Optimizing a Structural Constraint Solver for Efficient Software Checking

Junaid Haroon Siddiqui
UT Austin, Austin TX, 78712
jsiddiqui@ece.utexas.edu

Darko Marinov
Univ. of Illinois, Urbana, IL 61801
marinov@illinois.edu

Sarfraz Khurshid
UT Austin, Austin TX, 78712
khurshid@ece.utexas.edu

***Abstract*—Several static analysis techniques, e.g., symbolic execution or scope-bounded checking, as well as dynamic analysis techniques, e.g., specification-based testing, use constraint solvers as an enabling technology. To analyze code that manipulates structurally complex data, the underlying solver must support structural constraints. Solving such constraints can be expensive due to the large number of aliasing possibilities that the solver must consider. This paper presents a novel technique to selectively reduce the number of test cases to be generated. Our technique applies across a class of structural constraint solvers. Experimental results show that the technique enables an order of magnitude reduction in the number of test cases to be considered.**

I. Introduction

Manual generation of test inputs for programs with structurally complex inputs, such as heap-allocated complex data structures or XML documents, is expensive and likely ineffective. Automated generation of these tests require specifications or constraints that describe desired tests, as well as technology for solving the constraints. Recent years have seen much advances in constraint solving technology, e.g., in the development of general purpose automated theorem provers, such as Z3 [1] and CVC-Lite [2], as well as, specialized bounded checkers, such as Alloy [3] and Korat [4]. The availability of these solvers has enabled the development of several techniques for systematically checking rich properties of complex programs [5]–[8]. While these techniques apply to real code in principle, scaling them to effectively check properties of real code remains an active research area.

This paper presents a novel technique for increasing the effectiveness of constraint solving algorithms. We call this technique *focused generation*. In particular, this paper considers Korat [4]—an algorithm for solving structural constraints. Given a predicate that describes desired structural properties, Korat searches the predicate's input space systematically and enumerates inputs for which the predicate returns true. To test a method, Korat uses its precondition to generate tests and its postcondition to check correctness. Korat performs bounded exhaustive testing (BET): it runs the method against all non-isomorphic inputs within given bounds. BET has been used to find bugs in various applications, including a fault-tree analyzer, a resource discovery architecture, and an XPath compiler [9].

Korat implements a backtracking search and uses the predicate's executions to prune large portions of its input space. Korat runs the predicate on a candidate input, monitors the fields accessed by the predicate, backtracks on the last field accessed to generate a new candidate by making a non-deterministic assignment to that field, and re-executes the predicate. Korat's backtracking, driven by field-access monitoring, provides significant pruning of input spaces, which are very large even for small sized inputs.

Korat is a specialized solver that can optimize other analysis frameworks. For example, symbolic/concrete execution can delegate solving of structural constraints to Korat. Korat is likely to provide more efficient solving than general purpose solvers, such as SAT or SMT solvers.

Our technique addresses the concern of "what to generate" in a constraint solver. In previous work [10], we optimized the "how to generate" aspect of constraint solvers.

This paper makes the following contributions:

- **Focused Generation:** We propose a way to allow some structural constraints to be solved exhaustively but others solved for a single solution. For instance, we can test a structure that contains instances of pre-tested structures, by exploring the outer structure exhaustively but not the inner structure.
- **Evaluation:** We have implemented focused generation for Korat and evaluated it on sorted singly linked lists and red-black trees. We show the reduction in explored states and time.

II. Background & Example

We consider a binary search tree and explain the working of Korat using it. We present details of the Korat algorithm necessary to understand focused generation. More details are available elsewhere [4]. The definition of this tree is given in Figure 1. Nodes are modeled by the inner class `Node` having `left` and `right` pointers, and integer field `data`. The `BinaryTree` class contains the `root` pointer and `size` field. The `size` field caches the number of reachable nodes. The structural constraints are acyclicity along left and right pointers, search tree order of data fields, and that the number of reachable nodes equal the `size` field.

An imperative predicate is a predicate function written in an imperative language, as opposed to declarative language, to check the structural properties of a complex structure. It

```
class BinaryTree {
  struct Node {
    int data;  Node* left;  Node* right;
  };
  Node* root;  int size;
 public:
  bool repOk() {
    return acyclic() && ordered();
  }
  bool acyclic() {
    std::vector<Node*> visited;
    std::stack<Node*> worklist;
    if( root ) {
      worklist.push( root );
      visited.push_back( root );
    }
    while( !worklist.empty() ) {
      Node* current = worklist.top();
      worklist.pop();
      if( visited.find( current->left ))
        return false;
      if( current->left ) {
        visited.push_back( current->left );
        worklist.push( current->left );
      }
      if( visited.find( current->right ))
        return false;
      if( current->right ) {
        visited.push_back( current->right );
        worklist.push( current->right );
      }
    }
    return visited.size() == size;
  }
};
```

Figure 1. Definition of Binary Search Tree with its `repOk` function (Class Invariant). This function validates three constraints: acyclicity along all paths, validity of size field, and search tree order of data fields.

is conventionally called `repOk`. In object oriented paradigm, such a function is called a *class invariant*. The class invariant to verify structural constraints is provided by the `repOk` function in Figure 1. The part to check acyclicity and size is shown. It starts from root and visits the tree in depth first order. It returns false if the left pointer points to a visited part of the tree, if the right pointer points to a visited part of the tree, or if the total number of visited nodes does not match the `size` field. The part of class invariant to validate search tree ordering is not shown.

Korat is a solver for structural constraints. Its takes an input bound specification called *finitization*. Finitization contains one main object representing the structure to be tested. For our example, it is an object of class `BinaryTree`. For every field directly or indirectly accessible from this main object, Korat needs a domain of values to be used. These are called *field domains* of the respective fields. Different fields with the same *type* can share a field domain.

For our example, the field domain of `root` contains three objects of class `Node` and the field domain of `size` is the set $\{0,1,2,3\}$. The fields `left` and `right`, accessible indirectly through each of the three `Node` objects, share the same field domain as `root`. The `data` fields of each Node, also accessible indirectly, take a value from the set $\{1,2,3\}$. We have 14 fields in the finitization. Two fields are for `root` and

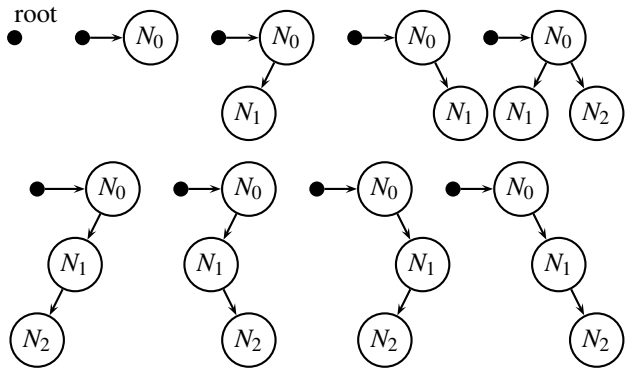

Figure 2. There are nine non-isomorphic binary trees of up to 3 nodes out of 65,536 possible structures. The remaining trees are cyclic, have invalid size, or are isomorphic to the ones shown here.

`size` of the single `BinaryTree` object. Each of the three `Node` objects take three fields for the values of `left` and `right` pointers, and the `data` field. All pointer fields take a value from the set $\{\text{NULL}, N_0, N_1, N_2\}$ where N_0, N_1, and N_2 are the three `Node` objects.

An assignment for a field in the finitization can be represented by an index into its field domain. For a fixed order of fields, the assignment of all fields can be given with an ordered tuple of integers. This is called a *candidate vector* in Korat. An entry in the candidate vector is an index into the field domain of the corresponding field. For this example, the candidate vector has 14 integers.

For simplicity, we ignore the `data` field for explaining the working of baseline Korat. Considering the given binary tree representation and ignoring the `data` field, there are nine valid non-isomorphic binary trees as shown in Figure 2. The number of possible field assignments is $4^8 = 65,536$. *Isomorphic structures* are obtained by swapping identities of objects. For example, the root is always N_0 is Figure 2. The same trees can be obtained by taking N_1 or N_2 as root and the other two as children. By pruning isomorphic candidates we avoid such similar structures.

For generation of binary trees up to three nodes using the given definition, Korat explores 90 candidates out of over 60 thousand possible candidates and discovers all 9 valid binary trees.

Korat algorithm is written as a recursive function in Figure 3. This recursive function starts with the candidate vector with all zeroes. It builds a C++ object structure using BUILDCANDIDATE. BUILDCANDIDATE makes field assignments according to candidate vector indices and generates an actual candidate from a candidate vector. It then tests the candidate using `repOk`. Then for all accessed fields up to a field pointing to a non-zero index, it recursively tests candidates for all non-zero indices of that field up to the maximum index given by NONISOMAX.

```
1: procedure KORAT(candidateV)
2:     candidate ← BUILDCANDIDATE(candidateV)
3:     (predicate, accessFields) ← REPOK(candidate)
4:     if predicate then
5:         VALIDCANDIDATE(candidate)
6:     end if
7:     while SIZE(accessFields)>0 ∧
           candidateV[TOP(accessFields)]=0 do
8:         for i ← 1, NONISOMAX(candidateV, accessFields) do
9:             candidateV[TOP(accessFields)] ← i
10:            KORAT(candidateV)
11:        end for
12:        candidateV[TOP(accessFields)] ← 0
13:        POP(accessFields)
14:    end while
15: end procedure

16: function NONISOMAX(candidateV, accessFields)
17:    f ← TOP(accessFields)
18:    if PRIMITIVE(f) then
19:        NonIsoMax ← MAXDOMAININDEX(f)
20:    else
21:        t ← 0;
22:        for all i ∈ accessFields do
23:            if SAMEDOMAIN(i, f) ∧ t<candidateV[i] then
24:                t ← candidateV[i]
25:            end if
26:        end for
27:        NONISOMAX ← MIN(t+1,MAXDOMAININDEX(f))
28:    end if
29: end function
```

Figure 3. The Korat algorithm written as a recursive function. It builds a C++ object structure using BUILDCANDIDATE and tests it using `repOk`. It recursively tests candidates for all non-zero indices of the last accessed field up to the maximum index given by NONISOMAX.

Further pruning is done in Korat to avoid isomorphic structures. Isomorphic structures are structures that only differ in object identities. Programs are not usually concerned with the identity of an object, e.g. the actual memory address in C++ or the object hash code in Java. The identities of n objects can be interchanged in $n!$ ways. Isomorphic structure avoidance therefore prunes a large portion of search space.

III. FOCUSED GENERATION

Focused generation means bounded exhaustive test generation where the exhaustive nature of test generation is *focused* on specific fields. For every possible assignment for the fields which are focused, the generation should find only one valid assignment for the remaining fields. This allows optimizing bounded exhaustive testing when objects are composed of objects of different classes that have been tested before. To illustrate, consider the binary search tree discussed above with some library class (e.g. Set) as data instead of integer data. Baseline Korat would generate the entire structure at the concrete level (including exhaustively testing the inner Set class). Focused generation allows more efficient testing of the subject class assuming the correctness of the classes it depends on. The knowledge

```
1: global lastvalid ← false
2: procedure KORAT(candidateV)
3:     candidate ← BUILDCANDIDATE(candidateV)
4:     (predicate, accessFields) ← REPOK(candidate)
5:     lastvalid ← predicate
6:     if predicate then
7:         VALIDCANDIDATE(candidate)
8:     end if
9:     while SIZE(accessFields)>0 ∧
           candidateV[TOP(accessFields)]=0 do
10:        if not (lastvalid ∧ onesol[TOP(accessV)]) then
11:            for i ← 1, NONISOMAX(candidateV, access-
                   Fields) do
12:                candidateV[TOP(accessFields)] ← i
13:                KORAT(candidateV)
14:            end for
15:        end if
16:        candidateV[TOP(accessFields)] ← 0
17:        POP(accessFields)
18:    end while
19: end procedure
```

Figure 4. Korat with Focused Generation. A new array *onesol* remembers which fields need only one solution. Global *lastvalid* remembers if the last candidate checked was valid or not.

of an abstraction function suffices for an effective application of this optimization in this scenario.

Focused generation also allows focussing generation towards some aspect of a data structure. For example in our tests we focus generation on the structure of a sorted singly linked list. Thus for a given node structure we get one sorted assignment of numbers however big the allowed range of numbers is. Without focused generation we will get a different test case for every possible sorted subsequence of the given range.

Since focused generation produces a smaller set of results, it cannot be strictly called an optimization of Korat algorithm. However, Korat provides no way of bounded exhaustive testing (BET) for a structure without doing BET for sub-structures. It also provides no way of doing BET for one aspect of the structure but not others. In such cases, we need to filter the results to reduce test suite size and retain the desired tests. Focused generation is an optimization for this procedure of applying Korat and filtering results.

To illustrate how to focus generation on an aspect of a structure, consider the same example of an ordered binary search tree. If we want to test some function that operates on structural aspect of the binary search tree, we may want to generate one binary search tree of each shape with valid data assignments. However, Korat will try to generate all possible data assignments for each valid tree shape. Focused generation can prune out these multiple assignments, leaving a single valid assignment for data. This results in fewer explored states and faster generation. For focused generation to work, the user marks the fields out of focus in the finitization. By default, all fields are in focus, and that means the same behavior as standard Korat. For ordered

Subject	Size	Without Focussed Generation			With Focussed Generation			
		Explored	Valid	Time (s)	Explored	Valid	Time (s)	Speedup
Sorted Singly Linked List	20	179307145	1048575	477.73	1770	20	0.00	>10X
	25	N/A	N/A	TIMEOUT	3275	25	0.01	>10X
	100	N/A	N/A	TIMEOUT	176850	100	1.82	>10X
	200	N/A	N/A	TIMEOUT	1373700	200	32.44	>10X
Red Black Tree	10	4901265	14101	26.34	1903225	377	8.96	2.9X
	11	20130233	40074	109.82	7301092	707	35.98	3.0X
	12	85201572	112813	477.77	28239078	1395	142.94	3.3X
	13	N/A	N/A	TIMEOUT	109591533	2835	563.92	>10X

Table I
RESULTS OF RUNNING KORAT WITH AND WITHOUT FOCUSED GENERATION ON SORTED SINGLY LINKED LIST AND RED-BLACK TREES. FOR EACH STRUCTURE, 4 DIFFERENT SIZES ARE TRIED. TIMEOUT IS SET TO 10 MINUTES.

binary search trees up to three nodes, standard Korat finds 15 valid candidates after exploring 178 candidates. All the trees of size 1 and 2 are repeated thrice with different data values. Focused generation only produces nine candidates; each of the trees in Figure 2 with one valid assignment. These nine trees are found after exploring 127 candidates. Thus an additional 51 candidates are pruned out.

A. Implementation

Korat with Focused Generation is shown in Figure 4. This modified Korat works on unmodified `repOk` constraints. The function VALIDCANDIDATE referred in Figure 4 can take any action on valid candidates, like count them, store them, or directly test some code using them.

To implement focused generation, we introduce a `onesol` array and a `lastvalid` boolean. On line 1 in Figure 4 the variable `lastvalid` is initialized to *false* and on line 5 it is assigned the result of last candidate checked. The `onesol` array is a boolean array containing one entry for each field. To enable this optimization, the user has to mark the field as requiring just one solution (out of focus). This is done using an additional parameter when adding the field in the finitization. This additional parameter results in the corresponding index in `onesol` array to become *true*. The check on line 10 in Figure 4 sees if the last candidate was valid and the user wanted only one solution for the current field, then skip over all remaining values of this field and move to the field accessed before it. If the previous field also needed just one solution, its values are also skipped and we move on. Thus significant portions of state space are pruned out and a lot fewer test cases are generated. All this requires no change in the predicate itself.

IV. EVALUATION

To evaluate focused generation, we have implemented it in Korat for C++. Our implementation can enable or disable focused generation. With it disabled, we get behavior identical to standard Korat. We compared outputs with the Java implementation to ascertain correctness. The experiments were run on a Pentium 2.8GHz processor with 4GB memory running Linux.

To see the effect of focused generation, we take a sorted singly linked list and a red-black tree. We focus testing on structural correctness and not on data ordering requirement. Thus we find all structures with one satisfying data assignment. Results of this experiment are given in Table I.

For both these structures, we took Standard Korat and Korat with Focused Generation. We test four sizes of each structure on both versions of Korat. We compare how focused generation improves Korat in terms of time and in terms of reducing valid candidates generated.

We observe that focused generation, where applicable, can improve time drastically in some cases. For sorted singly linked list, the benefit is enormous because we can avoid all sorted subsequences of data. For red-black trees, focused generation takes about three times less time.

The other observation is that because focused generation is generating fewer candidates, the actual testing time will reduce. For example test execution phase for red-black tree of up to size 14 will run 154 times faster in case of focused generation. This is assuming that all generated tests are used for testing.

V. RELATED WORK

Recent frameworks based on combined symbolic and concrete execution [5]–[7] that handle references/pointers are most closely related to Korat. A major difference is Korat's spirit of bounded *exhaustive* generation and backtracking based on last field accessed and not last branch taken. Generalized symbolic execution [8] follows Korat's spirit: lazy initialization of references has exactly the same effect as Korat's monitoring–both approaches consider the same candidates in the same order and generate the same structures. Practically, Korat is much faster since it is a specialized implementation–baseline Korat is an order of magnitude faster than a highly optimized version of lazy initialization on JavaPathFinder [11]. In our experiments using CUTE [5] for structural constraint solving, Korat outperformed CUTE because of the overhead to keep symbolic state and Korat's specialized nature to backtrack on last accessed field.

SAT-based static analysis tools (e.g. Alloy [3]) can perform bounded exhaustive checking for heap-allocated data. However, they require a translation of the whole program *and* its specification to a SAT formula: for non-small programs the formulas choke the solvers. Korat requires solving

only for input constraints, which are much simpler than the cumulative constraint that represents the correctness of the program under test. Static analysis tools that perform sound analysis of heap-allocated data (traditional shape analysis, verification conditions, separation logic) require more manual effort (in the form of loop invariants, additional predicates etc.) and have not been shown to scale to checking applications which Korat readily handles.

Efficient backtracking [10] optimizes Korat by improving the performance of exploring one candidate. Normally, for every candidate Korat runs `repOk` from the start. However most of the time, only the last accessed field is changed. This means that the initial part of the predicate will run unnecessarily. The initial part ends at the first access of a mutated field. Efficient backtracking uses this idea. Instead of Korat invoking `repOk` for every candidate, `repOk` invokes Korat for the next candidate and then undoes its operations up to the appropriate point. Unlike efficient backtracking, focused generation considers the other problem of what to generate and what not to generate.

Work on parallelizing Korat [9] uses a machine cluster with minimal communication. It runs Korat on appropriately partitioned portions of state space. The appropriate partitioning however requires an initial sequential execution of Korat. The other scheme proposed in the same paper divides the state space using approximations and ends up achieving about 8X speed up on a 1024 machine cluster. PKorat [12] is an alternate way to parallelize Korat on a cluster with efficient communication. Focused generation can directly improve both these techniques.

VI. Conclusions & Future Work

We have presented "focused generation", a novel optimization in Korat where we focus generation to some aspect of the structure like finding all structures with one data assignment or finding all structures with one valid instance of a previously tested contained structure. We have experimentally evaluated this optimization and shown how it reduces test suite size by focussing test generation when applicable.

We believe approaches that optimize and focus bounded exhaustive testing hold much promise in scaling it to bigger real software. Future work should develop heuristics for generating inputs that are more likely to find errors, as well as develop techniques to assist with writing predicates, e.g., by re-factoring poorly written predicates to enable efficient generation.

Acknowledgements

We thank Danhua Shao for detailed comments. This material is based upon work partially supported by the NSF under Grant Nos. IIS-0438967, CNS-0615372, CCF-0702680, CCF-0746856, and CCF-0845628, and AFOSR grant FA9550-09-1-0351.

References

[1] L. M. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS 2008: Proc. of the 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[2] C. Barrett and S. Berezin, "CVC Lite: A new implementation of the cooperating validity checker," in *CAV 2004: Proc. of the 16^{th} Int. Conf. on Computer Aided Verification*. Springer-Verlag, 2004, pp. 515–518.

[3] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, 2002.

[4] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: automated testing based on Java predicates," in *ISSTA '02: Proc. of the 2002 ACM SIGSOFT Int. Symp. on Softw. testing and analysis*. ACM, 2002, pp. 123–133.

[5] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *ESEC/FSE-13: Proc. of the 10th European Softw. Eng. Conf. held jointly with 13th ACM SIGSOFT Int. Symp. on Foundations of Softw. Eng.* ACM, 2005, pp. 263–272.

[6] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI '05: Proc. of the 2005 ACM SIGPLAN Conf. on Programming language design and implementation*. ACM, 2005, pp. 213–223.

[7] C. Cadar and D. R. Engler, "Execution Generated Test Cases: How to Make Systems Code Crash Itself," in *SPIN 2005: Proc. of the 12th Int. SPIN Workshop on Model Checking of Softw.*, 2005, pp. 2–23.

[8] S. Khurshid, C. Pasareanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *TACAS'03: Proc. of the Conf. on Tools and Algorithms for Construction and Analysis of Systems*. Springer, 2003, pp. 553–568.

[9] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov, "Parallel test generation and execution with Korat," in *ESEC-FSE '07: Proc. of the the 6th joint meeting of the European Softw. Eng. Conf. and the ACM SIGSOFT Symp. on The foundations of Softw. Eng.* ACM, 2007, pp. 135–144.

[10] B. Elkarablieh, D. Marinov, and S. Khurshid, "Efficient solving of structural constraints," in *ISSTA '08: Proc. of the 2008 Int. Symp. on Softw. testing and analysis*. ACM, 2008, pp. 39–50.

[11] M. Gligoric, T. Gvero, S. Lauterburg, D. Marinov, and S. Khurshid, "Optimizing Generation of Object Graphs in Java PathFinder," in *ICST '09: Proc. of the 2009 Int. Conf. on Softw. Testing Verification and Validation*. IEEE Computer Society, 2009, pp. 51–60.

[12] J. H. Siddiqui and S. Khurshid, "PKorat: Parallel Generation of Structurally Complex Test Inputs," in *ICST '09: Proc. of the 2009 Int. Conf. on Softw. Testing Verification and Validation*. IEEE Computer Society, 2009, pp. 250–259.