Noname manuscript No.
(will be inserted by the editor)

From Invariant Checking to Invariant Inference Using Randomized Search

Rahul Sharma · Alex Aiken

Received: date / Accepted: date

Abstract We describe a general framework C2I for generating an invariant inference procedure from an invariant checking procedure. Given a checker and a language of possible invariants, C2I generates an inference procedure that iteratively invokes two phases. The search phase uses randomized search to discover candidate invariants and the validate phase uses the checker to either prove or refute that the candidate is an actual invariant. To demonstrate the applicability of C2I, we use it to generate inference procedures that prove safety properties of numerical programs, prove non-termination of numerical programs, prove functional specifications of array manipulating programs, prove safety properties of string manipulating programs, and prove functional specifications of heap manipulating programs that use linked list data structures.

Keywords Verification · Loop Invariants · Markov Chain Monte Carlo (MCMC) · Satisfiability Modulo Theories (SMT)

This work was supported by National Science Foundation grant CCF-1160904, a Microsoft fellowship, and the Air Force Research Laboratory under agreement number FA8750-12-2-0020. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

R. Sharma
Department of Computer Science, Stanford University
E-mail: sharmar@cs.stanford.edu

A. Aiken
Department of Computer Science, Stanford University
E-mail: aiken@cs.stanford.edu

1 Introduction

In traditional program verification, a human annotates the loops of a given program with invariants and a decision procedure checks these invariants by proving some *verification conditions* (VCs). We explore whether decision procedures can also be used to infer the loop invariants; doing so helps automate one of the core problems in verification (discovering appropriate invariants) and relieves programmers from a significant annotation burden.

The idea of using decision procedures for invariant inference is not new [30, 18]. However, this approach has been applied previously only in domains with some special structure, e.g., when the VCs belong to theories that admit quantifier elimination, such as linear rational arithmetic (Farkas' lemma) or linear integer arithmetic (Cooper's method). For general inference tasks, such theory-specific techniques do not apply, and the use of decision procedures for such tasks has been restricted to invariant checking: to prove or refute a given manually provided candidate invariant.

We describe a general framework C2I that, given a procedure for checking invariants, uses that checker to produce an invariant inference engine for a given language of possible invariants. We apply C2I to various classes of invariants; we use it to generate inference procedures that prove safety properties of numerical programs, prove non-termination of numerical programs, prove functional specifications of array manipulating programs, prove safety properties of string manipulating programs, and prove functional specifications of heap manipulating programs that use linked list data structures. The two main characteristics of C2I are

- The decision procedure is only used to check a program annotated with candidate invariants (in contrast to approaches that use the decision procedure directly to infer an invariant).
- C2I uses a randomized search algorithm to search for candidate invariants. Empirically, the search technique is effective for generating good candidates for various classes of invariants.

The use of a decision procedure as a checker for candidate invariants is also not novel [37,39,53,54,46,22,21]. The main contribution of this paper is a general and effective search procedure that makes a framework like C2I feasible. The use of randomized search is motivated by its recent success in program synthesis [51,1] and recognizing that invariant inference is also a synthesis task. More specifically, our contributions are:

- We describe a framework C2I that iteratively invokes randomized search and a decision procedure to perform invariant inference. The randomized search combines random walks with hill climbing and is an instantiation of the well-known Metropolis Hastings MCMC sampler [11].
- We empirically demonstrate the generality of our search algorithm. We use randomized search for finding numerical invariants, *recurrent sets* [29], universally quantified invariants over arrays, invariants over string operators,

and invariants involving reachability predicates for linked list manipulating programs. These studies show that invariant inference is amenable to randomized search.
- Randomized search is effective only when done efficiently. We describe optimizations that allow us to obtain effective randomized search algorithms for invariant inference.

Even though we expect the general inference engines based on randomized search to be inferior in performance to the domain specific invariant inference approaches, our experiments show that randomized search has competitive performance with the more specialized techniques. This outcome does not prove that randomized search will always be competitive with techniques tuned to a particular domain, but does show that randomized search is worth evaluating, as it is usually simple to implement. A subset of the results shown in this paper appear in [52].[1] The rest of the paper is organized as follows. We describe our search algorithm in Section 2. Next, we describe inference of numerical invariants in Section 3, universally quantified invariants over arrays in Section 4, string invariants in Section 5, and invariants over linked lists in Section 6. Finally, we discuss related work in Section 7 and conclude in Section 8.

2 Preliminaries

An imperative program annotated with invariants can be verified by checking some *verification conditions* (VCs), which must be discharged by a decision procedure. As an example, consider the following program:

$$\texttt{assume}\ P; \texttt{while}\ B\ \texttt{do}\ S\ \texttt{od}; \texttt{assert}\ Q$$

The loop has a pre-condition P. The entry to the loop is guarded by the predicate B and S is the loop body (which, for the moment, we assume to be loop-free). We assert that the states obtained after execution of the loop satisfy Q. Given a loop invariant I, we can prove that the assertion holds if the following three VCs are valid:

$$P \Rightarrow I \tag{1}$$

$$\{I \wedge B\}S\{I\} \tag{2}$$

$$I \wedge \neg B \Rightarrow Q \tag{3}$$

Given a *candidate invariant* C, a decision procedure checks the conditions of Equations 1, 2, and 3. Since there are three conditions for a predicate to be an invariant, there are three queries that need to be discharged to check a candidate. Each query, if it fails, generates a different kind of counterexample. We discuss these next.

[1] The conference version, [52], lacks a comparison between pure random search and MCMC search. The treatment of numerical invariants also differs.

Equation 1 states that for any invariant I, any state that satisfies P also satisfies I. However, if $P \wedge \neg C$ has a satisfying assignment g, then $P(g)$ is *true* and $C(g)$ is *false* and hence g proves C is not an invariant. We call any state that must be satisfied by an actual invariant, such as g, a *good* state. Now consider Equation 2. A *pair* (s,t) satisfies the property that s satisfies B and if the execution of S is started in state s then S can terminate in state t. Since an actual invariant I is inductive, it should satisfy $I(s) \Rightarrow I(t)$. Hence, a pair (s,t) satisfying $C(s) \wedge \neg C(t)$ proves C is not an invariant. Finally, consider Equation 3. A satisfying assignment b of $C \wedge \neg B \wedge \neg Q$ proves C is inadequate to discharge the post-condition. For an adequate invariant I, $I(b)$ should be *false*. We call a state that must not be satisfied by an adequate invariant, such as b, a *bad* state. Hence, given an incorrect candidate invariant and a decision procedure that can produce counterexamples, the decision procedure can produce either a good state, a pair, or a bad state as a counterexample to refute the candidate.

Problems other than invariant inference can also be reduced to finding some unknown predicates to satisfy some VCs [23]. Consider the following problem: prove that the loop `while` B `do` S `od` goes into non-termination if executed with input i. One can obtain such a proof by demonstrating a *recurrent set* [9, 29] I which makes the following VCs valid.

$$I(i); \quad \{I \wedge B\}S\{I\}; \quad I \Rightarrow B \tag{4}$$

Our inference algorithm consumes such VCs with some unknown predicates. We use the term *invariant* for any such unknown predicate that we want to infer. In the rest of this section, we focus on the case when we need to infer a single predicate. The development here can be easily generalized to the case where we need to infer multiple predicates.

Our search algorithm is based on Markov Chain Monte Carlo (MCMC) sampling. Specifically, we use the Metropolis Hastings algorithm, which we describe next.

2.1 Metropolis Hastings

We denote the verification conditions by V, the unknown invariant by I, a candidate invariant by C, the set of predicates that satisfy V by $\mathcal{I}$ (more than one predicate can satisfy V), and the set of all possible candidates C by $\mathcal{S}$.

We view inference as a cost minimization problem. For each predicate $P \in \mathcal{S}$ we assign a non-negative cost $c_V(P)$ where the subscript indicates that the cost depends on the VCs. Suppose the cost function is designed to obey $C \in \mathcal{I} \Leftrightarrow c_V(C) = 0$. Then by minimizing c_V we can find an invariant. In general, c_V is highly irregular and not amenable to exact optimization techniques. In this paper, we use a MCMC sampler to minimize c_V.

The basic idea of a Metropolis Hastings sampler is given in Figure 1. The algorithm maintains a current candidate C. It also has a set of *moves*. A move, $m : \mathcal{S} \mapsto \mathcal{S}$, *mutates* a candidate to a different candidate. The goal of the search

Search(J: Initial candidate)
Returns: A candidate C with $c_V(C) = 0$.

1: $C := J$
2: **while** $c_V(C) \neq 0$ **do**
3: $m := SampleMove(rand())$
4: $C' := m(C)$
5: $c_o := c_V(C)$, $c_n := c_V(C')$
6: **if** $c_n < c_o$ or $e^{-\gamma(c_n - c_0)} > \frac{rand()}{RANDMAX}$ **then**
7: $C := C'$
8: **end if**
9: **end while**
10: **return** C

Fig. 1: Metropolis Hastings for cost minimization.

is to sample candidates with low cost. By applying a randomly chosen move, the search transitions from a candidate C to a new candidate C'. If C' has lower cost than C we keep it and C' becomes the current candidate. If C' has higher cost than C, then with some probability we still keep C'. Otherwise, we undo this move and apply another randomly selected move to C. Using these random mutations, combined with the use of the cost function, the search moves towards low cost candidates. We continue proposing moves until the search *converges*: the cost reduces to zero.

The algorithm in Figure 1, when instantiated with a suitable proposal mechanism (*SampleMove*) and a cost function (c_V), can be used for a variety of optimization tasks. If the proposal mechanism is designed to be *symmetric* and *ergodic* then the algorithm in Figure 1 has interesting theoretical guarantees.

A proposal mechanism is *symmetric* if the probability of proposing a transition from C_1 to C_2 is equal to the probability of proposing a transition from C_2 to C_1. Note that the cost is not involved here: whether the proposal is accepted or rejected is a different matter. Symmetry just talks about the probability that a particular transition is proposed from the available transitions.

A proposal mechanism is *ergodic* if there is a non-zero probability of reaching every possible candidate C_2 starting from any arbitrary candidate C_1. That is, there is a sequence of moves, $m_1, m_2, \ldots, m_k$, such that the probability of sampling each m_i is non-zero and $C_2 = m_k(\ldots(m_1(C_1)\ldots)$. This property is desirable because it says that it is not impossible to reach $\mathcal{I}$ starting from a bad initial guess. If the proposal mechanism is symmetric and ergodic then the following theorem holds [3]:

Theorem 1 *In the limit, the algorithm in Figure 1 samples candidates in inverse proportion to their cost.*

Intuitively, this theorem says that the candidates with lower cost are sampled more frequently. A corollary of this theorem is that the search always converges. The proof of this theorem relies on the fact that the *search space* $\mathcal{S}$ should be finite dimensional. Note that MCMC sampling has been shown to be effective in practice for extremely large search spaces and, with good cost

functions, is empirically known to converge well before the limit is reached [3]. Hence, we design our search space of invariants to be a large but finite dimensional space that contains most useful invariants by using templates. For example, our search space of disjunctive numerical invariants restricts the boolean structure of the invariants to be a DNF formula with ten disjuncts where each disjunct is a conjunction of ten linear inequalities. This very large search space is more than sufficient to express all the invariants in our numerical benchmarks.

Theorem 1 has limitations. The guarantee is only asymptotic and convergence could require more than the remaining lifetime of the universe. However, if the cost function is arbitrary then it is unlikely that any better guarantee can be made. In practice, for a wide range of cost functions with domains ranging from protein alignment [44] to superoptimization [51], MCMC sampling has been demonstrated to converge in reasonable time. Different cost functions do result in different convergence rates. Empirically, cost functions that provide feedback to the search have been found to be useful [51]. If the search makes a move that takes it closer to the answer then it should be rewarded with a decrease in cost. Similarly, if the search transitions to something worse then the cost should increase. We next present our cost function.

2.2 Cost Function

Consider the VCs of Equations 1, 2, and 3. One natural choice for the cost function is

$$c_V(C) = 1 - \mathit{Validate}(V[C/I])$$

where $\mathit{Validate}(X)$ is 1 if predicate X is valid and 0 otherwise. We substitute the candidate C for the unknown predicate I in the VCs and if the VCs are valid then the cost is zero and otherwise the cost is one. This cost function has the advantage that a candidate with cost zero is an invariant. However, this cost function is a poor choice for two reasons:

1. Validation is slow. A decision procedure takes several milliseconds in the best case to discharge a query. For a random search to be effective we need to be able to explore a large number of proposals quickly.
2. This cost function does not give any incremental feedback. The cost of all incorrect candidates is one, although some candidates are clearly closer to the correct invariant than others.

Empirically, search based on this cost function times out on even the simplest of our benchmarks. Instead of using a decision procedure in the inner loop of the search, we use a set of concrete program states that allows us to quickly identify incorrect candidates. As we shall see, concrete states also give us a straightforward way to measure how close a candidate is to a true invariant.

Recall from the discussion of the VCs that there are three different kinds of interesting concrete states. Assume we have a set of good states G, a set of bad states B, and a set of pairs Z. The data elements encode constraints that

a true invariant must satisfy. A good candidate C should satisfy the following constraints:

1. It should separate all the good states from all the bad states: $\forall g \in G. \forall b \in B. \neg(C(g) \Leftrightarrow C(b))$.
2. It should contain all good states: $\forall g \in G. C(g)$.
3. It should exclude all bad states: $\forall b \in B. \neg C(b)$.
4. It should satisfy all pairs: $\forall (s,t) \in Z. C(s) \Rightarrow C(t)$.

For most classes of predicates it is easy to check whether a candidate satisfies these constraints for given sets G, B, and Z without using decision procedures. For every violated constraint, we assign a penalty cost. In general, we can assign different weights to different constraints, but for simplicity, we weigh them equally. The reader may notice that the first constraint is subsumed by constraints 2 and 3. However, we keep it as a separate constraint as it encodes the amount of data that justifies a candidate. If a move causes a candidate to satisfy a bad state (which it did not satisfy before) then intuitively the increase in cost should be higher if the initial candidate satisfied many good states than if it satisfied only one good state. The third constraint penalizes equally in both scenarios (the cost increases by 1) and in such situations the first constraint is useful. The result is a cost function that does not require decision procedure calls, is fast to evaluate, and can give incremental credit to the search: the candidates that violate more constraints are assigned a higher cost than those that violate only a few constraints.

$$\begin{aligned} c_V(C) = & \textstyle\sum_{g \in G} \sum_{b \in B} \big(\neg C(g) * \neg C(b) + C(g) * C(b)\big) \\ & + \textstyle\sum_{g \in G} \neg C(g) + \sum_{b \in B} C(b) \\ & + \textstyle\sum_{(s,t) \in Z} C(s) * \neg C(t) \end{aligned} \tag{5}$$

In evaluating this expression, we interpret *false* as zero and *true* as one. The first line encodes the first constraint, the second line encodes the second and the third constraints, and the third line encodes the fourth constraint.

This cost function has one serious limitation: Even if a candidate has zero cost, still the candidate might not be an invariant. Once a zero cost candidate C is found, we check whether C is an invariant using a decision procedure; note this decision procedure call is made only if C satisfies all the constraints and therefore has at least some chance of actually being an invariant. If C is not an invariant one of Equation 1, 2, or 3 will fail and if the decision procedure can produce counterexamples then the counterexample will also be one of three possible kinds. If the candidate violates Equation 1 then the counterexample is a good state and we add it to G. If the candidate violates Equation 2 then the counter example is a pair that we add to Z, and finally if the candidate violates Equation 3 then we get a bad state that we add to B. We then search again for a candidate with zero cost according to the updated data. Thus our inference procedure can be thought of as a counterexample guided inductive synthesis (CEGIS) procedure [57], in particular, as an ICE learner [22]. Note that a pair (s,t) can also contribute to G or B. If $s \in G$ then t can be added to G. Similarly, if $t \in B$ then s can be added to B. If a state is in both G and

B then we abort the search. Such a state is both a certificate of the invalidity of the VCs and of a bug in the original program.

Not all decision procedures can produce counterexamples; in fact, in many more expressive domains of interest (e.g., the theory of arrays) generating counterexamples is impossible in general. In such situations the data we need can also be obtained by running the program. Consider the program point η where the invariant is supposed to hold. Good states are generated by running the program with inputs that satisfy the pre-conditions and collecting the states that reach η. Next, we start the execution of the program from η with an arbitrary state σ; i.e., we start the execution of the program "in the middle". If an assertion violation happens during the execution then all the states reaching η, including σ, during this execution are bad states. Otherwise, including the case when the program does not terminate (the loop is halted after a user-specified number of iterations), the successive states reaching η can be added as pairs. Note that successive states reaching the loop head are always pairs and may also be pairs of good states, bad states, or even neither.

The cost function of Equation 5 easily generalizes to the case when we have multiple unknown predicates. Suppose there are n unknown predicates $I_1, I_2, \ldots, I_n$ in the VCs. We associate a set of good states G_i and bad states B_i with every predicate I_i. For pairs, we observe that VCs in our benchmarks have at most one unknown predicate symbol to the right of the implication and one unknown predicate symbol to the left (both occurring positively), implying that commonly n^2 sets of pairs suffices: a set of pairs $Z_{i,j}$ is associated with every pair of unknown predicates I_i and I_j. A candidate $C_1, \ldots, C_n$ satisfies the set of pairs $Z_{i,j}$ if $\forall (s,t) \in Z_{i,j}.C_i(s) \Rightarrow C_j(t)$. For the pair $(s,t) \in Z_{i,j}$, if $s \in G_i$ then we add t to G_j and if $t \in B_j$ then we add s to B_i. Each of G_i, B_i, and $Z_{i,j}$ induces constraints and a candidate is penalized by each constraint it fails to satisfy.

In subsequent sections we use the cost function in Equation 5 and the search algorithm in Figure 1, irrespective of the type of program (numeric, array, string, or list) under consideration. What differs is the instantiation of C2I with different decision procedures and search spaces of invariants. Since a proposal mechanism dictates how a search space is traversed, different search spaces require different proposal mechanisms. In general, when C2I is instantiated with a search space, the user must provide a proposal mechanism and a function *eval* that evaluates a predicate in the search space on a concrete state, returning *true* or *false*. The function *eval* is used to evaluate the cost function; for the search spaces discussed in this paper, the implementation of *eval* is straightforward and we omit it. We discuss the proposal mechanisms for each of the search spaces in some detail in the subsequent sections.

3 Numerical Invariants

We describe the proposal mechanism for inferring numerical invariants. Suppose $x_1, x_2, \ldots, x_n$ are the variables of the program, all of type $\mathbb{Z}$. A program

state σ is a valuation of these variables: $\sigma \in \mathbb{Z}^n$. For each unknown predicate of the given VCs, the search space $\mathcal{S}$ is formulas of the following form:

$$\bigvee_{i=1}^{\alpha} \bigwedge_{j=1}^{\beta} \left(\sum_{k=1}^{n} w_k^{(i,j)} x_k \;\leq\; t^{(i,j)} \right)$$

Hence, predicates in $\mathcal{S}$ are boolean combinations of linear inequalities. We refer to w's as *coefficients* and t's as *constants*. The possible values that w's can take are restricted to a finite bag of coefficients $W = \{w_1, w_2, \ldots, w_{|W|}\}$. In our evaluation, the set $W = \{-1, 0, 1\}$ suffices. If needed, heuristics described in [2] can be used to obtain W. The possible values of t's are valuations of expression trees with leaves from a finite bag of constants $F = \{f_1, f_2, \ldots, f_{|F|}\}$. Binary multiplication and addition constitute the internal nodes of the expression trees. In our evaluation, the bag F contain all of the statically occurring constants in the program and their negations. The expression trees are created by the GEN-E procedure (Figure 2). Possible expression trees include $f_1 \times f_2$, $(f_1 + f_2) + f_3$, etc.

For our experiments, for the benchmarks that require conjunctive invariants we set $\alpha = 1$ and $\beta = 10$ and for those that require disjunctive invariants we set $\alpha = \beta = 10$. This search space, $\mathcal{S}$, is sufficiently large to contain invariants for all of our benchmarks.

3.1 Proposal Mechanism

We use $y \sim Y$ to denote that y is selected uniformly at random from the set Y and $[a : b]$ to denote the set of integers in the range $\{a, a{+}1, \ldots, b{-}1, b\}$. Unless stated otherwise, all random choices are derived from uniform distributions. Before a move we make the following random selections: $i \sim [1 : \alpha]$, $j \sim [1 : \beta]$, and $k \sim [1 : n]$ .We have the following three moves, each of which is selected with probability $\frac{1}{3}$:

- Coefficient move: select $l \sim [1 : |W|]$ and update $w_k^{(i,j)}$ to W_l.
- Constant move: update $t^{(i,j)}$ to GEN-E(F) (Figure 2).
- Inequality move: With probability $1 - \rho$, apply constant move to $t^{(i,j)}$ and coefficient move to $w_h^{(i,j)}$ for all $h \in [1 : n]$. Otherwise (with probability ρ) remove the inequality by replacing it with *true*.

These moves are motivated by the fact that prior empirical studies of MCMC have found that a proposal mechanism that has a good mixture of moves that make minor and major changes to a candidate leads to good results [51]. The first two moves make small changes and the last move can change an inequality completely.

This proposal mechanism is symmetric and ergodic. Using inequality moves we can transform any predicate in $\mathcal{S}$ to any other predicate in $\mathcal{S}$. For proving symmetry observe that the moves are themselves symmetric: if a move mutates C_1 to C_2 with probability p then the same move also updates C_2 to C_1 with

GEN-E(F: An array of integers)
Returns: An integer.

1: $t := r(F)$; O:=$\{+, \times, \bot, \bot\}$;
2: $o := r(O)$;
3: **while** $o \neq \bot$ **do**
4: $\quad f := r(F)$; $t := o(t, f)$; $o := r(O)$;
5: **end while**
6: **return** t

Fig. 2: Generate a random expression tree using leaves in F and operators in O; $r(A)$ returns an element selected uniformly at random from the array A.

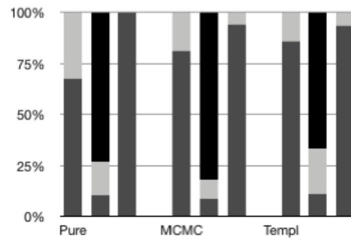

Fig. 3: Statistics for three different randomized searches applied to the `cgr2` benchmark.

probability p. It is easy to see that if all the moves are symmetric then the proposal mechanism is symmetric. Combining this proposal mechanism with the cost function in Equation 5 and the procedure in Figure 1 provides us a search procedure for numerical invariants. We call this procedure `MCMC` in the empirical evaluation of Section 3.3. Next, we describe two variations of this procedure.

In the first variation, we accept every move irrespective of the cost. The search terminates when a zero cost candidate is found. The resulting procedure is a pure random walk through the search space. The motivation for considering this variation is that it helps us evaluate the benefit of the cost function. We call this search strategy `Pure` in the evaluation in Section 3.3.

In the second variation, we further constrain the search space. The user provides templates to restrict the constituent inequalities of the candidate invariants. As an example, suppose we have a program with two variables x_1 and x_2 and the user restricts the invariants to boolean combinations of intervals. The inequalities must then be of the from $x_1 \leq d$, $x_1 \geq d$, $x_2 \leq d$, and $x_2 \geq d$. In general, the user can restrict the coefficients using other abstract domains like octagons [41] (bounds on sum or difference of at most two variables $\pm x_i \pm x_j \leq d$), octahedra [12] (bounds on all possible sums or differences of variables $\sum_i \pm x_i \leq d$), etc. In this variation, our moves need to ensure that the inequalities in the candidate invariants satisfy the templates. Hence, we need to modify the coefficient moves. The constant moves remain unchanged.

We replace the coefficient move with a template move: Select an inequality and replace all the coefficients with coefficients from a randomly chosen selection of coefficients permitted by the template. For example, for intervals, $x_1 \leq 2$ can be mutated to $-x_2 \leq 2$ by a template move. The inequality move applies a constant move and a template move. This variation is called `Templ` in the evaluation in Section 3.3.

Table 1: Inference of numerical invariants for proving safety properties.

Program	Z3-H	ICE	[54]	[30]	Pure	MCMC	Templ
cgr1 [27]	0.0	0.0	0.2	0.1	0.0 (10)	0.0 (10)	0.0 (10)
cgr2 [27]	0.0	7.3	?	?	0.4 (6)	0.4 (3)	0.5 (2)
fig1 [27]	0.0	0.1	?	?	0.1 (10)	0.3 (5)	0.3 (5)
w1 [27]	0.0	0.0	0.2	0.1	0.0 (10)	0.0 (10)	0.0 (10)
fig3 [24]	0.0	0.0	0.1	0.1	0.0 (10)	0.0 (10)	0.0 (10)
fig9 [24]	0.0	0.0	0.2	0.1	0.0 (10)	0.0 (10)	0.0 (10)
tcs [36]	TO	1.4	0.5	0.1	0.7 (2)	0.0 (7)	0.0 (10)
ex23 [35]	?	14.2	?	?	TO (0)	0.0 (10)	0.0 (10)
ex7 [35]	0.0	0.0	0.4	?	0.0 (10)	0.0 (10)	0.0 (10)
ex14 [35]	0.0	0.0	0.2	?	0.0 (10)	0.0 (10)	0.0 (10)
array [4]	0.0	0.3	0.2	?	1.0 (2)	0.6 (1)	0.9 (1)
fil1 [4]	0.0	0.0	0.4	0.1	0.0 (10)	0.0 (10)	0.0 (10)
ex11 [4]	0.0	0.6	0.2	0.1	0.0 (10)	0.0 (10)	0.0 (10)
trex1 [4]	0.0	0.0	0.4	0.1	0.0 (10)	0.0 (10)	0.0 (10)
monniaux	5.14	0.0	1.0	0.2	0.0 (10)	0.0 (10)	0.0 (10)
nested	0.0	?	1.0	0.0	0.1 (10)	0.2 (10)	0.0 (10)

3.2 Example

We give a simple example to illustrate the moves. Suppose we have two variables x_1 and x_2, $\alpha = \beta = 1$, the initial candidate is $C \equiv 0 * x_1 + 0 * x_2 \leq 0$, $W = \{0, 1\}$, and $F = \{0, 1\}$. Then a coefficient move leaves C unchanged with probability 0.5 and mutates it to $1 * x_1 + 0 * x_2 \leq 0$ or $0 * x_1 + 1 * x_2 \leq 0$ with probability 0.25 each. A constant move selects a new constant t (by calling Figure 2) to be one of $\{0, 1\}$ with probability 0.25 each, one of $\{0 + 1, 1 + 0, 0 + 0, 1 + 1, 1 \times 0, 0 \times 1, 0 \times 0, 1 \times 1\}$ with probability $\frac{1}{32}$ each, etc., and mutates C to $0 * x_1 + 0 * x_2 \leq t$ with the associated probability. An inequality move applies a constant move and a coefficient move to each coefficient. If we use intervals as templates then the possible values of the coefficients are $\{(0, 0), (0, 1), (0, -1), (1, 0), (-1, 0)\}$. Hence a template move leaves C unchanged with probability 0.2 and mutates it to $-x_1 \leq 0$, $x_1 \leq 0$, $-x_2 \leq 0$, or $x_2 \leq 0$ with probability 0.2 each. With templates, an inequality move applies a template move and a constant move.

3.3 Evaluation

We start with no data: $G = B = Z = \emptyset$. The initial candidate invariant J is the predicate in $\mathcal{S}$ that has all the coefficients and the constants set to zero: $\forall i, j, k.w_k^{(i,j)} = 0 \wedge t^{(i,j)} = 0$. The cost is evaluated using Equation 5 and when a candidate with cost zero is found then the decision procedure Z3 [42] is called. If Z3 proves that the candidate is indeed an invariant then we are done. Otherwise, Z3 provides a model. For better feedback, we ask Z3 for at most five distinct models. We extract counter-examples (good states, bad states, or pairs) from the models, they are incorporated in the data and

the search is restarted with J as the initial candidate. A *round* consists of one search-and-validate iteration: finding a predicate with zero cost and asking Z3 to prove/refute it.

Observe that since the search space is finite-dimensional, we can also use a decision procedure to search for a candidate invariant. This direction was pursued in recent work by Garg et al. [22]. Similar to us, they bound the search space to a finite set and iteratively invoke search and validate phases. Instead of using randomized search, they rely on Z3 to find an instantiation of the coefficients and the constants. Hence by comparing our approach against theirs, we can compare systematic search using a decision procedure with a randomized search.

The results of these comparisons are in Table 1. The first column is the name of the benchmark. For each benchmark, the problem is to find an invariant strong enough to discharge assertions in the program. All benchmarks except `monniaux` and `nested` are part of the benchmark suite described in [22]. The additional benchmarks are described below. Many of these benchmarks are flagship examples used by the respective papers to motivate a new technique for invariant inference. Five of these benchmarks require disjunctive invariants. The `Z3-H` column shows the time taken by Z3-Horn [32] in seconds. Z3-Horn is a decision procedure inside Z3 for solving VCs with unknown predicates. We observe that it is the fastest method for most of the programs. However, it crashes on `ex23` (? in the table), is slow for `monniaux`, and times out on `tcs` (TO in the table). The `ICE` column shows the search-and-validate approach of [22]. While slower, it is able to handle the benchmarks that trouble Z3-Horn. The fourth column evaluates a geometric machine learning algorithm [54] to search for candidate invariants and the next column is InvGen [30] a symbolic invariant inference engine that uses concrete data for constraint simplification. Columns `ICE`, [54], and [30] have been copied verbatim from [22] and the reader is referred to [22] for details.

The experiments in the last three columns (`Pure`, `MCMC`, and `Templ`) are performed on a 2.2 GHz Intel i7 with 4GB of memory. The experiments we compare to in Table 1 and in the rest of the paper were performed on a variety of machines. Our goal in reporting performance numbers is not to make precise comparisons, but only to show that c2i has competitive performance with other techniques. Indeed, we observe that the time measurements of the c2i searches in Table 1 are competitive with previous techniques. It is also worth emphasizing that because c2i is a stochastic technique, there is variation in the timing on repeated executions using the same input. Moreover, we have observed that some runs of MCMC never succeed and timeout eventually. Therefore, in practice, it is usual to run multiple stochastic searches in parallel that are stopped as soon as any of the searches succeeds [51]. The time measurements reported in this paper for randomized searches are the best of ten runs. The number of runs (out of ten) that succeed in under two seconds are shown alongside the time measurements in parentheses.

The `Pure` column shows the time taken by the pure random walk of Section 3.1. The time is the total time of all the rounds including the time for

both search and validation. The naive expectation is that the unguided search of `Pure` would fail for most of the benchmarks. However, the pure random walk is able to find the invariants for almost all the benchmarks including the ones on which the other tools fail. Moreover, the time required is comparable to the other tools. This observation suggests that the search space of numerical invariants is amenable to randomized search. Intuitively, there are many solutions in the search space and there are many possible sequences of transitions leading from one predicate to the other.

The `MCMC` column shows the effect of MCMC search that is guided by the cost function. The expectation is that it should perform much better than the pure random walk. We note that this expectation does not always hold (for `cgr2`, `fig1`, and `nested`). The reason is that MCMC search is slower. The pure random walk makes over a million proposals per second. On the other hand, MCMC search, with its overhead of computing the cost, is roughly an order of magnitude slower. Hence, even though MCMC search requires fewer proposals to converge for all the benchmarks in Table 1, it can take more time than the pure random walk. In Section 6.2, we show that `Pure` does not scale to more sophisticated search spaces and the cost function is essential. Also `Pure` times out on `ex23`: the reason is the absence of a cost function that guides the search towards the expression trees required for this benchmark.

The last column, `Templ`, shows the time when we manually provide templates to the search. The possible choices of templates are octagons and octahedra. Again, the expectation is that template based search should perform much better than `MCMC`. However, the templates adversely affect the desirable property of the MCMC proposal mechanism that there should be a good mixture of moves making small and large changes to the candidates.

Over all the benchmarks and all the different randomized searches fewer than 100 data points (good, bad, and pairs) and fewer than 30 rounds are sufficient to discover the invariant. The graph in Figure 3 shows some of the statistics for `cgr2`. We do not discuss the statistics for the other benchmarks as they are all quite similar and do not add additional insight.

The results in Figure 3 are divided into three groups. Each group corresponds to a different randomized inference engine. In each group the first bar represents the percentage of time spent in search (bottom) versus validation (top). All three approaches spend most of their time in search. The second bar shows the good states (bottom), bad states (middle), and pairs (top) as percentages of the number of data elements. The number of pairs is higher than the number of good or bad states. The third bar shows the number of proposals accepted (bottom) and rejected (top) as the percentage of total proposals. `Pure` accepts everything and the others reject only a small fraction of proposals.

The benchmark `monniaux`[2] illustrates a limitation of Z3-Horn.

```
for(i=0;i<1000;i++);assert(i<=10000);
```

[2] http://stackoverflow.com/questions/17547132/slow-invariant-inference-with-horn-clauses-in-z3

Table 2: Results on non-termination benchmarks.

Program	Z3-H	Pure	MCMC	Templ
`term1`	0.01	0.02	0.02	0.02
`term2`	TO	0.02	0.03	0.02
`term3`	TO	0.03	0.03	0.03
`term4`	0.01	0.03	0.05	0.02
`term5`	0.02	0.05	0.02	0.02
`term6`	TO	0.03	0.03	0.05

Intuitively, Z3-Horn is based on under-approximating strongest post-conditions and the large constant in the loop bound results in slow convergence. Empirically, for this example, the time taken by Z3-Horn appears to grow quadratically with the loop bound. On the other hand, the time taken by randomized search is independent of the size of the constants and is able to quickly find the invariant.

The benchmark `nested` has nested loops and cannot be handled by the implementation for simple loops described in [22]. Z3-Horn terminates on this example but instead of finding the simple invariants $i \geq 0$ and $0 \leq i < n \wedge 0 \leq j$, discovered by randomized search, it finds an existentially quantified invariant that cannot be consumed by most tools.

Since the randomized searches and Z3-Horn work with VCs, we can directly apply them to problems beyond invariant inference. Consider the problem addressed by the tool Looper [9]: Does a loop go into non-termination when executed with an input i? A certificate of non-termination is a *recurrent set* [29], a predicate that ensures the validity of the VCs in Equation 4. We consider the benchmarks for proving non-termination from TnT [29] and Looper in Table 2. Since these papers do not include performance results, we compare randomized search with Z3-Horn.

In Table 2, Z3-Horn is fast on half of the benchmarks and times out after thirty minutes on the other half. This observation suggests the sensitivity of symbolic inference engines to the search heuristics and the usefulness of Theorem 1. For half the benchmarks, the post-condition computation of Z3-Horn diverges. Randomized search, with no such systematic strategy and an asymptotic convergence guarantee, successfully handles all the benchmarks in less than a second.

4 Arrays

We consider the inference of universally quantified invariants over arrays. A program state for an array manipulating program contains the values of all the numerical variables and the arrays in scope. Given an invariant, existing decision procedures are robust enough to check that it indeed is an actual invariant. But in our experience, the decision procedures generally fail to find concrete counterexamples to refute incorrect candidates. This situation is a

real concern, because if our technique is to be generally applicable then it must deal with the possibility that the decision procedures might not always be able to produce counterexamples to drive the search. Fortunately, there is a solution to this problem. As outlined in Section 2.2, the good states, the bad states, and the pairs required for search can be obtained from program executions.

We use an approach similar to [54,21] to generate data. Let Σ_k denote all states in which all numerical variables are assigned values $\leq k$, all arrays have sizes $\leq k$, and all elements of these arrays are also $\leq k$. We generate all states in Σ_0, then Σ_1, and so on. To generate data, we run the loop with these states (see Section 2.2). To refute a candidate invariant, states from these runs are returned to the search. For our benchmarks, we did not need to enumerate beyond Σ_4 before an invariant was discovered. Better testing approaches are certainly possible [31].

We now define a search space of invariants to simulate the *fluid updates* abstraction for reasoning about arrays [17]. This abstraction is concerned with points-to relationships given by triples $(f[u], \phi, g[v])$, with the interpretation that ϕ is satisfied when $f[u]$ points to $g[v]$. In [17] both may relationships $(f[u] = g[v] \Rightarrow \phi)$ and must relationships $(\phi \Rightarrow f[u] = g[v])$ are used. The must relationships suffice for our benchmarks and we discuss only these here. If $x_1, \ldots, x_n$ are the numerical variables of the program and f and g are array variables, then we are interested in array invariants of the following form:

$$\forall u, v. T(x_1, x_2, \ldots, x_n, u, v) \Rightarrow f[u] = g[v] \tag{6}$$

The variables u and v are universally quantified variables and T is a numerical predicate in the quantified variables and the variables of the program. Using this template, we reduce the search for array invariants to numerical predicates $T(x_1, x_2, \ldots, x_n, u, v)$.

The search for T proceeds as described in Section 3. For all our benchmarks, the search space with $\alpha = 1$ and $\beta = 10$ suffices. The only significant difference between this search and the search in Section 3 is in the evaluation of the cost function. Since T has quantified variables, the evaluation of the cost function is more expensive: when evaluating whether a state satisfies a candidate, each quantified variable results in a loop. When applying moves, quantified and free variables are treated identically.

4.1 Evaluation

The principal difference between the evaluation here and in Section 3.3 is that there is no feedback between the search and the decision procedure. We manually wrote harnesses for generating data and then produced enough data that the search discovers a numerical predicate T that is an invariant of the array manipulating program. For all benchmarks, at most 150 data elements were sufficient to obtain an invariant. Just as in Section 3.3, we consider three variations of the search for T: `Pure` is a pure random walk, `MCMC` uses the cost

Table 3: Results on array manipulating programs

Program	[17]	Z3-H	ARMC	Dual	Pure	MCMC	Templ
init	0.01	0.06	0.15	0.72	0.06	0.02	0.01
init-nc	0.02	0.08	0.48	6.60	0.05	0.15	0.02
init-p	0.01	0.03	0.14	2.60	0.01	0.01	0.01
init-e	0.04	TO	TO	TO	TO	TO	TO
2darray	0.04	0.18	?	TO	0.02	0.41	0.02
copy	0.01	0.04	0.20	1.40	0.87	0.80	0.02
copy-p	0.01	0.04	0.21	1.80	0.09	0.13	0.01
copy-o	0.04	TO	?	4.50	TO	TO	0.50
reverse	0.03	0.12	2.28	8.50	TO	3.48	0.03
swap	0.12	0.41	3.0	40.60	TO	TO	0.21
d-swap	0.16	1.37	4.4	TO	TO	TO	0.51
strcpy	0.07	0.05	0.15	0.62	0.01	0.02	0.01
strlen	0.02	0.07	0.02	0.20	0.01	0.01	0.01
memcpy	0.04	0.20	16.30	0.20	0.02	0.03	0.01
find	0.02	0.01	0.08	0.38	2.23	0.30	0.02
find-n	0.02	0.01	0.08	0.39	0.07	0.95	0.01
append	0.02	0.04	1.76	1.50	TO	TO	0.12
merge	0.09	0.04	?	1.50	TO	TO	0.41
alloc-f	0.02	0.02	0.09	0.69	0.07	0.10	0.01
alloc-nf	0.03	0.03	0.13	0.42	0.49	0.14	0.07

function, and Templ restricts the inequalities in T to a user supplied abstract domain.

We evaluate these randomized search algorithms on the benchmarks of [17] in Table 3. The VCs for these benchmarks were obtained from the repository of the competition on software verification.[3] The first column is the name of the program. We have omitted benchmarks with bugs from the original benchmark set; these bugs are triggered during data generation. The second column shows the time taken by analyzing these benchmarks using the fluid updates abstraction [17]. Using a specialized abstract domain leads to a very efficient analysis, but the scope of the analysis is limited to array manipulating programs that have invariants given by Equation 6.

In [7], the authors use templates to reduce the task of inferring universally quantified invariants for array manipulating programs to numerical invariants and show results using three different back-ends: Z3-Horn [32], Armc [23], and Duality [40]. These are reproduced verbatim as columns Z3-H, ARMC, and Dual of Table 3. Details about these columns can be found in the original text [7]. Note that the benchmark init-e requires a divisibility constraint that none of these back-ends or our search algorithms currently support.

The next three columns describe our randomized searches. The time measurements are the total time to search (with sufficient data) and validate an invariant. We observe that Pure times out on several benchmarks, which suggests that these problems are harder than those for numerical invariants. Moreover, whenever the pure random search times out, both Armc and Duality

[3] https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/QALIA/

```
i := 0; x := "a";
while(non_det()){ i++; x := "(" + x + ")"; }
assert( x.length == 2*i+1 );
if(i>0) assert( x.contains( "(a)" ) );
```

Fig. 4: A string manipulating program.

take more than a second. Hence, there seems to be some correlation between which verification tasks are difficult for different techniques. Another factor that adversely affects the randomized searches is that, due to the quantified variables, the evaluation of the cost function is slower than the evaluation of the cost function for numerical candidates.

The surprising result is that `Pure` still terminates quickly on the majority of the benchmarks. The next column shows that `MCMC` also times out on several benchmarks (these are a subset of benchmarks on which `Pure` times out). For these benchmarks, as shown by the last column, just specifying the abstract domain in which the linear inequalities in the invariant belong suffices to make the search terminate in under a second. Moreover, `Templ` is faster than both ARMC and DUALITY on all the benchmarks of Table 3. These results suggest that randomized search is a suitable technique for inference of universally quantified invariants over arrays and can generate results competitive with state-of-the-art symbolic inference techniques.

5 Strings

Consider the string manipulating program of Figure 4 that computes the string $(^i\mathtt{a})^i$. To validate its assertions, the invariants must express facts about the contents of strings, integers, and lengths of strings; we are unaware of any previous inference technique that can infer such invariants. The string operations such as *length* (compute the length of a string), *indexof* (find the position of a string in another string), *substr* (extract a substring between given indices), etc., intermix integers and strings and pose a challenge for invariant inference. However, the decision procedure Z3-STR [60] can decide formulas over strings and integers. We use C2I to construct an invariant inference procedure from Z3-STR.

A program state contains the values of all the numerical and the string variables. The search space $\mathcal{S}$ consists of boolean combinations of predicates that belong to a given bag $\mathcal{P}$ of predicates: $\bigvee_{j=1}^{\alpha}\left(\bigwedge_{k=1}^{\beta} P_k^j\right)$ where $P_k^j \in \mathcal{P}$. The bag $\mathcal{P}$ is constructed using the constants and the predicates occurring in the program. We set $\alpha = 5, \beta = 10$, and for Figure 4, $\mathcal{P}$ has predicates $x.contains(y)$, $y_1 = y_2$, $w_1 i + w_2 x.length + w_3 \leq 0$ where $y \in \{x, \text{“}\mathtt{a}\text{”}, \text{“(”}, \text{“)”}, \text{“}(\mathtt{a})\text{”}\}$ and $w \in [-2 : 2]$. A move replaces a randomly selected P_k^j with a randomly selected predicate from $\mathcal{P}$. The current counterexample generation capabilities of Z3-STR are unreliable and we generate data using the process explained

Table 4: Results on string manipulating programs. The time taken (in seconds) by pure random search, by MCMC search, and by Z3-STR (for proving the correctness of the invariants) are shown.

	Figure 4	`replace`	`index`	`substring`
`Pure`	342.6	0.01	0.06	0.5
`MCMC`	0.8	0.02	0.06	0.05
Z3-STR	0.03	TO	114.6	0.01

in Section 4. (At most 25 data elements were sufficient to obtain an invariant.) For Figure 4, randomized search discovers the following invariant and discharges the assertions:

$$\big(x = \text{“}\mathtt{a}\text{”} \wedge i = 0\big) \vee \big(x.contains(\text{“}(\mathtt{a})\text{”}) \wedge x.length = 2i + 1\big)$$

Due to the absence of an existing benchmark suite for string-manipulating programs, our evaluation is limited to a few handwritten examples shown in Table 4. The program `replace` uses *replace* (replace the first occurrence of a string with another) in addition to the string operations present in Figure 4. This program checks that repeatedly replacing `"a"` by `"aa"` in a loop increases the length by the number of loop iterations. For this program Z3-STR times out in validating the candidate invariant. We confirmed that the candidate is an invariant manually. The program `index` uses *indexof* in addition to the string operations in `replace`. This program replaces all occurrences of one string by another and checks the relationship between the length of the output string and the number of iterations. Finally, `substring` uses *substr* and in this benchmark we prove that a loop which constructs an `http` request does not modify the domain name.

An alternative to C2I for proving these examples requires designing a new abstract interpretation [16,15], which entails designing an abstract domain that incorporates both strings and integers, an abstraction function, a widening operator, and abstract transfer functions that are precise enough to find disjunctive invariants like the one shown above. Such an alternative requires significantly greater effort than instantiating C2I. In our implementation, both the proposal mechanism and the *eval* function, required to instantiate C2I, are under 50 lines of C++ each.

6 Relations

In this section we define a proposal mechanism to find invariants over relations. We are given a program with variables $x_1, x_2, \ldots, x_n$ and some relations $R_1, R_2, \ldots, R_m$. A program state is an evaluation of these variables and these relations. For example, consider the program state $i = 1, j = 2, pts = \{(1,2),(2,1)\}, eq = \{(1,1),(2,2)\}$ where pts is the points-to relation and eq is the equality relation. In this state i and j point to two heap cells that form a circularly linked list. The invariants are composed of variables and such

relations. The search space consists of predicates F given by the following grammar:

$$\begin{array}{rcl} \textit{Predicate } F & ::= & \bigwedge_{i=1}^{\theta} F_i \\ \textit{Formula } F^i & ::= & \bigwedge_{j=1}^{\delta} G_j^i \\ \textit{Subformula } G^i & ::= & \forall u_1, u_2, \ldots, u_i.T \\ \textit{QF Predicate } T & ::= & \bigvee_{k=1}^{\alpha} \bigwedge_{l=1}^{\beta} L_l^k \\ \textit{Literal } L & ::= & A \mid \neg A \\ \textit{Atom } A & ::= & R(V_1, \ldots, V_a) \quad a = \textit{arity}(R) \\ \textit{Argument } V & ::= & x \mid u \mid \kappa \end{array} \tag{7}$$

A predicate in the search space is a conjunction of formulas F_i. The subscript of F_i denotes the number of quantified variables in its subformulas. A subformula is a quantified predicate with its quantifier free part T expressed in DNF. Each atomic proposition of this DNF formula is a relation whose arguments can be a variable of the program (x), a quantified variable (u), or some constant (κ) like *null*. We focus our attention on universally quantified predicates. Predicates with existential quantifiers and arbitrary alternations can also be incorporated easily in the search, but validating such candidates is much harder [59]. The variables *in scope* of a relation in a predicate are the program variables and the quantified variables in the associated subformula.

Next we define the moves of our proposal mechanism. We select a move uniformly at random from the list below and apply it to the current candidate C. As usual, we write "at random" to mean "uniformly at random".

1. Variable move: Select an atom of C at random. Next, select one of the arguments and replace it with an argument selected at random from the variables in scope and the constants.
2. Relation move: Select an atom of C at random and replace its relation with a relation selected at random from the set of relations of the same arity. The arguments are unaffected.
3. Atom move: Select an atom of C at random and replace its relation with a relation selected at random from all available relations. Perform variable moves to fill the arguments of the new relation.
4. Flip polarity: Negate a literal selected at random from the literals of C.
5. Literal move: Perform an atom move and flip polarity.

These moves are ergodic: using atom moves and flipping polarity it is possible to transform any candidate C_1 into any other candidate C_2. Moreover, these moves are symmetric and hence the proposal mechanism satisfies symmetry.

Next, we evaluate the MCMC algorithm in Figure 1 with this proposal mechanism and the cost function of Equation 5. We also evaluate a pure variation in which all moves are accepted. We do not evaluate a "template" variation as the relations can be seen as templates and it is unclear what additional template restrictions could be added.

6.1 Lists

We use the relational proposal mechanism to prove functional properties of linked list manipulating programs. The heap is composed of cells and each cell either contains *null* or the address of another cell. The reachability relation $n^*(i, j)$ holds if the cell pointed to by j can be reached from i using zero or more pointer dereferences. While writing post-conditions to express functional properties, it is useful to talk about the reachability relation that holds before the program begins execution. We denote this binary relation by $_n^*$. Using these relations, the predicates in our search space are universally quantified formulas over these reachability relations for linked list manipulating programs.

A recently published decision procedure is complete for such candidates via a reduction of such formulas to boolean satisfiability [33]. It takes a program annotated with invariants as input and checks the assertions. We use this decision procedure as our validator and randomized search to find invariants for some standard singly linked list manipulating programs. The evaluation of [33] shows that it can handle relations and hence can validate a variety of programs that have been hand-annotated with invariants. During our evaluation of various verification tasks, we observed that such decision procedures for advanced logics are not able to accept all formulas in their input language. Hence, sometimes we must perform some equality-preserving simplifications on the candidate invariants our search discovers. Currently we perform this step manually when necessary, but the simplifications could be automated.

6.2 Evaluation

For defining the search space using Equation 7 we set $\alpha = \beta = \delta = 5$ and $\theta = 2$, which is sufficient to express the invariants for all of our benchmarks. Our evaluation results on the benchmarks of [33] are in Table 5. The first column lists the programs, all of which perform basic manipulations of singly linked lists. The program `delete` removes a specific element of the list, `deleteall` deletes all the elements, `filter` deletes some specific elements if present in the list, `last` returns the last element of the list, and `reverse` is in-place reversal. The invariants for these programs are subtle and easy to get wrong.

Since these invariants are complex, pure random walk times out on all of these benchmarks. Hence, we show the results for only the MCMC search. Recall from Section 2.2 that it is easy to obtain good states: just run the program and collect the reachable states. However, it is more difficult to obtain bad states. We run our benchmarks on lists of length up to five to generate an initial set of good states, the size of which is shown in the column `G`. Starting from a non-empty set of good states results in faster convergence than starting from an empty set. Next, we start our search with zero bad states and zero pairs and generate candidate invariants. If the candidate is not an invariant we get a counterexample, which is added to the data (see discussion in Section 6.1).

Table 5: Results for list manipulating programs.

Program	#G	#R	Search	Valid	Prop	Accep	[34]
delete	50	2	0.20	0.04	4437	3949	9.32
delete-all	20	7	1.03	0.13	8482	7225	37.35
filter	50	26	10.41	0.11	160489	126389	55.53
last	50	3	0.90	0.04	98064	87446	7.49
reverse	20	54	55.11	0.08	582665	484208	146.42

The number of rounds for the search to converge to an invariant is shown in the column `R`. The next four columns show the statistics of the last (and also the most expensive) round, the one that produces an invariant. The column `Search` shows the time taken by the search to infer an invariant. The column `Valid` shows the time taken by the validator to validate the invariant discovered by the search. The next two columns show the number of proposals made (`Prop`) and the number of proposals accepted (`Accep`) by the search. Observe that the search converges in less than a million proposals for all the benchmarks.

On comparing these results with those for array invariants, we note that the time taken by the search is higher. However, with arrays we were able to execute many more proposals per second. The maximum number of proposals for the results in Table 3 are about seven hundred thousand (`MCMC` for `reverse`) which is more than the number of proposals for any benchmarks in Table 5. Note that shape analyses like TVLA [49] can also handle the benchmarks in Table 5 within seconds. The last column of Table 5 is a recent invariant inference engine for lists that also uses the decision procedure of [33]. However, due to the intermediate manual steps in our evaluation, we cannot perform a direct comparison.

On analyzing the invariants discovered by the search, we observe that they are different from the invariants in the manually annotated benchmarks of [33]. Consider the benchmark `reverse`. The variable h is the head of the initial list, i is the head of the remaining list to be reversed, and j is the head of the reversed list. The pre-condition is that the heap contains only the linked list and nothing else. The program is as follows:

```
i = h; j = null; while (i != null) { k=*i; *i=j; j=i; i=k; }
```

For `reverse` the search discovers the following invariant:

$$\forall u(u \neq \mathit{null} \Rightarrow (n^*(i,u) \vee n^*(j,u)))$$
$$\forall u,v(n^*(u,v) \Rightarrow (\neg_n^*(i,u) \wedge _n^*(v,u) \vee _n^*(u,v) \wedge \neg_n^*(u,j)))$$

The decision procedure of [33] is able to validate this invariant and uses it to show that `reverse` correctly reverses a linked list. Note that this invariant is more succinct and difficult to comprehend than the invariant written by hand in [33]:

$$\forall u(u \neq \mathit{null} \Rightarrow (n^*(i,u) \Leftrightarrow \neg n^*(j,u)))$$
$$\forall u,v(n^*(i,u) \Rightarrow (n^*(u,v) \Leftrightarrow _n^*(u,v)))$$
$$\forall u,v(n^*(j,u) \Rightarrow (n^*(u,v) \Leftrightarrow _n^*(v,u)))$$

This easier to understand invariant says that every node is either in the partially reversed list or the to be reversed list. In the to be reversed list, the reachability relation is unchanged and in the partially reversed list the reachability relation is reverse of the initial.

Since the decision procedure of [33] is complete, it is able to consume and verify any unusual invariants that the search produces. Generally the decision procedures for more advanced data structures are not so robust that they can consume arbitrary candidates. Often, one needs to write the invariants with care and might even need to provide additional axioms or lemmas to verify more advanced data structure manipulating programs [47]. Once the state of the art of these decision procedures improve, we can apply randomized search for these programs too. We do not need a complete decision procedure as we can generate data by running the programs, just as we do for arrays and strings. But the decision procedure should be robust enough to handle arbitrary candidate invariants automatically.

7 Related Work

The goal of this paper is a framework to obtain inference engines from decision procedures. C2I is parametrized by the language of possible invariants. This characteristic is similar to TVLA [49], which is a parametric shape analysis. TVLA requires specialized heuristics (focus, coerce, etc.) to maintain precision. We do not require these heuristics and this generality aids us in obtaining inference procedures for verification tasks beyond shape analysis. C2I is a template-based analysis that does not use decision procedures to instantiate the templates and limits their use to checking an annotated program. We do not rely on decision procedures to compute a predicate cover [28], or for fixpoint iterations [20,58], or on Farkas' lemma [27,30,14,6]. Hence, C2I is applicable to various decision procedures, including the incomplete procedures (Section 4 and Section 5).

The literature on invariant inference is huge. Most techniques for invariant inference are symbolic analyses that trade generality for effective techniques in specific domains [38,30,18,10,5,34]. We are not aware of any symbolic inference technique that has been successfully demonstrated to infer invariants for the various types of programs that we consider (numeric, array, string, and list). We have shown that invariant search using concrete data and general search procedures such as Metropolis Hastings has the potential to be a general solution to obtain inference procedures from checking procedures. We discuss the related techniques that learn invariants from concrete data and compare them with randomized search.

Concrete states can help maintain precision by aiding the computation of the best abstract transformers [48]. However, this approach suffers from irrecoverable imprecision as it relies on widening heuristics for termination. In contrast, randomized search can always recover from excessive under or over approximations, but only provides an asymptotic guarantee of termination if

an invariant exists. For numerical programs, machine learning techniques have been used to obtain invariants from concrete data. Daikon [19] uses conjunctive learning, [53,45] use equation solving, and [55] uses SVMs: these fail to infer disjunctive invariants over inequalities. The underlying machine learning algorithm of [54] uses geometry and hence is applicable to numerical predicates only. The inference algorithm for numerical invariants described in [22] uses decision procedures to search for candidate invariants. It is not clear whether decision procedures can effectively search for good candidates that satisfy data over quantified domains. There are several automata-based approaches for learning invariants [21,22,13]. It is unclear whether automata can express numerical invariants. Domain-specific search procedures like these seem unsuitable for a general framework for obtaining inference procedures from checking procedures.

Algorithmic learning [39,37] based approaches also iteratively invoke search and validate phases. They use a CDNF learning algorithm that requires membership queries (is a conjunction of atomic predicates contained in the invariant) and equivalence queries (is the candidate an invariant). Since the invariant is unknown, the membership queries are resolved heuristically. In contrast, C2I does not require membership queries. Other techniques that use concrete data to guide verification include [24,2,26,43].

We are unaware of the any previous work that uses Metropolis Hastings for invariant inference. In a related work, [25] uses Gibbs sampling for inference of numerical invariants. However, the inference works directly on the program (it computes pre-conditions and post-conditions) as opposed to concrete data. Handling programs with pointers and arrays is left as an open problem by [25].

We use efficiency to guide the choice of parameters for randomized search. E.g., in our evaluations, we set γ in Figure 1 to $log_e\ 2$. Systematic approaches described in [56] can also be used for setting such parameters.

MCMC based search has been found to be useful in superoptimization [51], where the goal is to infer an assembly program that satisfies some concrete input/output examples. MCMC search has also been used in program testing to achieve better coverage [50]. A randomized scheduler is used to find concurrency bugs in [8].

8 Conclusion

We have demonstrated a general procedure for generating an inference procedure from a checking procedure and applied it to a variety of programs. The inference procedure uses randomized search for generating candidate invariants that are proven or refuted by the checker. Furthermore, we have presented a cost function on concrete states that is useful in both guiding the search (especially for domains with more complex invariants, such as list-manipulating programs) and for performance. Finally, C2I produces results competitive with state of the art tools that are specialized for specific domains.

References

1. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: FM-CAD (2013)
2. Amato, G., Parton, M., Scozzari, F.: Discovering invariants via simple component analysis. J. Symb. Comput. 47(12) (2012)
3. Andrieu, C., de Freitas, N., Doucet, A., Jordan, M.I.: An Introduction to MCMC for Machine Learning. Machine Learning 50(1) (2003)
4. Beyer, D.: Competition on Software Verification (SV-COMP) benchmarks. https://svn.sosy-lab.org/software/sv-benchmarks/tags/svcomp13/loops/
5. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. STTT 9(5-6) (2007)
6. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: VMCAI (2007)
7. Bjørner, N., McMillan, K.L., Rybalchenko, A.: On solving universally quantified horn clauses. In: SAS (2013)
8. Burckhardt, S., Kothari, P., Musuvathi, M., Nagarakatte, S.: A randomized scheduler with probabilistic guarantees of finding bugs. In: ASPLOS (2010)
9. Burnim, J., Jalbert, N., Stergiou, C., Sen, K.: Looper: Lightweight detection of infinite loops at runtime. In: ASE (2009)
10. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL (2009)
11. Chib, S., Greenberg, E.: Understanding the Metropolis-Hastings Algorithm. The American Statistician 49(4) (1995)
12. Clarisó, R., Cortadella, J.: The octahedron abstract domain. In: SAS (2004)
13. Cobleigh, J.M., Giannakopoulou, D., Pasareanu, C.S.: Learning assumptions for compositional verification. In: TACAS (2003)
14. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: CAV (2003)
15. Costantini, G., Ferrara, P., Cortesi, A.: Static analysis of string values. In: ICFEM (2011)
16. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
17. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: Beyond strong vs. weak updates. In: ESOP (2010)
18. Dillig, I., Dillig, T., Li, B., McMillan, K.L.: Inductive invariant generation via abductive inference. In: OOPSLA (2013)
19. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. 69(1–3) (2007)
20. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: FME (2001)
21. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Learning universally quantified invariants of linear data structures. In: CAV (2013)
22. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A Robust Learning Framework for Synthesizing Invariants. In: CAV (2014)
23. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI (2012)
24. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: a new algorithm for property checking. In: FSE (2006)
25. Gulwani, S., Jojic, N.: Program verification as probabilistic inference. In: POPL (2007)
26. Gulwani, S., Necula, G.C.: Discovering affine equalities using random interpretation. In: POPL (2003)
27. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI (2008)
28. Gulwani, S., Srivastava, S., Venkatesan, R.: Constraint-based invariant inference over predicate abstraction. In: VMCAI (2009)

29. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In: POPL (2008)
30. Gupta, A., Majumdar, R., Rybalchenko, A.: From tests to proofs. In: TACAS (2009)
31. Harder, M., Mellen, J., Ernst, M.D.: Improving test suites via operational abstraction. In: ICSE (2003)
32. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: SAT (2012)
33. Itzhaky, S., Banerjee, A., Immerman, N., Nanevski, A., Sagiv, M.: Effectively-propositional reasoning about reachability in linked data structures. In: CAV (2013)
34. Itzhaky, S., Bjørner, N., Reps, T.W., Sagiv, M., Thakur, A.V.: Property-directed shape analysis. In: CAV (2014)
35. Ivancic, F., Sankaranarayanan, S.: NECLA Static Analysis Benchmarks http://www.nec-labs.com/research/system/systems_SAV-website/small_static_bench-v1.1.tar.gz
36. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: TACAS (2006)
37. Jung, Y., Kong, S., Wang, B.Y., Yi, K.: Deriving invariants by algorithmic learning, decision procedures, and predicate abstraction. In: VMCAI (2010)
38. Kannan, Y., Sen, K.: Universal symbolic execution and its application to likely data structure invariant generation. In: ISSTA (2008)
39. Kong, S., Jung, Y., David, C., Wang, B.Y., Yi, K.: Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In: APLAS (2010)
40. McMillan, K., Rybalchenko, A.: Combinatorial approach to some sparse-matrix problems. Tech. rep., Microsoft Research (2013)
41. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1) (2006)
42. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS (2008)
43. Naik, M., Yang, H., Castelnuovo, G., Sagiv, M.: Abstractions from tests. In: POPL (2012)
44. Neuwald, A.F., Liu, J.S., Lipman, D.J., Lawrence, C.E.: Extracting protein alignment models from the sequence database. Nucleic Acids Research 25 (1997)
45. Nguyen, T., Kapur, D., Weimer, W., Forrest, S.: Using dynamic analysis to discover polynomial and array invariants. In: ICSE (2012)
46. Nori, A.V., Sharma, R.: Termination proofs from tests. In: ESEC/SIGSOFT FSE (2013)
47. Qiu, X., Garg, P., Stefanescu, A., Madhusudan, P.: Natural proofs for structure, data, and separation. In: PLDI (2013)
48. Reps, T.W., Sagiv, S., Yorsh, G.: Symbolic implementation of the best transformer. In: VMCAI (2004)
49. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3) (2002)
50. Sankaranarayanan, S., Chang, R.M., Jiang, G., Ivancic, F.: State space exploration using feedback constraint generation and monte-carlo sampling. In: ESEC/SIGSOFT FSE (2007)
51. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. In: ASPLOS (2013)
52. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. In: CAV (2014)
53. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: ESOP (2013)
54. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Nori, A.V.: Program verification as learning geometric concepts. In: SAS (2013)
55. Sharma, R., Nori, A., Aiken, A.: Interpolants as classifiers. In: CAV (2012)
56. Sharma, R., Nori, A.V., Aiken, A.: Bias-variance tradeoffs in program analysis. In: POPL (2014)
57. Solar-Lezama, A.: The sketching approach to program synthesis. In: APLAS (2009)
58. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: PLDI (2009)
59. Srivastava, S., Gulwani, S., Foster, J.S.: VS3: SMT solvers for program verification. In: CAV (2009)
60. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: a Z3-based string solver for web application analysis. In: ESEC/SIGSOFT FSE (2013)