From Invariant Checking to Invariant Inference Using Randomized Search

Rahul Sharma and Alex Aiken

Stanford University, USA
{sharmar, aiken}@cs.stanford.edu

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.

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 [28, 16]. 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 [34, 36, 45, 46, 42, 20, 19]. 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 [44, 2] 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* [27], 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.
- Even though we expect the general inference engines based on randomized search to be significantly inferior in performance to the domain-specific invariant inference approaches, our experiments show that randomized search has competitive performance with the more specialized techniques.
- Randomized search is effective only when done efficiently. We describe optimizations that allow us to obtain practical randomized search algorithms for invariant inference.

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:

$$P \Rightarrow I; \quad \{I \wedge B\}S\{I\}; \quad I \wedge \neg B \Rightarrow Q \tag{1}$$

In this paper, we explore finding such an invariant I by randomized search. Given a candidate invariant, a decision procedure checks the conditions of Eqn. 1. 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.

Let C be a candidate invariant. The first condition 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 the second condition of Eqn. 1. 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 the third condition. 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 [21]. Consider the following problem: prove that the loop `while` B `do` S `od` fails to terminate if executed with input i. One can obtain such a proof by demonstrating a *recurrent set* [9, 27] I which makes the following VCs valid.

$$I(i); \quad \{I \wedge B\}S\{I\}; \quad I \Rightarrow B \tag{2}$$

Our inference algorithm consumes 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 generalizes easily to inferring multiple predicates.

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 candidate invariants 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.

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.

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 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 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 [4]:

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 [4]. 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 [40] to superoptimization [44], MCMC sampling has been demonstrated to converge in reasonable time. Empirically, cost functions that provide feedback to the search have been found to be useful [44]. 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 Eqn. 1. 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 Eqn. 1 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 is 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 weight 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.

$$\begin{aligned} c_V(C) = &\textstyle\sum_{g\in G}\sum_{b\in B}\big(\neg C(g) * \neg C(b) + C(g) * C(b)\big) + \\ &\textstyle\sum_{g\in G}\neg C(g) + \sum_{b\in B} C(b) + \sum_{(s,t)\in Z} C(s) * \neg C(t) \end{aligned} \tag{3}$$

In evaluating this expression, we interpret *false* as zero and *true* as one.

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 the three parts of Eqn. 1 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 the first condition of Eqn. 1 then the counterexample is a good state and we add it to G. If the candidate violates the second condition then the counter example is a pair that we add to

Z, and finally if the candidate violates the third condition 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 [49], in particular, as an ICE learner [20]. 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 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 Eqn. 3 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 Eqn. 3 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:

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

Hence, predicates in $\mathcal{S}$ are boolean combinations of linear inequalities. We refer to w's as *coefficients* and d's as *constants*. The possible values that w's and d's can take are restricted to a finite bag of coefficients $W = \{w_1, w_2, \ldots, w_{|W|}\}$ and a finite bag of constants $D = \{d_1, d_2, \ldots, d_{|D|}\}$ respectively. These bags contain all of the statically occurring constants in the program as well as their sums and differences, which has sufficed in our experience. If needed, heuristics to mine relevant constants from concrete states, as described in [46], can be used.

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: select $m \sim [1 : |D|]$ and update $d^{(i,j)}$ to D_m.
- Inequality move: With probability $1 - \rho$, apply constant move to $d^{(i,j)}$ 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 bias towards simple solutions and a good mixture of moves that make minor and major changes to a candidate leads to good results [44]. This proposal mechanism is symmetric and ergodic. Combining this proposal mechanism with the cost function in Eqn. 3 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. The user can also restrict the constituent inequalities of the candidate invariants to a given abstract domain. 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	[46]	[28]	MCMC	Templ
cgr1 [25]	0.0	0.0	0.2	0.1	0.0	0.0
cgr2 [25]	0.0	7.3	?	?	1.5	1.2
fig1 [25]	0.0	0.1	?	?	0.9	1.4
w1 [25]	0.0	0.0	0.2	0.1	0.0	0.0
fig3 [22]	0.0	0.0	0.1	0.1	0.0	0.0
fig9 [22]	0.0	0.0	0.2	0.1	0.0	0.0
tacas [33]	TO	1.4	0.5	0.1	0.5	0.0
ex23 [32]	?	14.2	?	?	0.1	0.1

Program	Z3-H	ICE	[46]	[28]	MCMC	Templ
ex7 [32]	0.0	0.0	0.4	?	0.0	0.0
ex14 [32]	0.0	0.0	0.2	?	0.0	0.0
array [5]	0.0	0.3	0.2	?	0.2	0.3
fil1 [5]	0.0	0.0	0.4	0.1	0.0	0.0
ex11 [5]	0.0	0.6	0.2	0.1	0.0	0.0
trex01 [5]	0.0	0.0	0.4	0.1	0.0	0.0
monniaux	5.14	0.0	1.0	0.2	0.0	0.0
nested	0.0	?	1.0	0.0	0.3	2.1

3.2 Example

We now 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 $D = \{0, 1\}$. 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 leaves C unchanged with probability 0.5 and mutates it to $0 * x_1 + 0 * x_2 \leq 1$ with probability 0.5. A predicate move (for $\rho = 0$) leaves C unchanged with probability 0.125 and mutates it to $x_1 \leq 0$, $x_2 \leq 0$, $0 \leq 1$, $x_1 \leq 1$, $x_2 \leq 1$, $x_1 + x_2 \leq 0$, or $x_1 + x_2 \leq 1$ with probability 0.125 each.

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: $\forall i, j, k. w_k^{(i,j)} = 0 \wedge d^{(i,j)} = 0$. The cost is evaluated using Eqn. 3 and when a candidate with cost zero is found then the decision procedure Z3 [38] is called. If Z3 proves that the candidate is indeed an invariant then we are done. Otherwise, Z3 provides a counterexample that is 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.

For each benchmark in Table 1, the problem is to find an invariant strong enough to discharge assertions in the program. The Z3-H column shows the time taken by Z3-Horn [30]. Z3-Horn is a decision procedure inside Z3 for solving VCs with unknown predicates. ICE shows the search-and-validate approach of [20]. The next column evaluates a geometric machine learning algorithm [46] to search for candidate invariants and the next column is InvGen [28] a symbolic invariant inference engine that uses concrete data for constraint simplification. Columns ICE, [46], and [28] have been copied verbatim from [20] and the reader is referred to [20] for details. The MCMC column shows for MCMC search the total time of all the rounds including the time for both search and validation.The

Table 2. Results on non-termination benchmarks.

Program	Z3-H	MCMC	Templ
term1	0.01	0.02	0.01
term2	TO	0.04	0.05
term3	TO	0.04	0.06
term4	0.01	0.04	0.06
term5	0.01	0.01	0.02
term6	TO	0.12	0.07

`Templ` column shows the time when we manually provide abstract domains (octagons/octahedra) to the search. All of our experiments were 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.

We consider the benchmarks for proving non-termination from TNT [27] 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. Randomized search, with 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 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. As outlined in Section 2.2, the good states, the bad states, and the pairs required for search can also be obtained from program executions.

We use an approach similar to [46, 19] 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

Table 3. Results on array manipulating programs

Program	[15]	Z3-H	ARMC	Dual	MCMC	Templ
init	0.01	0.06	0.15	0.72	0.02	0.01
init-nc	0.02	0.08	0.48	6.60	0.15	0.02
init-p	0.01	0.03	0.14	2.60	0.01	0.01
init-e	0.04	TO	TO	TO	TO	TO
2darray	0.04	0.18	?	TO	0.41	0.02
copy	0.01	0.04	0.20	1.40	0.80	0.02
copy-p	0.01	0.04	0.21	1.80	0.13	0.01
copy-o	0.04	TO	?	4.50	TO	0.50
reverse	0.03	0.12	2.28	8.50	3.48	0.03
swap	0.12	0.41	3.0	40.60	TO	0.21

Program	[15]	Z3-H	ARMC	Dual	MCMC	Templ
d-swap	0.16	1.37	4.4	TO	TO	0.51
strcpy	0.07	0.05	0.15	0.62	0.02	0.01
strlen	0.02	0.07	0.02	0.20	0.01	0.01
memcpy	0.04	0.20	16.30	0.20	0.03	0.01
find	0.02	0.01	0.08	0.38	0.30	0.02
find-n	0.02	0.01	0.08	0.39	0.95	0.01
append	0.02	0.04	1.76	1.50	TO	0.12
merge	0.09	0.04	?	1.50	TO	0.41
alloc-f	0.02	0.02	0.09	0.69	0.10	0.01
alloc-nf	0.03	0.03	0.13	0.42	0.14	0.07

(at most 150 states) before an invariant was discovered. Note that [46, 19] test only on reachable states. We additionally test on unreachable states to obtain bad states and pairs. Better testing approaches are certainly possible [29].

We now define a search space of invariants to simulate the fluid updates abstraction for reasoning about arrays [15]. 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:

$$\forall u, v. T(x_1, x_2, \ldots, x_n, u, v) \Rightarrow f[u] = g[v] \tag{4}$$

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.

4.1 Evaluation

We evaluate the randomized search algorithms on the benchmarks of [15] in Table 3. The VCs for these benchmarks were obtained from the repository of the competition on software verification.[1] 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 to analyze these benchmarks using the fluid updates abstraction in [15]. 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 Eqn. 4.

In [8], 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 [30], Armc [21], and Duality [37]. 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 [8].

[1] 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. 2. A string manipulating program

Note that the benchmark `init-e` requires a divisibility constraint that none of these back-ends or our search algorithms currently support.

Columns `MCMC` and `Templ` describe our randomized searches: the total time to search (with sufficient data) and validate an invariant. Again the results are competitive with previous domain-specific approaches. Also, a comparison of `MCMC` and `Templ` shows that convergence depends crucially on the proposal mechanism.

5 Strings

Consider the string manipulating program in Figure 2. 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 [51] 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 2, $\mathcal{P}$ has predicates $x.contains(y)$, $y_1 = y_2$, $w_1 i + w_2 x.length + w_3 \leq 0$ where $y \in \{x, \text{“a”}, \text{“(”}, \text{“)”}, \text{“(a)”}\}$ 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 in Section 4. (At most 25 data elements are sufficient to obtain an invariant.) For the program in Figure 2, randomized search discovers the following invariant:

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

We consider some additional examples in Table 4 and the name indicates the string operations they use. Due to the absence of an existing benchmark suite for string-manipulating programs, our evaluation is limited to a few handwritten examples.

One alternative to C2I for proving these examples involves designing a new abstract interpretation [14, 13], which requires 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. The search space consists of predicates F given by the following grammar:

$$\begin{aligned}
\textit{Predicate } F &::= \textstyle\bigwedge_{i=1}^{\theta} F^i \\
\textit{Formula } F^i &::= \textstyle\bigwedge_{j=1}^{\delta} G^i_j \\
\textit{Subformula } G^i &::= \forall u_1, u_2, \ldots, u_i.T \\
\textit{QF Predicate } T &::= \textstyle\bigvee_{k=1}^{\alpha} \bigwedge_{l=1}^{\beta} L^k_l \\
\textit{Literal } L &::= A \mid \neg A \\
\textit{Atom } A &::= R(V_1, \ldots, V_a) \quad a = \textit{arity}(R) \\
\textit{Argument } V &::= x \mid u \mid \kappa
\end{aligned} \tag{5}$$

A predicate in the search space is a conjunction of formulas. The superscript of F^i denotes the number of quantified variables in its subformulas. A subformula G^i 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*. 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 symmetric and ergodic. Next, we evaluate the MCMC algorithm in Figure 1 with this proposal mechanism and the cost function of Eqn. 3.

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

	Figure 2	`replace`	`index`	`substring`
`Search`	0.8	0.02	0.06	0.05
Z3-STR	0.03	TO	114.6	0.01

Table 5. Results for list manipulating programs.

`Program`	`#G`	`#R`	`Search`	`Valid`
`delete`	50	2	0.20	0.04
`delete-all`	20	7	1.03	0.13
`find`	50	9	0.42	0.04
`filter`	50	26	10.41	0.11
`last`	50	3	0.90	0.04
`reverse`	20	54	55.11	0.08

We instantiate the relational proposal mechanism with reachability relations: 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. A recently published decision procedure is complete for such candidates via a reduction of such formulas to boolean satisfiability [31]. We use this decision procedure as our validator and randomized search to find invariants for some standard singly linked list manipulating programs (described in [31]) in Table 5.

6.1 Evaluation

For defining the search space using Eqn. 5 we set $\alpha = \beta = \delta = 5$ and $\theta = 2$, which is sufficient to express the invariants for benchmarks in Table 5. 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. The number of rounds for the search to converge to an invariant is shown in the column `#R`. Later rounds take more time than the initial rounds. Columns `Search` and `Valid` describe the time to search (with sufficient data) and to validate an invariant respectively.

During our evaluation of various verification tasks, we observe that the 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.

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 [43]. 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 [26], or for fixpoint iterations [18, 50], or on Farkas' lemma [25, 28, 12, 7]. Hence, C2I is applicable to various decision procedures, including 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 [35, 28, 16, 10, 6, 1]. 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). Daikon [17] and Houdini [18] use conjunctive learning, [45, 41] use equation solving, and [47] uses SVMs: these fail to infer disjunctive invariants over inequalities. The underlying machine learning algorithm of [46] uses geometry and hence is applicable to numerical predicates only.

Algorithmic learning [36, 34] 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?", that are resolved heuristically. We do not require membership queries. Other techniques that use concrete data to guide verification include [22, 3, 24, 39].

We are unaware of the any previous work that uses Metropolis Hastings for invariant inference. In a related work, [23] uses Gibbs sampling for inference of numerical invariants. However, the inference does not use concrete states and the resulting cost function is expensive to evaluate. Handling programs with pointers and arrays is left as an open problem by [23].

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 [48] can also be used for setting such parameters.

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. While C2I is general and can handle many classes of useful invariants, its performance is still competitive with state of the art tools that are specialized for specific domains.

Acknowledgements We thank Eric Schkufza, Manolis Papadakis, and the anonymous reviewers for their comments. This work was supported by NSF 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.

References

1. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: SAFARI: SMT-based abstraction for arrays with interpolants. In: CAV (2012)
2. 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: FMCAD (2013)
3. Amato, G., Parton, M., Scozzari, F.: Discovering invariants via simple component analysis. J. Symb. Comput. 47(12) (2012)
4. Andrieu, C., de Freitas, N., Doucet, A., Jordan, M.I.: An Introduction to MCMC for Machine Learning. Machine Learning 50(1) (2003)
5. Beyer, D.: Competition on Software Verification (SV-COMP) benchmarks. https://svn. sosy-lab.org/software/sv-benchmarks/tags/svcomp13/loops/
6. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. STTT 9(5-6) (2007)
7. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: VMCAI (2007)
8. Bjørner, N., McMillan, K.L., Rybalchenko, A.: On solving universally quantified horn clauses. In: SAS (2013)
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. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: CAV (2003)
13. Costantini, G., Ferrara, P., Cortesi, A.: Static analysis of string values. In: ICFEM (2011)
14. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
15. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: Beyond strong vs. weak updates. In: ESOP (2010)
16. Dillig, I., Dillig, T., Li, B., McMillan, K.L.: Inductive invariant generation via abductive inference. In: OOPSLA (2013)
17. 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)
18. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: FME (2001)
19. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Learning universally quantified invariants of linear data structures. In: CAV (2013)
20. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A Robust Learning Framework for Synthesizing Invariants. In: CAV (2014)
21. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI (2012)
22. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: a new algorithm for property checking. In: FSE (2006)

23. Gulwani, S., Jojic, N.: Program verification as probabilistic inference. In: POPL (2007)
24. Gulwani, S., Necula, G.C.: Discovering affine equalities using random interpretation. In: POPL (2003)
25. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI (2008)
26. Gulwani, S., Srivastava, S., Venkatesan, R.: Constraint-based invariant inference over predicate abstraction. In: VMCAI (2009)
27. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In: POPL (2008)
28. Gupta, A., Majumdar, R., Rybalchenko, A.: From tests to proofs. In: TACAS (2009)
29. Harder, M., Mellen, J., Ernst, M.D.: Improving test suites via operational abstraction. In: ICSE (2003)
30. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: SAT (2012)
31. Itzhaky, S., Banerjee, A., Immerman, N., Nanevski, A., Sagiv, M.: Effectively-propositional reasoning about reachability in linked data structures. In: CAV (2013)
32. 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
33. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: TACAS (2006)
34. Jung, Y., Kong, S., Wang, B.Y., Yi, K.: Deriving invariants by algorithmic learning, decision procedures, and predicate abstraction. In: VMCAI (2010)
35. Kannan, Y., Sen, K.: Universal symbolic execution and its application to likely data structure invariant generation. In: ISSTA (2008)
36. 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)
37. McMillan, K., Rybalchenko, A.: Combinatorial approach to some sparse-matrix problems. Tech. rep., Microsoft Research (2013)
38. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS (2008)
39. Naik, M., Yang, H., Castelnuovo, G., Sagiv, M.: Abstractions from tests. In: POPL (2012)
40. 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)
41. Nguyen, T., Kapur, D., Weimer, W., Forrest, S.: Using dynamic analysis to discover polynomial and array invariants. In: ICSE (2012)
42. Nori, A.V., Sharma, R.: Termination proofs from tests. In: ESEC/SIGSOFT FSE (2013)
43. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3) (2002)
44. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. In: ASPLOS (2013)
45. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: ESOP (2013)
46. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Nori, A.V.: Program verification as learning geometric concepts. In: SAS (2013)
47. Sharma, R., Nori, A., Aiken, A.: Interpolants as classifiers. In: CAV (2012)

48. Sharma, R., Nori, A.V., Aiken, A.: Bias-variance tradeoffs in program analysis. In: POPL (2014)
49. Solar-Lezama, A.: The sketching approach to program synthesis. In: APLAS (2009)
50. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: PLDI (2009)
51. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: a Z3-based string solver for web application analysis. In: ESEC/SIGSOFT FSE (2013)