Predictive Constraint Solving and Analysis

Alyas Almaawi
alyas.mohammed@utexas.edu
University of Texas at Austin

Nima Dini
nima.dini@utexas.edu
University of Texas at Austin

Cagdas Yelen
cagdas@utexas.edu
University of Texas at Austin

Milos Gligoric
gligoric@utexas.edu
University of Texas at Austin

Sasa Misailovic
misailo@illinois.edu
University of Illinois at Urbana-Champaign

Sarfraz Khurshid
khurshid@utexas.edu
University of Texas at Austin

ABSTRACT

We introduce a new idea for enhancing constraint solving engines that drive many analysis and synthesis techniques that are powerful but have high complexity. Our insight is that in many cases the engines are run repeatedly against input constraints that encode problems that are related but of increasing complexity, and domain-specific knowledge can reduce the complexity. Moreover, even for one formula the engine may perform multiple expensive tasks with commonalities that can be estimated and exploited. We believe these relationships lay a foundation for making the engines more effective and scalable. We illustrate the viability of our idea in the context of a well-known solver for imperative constraints, and discuss how the idea generalizes to more general purpose methods.

KEYWORDS

History-aware analysis, approximate model counting, Korat, SAT

ACM Reference Format:
Alyas Almaawi, Nima Dini, Cagdas Yelen, Milos Gligoric, Sasa Misailovic, and Sarfraz Khurshid. 2020. Predictive Constraint Solving and Analysis. In *New Ideas and Emerging Results (ICSE-NIER'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377816.3381740

1 INTRODUCTION

Many analysis and synthesis techniques, including symbolic execution, fuzzing, and program repair, rely critically on the efficiency of backend constraint solvers and decision procedures, which often suffer from scalability issues [6, 10, 15, 18, 20]. This paper introduces a new idea for enhancing the performance of the engines by tackling the challenge they face in exploring very large state spaces of possible solutions. Our insight is that in many application scenarios the engines are run repeatedly against input formulas that encode problems that are related but of increasing complexity, and domain-specific knowledge can be exploited to mitigate the increase in complexity. Moreover, even for one formula, the engine may perform multiple expensive computations with commonalities that can be estimated and exploited likewise. Our thesis is that these relationships lay a foundation for making the engines more effective and their applications more scalable.

Our focus is tools that take as input *logical constraints*, which arise naturally in the context of many analysis and synthesis problems [6, 10, 15, 18, 20]. For example, a *propositional satisfiability* (SAT) solver takes as input a formula in *conjunctive normal form* (CNF) to determine its satisfiability, and outputs a solution (if one exists). As another example, a *model counter* [3, 5, 9, 17] computes the number of solutions that a given formula has.

Our work is motivated by two general strategies for analysis of code – *iterative deepening* and *bounded exhaustive checking* – which have their origins in *model checking* [10]. The strategies are effective in both *white-box* analysis (e.g., symbolic execution) and *black-box* analysis, say using *bounded-exhaustive testing* [6] where the code is tested against all (non-equivalent) inputs within a bound on the input size, which is likewise increased.

As an example, consider creating solutions to a formula that represents a precondition in bounded-exhaustive testing. The solver's task is to enumerate solutions for the given formula, which, once defined, remains fixed (in terms of the precondition it characterizes). Our idea is to exploit any commonalities in different search steps that the solver must perform during enumeration. We expect many such commonalities to exist – after all, the solver creates many inputs that may have a lot in common, e.g., in the form of sub-structures they contain. We believe we can abstract key steps of the solver's search and re-use them, e.g., for more aggressive pruning, as it enumerates the solutions.

As another example, consider computing the model count for a formula that encodes a domain-specific problem for size bound k in the context of iterative deepening where the tool already computed the counts for the same problem for some smaller bounds, say $k-1$ and $k-2$. We believe the previous runs of the tool can provide vital information that enables the model count for the desired bound k to be computed without a full execution of the model counter for this bound, e.g., by hypothesizing relations among the partial results for different bounds and projecting the observed results for the smaller bounds to estimate the result for the desired bound.

To demonstrate the viability of our idea, we develop it in the context of the well-known Korat [6] solver for *imperative* constraints, which has been used for automated testing [6] and quantitative analysis [14]. As supporting evidence we present empirical data that enables insights into the Korat search steps and key observations that show the potential for our idea to lead to new methods

ICSE-NIER'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7126-1/20/05…$15.00
https://doi.org/10.1145/3377816.3381740

```
class Node {
    Node left, right; /* children */
    int info; /* data */ }

class SearchTree {
    Node root; /* root node */
    int size; /* number of nodes in the tree */

    boolean repOK() {
       if (root == null) return size == 0; // checks that empty tree has size zero
       if (!isAcyclic()) return false; // checks that the input is a tree
       if (numNodes(root) != size) return false; // checks that size is consistent
       if (!isOrdered(root)) return false; // checks that data is ordered
       return true; }
    ...
```

Figure 1: Search tree declaration and constraints in Java.

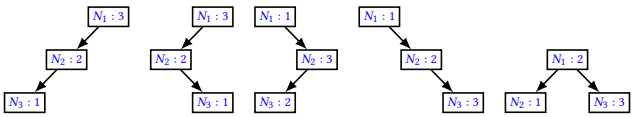

Figure 2: Five non-isomorphic search trees with 3 nodes N_1, N_2, and N_3, and data 1, 2, and 3. N_1 is the root.

for more efficient constraint solving and model counting. Specifically, we observe how solutions found by Korat are distributed with respect to the candidates it considers during its search. We build on these observations to lay the foundation of new techniques for constraint solving and model counting using Korat, and discuss how these can generalize to a much broader class of tools, which can substantially impact software analysis and synthesis techniques by making them much more scalable.

Related work. Our approach is rooted in the general principles of *memoization* and *incremental methods* in computer science [11], which have been employed in many previous analysis and synthesis techniques, including the following that are most closely related to our proposal: techniques for enhancing symbolic execution using memoization where constraint solving results are reused [7, 12, 24, 25], transcoping where results for smaller bounds on analysis size are used to optimize analysis for larger bounds [21], and model counting for complex structures [13], and regression testing where the cost of executing tests is reduced based on previous runs [26].

The key novelty of our work is to define an integration of the principles of memoization and incremental methods in the areas of constraint solving and model counting with a focus on the engine's internal state. Previous work in constraint solving and model counting has neither studied solution distributions during the progress of the engine nor leveraged the distributions (whether estimated or known) to optimize the engine.

2 BACKGROUND: KORAT

This section describes the basics of Korat using binary search trees (Figure 2) as an example [6]. Given an imperative constraint and bound on the input size, Korat performs a backtracking search with pruning and symmetry breaking to enumerate all non-isomorphic solutions within the bound. Figure 1 shows an example constraint solving problem [2]. The `repOk` method defines an imperative predicate that inspects (using helper methods) its input structure, i.e., `this`, and returns true if all expected properties hold, and false otherwise. Specifically, `repOk` checks: 1) the input is an acyclic object graph (along `left` and `right`); 2) its nodes contain data in the correct search order; and 3) its `size` field equals the number of nodes in the tree. To bound the input size, Korat uses a *finitization*, which is given as a Java method. The helper methods for `repOk` and finitization are omitted here but available in the Korat distribution [2].

The imperative constraint and finitization define a *parameterized* constraint solving problem. Providing the number of possible values for the fields of different types using the finitization bounds the problem space. Korat solves the given problem by iteratively creating a *candidate structure*, invoking `repOk` to check the structure, outputting it if valid, and creating the next structure based on the object fields accessed by `repOk`. Internally, Korat represents each structure as a *candidate vector*: a sequence of integer indices into ordered domains of values for each field.

Consider the constraint solving problem with finitization bound of 2, i.e., each solution must have exactly 2 nodes. The `root` field takes a value from the domain [`null`, $N1$, and $N2$], where $N1$ and $N2$ are the two possible nodes that can be in the tree. Each of the two reference fields (`left` and `right`) of the two nodes also take a value from this domain. The field `size` has only 1 possible value (i.e., 2). The `info` field of each node takes a value from [0, 1]. The following partial list shows the first five and the last candidate vectors that Korat considers and the fields accessed for each candidate:

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

Each row contains a candidate vector that has 8 elements – `root`, `size`, $N1$.`left`, $N1$.`right`, $N1$.`info`, $N2$.`left`, $N2$.`right`, and $N2$.`info` – and list of fields accessed with "`::`" as the separator. The valid candidates are marked "`***`". Each vector element is an index in the corresponding field domain. The first candidate, i.e., all 0's, has all 5 reference fields set to `null` (since index 0 represents the first value, i.e., null, from the corresponding domain), `size` set to 2 (since its domain only has that single value), and each node's `info` is 0; this candidate is invalid because the value of `size` is incorrect.

Setting the bound to 5 in the finitization and running Korat takes 0.29 seconds to find 42 solutions where each solution is a (valid) search tree with exactly 5 nodes. During the search, Korat explicitly checks 6,155 candidate vectors, i.e., runs `repOk` on each of them, before finding all the solutions. Increasing the bound to 9 and re-running Korat we get 4,862 solutions in 17.37 seconds after it explores 20,086,300 candidate vectors; the space of candidate vectors for this problem has size $> 2^{63}$– Korat prunes much of it.

3 OUR APPROACH

3.1 Study of Solution distributions

Our idea is motivated by our initial study of the solution distributions during the Korat search, specifically how the number of solutions found by Korat grows with the number of candidates it explores. Figure 3 plots the distributions for 10 subjects from the Korat distribution: binary tree (*BT*), binomial heap (*BH*), directed acyclic graph (*DAG*), disjoint set (*DS*), doubly-linked list (*DLL*), Fibonacci heap (*FH*), heap array (*HA*), red-black trees (*RBT*), search tree (*BST*), sorted list (*SL*), and singly-linked list (*SLL*). For each subject, two consecutive sizes are plotted; the maximum size is set to 10 if the generation completes in 300 seconds; otherwise, the largest size that completes in 300 seconds is used. For each subject,

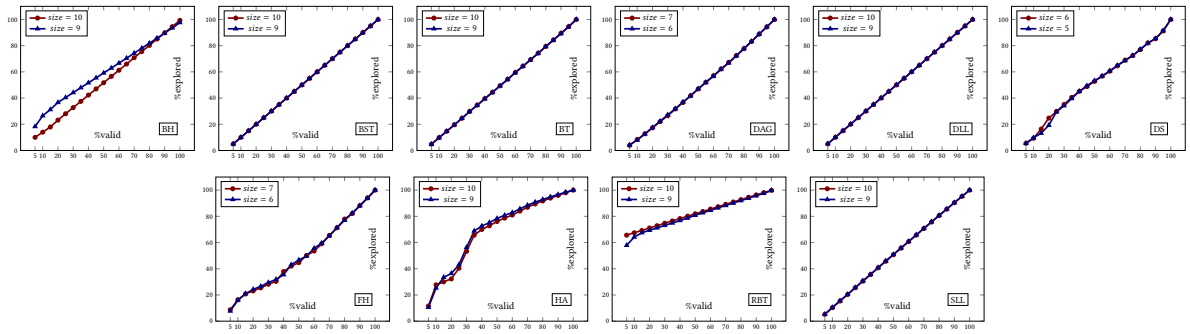

Figure 3: Solution distribution for subject constraints for select sizes.

the figure shows the *%explored* (y-axis) against *%valid* (x-axis) for two cases: 1) maximum size s for the subject; and 2) size $s-1$.

For six of the ten structures, namely BT, BST, DAG, DLL, FH, and SLL, the lines (for two sizes) are almost identical. Moreover, for two of the subjects, namely DS and RBT, there are small differences up to 30% solutions found – with the maximum distance for DS at 5.42 for 20% solutions found and for RBT at 3.29 for 10% solutions found – and then the lines become almost identical. For HA, the two lines are different initially but converge in the last 20%. For BH the lines are the most different among all these subjects and the maximum difference is as high as 13.47 (for 10% solutions found).

We also computed the R-squared values for the subject constraints based on the distributions we observed for the maximum sizes selected. Eight of the ten subjects (BH, DLL, BT, RBT, DAG, DS, SLL, BST) have R-squared value > 0.99; FH has R-squared value between 0.98 and 0.99, and HA has R-squared value < 0.98, around 0.91. Overall, the R-squared and corresponding trend-lines show that for majority of the subjects the solution count obtained during partial search predicts accurately the results for the full search.

3.2 Predictive constraint solving and analysis

The solution distributions provide two key observations that allow us to propose predictive constraint solving and analysis techniques:

(1) Running the solver to find solutions up to a fraction, e.g., 30%, of the total number of structures considered by the standard Korat search can help estimate the total solution count; and
(2) The solution distribution for a constraint c solved for a size bound s can provide an estimate for the model count for a larger bound u ($> s$) for the constraint c.

Predictive model counting. Perhaps the most direct application of the two observations is to predict model counts using Korat. While our observations provide insights into new techniques, the observations do not lend themselves directly into automated techniques. Consider Observation 1 for example. Even if we know that the partial model count with respect to 30% of the total number of structures considered is an excellent predictor for the full model count, we still face the problem of not knowing how many structures the full Korat search would actually consider, and hence not know when the search reaches the 30% mark.

We next outline a technique to tackle this problem. Assume Korat is used to estimate the model count for formula f for size bound s. Observe that the state spaces (typically) grow at an exponential rate with respect to the input size. We hypothesize that the number of candidates explored by the Korat search also grows at an exponential rate, i.e., $x = ab^s$, where x is the number of candidates explored and s is the size bound, and a and b are constants. Under this hypothesis, it is simple to compute the values of a and b using the results of the Korat search for sizes smaller than s, e.g., using the two sizes $s-1$ and $s-2$. Table 1 shows the results of estimating the number of candidates explored using this technique. In 5 of 10 cases the estimate is within 10% of the actual count; the worst case is for DAG where the estimate is 48.49% of the actual count.

Approximate sampling. As a second application we show how to use Korat to create a desired sample of test inputs, say to complement bounded-exhaustive testing by creating some larger inputs, and testing against them. Let us sample approximately k% of inputs in bound s. Our technique for estimating the model count (Section 3.2) immediately lends to a sampling technique: estimate the model count c, and run Korat until it finds $\frac{k \times c}{100}$ solutions.

Preemptive pruning. As a third application we discuss how to enhance the standard Korat search for enumerating solutions for formula f and bound s. Observation 1 in an instance of a more general property: the past behavior of the search can predict the future behavior. We envision exploiting this property by abstracting and reusing key steps that the search takes in finding the next solution. To illustrate, observe the following consecutive steps in the Korat search for creating search trees with 3 nodes where all non-zero values for the 10th field, which represents the right child of the third node, are tried, but none of them creates a valid tree:

```
1 0 0 2 0 0 3 0 0 1 0
1 0 0 2 0 0 3 0 0 2 0
1 0 0 2 0 0 3 0 0 3 0
```

Later in the search this sequence of assignments repeats for the same field, and once again creates only invalid trees:

Table 1: Estimating the number of candidates explored for size s using the known numbers for sizes $s-1$ and $s-2$.

Subject	*size*	*space*	*#valid*	*#explored*	*#est. expl.*	*est./actual [%]*
BH	10	2^{178}	117,157,172	150,727,471	104,840,109	69.56
BST	10	2^{105}	16,796	155,455,872	154,761,948	99.55
BT	10	2^{72}	16,796	815,100	813,824	99.84
DAG	7	2^{157}	1,410,723	20,128,126	9,760,786	48.49
DLL	10	2^{121}	562,595	562,823	525,969	93.45
DS	6	2^{52}	2,967,087	33,436,639	21,639,413	64.72
FH	7	2^{132}	49,698,272	175,980,937	140,719,184	79.96
HA	10	2^{42}	111,511,015	583,317,405	506,210,306	86.78
RBT	10	2^{151}	260	7,530,712	7,063,431	93.79
SLL	10	2^{82}	115,975	1,702,171	1,592,413	93.55

```
1 0 0 2 0 3 0 0 0 1 0
1 0 0 2 0 3 0 0 0 2 0
1 0 0 2 0 3 0 0 0 3 0
```

In fact, during the search, this sequence of assignments is also made for some other fields, e.g., the left child of the third node, which also creates only invalid trees. For this subject, the reason is that these failed assignments create a "back" edge, i.e., a cycle, and hence we get an invalid tree. We envision techniques that detect and exploit such commonalities for more effective pruning, perhaps using machine learning methods [19] to detect more complex relations, can make the search much more efficient.

3.3 #Korat

We envision #Korat, a new core engine for model counting for imperative predicates, which is based on insights into the Korat search but unlike Korat, does not need to enumerate each solution. A key restriction of the current Korat search is the need to concretely initialize each candidate input and check it using *repOk*. Our insight is to re-define Korat's backtracking such that it prunes *valid* candidates when their validity can be determined by reasoning about the *repOk* checks of other similar valid candidates. We envision #Korat will enable new classes of optimized model counters that support analyses of important classes of programs, including probabilistic programs [8], especially in the context of dynamic data structures.

3.4 Generalization to propositional logic

We next discuss how our idea may be generalized to other tools, specifically general purpose propositional satisfiability solvers (SAT) and model counters, which take as input CNF formulas. While there are many differences in Korat and the search of say, a SAT solver, there is something basic in common: both are search engines that maintain internal state to explore very large state spaces.

For SAT solvers, a starting point is the two observations we made about Korat (Section 3). We can check if they hold for a modern SAT solver [1], e.g., one based on *CDCL*, which maintains a history of *conflict* clauses for enhanced pruning [22]. There are several ways we can perform the check: 1) observe the number of *backjumps* the SAT solver makes; 2) introduce and observe a counter in the SAT solver that tracks the candidate solutions it prunes; and 3) observe the number of conflict clauses that the SAT solver adds.

For model counters, the generalization is less direct because they (except under exceptional cases) do not just enumerate all solutions and report the count, rather they employ dedicated algorithms for directly computing the counts. We discuss the generalization for two classic techniques that lay at the heart of modern model counters: 1) caching sub-formulas in the case of DPLL-based exact model counting [4]; and 2) repeatedly adding *XOR* clauses and solving in the case of probabilistic approximate model counting [16]. Assume the model counter is used for computing counts for two formulas derived from the same domain-specific problem but with different size bounds s and u where $s < u$. We hypothesize the cache hits and misses for bound s can predict the hits and misses for the bound u, and hence allow improving the caching strategy. Likewise, we hypothesize the number of XOR clauses that are added for bound s can predict the number to add for u; to notice the benefit of this prediction, observe that the addition of each XOR clause is followed by a constraint solving call to a non-traditional SAT solver that can also handle XOR constraints [23]; therefore, if the number of clauses to add can be estimated, many solver calls can be saved.

4 CONCLUSION

We proposed an approach to enhance constraint solvers and model counters based on observing the steps they take during their exploration in one run or across multiple runs, and exploiting the commonalities in the subproblems encountered. We presented supporting evidence in the context of the Korat solver. We also proposed a generalization to propositional solvers and model counters. We believe our work opens a new direction for tackling the complexity of very large state spaces, and can substantially impact analysis and synthesis techniques by making them much more scalable.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and Marko Vasic for valuable comments. This work was partially supported by NSF grants CCF-1703637, CCF-1704790, and CCF-1846354, and a Facebook Testing and Verification Research Award.

REFERENCES

[1] 2019. International SAT Competitions Webpage. http://www.satcompetition.org/.
[2] 2019. Korat GitHub Repository. https://github.com/korattest/korat.
[3] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. 2015. Automata-Based Model Counting for String Constraints. In *CAV*. 255–272.
[4] Paul Beame, Russell Impagliazzo, Toniann Pitassi, and Nathan Segerlind. 2003. Memoization and DPLL: Formula Caching Proof Systems. In *Complexity*.
[5] A. Biere, M. Heule, H. van Maaren, and T. Walsh (Eds.). 2009. *Handbook of Satisfiability*. Vol. 185.
[6] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated testing based on Java predicates. In *ISSTA*.
[7] C. Cadar, D. Dunbar, and D. R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*.
[8] Bob Carpenter et al. 2017. Stan : A Probabilistic Programming Language. *JSS* 76, 1 (2017).
[9] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2013. A Scalable Approximate Model Counter. In *CP*. 200–216.
[10] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. 1999. *Model Checking*. MIT Press.
[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2009. *Introduction to Algorithms*. The MIT Press.
[12] Nima Dini. 2016. *MKorat: A novel approach for memorizing the Korat search and some potential applications*. Master's thesis. University of Texas at Austin.
[13] Antonio Filieri, Marcelo F. Frias, Corina S. Pasareanu, and Willem Visser. 2015. Model Counting for Complex Data Structures. In *SPIN*. 222–241.
[14] Antonio Filieri, Corina S. Păsăreanu, and Willem Visser. 2013. Reliability Analysis in Symbolic Pathfinder. In *ICSE*. 622–631.
[15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *PLDI*. 213–223.
[16] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. 2006. Model Counting: A New Strategy for Obtaining Good Bounds. In *AAAI*. 54–61.
[17] Seonmo Kim and Stephen McCamant. 2018. Bit-Vector Model Counting Using Statistical Estimation. In *TACAS*. 133–151.
[18] J. C. King. 1976. Symbolic Execution and Program Testing. *CACM* 19, 7 (1976).
[19] Thomas M. Mitchell. 1997. *Machine Learning* (1 ed.). McGraw-Hill, Inc.
[20] M. Monperrus. 2018. Automatic Software Repair: A Bibliography. *CUSR* 51, 1 (2018).
[21] N. Rosner, C. Pombo, N. Aguirre, A. Jaoua, A. Mili, and M. F. Frias. 2013. Parallel Bounded Verification of Alloy Models by TranScoping. In *VSTTE*. 88–107.
[22] João P. Marques Silva and Karem A. Sakallah. 1996. GRASP – a New Search Algorithm for Satisfiability. In *ICCAD*. 220–227.
[23] Mate Soos, Karsten Nohl, and Claude Castelluccia. 2009. Extending SAT Solvers to Cryptographic Problems. In *TACAS*. 244–257.
[24] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In *FSE*. 58:1–58:11.
[25] Guowei Yang, Corina S. Pasareanu, and Sarfraz Khurshid. 2012. Memoized symbolic execution. In *ISSTA*. 144–154.
[26] S. Yoo and M. Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *STVR* 22, 2 (2012), 67–120.