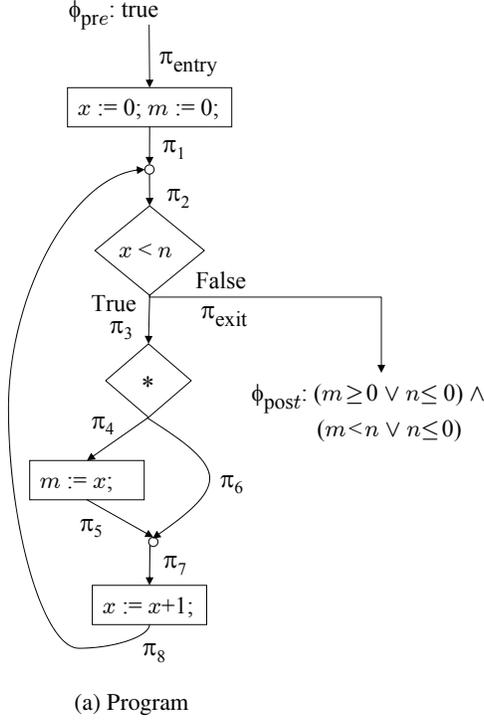
$$\phi_2 = (x < 50 \vee x = y) \vee (x < 100 \vee y = 100)$$

Note that the discovery of invariant $x < 50 \vee x = y$ at π_2 is crucial in order to validate the program. There are two crucial things to observe as to how the algorithm discovered $x < 50 \vee x = y$ at π_2 in the above instance. Observe that ϕ_5 and hence ϕ_4 did not get updated, and that the algorithm tried to weaken ϕ_2 to make it more consistent in the forward direction, and was able to find something that made it fully consistent in both directions.

Random choices An important aspect of the algorithm is that it makes some random choices.

One of the random choices that the algorithm makes is to decide which program point to update. Observe that in the above proof search instance, if ϕ_4 also got updated at the same time when ϕ_8, ϕ_7, ϕ_6 and ϕ_3 got updated between snapshot 1 and 2, the algorithm might have chosen something else for ϕ_2 because then it would not have been possible to choose anything for ϕ_2 that makes it consistent in both directions. There is no clear strategy to decide in which order to update the program points. Hence, randomness plays an important role here.

However, the interesting thing is that the chances of such incidents happening, though small, are not rare. For example, another sequence of updates that we saw in another proof search in producing the invariant $(x < 51 \vee x = y)$ at program point π_2 after snapshot 1 is as follows. The formulas ϕ_4 and ϕ_5 get updated with the forward information from ϕ_3 . Then, ϕ_3, ϕ_6 and ϕ_7 get updated



Program Point	Invariant
π_0	true
π_1	$(x = 0) \wedge (m = 0)$
π_2	$(x \geq 0 \vee n \leq 0) \wedge (m \geq 0 \vee n \leq 0) \wedge (m < n \vee n \leq 0)$
π_3	$(x \geq 0 \vee n \leq 0) \wedge (m \geq 0 \vee n \leq 0) \wedge (m < n \vee n \leq 0)$
π_4	$(x \geq 0 \vee n \leq 0) \wedge (m < n \vee n \leq 0)$
π_5	$(x \geq -1 \vee n \leq 0) \wedge (m \geq 0 \vee n \leq 0) \wedge (m < n \vee n \leq 0)$
π_6	$(x \geq -1 \vee n \leq 0) \wedge (m \geq 0 \vee n \leq 0) \wedge (m < n \vee n \leq 0)$
π_7	$(x \geq -1 \vee n \leq 0) \wedge (m \geq 0 \vee n \leq 0) \wedge (m < n \vee n \leq 0)$
π_8	$(x \geq 0 \vee n \leq 0) \wedge (m \geq 0 \vee n \leq 0) \wedge (m < n \vee n \leq 0)$
π_9	$(m \geq 0 \vee n \leq 0) \wedge (m < n \vee n \leq 0)$

Figure 4. (a) shows an example program with pre and post conditions. (b) describes the proof of correctness, which consists of invariants at each program point such that the invariants can be locally verified.

based on the forward information from ϕ_2 . This results in the following snapshot:

$$\begin{aligned}
 \phi_1 &= (x = 0) \wedge (y = 50) \\
 \phi_2 &= (x = y) \wedge (x < 100 \vee y = 100) \\
 \phi_3 &= (x < 100) \wedge (x = y) \\
 \phi_4 &= (x < 50) \\
 \phi_5 &= (x < 51) \\
 \phi_6 &= (x \geq 50) \wedge (x < 100) \wedge (x = y) \\
 \phi_7 &= (x \geq 51) \wedge (x < 101) \wedge (x = y)
 \end{aligned}$$

Now, ϕ_8 is updated to the following formula to minimize its local inconsistency at π_8 (which is a function of ϕ_5 , ϕ_7 , and ϕ_2):

$$(x < 51 \vee x = y) \wedge (x < 101)$$

This is followed by ϕ_2 getting updated to ϕ_8 to minimize the local inconsistency. The crucial point in the above sequence of updates is that ϕ_4 and ϕ_6 get updated from choice of ϕ_3 before ϕ_3 gets updated from choice of ϕ_2 .

The other random choice that the algorithm makes is in choosing the abstract element at a given program point π_k . It is possible that there are several abstract elements that are locally consistent at π_k , but some of those choices may be better than the others. For example, consider the following snapshot that arose in one of the proof searches in our experiments.

$$\begin{aligned}
 \phi_1 &= (x = 0) \wedge (y = 50) \\
 \phi_3 &= (x < 99 \vee y = 99) \wedge (x \neq 98 \vee y = 98) \wedge \\
 &\quad (x \neq 97 \vee y = 97) \\
 \phi_8 &= (x < 100 \vee y = 100) \wedge (x \neq 99 \vee y = 99) \wedge \\
 &\quad (x \neq 98 \vee y = 98)
 \end{aligned}$$

$\text{Pre}(\pi_2)$ can be represented by the following formula:

$$(x < 100 \vee y = 100) \wedge (x < 97 \vee x > 99 \vee x = y)$$

Note that we have fixed the abstract domain \mathcal{A} to consist of Boolean formulas involving at most 3 clauses, with each clause being a disjunction of at most 2 difference constraints. Hence, dropping any one or more of the disjuncts from the clause $(x < 97 \vee x > 99 \vee x = y)$ would satisfy the size restriction. However, guidance from $\text{Post}(\pi_2)$ suggests that $x < 97$ must be included to minimize the inconsistency. Thus, the following 3 are good choices for ϕ_2 .

$$\text{Choice 1} : (x < 100 \vee y = 100) \wedge (x < 97)$$

$$\text{Choice 2} : (x < 100 \vee y = 100) \wedge (x < 97 \vee x > 99)$$

$$\text{Choice 3} : (x < 100 \vee y = 100) \wedge (x < 97 \vee x = y)$$

Choice 3 is better than the other two choices since it yields the important predicate $x = y$ required later to discover the invariant $x < 50 \vee x = y$. Since there is no clear strategy what to choose, randomness plays a crucial role here.

6. Case Study: Boolean Combinations of Difference Constraints

We implemented the learning algorithm for programs P whose assignment statements s and conditional predicates p have the following form:

$$\begin{aligned}
 s &: x := e \\
 p &: x = e \mid x \neq e \mid x < e \mid x > e \mid x \leq e \mid x \geq e \\
 e &: c \mid y + c
 \end{aligned}$$

Here x and y refer to some integer program variables, while c refers to some integer constant. The predicates p as defined above are also called *difference constraints*.

We chose the abstract domain \mathcal{A} whose elements are boolean combinations of difference constraints among program variables. In particular, for computational reasons, we restricted the abstract domain \mathcal{A} to include boolean formulas with a specific template, namely boolean formulas that when expressed in a conjunctive normal form have at most m conjuncts, and each conjunct having at most n disjuncts, each of which is a difference constraint (also referred to as $m \times n$). In our experiments that are described in Section 6.1, we chose $m \in \{3, 4, 5\}$ and $n \in \{2, 3\}$. Such a template choice is also justified by the fact that most programs are correct for simple reasons, and their proof of validity is expressible using some appropriate small representation.

We used the following inconsistency measure on the above abstract domain \mathcal{A} . The inconsistency of $\mathcal{M}(\phi, \phi')$ is the sum of the inconsistencies of $\mathcal{M}(\phi, C_i)$ for each clause C_i in ϕ' , divided by the total number of clauses in ϕ' .

$$\mathcal{M}(\phi, \bigwedge_{i=1}^m C_i) = \sum_{i=1}^m \frac{1}{m} \times \mathcal{M}(\phi, C_i)$$

The inconsistency $\mathcal{M}(\phi, C_i)$ is proportional to the number of disjuncts D_j in the disjunctive normal form of ϕ that do not imply clause C_i .

$$\mathcal{M}(\bigvee_{j=1}^k D_j, C_i) = \sum_{j=1}^k \frac{1}{k} \times \mathcal{M}(D_j, C_i)$$

The inconsistency of $\mathcal{M}(D_j, C_i)$ is defined to be 0 or 1 depending on whether $D_j \Rightarrow C_i$ or not respectively.

We implemented Line 4 in the `FindProof` procedure as follows. We considered the set of abstract elements that minimize the penalty at program point π_k (given `Post`(π_k) and `Pre`(π_k)) and chose an element randomly from it. However, in order to expedite the convergence process, we implemented a simple version of widening and narrowing, which can be regarded as choosing abstract elements that do not minimize the penalty with a lesser probability.

6.1 Experiments

We have built a prototype tool in C called `Magic`⁸. We describe our preliminary experience with this tool on the two programs shown in Figure 3 and Figure 4. We do not know of any current tool that can automatically generate the proof of validity of the example in Figure 3. The program in Figure 4 was chosen as a contrasting example - its structure resembles very closely to that of the program in Figure 3, but it is easier to validate, and has been used as a motivating example for some existing verification techniques [11, 19]. Figure 5 contains histograms of the numbers of updates over different runs of our tool on these programs. For different algorithm parameters, we ran our tool 200 times and recorded average number of updates per program point needed to discover the invariant. The figures show histograms of the number of updates over the 200 tests: the y-axis shows the number of runs that ended up in the average number of runs within the boundaries of the bin centered at the values on the x-axis.

Proof of validity of example in Figure 3 We first ran our tool on the program shown in Figure 3(a) with the constants 50 and 100 replaced by the bigger constants 5000 and 10000 respectively. We chose the size of boolean formulas over difference constraints as 4×3 (i.e., at most 4 clauses, with each clause having at most 3 disjuncts in the CNF representation). Our tool is successfully able to generate the proof of validity of this example. One such

⁸`Magic` is an acronym for Machine-learning based Automatic Generation of Invariants in Code

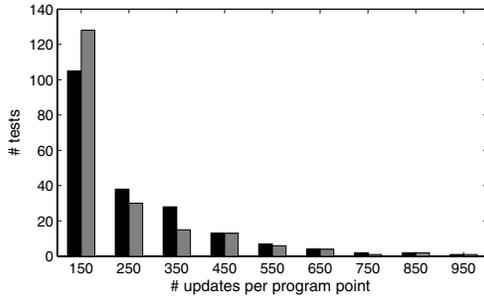
proof is shown in Figure 3(b). The chart in Figure 5 shows the average number of updates (per program point) required to discover the proof of validity over 200 different runs of the algorithm. For example, the first dark bin in Figure 5(a) has 105 of the 200 tests and is centered at 150 updates per program point, which means that around 50% of cases needed around 150 updates before the program invariant is found, on the other hand 38 of the 200 tests needed between 200 and 300 updates before convergence. The graph shows that it is unlikely to have to run the tool for longer than 500 updates per program point before discovering the invariant.

Incremental proof of validity At the end of each of the 200 runs of the tool in the above case, we changed the program slightly (we replaced occurrences of the constant 5000 by 6000) and continued the algorithm (with its current state of formulas ϕ_k at each program point π_k) to discover the proof of validity of the modified program. Observe that the modified program requires a small change in the proof of validity. The goal of this experiment was to illustrate that the algorithm is smart enough to converge faster if starting from a partially correct proof as opposed to starting from scratch. This is indeed what we find experimentally: the gray histogram shows the distribution of the number of additional updates the tool needed to refine the invariant. On average, the 200 tests starting from scratch required 235 updates per program point to discover the invariant, but recovering from a small program change required on average additional 195 updates. The trend is statistically significant ($p < 10^{-3}$).

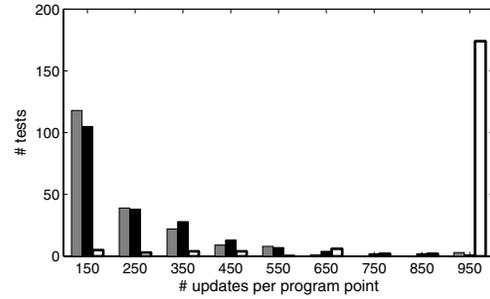
Effect of program constants One way to discover the proof of validity is to run the program fully, i.e. to run through the program loops until all the termination conditions are fulfilled and the end of program is reached. However, our algorithm always finds true invariants in a manner different than this. To show this, we rerun the tool 200 times using a smaller constants (this time using the constants 50 and 100, as is the case in the program shown in Figure 3(a)) as shown in loop termination conditions to see how this would affect the program. The distributions over the number of updates did not differ significantly between the two cases. In fact, under a randomized pairing, we found that the verification of the program with the larger constant terminated faster than that of the one with a smaller constant in 101 out of 200 tests. This agrees with our visual inspection of the results which show qualitatively the same invariants found in both cases (only the constants are different), but it also indicates that the tool is highly unlikely to go through the entire loop before getting a clue about what an invariant should be.

Changing template of boolean formulas Next, in Figure 5(b), we compare the number of updates per program point for the tools that use different size of the abstract representation ϕ_k at each program point. The dark histogram is the same as in (b), while the gray histogram corresponds to the tests with larger representation (5×3), and the white histogram corresponds to the smaller representation (3×2). The important conclusion is that using tight representation (of exactly the right size), reduces the tool's ability to reach all possible expression combinations. When the representation is larger, then the extra room effectively smoothes the probability distribution by making it possible to express the invariant in multiple redundant ways.

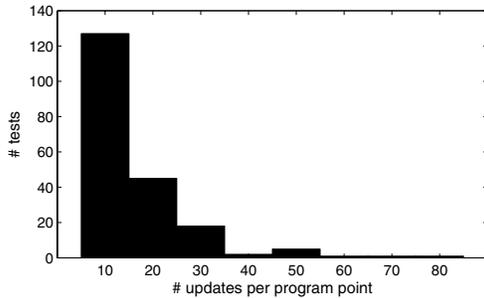
Proof of validity of example in Figure 4 To illustrate that the algorithm performance will depend on the difficulty of the program, we also ran our algorithm on second verification problem, known to be verifiable by other techniques [11, 19]. Figure 5(c) shows the histogram of the number of updates for this problem. Even though the number of program points is the same, and the tool's settings



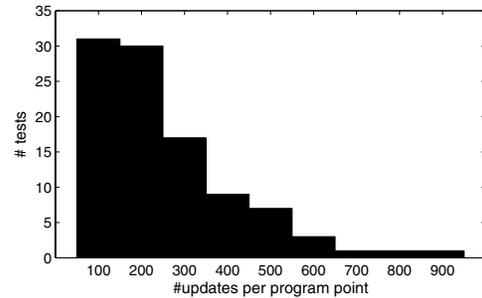
(a): Proof/Incremental Proof of Validity of program in Figure 3



(b): Different sizes of boolean formulas



(c) Validity proof of program in Figure 3



(d) Invalidity proof of program in Figure 4

Figure 5. The distribution over the number of updates in the Gibbs sampling algorithm before the program invariant is discovered. Dark bins in (a) show the histogram over the runs that all used a slightly enlarged representation than necessary (4×3 instead of 3×2 , which is sufficient to represent invariants. This version converges the fastest among the ones we tried. The gray bins in (a) show how many updates are typically necessary to change the found invariant into a new one that satisfy a slightly changed program. Since the invariant does not change significantly, the number of updates per program point is lower than it is when the sampling is started from scratch. In (b) we illustrate the effect of the limit on the size of the abstract representation to program convergence. Gray denotes lot of extra room (5×3), black denotes some extra room (4×3), and white denotes tight space (3×2). In (c) we show the update histogram for solving the second verification problem, and in (d) the update histogram for solving the invalidity problem.

were the same, this problem is easier, and the algorithm discovers the invariant much faster.

Proof of Invalidity of Example 1 Finally, in Figure 5(d), we illustrate the performance on the invalidity proof of the program in Figure 3 when its precondition is changed to `true`. This is to show that our tool works equally for discovering proofs of invalidity as well proofs of validity (under the assumption that the program terminates on all inputs). Figure 3(c) shows one of the proofs of invalidity generated by our tool. In a different invalidity proof, the tool even generated the weakest condition $x \geq 51$ at π_{entry} .

7. Related Work

The idea of applying machine learning techniques in programming languages has been used recently, though for the different problem of discovering small programs given input-output pairs [17]. In this paper, we use machine learning techniques for the problem of program verification.

Combination of Forward and Backward Analyses Cousot and Cousot proposed a technique to combine forward and backward analyses by refining the over-approximation of the intersections of precondition and negated postcondition by an iterative forward and backward analysis [4, 5]. Dill and Wong Toi proposed a different kind of forward-backward combination that consists of computing separate upper-approximation and lower-approximation of precondition and postcondition respectively [8]. Leino and Logozzo also combine the forward inference procedure with the goal-

driven nature of going backward by invoking the forward analysis along infeasible error traces reported during a backward analysis in the hope of generating stronger loop invariants to rule out those traces [19]. We have a different kind of forward and backward combination in which we do not distinguish between the forward and backward information, and information flows in both forward and backward directions in each step of the algorithm.

Predicate abstraction with counter-example guided refinement

This technique involves using a model-checker to compute an over-approximation of a set of reachable states of a program using boolean formulas over a given set of predicates. If this set of reachable states intersects with the set of error states, the model-checker provides a counter-example. A theorem prover is then used to check the validity of that counter-example. If the counter-example is found to be invalid, the proof of invalidity provides additional predicates that should be considered to avoid this counterexample next time. The process is then repeated with these new set of predicates.

There are some interesting differences between this technique of predicate abstraction with counter-example guided refinement and our technique.

- Computation of reachable states can be regarded as a forward analysis, while counter-example discovery and its feedback to refine the set of predicates can be regarded as a backward analysis. However, this forward analysis and backward analysis happens in two different phases. Our forward and backward

analysis is more tightly integrated and happens in one phase, thus providing an immediate feedback.

- Predicate abstraction is limited to computing invariants using a given set of predicates inside a fixed iteration. However, our technique is not limited to considering a fixed set of predicates during any step. The only restriction is that the invariant at any program point should come from some language whose elements can be represented finitely. For example, in our implementation, we choose this language to consider all Boolean formulas with bounded number of clauses, with each clause being a disjunction of bounded number of difference constraints. Such a choice of language in our technique has the ability to consider the space of all predicates (but boolean formulas of bounded size) as opposed to predicate abstraction, which considers a fixed set of predicates (but arbitrary boolean formulas over them).

Interpolants Line 4 in the procedure `FindProof` in our algorithm involves choosing an abstract element $\phi' \in \mathcal{A}$ at program point π , such that ϕ' is likely to be least inconsistent with ϕ and ϕ'' , where $\phi = \bigwedge_{\tilde{\phi} \in \text{Post}(\pi)} \tilde{\phi}$ and $\phi'' = \bigvee_{\tilde{\phi} \in \text{Pre}(\pi)} \tilde{\phi}$. More formally, this

means that we want to find ϕ' such that the set of program states that violate $\phi \Rightarrow \phi'$ or $\phi' \Rightarrow \phi''$ is minimal.⁹ We refer to such a process as the *sandwich step*.

The sandwich step in our algorithm can be viewed as a generalization of the well-known interpolant procedure in theorem proving. Given a pair of formulas $\langle \phi_1, \phi_3 \rangle$ such that $\phi_1 \Rightarrow \phi_3$, an interpolant for $\langle \phi_1, \phi_3 \rangle$ consists of a formula ϕ_2 such that $\phi_1 \Rightarrow \phi_2$, $\phi_2 \Rightarrow \phi_3$, and $\phi_2 \in \mathcal{L}(\phi_1) \cap \mathcal{L}(\phi_3)$. The Craig interpolation lemma [7] states that an interpolant always exists when ϕ_1 and ϕ_3 are formulas in first-order logic.

The sandwich step in our algorithm generalizes the interpolant problem in several dimensions in the context of abstract computation over programs. Instead of enforcing that $\phi_2 \in \mathcal{L}(\phi_1) \cap \mathcal{L}(\phi_3)$, the sandwich step imposes the constraint that $\phi_2 \in \mathcal{A}$, for any given language \mathcal{A} . Furthermore, the sandwich step does not insist that the inputs ϕ_1 and ϕ_3 satisfy $\phi_1 \Rightarrow \phi_3$. The immediate consequence of (either of) these generalizations is that the existence of ϕ_2 such that $\phi_1 \Rightarrow \phi_2$ and $\phi_2 \Rightarrow \phi_3$ is no longer guaranteed, even when ϕ_1 and ϕ_3 are formulas in first-order logic. However, the sandwich step insists on finding a formula ϕ_2 that fits as best as possible between ϕ_1 and ϕ_3 , i.e., the number of program states that violate $\phi_1 \Rightarrow \phi_3$ or $\phi_3 \Rightarrow \phi_2$ is minimal.

Jhala and McMillan have recently proposed a predicate refinement [16] approach based on interpolants, wherein the search of interpolants is restricted to a richer class of languages in successive stages of their algorithm. This is similar to one of the generalizations of the interpolant problem mentioned above, wherein ϕ_2 is constrained to belong to \mathcal{A} . However, the choice of the languages suggested in Jhala and McMillan’s work is quite different from the language \mathcal{A} that we use in our implementation. Their language L_k involves predicates that contain numeric constants no larger in absolute value than k from the constants that already occur in the program.

However, the choice of the language used in our implementation places restrictions on the size of the boolean formulas, which is motivated by the fact that most programs are usually correct for simple reasons that can be expressed using some small representation. It would be interesting to consider the intersection of these languages for faster convergence.

⁹In other words, the following set of program states is minimal:

$$(\gamma(\phi) - \gamma(\phi')) \cup (\gamma(\phi') - \gamma(\phi''))$$

Our choice of languages however raises the important issue of how to choose between several solutions ϕ_2 that are equally consistent with ϕ_1 and ϕ_3 (i.e., the number of program states that violate $\phi_1 \Rightarrow \phi_3$ or $\phi_3 \Rightarrow \phi_2$ is same for all of them) and fit within the space restriction of the language \mathcal{A} . The strategy that we use in our implementation is to choose solutions ϕ_2 that are more close to ϕ_1 (i.e., the number of program states that violate $\phi_2 \Rightarrow \phi_1$ is small) or ϕ_3 (i.e., the number of program states that violate $\phi_3 \Rightarrow \phi_2$ is small). The former choice has the effect of a forward analysis that is guided by the backward information, while the latter choice has the effect of a backward analysis that is guided by the forward information.

However, the biggest difference with Jhala and McMillan’s work is that their work is an instance of predicate abstraction. Their approach involves computing interpolants (which can also be seen as combination of forward and backward analysis) only during the predicate-refinement step, which happens in every alternative phase after performing the standard reachability computation. On the other hand, our algorithm performs the sandwiching procedure (which is a combination of forward and backward analysis) at each step.

Probabilistic Algorithms Gulwani and Necula developed a probabilistic technique for program analysis (called *random interpolation* [12–14]) that combines the complementary strengths of abstract interpretation and random testing. This technique involves computing and manipulating program invariants efficiently by representing them by a small random sample of the set of program states that they represent (as opposed to manipulating program invariants symbolically). However, there are some differences worth-mentioning with the probabilistic technique described in this paper: (a) Random Interpretation terminates in a bounded time, but may output incorrect program invariants with a (infinitesimally) small probability. On the other hand, the technique described in this paper always outputs the correct answer, but may take long to execute. (b) Random Interpretation is a forward technique that is used to discover program properties. The technique described in this paper is a combination of forward and backward analyses and is used to discover the proof of correctness of a given pre/postcondition pair.

8. Conclusion and Future Work

In this paper, we have described a simple probabilistic inference algorithm that searches for proofs of validity or invalidity of given Hoare triples. The algorithm works by randomly choosing a program point and randomly updating its guess for the invariant at that point to make it less inconsistent with the neighboring guesses until a valid proof is found. This simple algorithm combines forward and backward analyses in a novel manner.

Furthermore, we pose the invariant inference problem as inference in probabilistic graphical models, which allows for a large class of other probabilistic inference techniques to be applied to program verification. To this end, we have introduced the notion of an inconsistency measure for an abstract domain equipped with a partial order. This measure can be used to create a probability distribution over the program states, described by a graphical model amenable to various probabilistic inference techniques (e.g., the ones reviewed in [9, 20]). It is important to note that, even though the problem is re-formulated as an inference of hidden variables in a probability model, many inference algorithms used in machine learning are *deterministic*, and also new classes of deterministic algorithms can be developed to leverage the real-valued inconsistency measures to search for proofs of validity or invalidity. These measures can be an effective measure of algorithm progress.

Our algorithm is based on one of the simplest sampling approaches, but a plethora of other related sampling algorithms is reviewed in [20].

Possible extensions of this work include discovering invariants that involve pointer variables, and performing an interprocedural analysis by learning procedure summaries. It would also be interesting to experiment with the techniques described in this paper to learn invariants in richer abstract domains such as first order logic invariants, which would be useful to reason about programs with arrays.

9. Acknowledgments

We thank Vladimir Jovic for general discussions on applying machine learning techniques to reason about programs.

References

- [1] T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.
- [2] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE*, pages 385–395. IEEE Computer Society, May 2003.
- [3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on POPL*, pages 234–252, 1977.
- [4] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, July 1992.
- [5] P. Cousot and R. Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering: An International Journal*, 6(1):69–95, Jan. 1999.
- [6] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on POPL*, pages 84–97, 1978.
- [7] W. Craig. Three uses of the Herbrand-Genzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22:269–285, 1957.
- [8] D. Dill and H. Wong-Toi. Verification of real-time systems by successive over and under approximation. *Lecture Notes in Computer Science*, 939, 1995.
- [9] B. J. Frey and N. Jovic. A comparison of algorithms for inference and learning in probabilistic graphical models. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 27(9):1392–1416, 2005.
- [10] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.
- [11] B. Gulavani and S. Rajamani. Counterexample driven refinement for abstract interpretation. In *TACAS*, volume 3920 of *LNCIS*, pages 474–488. Springer, Mar. 2006.
- [12] S. Gulwani and G. C. Necula. Discovering affine equalities using random interpretation. In *30th ACM Symposium on POPL*, pages 74–84. ACM, Jan. 2003.
- [13] S. Gulwani and G. C. Necula. Global value numbering using random interpretation. In *31st ACM Symposium on POPL*, pages 342–352, Jan. 2004.
- [14] S. Gulwani and G. C. Necula. Precise interprocedural analysis using random interpretation. In *32nd ACM Symposium on POPL*, pages 324–337, Jan. 2005.
- [15] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.
- [16] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In H. Hermanns and J. Palsberg, editors, *TACAS*, volume 3920, pages 459–473. Springer, 2006.
- [17] V. Jovic, S. Gulwani, and N. Jovic. Probabilistic inference of programs from input/output examples. (MSR-TR-2006-103), July 2006.
- [18] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Trans. Information Theory*, 47(2):7–47, 2001.
- [19] K. R. M. Leino and F. Logozzo. Loop invariants on demand. In *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 119–134. Springer, 2005.
- [20] R. Neal. Probabilistic inference using markov chain monte carlo methods. Technical Report CRG-TR-93-1, University of Toronto, Sept. 1993.