Quantifying the Exploration of the Korat Solver for Imperative Constraints

Alyas Almaawi, Hayes Converse, Milos Gligoric, Sasa Misailovic*, and Sarfraz Khurshid
University of Texas at Austin, *University of Illinois at Urbana-Champaign
{alyas.mohammed,hayesconverse,gligoric,khurshid}@utexas.edu,misailo@illinois.edu

ABSTRACT

Tools that explore very large state spaces to find bugs, e.g., when model checking, or to find solutions, e.g., when constraint solving, can take a considerable amount of time before the search terminates, and the user may not get useful feedback on the state of the search during that time. Our focus is a tool that solves imperative constraints to provide automated test input generation for systematic testing. Specifically, we introduce a technique to quantify the exploration of Korat, a well-known tool that explores the bounded space of all candidate inputs and enumerates desired inputs that satisfy given constraints. Our technique quantifies the size of the input space as it is explored by the Korat search, and provides the user exact information on the size of the remaining input space. In addition, it allows studying key characteristics of the search, such as the distribution of solutions as the search finds them. We implement the technique as a listener for the Korat search, and present initial experimental results using it.

1. INTRODUCTION

Constraint solvers, e.g., propositional satisfiability (SAT) [9] or satisfiability modulo theory (SMT) [8] solvers, form the backbone of many software analysis and synthesis techniques [5,6,11,15,19, 22,23]. This increasing use of solvers has in part been driven by improvements in the core solving methods and in part by technological advances in the computation machinery that is now readily available. Despite the advances, in many real-world application scenarios, the constraints that arise and must be solved can be problem instances that are particularly hard and take a long time to solve. In such scenarios, the user may have to painstakingly wait a long time before the solving results become available. For many solvers, there is very little, if any, feedback to the users, on the progress of the solver. As a result, it is typical for the users to timeout the search after some period based on their intuition or frustration. Unfortunately, when the search times out (or a user terminates the search), the users often do not have much information about what the search did, e.g., how much of the state space was explored and how much remained to be explored.

The focus of this paper is on *imperative constraints*, i.e., logical constraints that are written in an imperative language, such as Java. To illustrate, consider a Java predicate (method) that inspects its input and returns true if and only if the input is valid, i.e., satisfies the desired constraints. Such predicates are termed *repOk* methods [18].

This paper introduces a technique that precisely *quantifies the search performed* – in terms of the size of the input space explored – by a solver for imperative constraints. Our specific focus is the Korat solver for Java predicates, which is a well-known tool for automated test generation for bounded-exhaustive testing [5].

Korat implements a backtracking search that enumerates all inputs within a size bound, termed *finitization*, such that the given predicate returns true for each enumerated input; the set of all such inputs is the set of all solutions, also termed *valid* inputs, to the imperative constraint in the bounded input space. The finitization bounds the number of objects and values for the relevant types and defines domains of values for each object field.

The Korat search iteratively considers many candidate inputs. For each candidate, Korat runs *repOk* to check its validity. Korat uses feedback from *repOk* to create the next candidate. Specifically, Korat monitors the field accesses made by *repOk* and systematically backtracks only on the fields that are accessed by *repOk*, thereby creating a *dynamic* lexicographic ordering of the input space and pruning from search large parts of the input space. To optimize further, Korat restricts the search to only *non-isomorphic* candidates by adapting the least number heuristic [27] to dynamically break symmetries.

While this integration of field access monitoring, dynamic lexicographic ordering, and isomorphism breaking is the key to the effectiveness of the Korat search, it is also what makes computing the progress of the search challenging. In contrast, a search that followed a *static* lexicographic ordering is straightforward to characterize: each candidate's position in the order is known statically, and since no candidate that is lexicographically smaller can be explored later in the search, the space explored is exactly the position of the current candidate.

We introduce the first technique that – at any point in the Korat search precisely quantifies the size of the input space that has exhaustively been explored up to that point, and as a result, can report the size of the input space that has yet to be explored at any point during the search. Our key insight is two-fold: 1) each candidate that Korat checks (using *repOk*) is a representative of a class of candidates that is checked together using just one invocation of *repOk* and the cardinality of this class can be computed using the list of fields accessed by *repOk* and the finitization bounds; 2) when Korat checks candidate v and creates next candidate w, a space of candidates is pruned from the search, and the size of the space pruned can be computed using the two candidates v and w together with the list of field accesses.

We embody the technique as a part of the Korat implementation (via a novel listener) and experimentally validate it using a standard suite of subject constraints to demonstrate its efficacy. Moreover, as a potential application of our technique, we demonstrate how it allows studying the input space coverage of Korat as its search progresses and it finds valid inputs for a select subject.

This paper makes the following key contributions:

1) **Quantifying space covered during imperative constraint solving.** We introduce the first approach that computes the input space covered by an imperative constraint solver as it explores the space. 2) **Embodiment in Korat.** We embody our technique in Korat, which is a well-known solver for imperative constraints. 3) **Evaluation.** We experimentally validate our technique using a suite of standard subjects, and apply it to study the solution distribution for a select subject, and report our observations.

In future work, we plan to further evaluate our approach and its benefits in quantifying the search. We also plan to investigate applying it for new applications, e.g., to approximate the number of solutions, i.e., *model counts*, say based on using the results of a partial search, taking into consideration the space covered, and projecting the partial search results to the full input space.

2. ILLUSTRATIVE OVERVIEW

This section presents an illustrative example using binary trees to describe the key elements of the Korat search [5] and gives an overview of our technique for quantifying the input space covered.

Figure 1 declares a binary tree that has a *root* node and caches the number of nodes in the *size* field. Each node has a *left* and a *right* child. The predicate *repOk* describes the structural invariants that any valid binary tree must satisfy. The predicate traverses its input structure (*this*), checks that it has no cycles and that the value of size correctly states the number of nodes, and returns false if any check fails, and true if all checks pass. The *repOk* predicate is an example of an imperative constraint – a logical constraint that is written using imperative code. The method *finBinaryTree(int, int, int)* defines the space of all candidate inputs to consider for constraint solving, specifically each candidate can have at most *numNodes* nodes, and has *size* range between *minSize* and *maxSize* (inclusive). The method *finBinaryTree(int)* requires each candidate to have exactly *size* nodes.

Given the input constraint (*repOk*) and finitization (*finBinaryTree(int)*) with *size* set to 3, Korat uses the *repOk* to inspect 63 candidate structures (from a space of 16384 total structures) and enumerates 5 solutions – each a valid non-isomorphic binary tree with exactly 3 nodes (Figure 2).

The Korat search internally represents each candidate structure as a vector of integer elements; the length of the candidate vector is fixed and is defined by the bound on the input size. Each vector element represents a value of an object field; specifically the element is an index into an array of values that define all possible values for the corresponding field. The candidate vectors admit a *natural lexicographic order*: for vectors v and w, $v < w$ if $\exists i \cdot v[i] < w[i] \wedge \forall j < i \cdot v[j] = w[j]$ (intuitively, read the vector elements left to right, and order smaller values before higher values). The Korat search starts with the smallest candidate in the natural order. However, the search does not proceed in conformance with the natural order. Instead, the search considers the candidates in a *dynamic* lexicographic order, where the order is based on the fields accessed by *repOk* executions. For example, if *repOk* accesses the field at index 2 of the first candidate vector, that field is the first to be considered for lexicographic ordering. Moreover, if a field f is accessed with value x for candidate vector v and the next field accessed is g, g becomes the successor of f for lexicographic ordering. However, g may not remain the successor of f in the dynamic lexicographic ordering throughout the search, since for another candidate vector $w(\neq v)$ that for field f has a value $y(\neq x)$, the next field accessed may be $h(\neq g)$, in which case h becomes the successor of f in the lexicographic order from that point onward in the search.

```
class BinaryTree {
   static class Node { Node left, right; }

   Node root; int size;

   boolean repOk() {
      if (root == null) return size == 0;
      // checks that tree has no cycle
      Set visited = new HashSet();
      visited.add(root);
      LinkedList workList = new LinkedList();
      workList.add(root);
      while (!workList.isEmpty()) {
         Node current = (Node) workList.removeFirst();
         if (current.left != null) {
            if (!visited.add(current.left))
               return false;
            workList.add(current.left);
         }
         if (current.right != null) {
            if (!visited.add(current.right))
               return false;
            workList.add(current.right);
         }
      }
      // checks that size is consistent
      return (visited.size() == size); }

   static IFinitization finBinaryTree(int size) {
      return finBinaryTree(size, size, size); }

   static IFinitization finBinaryTree(int nodesNum,
         int minSize, int maxSize) {
      IFinitization f =
         FinitizationFactory.create(BinaryTree.class);
      IObjSet nodes =
         f.createObjSet(Node.class, nodesNum, true);
      f.set("root", nodes);
      f.set("size", f.createIntSet(minSize, maxSize));
      f.set("Node.left", nodes);
      f.set("Node.right", nodes);
      return f; }
}
```

Figure 1: Binary tree declaration, class invariant (*repOk*), and finitization (*finBinaryTree*) [1].

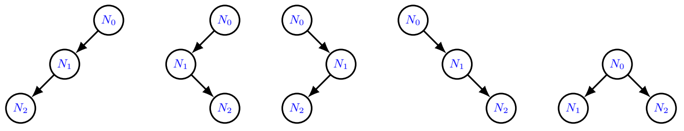

Figure 2: Five non-isomorphic binary trees with 3 nodes generated by Korat. N_0 is the root node.

For binary tree size 3, the candidate vector has 8 elements – one each for the *root* and *size* fields of the tree, and 2 each for the *left* and *right* fields of each of the 3 nodes (which we denote N_0, N_1, and N_2). Each element of the vector is an index into a domain of values for the corresponding field. For size 3, the *root* field of the tree, and the *left* and *right* fields of each of the 3 nodes take a value from the domain [*null*, N_0, N_1, N_2], and the field *size* takes a value from the domain [3] that has just 1 element.

The following list shows the sequence of candidates inspected by the Korat search starting at the first candidate (all 0's) until the first solution (identified by "***") is found:

```
/*  1*/ 0 0 0 0 0 0 0 0  :: 0 1
/*  2*/ 1 0 0 0 0 0 0 0  :: 0 2 3 1
/*  3*/ 1 0 0 1 0 0 0 0  :: 0 2 3
/*  4*/ 1 0 0 2 0 0 0 0  :: 0 2 3 4 5 1
/*  5*/ 1 0 0 2 0 1 0 0  :: 0 2 3 4 5
/*  6*/ 1 0 0 2 0 2 0 0  :: 0 2 3 4 5
/*  7*/ 1 0 0 2 0 3 0 0  :: 0 2 3 4 5 6 7 1 ***
```

Each row has an identifier (in block comments) and contains a candidate vector, the separator "`::`", and the list of fields accessed by *repOk* for that candidate (based on the first time *repOk* accesses the field). For example, for Candidate #1 (all 0s), *repOk* only accesses the *root* and *size* fields and returns *false* since *root* is *null* (first value in the domain [*null*, N_0, N_1, N_2]) but *size* is 3 (first value in the domain [3]). Korat backtracks on the last (most recent) field accessed. Since *size* has only 1 possible value, Korat increments the value for the `root` field to create the second candidate vector.

To illustrate coverage of input space, observe that when Korat inspects a candidate c, c serves as a representative for a *set* E of unique candidates for the Korat search such that *c.repOk()* == *e.repOk()* for all $e \in E$, and the field access list of each candidate in E is identical to that of c; we term the size of set E the *reach-space*. For example, for Candidate #1 Korat only reads the first 2 fields, so it represents all $4 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \cdot 4 = 4096$ candidates of the form "`0 0 * * * * * *`" (since each of the 6 fields that are not accessed is either a *left* or a *right* field and has 4 possible values). Thus, the space covered by Korat by just *inspecting* the first candidate is of size 4096 (i.e., *reach-space* with respect to the first candidate).

Korat may cover *additional* space in a single search step due to isomorphism breaking; we term the size of this additional space the *prune-space*. In particular, when Korat increments a field index, it may reset it to 0 (and backtrack) without trying all remaining values for that field when those values are equivalent [27]. To illustrate, consider the following sequence of two candidate trees that Korat inspects:

```
...
/* 25*/ 1 0 0 2 3 3 0 0  :: 0 2 3 4 5
/* 26*/ 1 0 1 0 0 0 0 0  :: 0 2
...
```

To create Candidate #26, Korat considers incrementing the value of the sixth field (*id*: 5) since that is the last field accessed but resets it since the largest index possible is already used. Next, Korat considers incrementing the value of the fifth field (*id*: 4) but also resets it. Next, Korat considers incrementing the value of the fourth field (*id*: 3). Even though this field can take the value 3 according to the finitization, Korat resets it because the fields accessed *before* it only use a maximum index of 1 into the corresponding domain, and it suffices to try just one of the remaining values since all unused values are interchangeable [27]. Thus, by generating Candidate #26 Korat covers another $1 \cdot 4 \cdot 4 \cdot 4 \cdot 4 = 256$ candidates of the form "`1 * 0 3 * * * *`" without inspecting even one of them.

Our technique for space coverage computation supports two execution modes: (1) to compute distribution for the number of solutions found by Korat with respect to the size of the input space it covers; (2) to periodically output to console the size of the input space covered up to the end of each period. For the first mode, the user specifies the number k of *bins* that partition the input space into k equal parts. The second mode integrates with Korat's built-in progress reporting facility, which allows the user to specify a number n such that after every n candidates, it outputs the solution count and number of candidates explored so far, and the last candidate explored. Our technique adds the information on the input space covered to Korat's progress report.

Figure 3 presents the distribution of the solutions that we obtain by plotting the command line outputs. For this benchmark, it shows that most of the solutions are concentrated between 20% and 50% of the explored states.

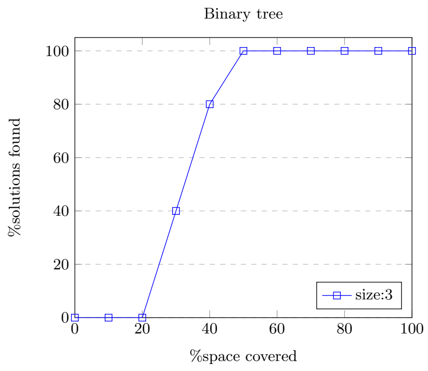

Figure 3: Korat solving of the binary tree constraints.

Algorithm 1 ReachSpace Algorithm

Inputs:
Candidate Vector CV : `Field` $\rightarrow$ `InstanceId`
List of Accessed Fields F_{acc} : **list**($Field$)
Output: Number of Instances in the Reached Space

```
1: function ReachSpace
2:     countInstances ← 1
3:     for $f \in fields(CV)$ do          ▷ Add all instances the search skipped
4:         if ($f \notin F_{acc}$) then
5:             $countInstances \leftarrow countInstances \cdot size(instances(f))$
6:         end if
7:     end for
8: return countInstances
9: end function
```

3. TECHNIQUE

Our technique is embodied by two main algorithms. One algorithm, called *ReachSpace*, computes the space covered based on the field access information when Korat inspects one candidate vector (Algorithm 1). The other algorithm, called *PruneSpace*, computes the space covered based on isomorphism breaking when Korat creates the next candidate vector (Algorithm 2).

ReachSpace The algorithm quantifies the part of the visited space as the count consisting of 1) the current instance, and 2) all instance that are not visited because the corresponding fields were not visited inside the `repOK` method.

PruneSpace The algorithm quantifies the part of the visited space that was skipped because of the pruning. Recall that the Korat search resets the potentially isomorphic candidate fields, meaning that the instance identifier is equal to zero. The algorithm compares two neighboring candidate vectors to estimate the state: CV_{prev} is the previous candidate vector and CV_{cur} is the subsequent candidate vector. F_{acc} is the set of fields accessed while executing `repOK` on the data structure represented by the *prev* state (and that led to the computation of the structure represented by *cur*).

The algorithm iterates over the visited fields, starting from the most recently accessed field, and computes the number of the instances that were skipped due to the resetting (i.e., pruning based on that field). The algorithm stops when it encouters the first field that was not reset, i.e., its instance identifier is only incremented by one.

Implementation We implemented our technique as a *listener* (called *TrackingListener*) for the Korat search. The search notifies the listener every time it creates a new candidate, and also when it terminates. In addition to tracking progress of the Korat search,

Algorithm 2 PruneSpace Algorithm

```
Inputs:
    Previous Candidate Vector $CV_{prev} : Field \rightarrow InstanceId$,
    Current Candidate Vector $CV_{cur} : Field \rightarrow InstanceId$
    List of Accessed Fields $F_{acc}$ : list($Field$)
Output: Number of Instances Pruned by the Search

function PruneSpace
    $countPruned \leftarrow 0$
    for $f \in reverse(F_{acc})$ do
        $prevInstanceId \leftarrow CV_{prev}(f)$
        $curInstanceId \leftarrow CV_{cur}(f)$
        if $curInstanceId = prevInstanceId + 1$ then
            break
        end if
        if $curInstanceId = 0$ then                    ▷ The field was reset
            $skipped \leftarrow size(instances(f)) - prevInstanceId - 1$
            if $skipped > 0$ then            ▷ Reset was isomorphism breaking
                $F_{prefix} \leftarrow prefix :: f$ , where $F_{acc} = prefix :: f :: postfix$
                $reached \leftarrow$ ReachSpace($CV_{prev}, F_{prefix}$)
                $countPruned \leftarrow countPruned + skipped \cdot reached$
            end if
        end if
    end for
return $countPruned$
end function
```

the listener also initializes *bins* that keep track of the distribution of the solutions with respect to the amount of space covered. The listener provides a constructor that allows specifying the number of bins. We add a command-line option to attach the listener to the Korat search. Specifically, we added the option "$--$*track*" that takes a numeric argument that is the number of bins desired. The tracking option can be used together with the built-in option "$--$*progress*" that takes an integer argument k such that Korat outputs its progress every k candidates checked. When both these options are used together our listener outputs the space covered up to that point with each progress report. We plan to release our implementation as part of the Korat code-base [1].

4. EXPERIMENTS

This section presents results of experiments we conducted using our prototype implementation. Our primary goal is to empirically validate the correctness of the implementation of our technique and study some basic characteristics of the Korat search based on the input space coverage information computed by our technique. In addition, we demonstrate for a select subject how our technique enables making observations that can lead to new applications.

4.1 Subjects and methodology

We use ten subjects that are implemented for the standard Korat search and included in the standard distribution of Korat[1] [1]. The subjects are heap-allocated data structures: binary tree (BT), binomial heap (BH), disjoint set (DS), doubly-linked list (DLL), Fibonacci heap (FH), heap array (HA), red-black tree (RBT), search tree (ST), singly-linked list (SLL), and sorted list (SL). We run Korat with space coverage tracking turned on and number of bins set to 40. We use all sizes between 0 and the largest size that does not time-out. All experiments were performed on Linux Ubuntu 18.04.2 running on an Intel Core i7-7500U CPU @ 2.70GHz × 4 with 8GB RAM.

4.2 Results

Table 1 shows the results for all subjects for select sizes such that the time taken is between 0.5 second and 216 seconds. For each subject and size, the number of non-isomorphic solutions found by Korat (*#Valid*), the number of candidates explored by Korat (*#Explored*), the space covered computed by our technique using *reach-space* and *prune-space* calculations (*#Cover*), the percent of space covered based on Korat's backtracking using the field access information (*Reach[%]*), the percent of space covered based on Korat's symmetry breaking (*Prune[%]*), and the total time taken (*Time[s]*) in seconds are shown.

[1]The distribution includes one additional subject, DAG, which we do not consider here since it does not work with the standard Korat search, rather it uses a specialized search implemented using a listener for Korat.

Table 1: Results of Korat Search With Our Listener. All Sizes for Which it Completes Between 0.5 and 216 Seconds are Shown.

Subj.	*Sz.*	*Valid* [#]	*Expl.* [#]	*Cover* [#]	*Reach* [%]	*Prune* [%]	*P:R*	*Time* [s]
BT	10	1.7e4	8.2e5	7.4e21	10.3	89.7	8.7	0.9
	11	5.9e4	3.2e6	6.6e24	9.3	90.7	9.8	3.2
	12	2.1e5	1.2e7	7.1e27	8.5	91.5	10.8	11.8
	13	7.4e5	4.8e7	8.8e30	7.8	92.2	11.8	50.5
BH	7	1.1e5	2.6e5	1e33	17.1	82.9	4.9	0.5
	8	6e5	1.3e6	4.7e39	14.7	85.3	5.8	2.4
	9	8.7e6	1.2e7	3.9e46	12.9	87.1	6.7	21.2
DS	5	4.2e4	4.1e5	1.1e12	87.4	12.6	0.1	0.7
	6	3e6	3.3e7	5e15	88.5	11.5	0.1	23
DLL	10	5.6e5	5.6e5	4.9e36	9.1	90.9	10	0.9
	11	3.5e6	3.5e6	1.6e41	8.4	91.7	11	5.9
	12	2.3e7	2.3e7	7e45	7.7	92.3	12	43.5
FH	6	1.5e6	4.8e6	1.4e32	7.8	92.2	11.9	5.3
	7	5e7	1.8e8	6.7e39	5.8	94.2	16.4	216.7
HA	8	1e6	5.2e6	8.1e9	100	0	0	1.9
	9	1e7	5.1e7	2.4e11	100	0	0	19.6
RBT	8	64	3.2e5	7.9e33	13.8	86.2	6.2	0.8
	9	122	1.5e6	5.1e39	12.2	87.8	7.2	3.9
	10	260	7.5e6	5.1e45	10.9	89.1	8.2	20.7
	11	586	3.9e7	7.5e51	9.8	90.2	9.2	116.1
ST	8	1.4e3	2.6e6	2.8e23	13	87	6.7	2.4
	9	4.9e3	2e7	3.9e27	11.5	88.5	7.7	20.4
	10	1.7e4	1.6e8	7.4e31	10.3	89.7	8.7	176.4
SLL	10	1.2e5	1.7e6	6.6e24	9.2	90.9	9.9	1.3
	11	6.8e5	1.1e7	7.1e27	8.4	91.6	10.9	8.5
	12	4.2e6	7e7	8.8e30	7.7	92.3	11.9	63.2
SL	10	9.2e4	9.2e5	1.9e36	16	84	5.3	0.8
	11	3.5e5	3.9e6	6.3e40	14.8	85.2	5.7	3.6
	12	1.4e6	1.6e7	2.7e45	13.8	86.2	6.3	16.8
	13	5.2e6	6.8e7	1.4e50	12.9	87.1	6.8	73.3

As expected, in all cases, the total space covered compute matched the size of the input space defined by the finitization, thereby validating the correctness of our implementation.

Comparison of *reach-space* and *prune-space* For all subjects except disjoint set and heap array, the total *reach-space* is substantially less than the total *prune-space*, and *prune-space* is between 6.7× (for binomial heap) and 16.3× (Fibonacci heap) the *reach-space* for the largest sizes used. For disjoint set, the *reach-space* is 7.7× the *prune-space*. Disjoint set is represented using an array of *records* and each record only has integer fields. Since integers (and other primitives) are not interchangable, there are fewer symmetries to be broken. An extreme case is heap array, where the *prune-space* is 0. The data structure is an array of integers and there are no symmetries that can be broken. Overall, symmetry breaking plays an important role in reducing the number of candidates considered by Korat. Field access monitoring plays an even more important role since it directly contributes to the *reach-space*, and in addition, is necessary to compute *prune-space* since symmetry breaking on a field f requires the highest index used on (compatible) fields accessed *before* f.

Distribution of solutions with respect to space covered Next we illustrate a new potential application of our technique. Figure 4 shows the disribution of solutions found by Korat with respect to the space it covers for the binary tree subject. The results for the five largest sizes used for evaluation are graphically plotted. Specifically, each graph shows how the percent of solutions

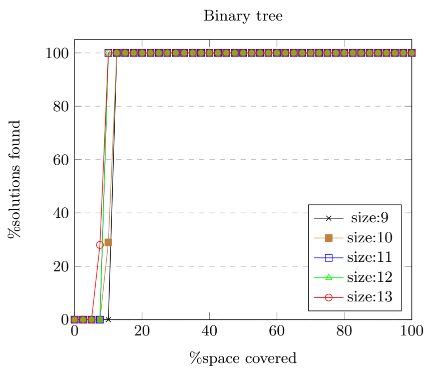

Figure 4: Space coverage for binary tree.

found varies with the percent of input space covered. The plots for different sizes show a similar trend: almost all solutions are found when less than 20% of the space is covered. Such similarities can allow estimating results, such as solution counts, by computing results for a smaller size, performing partial exploration of the desired size to compute a partial result, and extrapolating it to an estimate of the desired result [2].

5. RELATED WORK

Recent work studied the distribution of solutions found by the Korat search with respect to the candidates it explored and observed that after some initial exploration, the Korat search tends to find solutions linearly with the candidates explored; it also identified opportunities for estimating solution counts by observing the distribution for a smaller size and using it for a larger size [2]. In this paper, we introduce a new metric that precisely quantifies the space covered as the search progresses and provides new insights into the search.

The process of creating a tool that quantifies and reports Korat's space exploration progress requires a partial answer to the problem of estimating the size of a backtrack search tree. This problem was addressed by Knuth using probing samples over forty years ago [16], which was expanded upon by others in the decades to follow once, especially when propositional satisfiability (SAT) solvers started to become practical [3, 9, 14, 17, 21].

This problem is particularly notable in the domain of symbolic execution [15] and model checking [7], where the goal is to estimate the size of the exploration tree, or more generally, graph. The symbolic execution engine Java PathFinder (JPF) [25] has two existing state space size estimators that provide mid-execution progress reports: *StateCountEstimator*, which estimates the number of possible program states as they are explored, and *JPF-Bar* [26], which estimates the search's position in the graph based on the number of paths discovered so far. Taleghani and Atlee [24] created a Monte Carlo-based algorithm that estimates state-space coverage by sending additional probes into the search space.

Since our application domain uses finitizations, the size of the total input space is *a priori* knowledge. The reduced version of the problem we address here is to determine the size of the space of the explored space and the size that remains to be explored. While our technique is the first to perform this sort of analysis and reporting for an imperative constraint solver, efforts to add declarative constraints to imperative languages date several decades [4], ultimately resulting in tools such as Babelsberg/R for Ruby [10] and Turtle++ for C++ [12]. A notable tool to enable constraint solving using purely declarative programming is Alloy [13, 20].

6. CONCLUSION

This paper introduced a technique for precisely quantifying the size of the input space explored by the constraint solver Korat during its systematic search. The technique was implemented as a listener for Korat and experimentally validated using a suite of standard Korat subjects. The technique provides exact feedback to the user about the size of the input space that is yet to be explored by Korat as it executes, and allows collecting useful statistics about the search, which can enable new approximation methods in software analysis.

Acknowledgments

This work was partially supported by the US National Science Foundation under Grants Nos. CCF-1652517, CCF-1703637, CCF-1704790, and CCF-1846354.

7. REFERENCES

[1] Korat GitHub repository, 2019. https://github.com/korattest/korat.
[2] A. Almaawi, N. Dini, C. Yelen, M. Gligoric, S. Misailovic, and S. Khurshid. A study of the solution distributions for structural constraints using Korat. Under submission, 2019.
[3] F. A. Aloul, B. D. Sierawski, and K. A. Sakallah. Satometer: How much have we searched? In *DAC*, 2002.
[4] K. R. Apt, J. Brunekreef, V. Partington, and A. Schaerf. Alma-0: An imperative language that supports declarative programming. Technical report, Amsterdam, The Netherlands, 1997.
[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, 2002.
[6] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.
[7] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.
[8] L. de Moura and N. Bjorner. Z3: An efficient SMT solver. In *TACAS*, 2008.
[9] N. Een and N. Sorensson. An extensible SAT-solver. In *SAT*, 2003.
[10] T. Felgentreff, A. Borning, and R. Hirschfeld. Specifying and solving constraints on object behavior. *The Journal of Object Technology*, 13:1:1, 04 2014.
[11] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223, 2005.
[12] P. Hofstedt and O. Krzikalla. TURTLE++ – a CIP-library for C++. In *INAP*, pages 12–24, 2005.
[13] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012.
[14] P. Kilby, J. K. Slaney, S. Thiébaux, and T. Walsh. Estimating search tree size. In *AAAI*, 2006.
[15] J. C. King. Symbolic execution and program testing. *CACM*, 19(7), 1976.
[16] D. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129), 1975.
[17] D. Kokotov and I. Shlyakhter. Progress bar for SAT solvers. Unpublished manuscript, 2000.
[18] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. 2000.
[19] D. Marinov and S. Khurshid. TestEra: A novel framework for automated testing of Java programs. In *ASE*, 2001.
[20] A. Milicevic, J. P. Near, E. Kang, and D. Jackson. Alloy*: A general-purpose higher-order relational constraint solver. In *ICSE*, pages 609–619, 2015.
[21] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, June 2001.
[22] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/SIGSOFT FSE*, 2005.
[23] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *PLDI*, pages 281–294, 2005.
[24] A. Taleghani and J. M. Atlee. State-space coverage estimation. In *ASE*, 2009.
[25] W. Visser, K. Havelund, G. P. Brat, and S. Park. Model checking programs. In *ASE*, 2000.
[26] K. Wang, H. Converse, M. Gligoric, S. Misailovic, and S. Khurshid. A progress bar for the JPF search using program executions. In *JPF, 2018.*
[27] J. Zhang. *The generation and application of finite models*. PhD thesis, Institute of Software, Academia Sinica, Beijing, 1994.