Extension-Aware Automated Testing Based on Imperative Predicates

Nima Dini, Cagdas Yelen, Milos Gligoric and Sarfraz Khurshid
University of Texas at Austin, Austin, TX-78712, USA
nima.dini@utexas.edu, cagdas@utexas.edu, gligoric@utexas.edu, khurshid@utexas.edu

***Abstract*—Bounded exhaustive testing (BET) techniques have been shown to be effective for detecting faults in software. BET techniques based on imperative predicates, enumerate all test inputs up to the given bounds such that each test input satisfies the properties encoded by the predicate. The search space is bounded by the user, who specifies the number of objects of each type and the list of values for each field of each type. To optimize the search, existing techniques detect isomorphic instances and record accessed fields during the execution of a predicate. However, these optimizations are extension-unaware, i.e., they do not speed up the search when the predicate is modified, say due to a fix or additional properties.**

We present a technique, named iGen, that speeds up test generation when imperative predicates are extended. iGen memoizes intermediate results of a test generation and reuses the results in a future search – even when the new search space differs from the old space. We integrated our technique in two BET tools (one for Java and one for Python) and evaluated these implementations with several data structure pairs, including two pairs from the Standard Java Library. Our results show that iGen speeds up test generation by up to 46.59× for the Java tool and up to 49.47× for the Python tool. Additionally, we show that the speedup obtained by iGen increases for larger test instances.

I. Introduction

Software testing is the most common approach in industry for detecting software bugs. Unfortunately, manually writing tests is tedious and time consuming. Automated test generation has been widely studied, over several decades, in both research and industry [1]–[4].

Bounded exhaustive testing (BET) [5]–[12], an approach to automated test generation, has been shown effective for detecting faults in various software applications [5], [6], [11]. BET techniques based on *imperative predicates* generate all test inputs, up to the given bounds, that satisfy a set of *properties* encoded by the imperative predicate [13]–[15].

One of the most widely studied BET techniques based on imperative predicates is Korat [13], [16], which was originally implemented for Java, but variations of Korat have been implemented for other programming languages [7], [9] (including C/C++, C#, CUDA, and Python). Korat takes as input: (1) a boolean predicate, traditionally termed `repOK` [17], which is written in an imperative language, e.g., Java; and (2) bounds on the space to explore. The bounds are given by the user, as the number of objects of each type and the domain, i.e., list of values, for each field of each type. Korat searches through the bounded space by generating *candidate vectors* (i.e., concrete assignment of values to each field for each object) within specified bounds, and executing `repOK` on each candidate; `repOK` returns `true` if the given candidate is *valid*, i.e., satisfies all the properties.

Due to a large number of candidates to explore, even for small bounds, the test generation (and execution of those tests) quickly becomes intractable [7], [8]. To optimize the search, Korat implements backtracking search with *pruning* by monitoring field accesses during the execution of the `repOK` predicate on each enumerated test input. Additionally, Korat detects isomorphic object graphs to optimize the search space.

To further reduce the cost of test generation, researchers and practitioners have explored several algorithms to parallelize BET techniques. Parallel Korat [8] includes a set of algorithms for test generation and execution on map-reduce model; PKorat [9] improves parallel test generation by dynamic partitioning of the search space; and intKorat [7] speeds up test generation by utilizing graphics processing units (GPUs).

Motivation. These recent techniques not only require specialized hardware, but they are also, as the original Korat, *extension-unaware*, i.e., if the imperative predicate is modified by the user, the entire search for finding valid candidate vectors has to be done from scratch.

An imperative predicate can undergo modification for various reasons, including a bug fix (e.g., a fix in `java.util.TreeSet` between Java 6 and Java 7 [18]) or reuse of a predicate to generate test inputs for more complex data structures, e.g., if the original predicate encoded properties for `java.util.TreeMap` (a common form of height-balanced binary search trees), the predicate can be extended to encode properties for `org.apache.commons.collections4.bag.TreeBag` (the standard implementation of a sorted bag).

Technique. We present the first technique, named iGen, that speeds up test generation for *extended imperative predicates*. iGen implements *mixed ranges*, a novel approach for memoizing test inputs generated based on a predicate; mixed ranges track ranges of candidates that need not be explored if the predicate is extended with extra properties on the candidate vectors (e.g., a previously unbalanced tree should be balanced). Our approach also works if the search space is different (e.g., a new field was added in one of the types) from prior searches; we currently do not support a removal of a field or property. Last but not least, our approach is orthogonal to prior efforts to speed up test generation by parallelizing the algorithm [7]–[9].

We implemented the iGen technique by extending two existing BET tools, written in two programming languages:

Korat [16] (for Java) and PyIG [19] (for Python). We named the two new (extension-aware) implementations Korat$_i$ and PyIG$_i$, respectively.

Evaluation. We evaluated the benefits of Korat$_i$ and PyIG$_i$ in several ways. First, we simulated the extension of predicates written for one data structure to predicates written for another data structure, e.g., an extension of Binary Tree to Binary Search Tree to Red Black Tree. The data structures used in this part of our evaluation are commonly used in prior work on bounded exhaustive test generation [7], [8], [10], [13]; we used the exact code, which is available online, as used in prior studies. We also evaluated the benefits of combining parallel generation and iGen and compared it with Parallel Korat. Second, we used two pairs of data structures from popular Java libraries; the first pair illustrates extension due to a bug fix (aforementioned fix in `java.util.TreeSet`), and the second pair illustrates an extension of one data structure to implement another data structure (`org.apache.commons.collections4.bag.TreeBag` uses `java.util.TreeMap`). Third, we compared our technique to an incremental technique for constraints written in Alloy [20]. Finally, we show an application of our technique for speeding up model counting [21], [22].

Results. Our results show that Korat$_i$ and PyIG$_i$ speed up test generation up to 46.59× and 49.47×, respectively. Parallel generation scales almost linearly and compares favorably with Parallel Korat. Also, our results show that iGen provides increasing benefits, unlike Titanium [20], as the size of the generated structures increases. Finally, iGen speeds up an existing model counting technique between 4.37× and 29.97×.

II. Background and Example

This section introduces Korat as a representative BET technique and illustrates the key ideas behind mixed ranges, implemented in Korat$_i$ and PyIG$_i$, through an example.

Korat (extension-unaware technique). Consider using Korat for generating a bounded exhaustive suite of test inputs that are binary search trees with parent pointers (or BSTP for brevity). The user writes the `repOK` method that checks the properties of desired trees, and *finitization* that bounds the search size, i.e., number of nodes in a tree instance. Figure 1 shows the class declaration (BSTP), the `repOK` method, and finitization (`finBinaryTree`). The `repOK` method checks acyclicity along `left` and `right` fields; correctness of the `size` field, which caches the number of nodes reachable from the root node; binary search constraints on the node elements (i.e., for every node the value of the `elem` field is larger than the values of `elem` fields for nodes in a left subtree); and correctness of the `parent` pointers.

Figure 2 shows an example BSTP instance with values 0, 1 and 2. Corresponding candidate vector is shown on the left of the tree. *Field domains* (fd) for size=3 is shown at the top. Field domains specifies that each of the fields `left`, `right`, and `parent` have 4 possible values (i.e., `null` or one of three possible nodes `N0`, `N1`, and `N2`), and the `size` field has only one possible value (i.e., 3). For this size, the exploration

```
1  public class BSTP {
2    Node root;
3    int size;
4    static class Node {
5      Node left, right, parent;
6      int elem;
7    }
8    boolean repOK() {
9      return isAcyclic() && sizeOK() &&
10            ordersOK() && parentsOK();
11   }
12   static IFinitization finBinaryTree(int num) {
13     IFinitization f = FinitizationFactory.create(
       BSTP.class);
14     IObjSet nodes = f.createObjSet(Node.class, num,
       true);
15     f.set("root", nodes);
16     f.set("size", f.createIntSet(num, num));
17     f.set("Node.left", nodes);
18     f.set("Node.right", nodes);
19     f.set("Node.elem", f.createIntSet(0, num-1));
20     f.set("Node.parent", nodes);
21     return f;
22   } ...
23 }
```

Fig. 1: Binary search tree with parent pointers (BSTP), with properties and finitization

fd(BSTP.root) = $[null, N0, N1, N2]$, fd(BSTP.size) = [3]

fd(Node.left) = fd(Node.right) = fd(Node.parent) = fd(BSTP.root)

fd(Node.elem) = [0, 1, 2]

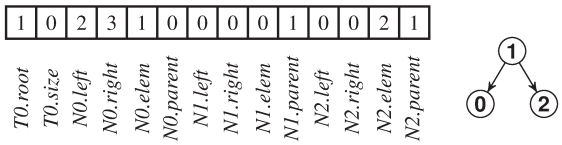

Fig. 2: An example candidate vector with its corresponding tree instance, and the field domains (for size=3)

```
1  public class RBT {
2    static class Node {
3      ...
4      int color; // New field.
5    }
6    boolean repOK() {
7      return isAcyclic() && sizeOK() && ordersOK() &&
8        parentsOK() && colorsOK(); // New property.
9    }
10   static IFinitization finBinaryTree(int num) {
11     ...
12     // Set bounds on the newly defined field.
13     f.set("Node.color", f.createIntSet(0, 1));
14     return f;
15   } ...
16 }
```

Fig. 3: Red-black tree with parent pointers (RBT), with new field, property, and bound, which are highlighted

space has size $4 \cdot 1 \cdot (4 \cdot 4 \cdot 3 \cdot 4)^3 > 2^{24}$. The space grows exponentially. For example, for size 8, there are more than 2^{103} candidate inputs. Korat prunes most of this input space, exploring 2,698,488 candidates, and generating 1430 trees in 1.95 seconds.

iGen (extension-aware technique). Assume the user now wants to implement the data structure to represent height-

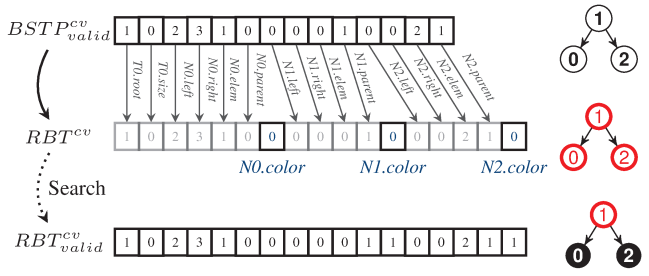

Fig. 4: For size=3, augment a valid candidate vector from the base subject ($BSTP^{cv}_{valid}$) to a candidate in the extended subject (RBT^{cv}) by introducing `color` fields, initialized to their default values, i.e., 0. Next, start bounded exhaustive exploration from RBT^{cv} to find a valid candidate vector for the extended subject (RBT^{cv}_{valid})

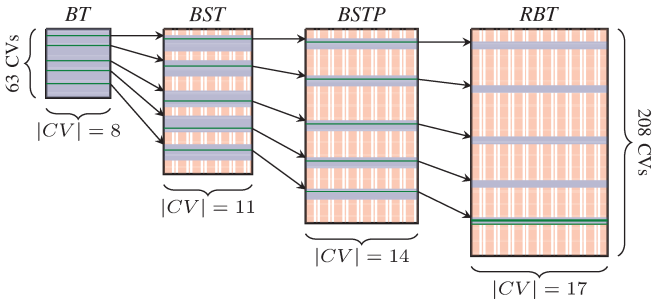

Fig. 5: Visualization of incremental BET for binary tree extension, size=3. The space of candidate vectors explored is colored blue. Green horizontal lines indicate valid candidate vectors, mapped from a base subject to an extended subject. Regions that need not be explored are colored in red and white

balanced binary search trees, specifically *red-black* trees (RBT) [23]. The user could do so by reusing the BSTP class. Figure 3 shows (a part of) the new class declaration (RBT), `repOK` method, and finitization; we highlight new code and do not repeat code from the BSTP class. The user has added one new field (`color`), which represents a node's red or black color. In addition, the user has extended the `repOK` to add a check for coloring constraints, i.e., (1) red nodes cannot have red children, and (2) the number of black nodes on any path from root to a leaf is the same. Given this *extended predicate*, Korat constructs the new search space, which for size=3 is 8× larger (because each node now has the "color" field that has one of two values) than the old search space. The extended candidate vector has 17 elements, i.e., 3 more elements than the base candidate vector. Korat explores the extended space by creating a new *independent* search and generates 3 red-black trees, exploring 208 candidates.

Unlike the extension-unaware BET techniques, Korat$_i$ (i.e., integration of our iGen technique into Korat) generates tests *incrementally* by re-using the base tests. Figure 4 illustrates an example of how Korat$_i$ transforms a base test for size=3 ($BSTP^{cv}_{valid}$) into an extended candidate vector (RBT^{cv}), invokes Korat on it, and generates an extended test for the same size=3 (RBT^{cv}_{valid}).

Figure 5 shows the amount of savings in the state space exploration (the red and white regions), achieved by Korat$_i$ for size=3, when different pairs of base and extended $subject_b \rightarrow subject_e$ subjects are used. Specifically, we show the savings in the explored space for the following extensions: *BT→BST*, *BST→BSTP*, and *BSTP→RBT*. For instance, for *BSTP→RBT*, Korat$_i$ uses the existing suite of 5 base tests for *BSTP*, and generates the suite of all 3 extended tests for *RBT*, by exploring only 35 candidates (compared to 208 that Korat explored), i.e., a savings of 5.94× in the space exploration over the extension-unaware technique.

III. iGen

This section describes our approach for memoizing the Korat search. We describe the key concepts, including mixed ranges, and show the way they enable incremental test generation by building on a Korat-like algorithm.

Definitions. We define a *subject*, e.g., Binary Tree, as a triple:

$$(Classes, Properties, Bounds)$$

where $Classes$ is a set of classes used to define the subject (e.g., `BSTP` and `Node`), $Properties$ is a set of properties that a valid instance of the subject should satisfy (i.e., the `repOK` method), and $Bounds$ specifies the number of instances of each class in $Classes$ and the list of values for each field (i.e., finitization). We define an *instance of a subject* as a *candidate vector*, i.e., a concrete assignment of values to each field from the domain for that field as specified in the $Bounds$.

Further, we define a function $bet(subject, cv, cv')$, which takes as input a subject, and a *range* specified by the starting and ending candidate vectors; the function enumerates all candidate vectors in the given bounds in the given range, and returns a set of all candidate vectors that are valid. If the last two arguments are not given, the function enumerates all candidates within the bounds specified by the subject. We also define a function $nextCv(cv)$, which takes as input a candidate vector and returns a *subsequent candidate vector* in order in which the invocation of the bet function enumerated the instances.

We define a *simple mixed range*, written as $[cv, cv')$, as a pair of candidate vectors such that the first element is a valid candidate and the second element (cv') is the subsequent candidate of the first element (cv), which may be valid or invalid, i.e., $\texttt{repOK}(cv) \wedge cv' = nextCv(cv)$.

Next, we consider a pair of base and extended subjects $subject_b \rightarrow subject_e$ (e.g., $BSTP \rightarrow RBT$) that are *compatible*; we define the conditions when two subjects are compatible later in Section III-A. We introduce a function $extend(cv_b)$ that takes an instance (i.e., candidate vector) of a base subject and creates an instance of the extended subject by augmenting the candidate vector to include new elements (classes and/or fields) and assigning to each new element its first value from the domain for that element as specified with the bounds of the extended subject, i.e., $Bounds_e$ (for an example of how a candidate vector is extended see Figure 4). We say that a *mixed range* is a simple mixed range augmented for $subject_e$, i.e., if $[cv, cv')$ is a simple mixed range, then $[extend(cv), extend(cv'))$ is a mixed range.

```
Input: subject_b - base subject
Input: subject_e - extended subject
 1: function ISCOMPATIBLE(subject_b, subject_e)
 2:     if subject_b.Classes.flds ⊈ subject_e.Classes.flds then:
 3:         return False
 4:     fi
 5:     if subject_b.Properties ⊀ subject_e.Properties then:
 6:         return False
 7:     fi
 8:     commonFields ← subject_e.Classes.flds ∩ subject_b.Classes.flds
 9:     for each field in commonFields do:
10:         if subject_e.Bounds(field) != subject_b.Bounds(field) then:
11:             return False
12:         fi
13:     done
14:     return True
15: end function
```

Fig. 6: Procedure to check compatibility, i.e., if the given subject can be used as the base for the extended subject

Finally, we introduce a function $igen(subject_b, subject_e)$, which generates instances for the extended subject based on previously generated instances of the base subject. Specifically, we define the $igen$ function as:

$$\bigcup_{cv_b \in bet(subject_b)} bet(subject_e, extend(cv_b), extend(nextCv(cv_b)))$$

In other words, $igen$ re-uses the base tests by partitioning the search problem into several (non-overlapping) sub-problems. Conceptually, each sub-problem is simply a ranged search [8] over a mixed range.

We give a brief intuition that $igen$, as defined above, generates the same set of tests as $bet(subject_e)$. First, for every simple mixed range $[cv, cv')$, the augmented candidates ($extend(cv)$ and $extend(cv')$) would be enumerated in that order by $bet(subject_e)$. Second, any valid candidate in $subject_e$ must belong to one of the mixed ranges. Assume that this does not hold and there is a valid candidate that does not belong to one of the mixed ranges. If the candidate is valid, it means that its backbone structure, i.e., properties for the base subject, is valid. If that was the case, based on the definition of a simple mixed range, the candidate would be the first element for one of the simple ranges, which contradicts our assumption that it does not belong to any mixed range.

A. Checking Applicability

Our technique supports incremental generation if the base and extended subjects are *compatible*, defined by the following conditions: (1) the set of fields in the extended subject is a super set of the set of fields for the base subject, (2) the set of properties for the extended subject is stronger than the set of properties for the base subject, and (3) the bounds on the input fields for common fields for the subjects remains the same. We discuss other code transformations, which we currently do not support, in Section V.

Figure 6 illustrates the algorithm that checks if the incremental generation is feasible. For a previously run subject with iGen, we check if it can be used as the base subject for extended; we check the set of fields on Line 2, the set of properties on Line 5, and the bounds on common input fields on Line 10. Note that the relevant set of fields only includes fields that are used in the finitization; the existence of other fields has no impact on search and therefore they are ignored by our algorithm. Similarly, we can ignore fields whose domain has only a single value because their existence does not impact the test generation. Note that the new fields in the extended subject could be added in classes for the base subject or they could be in a new set of classes.

```
Input: subject_b - base subject
Input: subject_e - extended subject
 1: function IGEN(subject_b, subject_e)
 2:     if not ISCOMPATIBLE(subject_b, subject_e) then:
 3:         return BET(subject_e)
 4:     fi
 5:     tests_e ← ∅
 6:     tests_b ← LOADTESTS(subject_b)
 7:     for each test in tests_b do:
 8:         cv ← BUILDCANDIDATEVECTOR(subject_b, test)
 9:         cv' ← NEXTCV(subject_b, cv)
10:         startCV ← EXTEND(subject_b, subject_e, cv)
11:         endCV ← EXTEND(subject_b, subject_e, cv')
12:         mixedRange ← tuple(startCV, endCV)
13:         testsFound ← BET(subject_e, mixedRange)
14:         ADDALL(tests_e, testsFound)
15:     done
16:     return tests_e
17: end function
```

Fig. 7: Procedure for incremental test generation for the given extended subject by reusing tests from the base subject

Regarding the set of properties, we detect condition strengthening by statically analyzing `repOK` method. Specifically, we look for the cases that are a conjunction of new properties with already existing properties. The conjunction can be explicit (e.g., `isAcyclic() && sizeOK() && ordersOK()`) or it can be implicit by extending the code of the existing functions. It is important that the order of field accesses, for the fields used in finitization, does not change.

B. Incremental Generation

Figure 7 shows the incremental generation algorithm. First, we identify if the incremental technique is applicable for the given pair of base and extended subjects (Lines 2-4); if not, no re-use is possible, and the traditional extension-unaware BET will be used to generate tests (Line 3) for the extended subject. Otherwise, we initialize the list of generated tests to an empty set (Line 5), and retrieve the list of previously generated tests for the base subject (Line 6). We then iterate over each test of the base subject (Lines 7-15). In each iteration, we load the candidate vector for the base test (we know that the candidate vector was valid) and find its subsequent candidate vector for the base subject, which can be valid or invalid (Line 9). Next (Lines 10 and 11), we augment these two consecutive candidate vectors to the candidate vectors for the extended subject by adding new fields and setting them to their initial values (i.e., zeros). These two extended candidate vectors form a mixed range of candidates to be explored for the extended subject

(Line 13). All tests (i.e., valid candidates) found in this range are added to the set of tests for the extended subject (Line 14); the union of results for all ranges is the result of the function. Recall the functions `nextCv`, `extend`, and *bet* in the first part of Section III, e.g., *bet* explores the given bounds, to find all valid candidates in the given range of candidates.

IV. Evaluation

To evaluate the benefits of incremental generation with iGen, we implemented our technique in two tools, one for Java and one for Python, by extending the existing publicly available tools, namely Korat (for Java) and PyIG (for Python); we call our prototype implementations Korat$_i$ and PyIG$_i$, respectively. We answer the following research questions:

- **RQ1:** What is the amount of savings in state space exploration, when a predicate is extended, due to the iGen technique?
- **RQ2:** What is the speedup, in terms of test generation time, due to the iGen technique?
- **RQ3:** What is the speedup trend due to the iGen technique as the size of generated instances increases, and how does this speedup trend compare to the trend of Titanium [20], a recent incremental technique for constraints written in the Alloy specification language?
- **RQ4:** What is the benefit of using the iGen technique in a parallel setting, and how does it compare to prior work on Parallel Korat [8], namely SEQ-ON algorithm?
- **RQ5:** What is the reduction in the amount of memory used due to the iGen technique?
- **RQ6:** What is the benefit of using the iGen technique for classes in Standard Java Libraries?
- **RQ7:** What benefits does the iGen technique provide when applied in the context of recent work [24] that used Korat as a backend model counter?

Execution platform: We obtained all data on a machine with 4-core 2.2 GHz Intel Xeon CPU and 16GB of RAM, running Ubuntu 16.04 LTS. We used Oracle Java 1.8.0_121 and PyPy 5.8.0, for Java and Python executions, respectively. For each run, we limited the heap size to 8GB.

We first discuss the subjects and subject pairs used in our study and then answer our research questions.

A. Subjects and Subject Pairs

To evaluate iGen, we consider several pairs of subjects: $subject_b \rightarrow subject_e$. We split all pairs of subjects into three groups. The first group (named GroupT) contains various tree data structures, the second group (named GroupL) contains various list data structures, and the third group (named GroupJL) contains data structures from popular Java libraries.

Table I shows the groups. The first column of the table shows the name and the acronym for each data structure. The second column shows the set of fields used in the finitization for the structure; the fields may belong to various classes. For example, for `BT` there are 4 fields used in the finitization; these fields belong to two classes: `BT` and `Node`. Finally, the third column shows the set of properties that each instance has to satisfy, i.e., the `repOK` method. For example, for `BT` there should be no cycles (`isAcyclic`) and `size` should match the number of nodes in the tree (`sizeOK`). All subjects in the first two groups are taken from the existing online source code repositories [16], [19], and the same subjects were used in prior work on automated test generation (e.g., [8]).

GroupT includes (1) binary tree (`BT`), (2) binary search tree (`BST`), which has the same properties as `BT` and additional properties on the order of elements, (3) binary search tree where each node keeps a pointer to its parent (`BSTP`), which has additional properties on these pointers, and (4) red black tree (`RBT`), which extends the properties of `BSTP` with properties on the color of each node. Note that `RBT` is actually `java.util.TreeMap` class, but we use the `RBT` acronym to better reflect the underlying data structure.

GroupL includes (1) doubly linked list (`DLL`), (2) doubly linked list that has unique elements (`DLLU`), and (3) sorted doubly linked list with unique elements (`DLLS`).

GroupJL includes (1) `TreeMap` which is a red-black tree navigable map implementation available in the Standard Java Library, (2) `TreeSet`$_6$ which is a navigable set (available in Java 6 and earlier versions) implemented based on `TreeMap` (this version allowed insertion of `null` values in an empty set), (3) `TreeSet`$_7$ which is a navigable set available in Java 7 and later versions (this version of the set does not allow insertion of `null` values in an empty set). (4) `TreeBag` which is the standard implementation of a sorted bag, using a `TreeMap`, available in the Apache commons.

To answer the research questions, we consider one pair of subjects at a time. We evaluated our technique on all pairs of adjacent subjects for GroupT and GroupL. Specifically, we evaluated three pairs for GroupT: *BT*→*BST*, *BST*→*BSTP*, and *BSTP*→*RBT*, and two pairs for GroupL: *DLL*→*DLLU* and *DLLU*→*DLLS*. Additionally, we evaluate two pairs in GroupJL: TreeSet$_6$→TreeSet$_7$ (which illustrates extension of an imperative predicate due to change in properties of a data structure) and TreeMap→TreeBag (which illustrates extension of a an imperative predicate due to a reuse of one data structure to implement another data structure). Note that the first pair in this group is a trivial extension in the predicate.

B. Results and Answers

1) RQ1: Savings in the State Space Exploration: Table II shows the number of candidate vectors explored by the extension-unaware BET techniques (i.e., Korat and PyIG), for sizes from 4 to 10, for each subject. Although Korat and PyIG do not execute the exact same algorithm, the number of generated instances is the same for our examples. One key observation from this table is that different properties affect the number of candidates explored by BET differently. This case is specially evident for larger sizes. For instance, for size=10, there is a 155 million difference in the number of candidates explored for *BT*→*BST* (due to addition of `ordersOK()` constraint), while this number is only 2 million for *BST*→*BSTP*, when `parentsOK()` constraint is added. Further, note that the number of explored candidates for *DLLU* and *DLLS* is the same, for all sizes, as there is no new field introduced by *DLLS*,

TABLE I: Subjects Used in Evaluation with their Class Fields and `repOK`

	Subjects	Class Fields	repOK Method
GroupT	BinaryTree (*BT*)	fields(*BT*) ≡ {`root`, `size`, `Node.left`, `Node.right`}	isBT() ≡ isAcyclic() && sizeOK()
	BinarySearchTree (*BST*)	fields(*BST*) ≡ fields(*BT*) ∪ {`Node.key`}	isBST() ≡ isBT() && ordersOK()
	BST w/ parent pointers (*BSTP*)	fields(*BSTP*) ≡ fields(*BST*) ∪ {`Node.parent`}	isBSTP() ≡ isBST() && parentsOK()
	RedBlackTree (*RBT*)	fields(*RBT*) ≡ fields(*BSTP*) ∪ {`Node.color`}	isRBT() ≡ isBSTP() && colorsOK()
GroupL	DoublyLinkedList (*DLL*)	fields(*DLL*) ≡ {`head`, `size`, `Entry.prev`, `Entry.next`}	isDLL() ≡ isStructOK()
	DLL w/ unique elems (*DLLU*)	fields(*DLLU*) ≡ fields(*DLL*) ∪ {`Entry.value`}	isDLLU() ≡ isDLL() && elementsUnique()
	DLLU w/ sorted elems (*DLLS*)	fields(*DLLS*) ≡ fields(*DLLU*)	isDLLS() ≡ isDLLU() && elementsSorted()
GroupJL	JDK TreeMap (TreeMap)	fields(TreeMap) ≡ fields(*RBT*)	isTreeMap()
	JDK 6 TreeSet (TreeSet_6)	fields(TreeSet_6) ≡ fields(*RBT*)	isTreeSet$_6$()
	JDK 7 TreeSet (TreeSet_7)	fields(TreeSet_7) ≡ fields(TreeSet_6)	isTreeSet$_7$() ≡ isTreeSet$_6$() && noNullValues()
	Apache Commons TreeBag	fields(TreeBag) ≡ fields(TreeMap) ∪ {`TreeBag.size`}	isTreeBag() ≡ isTreeMap() && bagSizeOK()

TABLE II: Number of Candidates Explored by Extension-Unaware Techniques

Size	*BT*	*BST*	*BSTP*	*RBT*	*DLL*	*DLLU*	*DLLS*
4	245	875	1099	1251	39	162	162
5	947	6155	7205	7989	57	881	881
6	3653	45,233	49,985	54,117	78	6263	6263
7	14,092	340,990	362,011	384,231	102	52,062	52,062
8	54,418	2,606,968	2,698,488	2,820,170	129	485,096	485,096
9	210,444	20,086,300	20,480,122	21,157,272	159	4,988,399	4,988,399
10	815,100	155,455,872	157,135,472	160,957,128	192	56,117,901	56,117,901

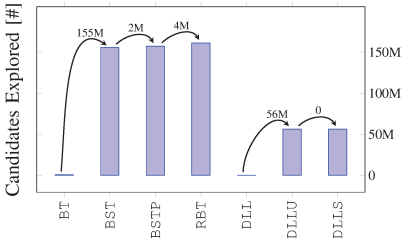

Fig. 8: Difference in number of candidates explored by BET (Size=10)

TABLE III: Percent [%] of Candidates Pruned by iGen for Various $subject_b \rightarrow subject_e$ Pairs; Percent is Computed out of All Candidates That are Explored by Extension-Unaware Techniques for $subject_e$ Subject (Table II)

Size	iGen savings in explored space [%]				
	BT→*BST*	*BST*→*BSTP*	*BSTP*→*RBT*	*DLL*→*DLLU*	*DLLU*→*DLLS*
4	26.40	78.34	86.73	23.46	85.19
5	14.70	84.84	89.66	6.36	86.38
6	7.78	90.23	92.12	1.23	88.50
7	4.01	94.07	94.11	0.19	90.32
8	2.03	96.56	95.63	0.03	91.69
9	1.02	98.05	96.78	< 0.01	92.73
10	0.51	98.92	97.62	< 0.01	93.53

and the new property (`elementsSorted()` constraint) does not change the order of field access in the BET search.

Figure 8 shows the difference in the number of candidate vectors explored by extension-unaware BET tools for size=10, based on the raw numbers shown in the last row of Table II (highlighted in gray). Note that for the two subject pairs *BT*→*BST* and *DLL*→*DLLU*, there is a large difference in number of explored candidates. This is because the search space is considerably larger for the extended subjects *BST* and *DLLU*. Therefore, we expect iGen, which is an incremental technique, to provide smaller speedup for these two subject pairs (*BT*→*BST* and *DLL*→*DLLU*). However, for the other three subject pairs, there is a smaller difference in the number of explored candidates between base subject and extended subject; therefore, we expect iGen to provide higher speedup.

Table III shows iGen savings in the state space exploration, over the traditional extension-unaware BET. For each subject pair ($subject_b \rightarrow subject_e$), we first applied BET on the extended subject and collected the number of explored candidates (N_{bet}). Next, we applied iGen on the pair, and we measured only the number of candidates explored by iGen (N_{igen}) for the $subject_e$. Then, we computed savings in explored space as (1 - N_{igen}/ N_{bet}) · 100. For instance, for *BST*→*BSTP* (size=10), BET explored 157,135,472 candidates, while iGen explored only 1,696,396 candidates, i.e., savings of 98.92%.

Further, as we expected based on our reasoning of numbers in Figure 8, iGen provided higher savings for three subject pairs *BST*→*BSTP*, *BSTP*→*RBT*, and *DLLU*→*DLLS*, as there was a small amount of incremental solving to be done, while solving the extended constraints from scratch (using BET) was prohibitively expensive. In addition, for these three subject pairs we observed an increasing trend in the amount of savings, as the size of the generated instances increased. For the remaining two subject pairs, *BT*→*BST* and *DLL*→*DLLU*, the achieved savings decreased for larger instances, as the incremental solving required became considerably large.

2) RQ2: Speedup in Test Generation Time: For each subject pair ($subject_b \rightarrow subject_e$), we first measured test generation time for the extended subject, using extension-unaware BET tools (T_{bet}). Next, we measured incremental test generation (iGen) time for the adjacent subject, using our implementations (T_{igen}). We then calculated the speedup based on the recorded execution times as T_{bet}/ T_{igen}.

Table IV shows the results. For RQ2 (this question), we only focus on the top two parts: Korat_i\Korat and PyIG_i\PyIG. The left half of the table shows the test generation time using extension-unaware BET tools (Korat and PyIG) for each extended subject. For any cell on the left half, there is a corresponding cell on the right half, which shows the speedup in execution time using our implementations. The reported execution time is computed as an average value of 3 runs.

TABLE IV: Extension-Aware Test Generation Speedup over Extension-Unaware Techniques

	Size	Extension-Unaware Generation Time [ms]					Extension-Aware Generation Speedup [×]				
		BST	*BSTP*	*RBT*	*DLLU*	*DLLS*	*BT→BST*	*BST→BSTP*	*BSTP→RBT*	*DLL→DLLU*	*DLLU→DLLS*
Korat \ Korat$_i$	4	144	158	174	126	155	1.00	1.05	1.07	0.95	1.14
	5	164	171	194	135	197	1.01	1.04	1.10	0.98	1.46
	6	236	242	265	152	194	1.04	1.34	1.29	0.96	1.31
	7	420	456	500	219	287	1.01	1.95	1.28	0.98	1.68
	8	1697	1918	2197	485	561	1.04	3.71	3.22	0.99	1.68
	9	11,434	12,074	13,669	3434	2882	1.02	18.02	9.67	1.01	5.98
	10	93,929	97,281	106,800	39,540	29,026	1.04	46.59	16.82	1.01	10.45
PyIG \ PyIG$_i$	4	167	242	248	49	42	1.14	2.66	3.35	1.26	7.00
	5	358	480	452	188	155	1.09	1.88	1.57	1.11	2.54
	6	724	811	1357	286	297	1.05	1.57	2.24	1.04	1.66
	7	3190	3826	5949	625	623	1.01	4.50	3.95	1.01	1.53
	8	22,812	27,743	28,294	4051	4310	1.02	11.37	6.39	1.06	3.79
	9	196,032	217,523	233,054	46,597	45,657	1.03	29.94	9.09	1.00	4.84
	10	1,605,420	1,673,692	3,111,485	521,909	534,937	1.01	49.47	25.17	1.01	8.89
Alloy \ Titanium	4	371	405	580	486	418	1.74	2.76	4.79	1.81	2.99
	5	660	648	778	915	583	1.64	1.37	4.27	1.36	2.80
	6	1560	1638	941	2566	688	1.58	1.41	2.14	1.19	2.51
	7	3796	3669	1333	12,140	861	1.13	1.51	1.37	1.01	2.53
	8	9194	10,661	2340	117,726	815	1.21	1.04	1.65	1.00	2.14
	9	43,167	55,255	4230	memory	958	1.08	1.15	1.33	n/a	n/a
	10	189,216	222,161	6501	memory	1397	1.01	1.11	1.18	n/a	n/a

For instance, Korat and PyIG required 97.3 and 1673.7 seconds, respectively, to generate all *BSTP* instances with 10 nodes, while Korat$_i$ and PyIG$_i$ only needed 2.1 and 33.8 seconds, to generate the same instances incrementally on top of existing *BST* tests, i.e., 46.59× and 49.47× speedup in test generation time, respectively.

As we expected based on our reasoning of numbers in Figure 8 and Table III, our implementations did not provide considerable speedup for the two subject pairs *BT→BST* and *DLL→DLLU*. For smaller sizes, we even observed minor slowdowns, e.g., sizes 4-8 for *DLL→DLLU*, because the slight overhead of iGen (which includes loading tests previously generated for the base subject) did outweigh the savings achieved. Further, we observed that Korat$_i$ and PyIG$_i$ show similar behaviors for most subject pairs and sizes, and their speedup is higher for larger instances.

3) RQ3: Trend of iGen Speedup vs. Trend of Titanium Speedup: Titanium [20] is a recent technique to efficiently analyze *extended* Alloy specifications [25]. Unlike iGen, which introduced mixed ranges for BET techniques based on imperative predicates, Titanium achieves speedup by tightening input bounds to speedup the underlying constraint solving; the tightening (when exploring $subject_e$) is based on the knowledge from the execution on $subject_b$.

For our experiment, we implemented each subject from GroupT and GroupL in Alloy. For each subject, we only generated non-isomorphic instances by enforcing a linear order on node instances [26], in Alloy code. This was required as both BET techniques (Korat and PyIG), by construction, only generate non-isomorphic instances.

Next, we took Titanium, which is publicly available [27]. For each subject pair ($subject_b \rightarrow subject_e$), we measured (1) the time needed by Alloy Analyzer to generate instances for the extended subject (T_{alloy}), and (2) the time spent by Titanium for efficient solving of the $subject_e$ given the $subject_b$ ($T_{titanium}$). The last/third segment of Table IV shows the speedup achieved by Titanium over the default Alloy Analyzer; we measured the speedup the same way as for iGen, i.e., T_{alloy}/ $T_{titanium}$.

As expected, Titanium achieved speedup to generate tests for all subjects. For *DLLU* (sizes 9 and 10), Titanium encountered a `java.lang.OutOfMemoryError`, while running the Alloy Analyzer on the $subject_b$, as the 8GB memory limit was exceeded. Hence, we do not report numbers for columns *DLL→DLLU* and *DLLU→DLLS* for these two sizes.

In most cases, we observed higher speedup for iGen compared to Titanium, e.g., for *BST→BSTP* (size=10) Korat$_i$ and PyIG$_i$ provided 46.59× and 49.47× speedup respectively, compared to the 1.11× speedup for Titanium; however, as these two techniques are inherently different (one is for declarative languages using backend SAT solving, while the other uses bounded exhaustive testing based on imperative predicates), we do not account this greater speedup as being superior. However, we do still compare the *trend of speedup* across the two techniques. Unlike iGen, we observed that Titanium shows a descending speedup trend as the size of generated instances increases, and for larger sizes, e.g., 10, the speedup seems to be converging towards 1×. Speedup trends are similar for two subject pairs – *BT→BST* and *DLL→DLLU* – due to the reasons discussed in Section IV-B1 and Figure 8.

4) RQ4: Potential Benefits of Parallel Generation: iGen with mixed ranges is embarrassingly parallel. Hence, we evaluated potential benefits of parallel generation by simulating parallel runs: we run one worker at a time on a single machine and compute maximum, and average running time across all workers. Each worker takes the same number of valid instances for base subject, and consequently, creates the same number of mixed ranges. Note that there is no overlap among explorations done by individual workers. Further, there are no inter-worker communications after the work has been distributed, and therefore, similar results are expected in a distributed setting.

TABLE V: Parallel Korat$_i$ ($subject_b \rightarrow subject_e$) and SEQ-ON ($subject_e$) Speedups in Worker Execution Time, Compared to a Sequential Run of Korat (Size=10)

	Workers	Speedup over Korat [×]							
		BT→BST		*BST→BSTP*		*BSTP→RBT*		*DLLU→DLLS*	
		avg	*min*	*avg*	*min*	*avg*	*min*	*avg*	*min*
Korat$_i$	2	2.0	2.0	79.3	78.3	31.7	30.3	19.0	18.6
	4	4.1	4.0	110.4	108.5	53.1	50.7	29.4	27.6
	8	8.1	8.0	146.2	136.6	83.9	77.1	46.9	44.5
	16	15.3	14.8	172.0	163.2	119.7	104.6	62.9	58.3
	32	28.7	25.5	216.0	203.1	161.0	124.5	79.0	72.2
	64	51.2	47.0	268.3	209.7	194.3	161.6	88.2	80.2
SEQ-ON	2	2.0	2.0	2.0	2.0	2.0	1.9	1.9	1.9
	4	4.0	4.0	4.0	4.0	4.0	3.9	3.8	3.7
	8	7.9	7.8	7.9	7.8	7.8	7.7	7.3	7.0
	16	15.3	14.4	14.9	14.2	15.0	14.0	13.1	12.8
	32	28.6	27.5	27.3	25.7	27.8	26.2	23.5	22.6
	64	51.3	48.8	46.8	41.3	47.6	43.9	39.9	37.3

Next, we implemented the equi-distancing for SEQ-ON algorithm [8] to simulate parallelizing Korat runs across workers. Note that unlike Korat$_i$ (and generally iGen), the SEQ-ON algorithm is extension-unaware; it uses key intermediate results stored during the first execution of Korat, to distribute the work and speed up *future* executions for *the same* constraint solving problem. Similar to the evaluation method in prior work [8], we first found the equi-distant candidate vectors using the equi-distancing algorithm (for the extended subject), and then used those candidates to evenly distribute the work. We did not account for the resource allocation time, as prior work showed it would be near constant across the workers [8]. Further, we did not consider the time spent to distribute inputs among the workers.

Table V shows the average and minimum speedup in execution time, across the workers, that the two parallel techniques Korat$_i$ and SEQ-ON, achieved over Korat. Specifically, for each subject pair, we measured the time needed to run: (1) Korat on the $subject_e$ (T_{korat}), (2) parallel Korat$_i$ on the $subject_b \rightarrow subject_e$ pair (T_{Korat_i}), and (3) SEQ-ON on the $subject_e$ (T_{SEQ-ON}). The top half of the table shows numbers for T_{korat}/ T_{Korat_i}, and the bottom half shows numbers for T_{korat}/ T_{SEQ-ON}. We chose size=10 for all subjects, and considered a range of 2 to 64 workers.

As expected, both parallel techniques, achieved considerable speedup over running Korat sequentially. For instance, parallel Korat$_i$ achieved up to 268.3× speedup on average worker time, over a sequential Korat run. For the 3 subject pairs, *BST→BSTP*, *BSTP→RBT*, and *DLLU→DLLS*, even for small number of workers, e.g., two, the speedup was substantial; 79.3×, 31.7×, and 19.0×, respectively.

We observed that parallel Korat$_i$ (extension-aware) outperforms SEQ-ON (extension-unaware) in most cases. For instance, for *BT→BST* with 2 workers, Korat$_i$ achieved on average the speedup of 79.3×, while SEQ-ON achieved the speedup of 2.0×. As we expected, based on our reasoning of numbers in Figure 8, for *BT→BST*, with 32 and 64 workers, we observed minor slowdown (less that 8%) compared to SEQ-ON.

Note that there is no column for *DLL→DLLU*, because the base subject, i.e., *DLL*, had only one valid instance (i.e., one range), a case that would not be impacted by a parallel run. For other pairs, the base subject had at least 64 valid instances, facilitating the requirement for our parallel experiment.

5) RQ5: Reduction in Memory Usage: Table to the right, shows the memory usage (in megabytes), for end-to-end runs of Korat and Korat$_i$. We used `java.lang.management` standard Java library to measure memory usage. Specifically, we manually invoked JVM's *garbage collection* (`GC`) in code (`System.gc()`), right before test generation, and measured memory usage right after test generation. We considered instances of size=6, as this was the largest size that did not require a `GC` invocation during each execution. Our preliminary evaluation shows Korat$_i$ requires less memory for all subjects (up to 3.77× for *BST→BSTP*) compared to Korat. We expected this memory savings, as Korat$_i$ explores fewer candidates (Table III), meaning it runs `repOK` fewer times; each `repOK` invocation allocates objects, e.g., work-list and visited-set. Further, for *BT→BST* and *DLL→DLLU*, we expected smaller savings as there is a larger extension between the two subject pairs (Figure 8).

		Memory Usage [MB]
Korat	*BST*	28.83
	BSTP	32.86
	RBT	38.23
	DLLU	8.70
	DLLS	8.71
Korat$_i$	*BT→BST*	27.49
	BST→BSTP	8.72
	BSTP→RBT	11.38
	DLL→DLLU	8.69
	DLLU→DLLS	7.35

6) RQ6: Applicability to Data Structures in Standard Java Libraries: We report the results for subject pairs in GroupJL. Table VI shows the results for TreeMap→TreeBag. We first describe the reason for two finitizations for this pair. `TreeBag`, as mentioned earlier, implements a sorted bag (e.g., {"a": 2, "b": 1, "c": 2}), using a `TreeMap`. Namely, `TreeMap` keeps a mapping from an element to the count of that element in the bag. As the count for any element can be arbitrarily large, we chose two different values for the max count of each element: 2 for Finitization I and 3 for Finitization II.

The first column in Table VI shows size for the underlying map in `TreeBag`. Columns two and three show search time and number of explored candidate vectors for Korat, respectively. Finally, columns four and five show speedup (time), and savings on the number of explored candidate vectors (percent) for Korat$_i$. We can observe increasing speedup for both finitizations although speedup for the second finitization is smaller due to a larger space of the extended subject. Clearly, choosing a larger value for the max count would further increase the search space for Korat$_i$ and reduce the achieved speedup.

We also briefly discuss the second pair in the GroupJL: TreeSet$_6$→TreeSet$_7$. This is an example when the semantics of a data structure (in the Standard Java Library) has changed: having a set with a single element that has `null` value was allowed in earlier versions of Java. The extension in `repOK` simply checks that a set with a single value cannot have `null` element. Interestingly, even in this simple case Korat would still re-explore the entire search space only to obtain the same result as in the run for the base subject. On the other hand, this

TABLE VI: Test Generation for TreeMap→TreeBag

	Size	Korat		Korat$_i$	
		Time [ms]	Explored [#]	Speedup [×]	Explored [%]
Finitization I	4	341	2213	1.04	65.66
	5	365	12,399	1.05	74.82
	6	488	72,633	1.10	85.93
	7	877	478,964	1.66	91.59
	8	2632	3,348,532	3.47	95.11
	9	14,834	24,328,614	8.01	97.18
	10	111,324	181,333,796	16.65	98.24
	11	929,518	1,371,394,558	26.78	98.86
Finitization II	4	369	7605	1.02	19.11
	5	542	48,393	1.04	19.18
	6	868	259,233	1.15	24.08
	7	2599	1,625,109	1.16	27.00
	8	13,976	10,533,044	1.17	30.26
	9	96,962	70,467,428	1.23	33.55
	10	770,904	508,222,956	1.24	35.05

TABLE VII: Using Korat [24] and Korat$_i$ as a Backend Model Counter to Solve 100 Heap-PC Problems per Subject

Subject	Korat [24]		Korat$_i$	
	Time [ms]	Explored [#]	Speedup [×]	Space Saved [%]
DLL	2550	3,233,440	4.37	88.09
HBST	3287	3,716,300	14.93	98.98
RBT	7283	7,056,576	27.44	99.00
SLL	7373	18,305,148	29.97	98.98

is an ideal case for Korat$_i$, which explores no new candidates. As the number of explored instances with Korat$_i$ is equal to the number of valid instances obtained for the base subject (and the overhead of Korat$_i$ is negligible), we do not show the results in a table; speedup goes from 1.05× for size 4 up to 320.15× for size 10.

7) RQ7: Application to Model Counting: Recent work [21], [22] used Korat as a backend model counter, to find the number of solutions for a constraint derived from path conditions that occur in symbolic execution. Specifically, to solve problems modeled as `repOK() && heap-PC`, where the term `heap-PC` is used to emphasize that the nature of the constraints are on the fields of heap allocated objects.

Korat-API [24] is a recent framework that systematically applied Korat to solve model counting problems for 4 data structure subjects with size=10 nodes. For each subject, 100 path conditions were generated, that all shared the same `repOK()`, but had different `heap-PC` conditions. We used the same publicly available Java subjects [28], to evaluate how well Korat$_i$ performs compared to Korat.

Table VII summarizes the results. The first column shows the acronyms of subjects used in [24], namely: *DLL* for doubly linked list, *HBST* for height-balanced binary search tree, *RBT* for red-black tree, and *SLL* for singly linked list. The subsequent columns show the average values to solve 100 `heap-PC` problems. Specifically, Columns 2 and 3 show the average execution time and average number of explored candidates for Korat, respectively. Column 4 shows the average speedup in the execution time Korat$_i$ achieved over Korat. Column 5 shows the average percent of explored space saved when using Korat$_i$ compared to Korat.

Similar execution times (Column 2) were reported in Table 4 of [24]; our numbers are slightly smaller, as we used the vanilla Korat which does not have the dynamic compilation overhead of Korat-API, and we used a different execution platform. We recalculated the base execution times for Korat on our execution platform, to accurately measure the Korat$_i$ speedup. As shown, Korat$_i$ outperformed Korat for all 4 subjects, ranging from 4.37× for *DLL*, to 29.97× for *SLL*. We expected the speedup, as unlike Korat that needs to solve `repOK()` for each problem, Korat$_i$ can solve `repOK()` once, and re-use the results to incrementally solve future path condition problems where `repOK()` stays the same and `heap-PC` changes.

V. Discussion

Limitations. The subjects used in our evaluation may not be representative. We used subjects distributed with Korat [16] that have been used independently in many prior studies on BET [8]–[10], [13]. Moreover, data structures that we used are the backbone of many complex data structures, e.g., DOMs, XML schema, ASTs [6], [11], [22]. Additionally, we evaluated Korat$_i$ on standard data structures from Java libraries.

Our prototype tools and our scripts for processing the results may contain bugs. We built our prototype on top of Korat and PyIG that are thoroughly tested and used differential testing (between Korat$_i$, PyIG$_i$, Korat, and Alloy) for test generation.

Code transformations. Our work focuses, as discussed in Section III-A, on cases that a developer starts from a weaker `repOK` and then strengthens it to generate instances of a more complex data structure. Basically, we naturally cover cases when one uses an existing data structure to implement another data structure, e.g., using TreeMap to implement TreeBag. One can also imagine that a developer may go too far (with strengthening) and then has to revert some of the properties. While the latter case could happen in practice, it is somewhat less interesting algorithmically; to support backward changes, we could memoize valid instances at various places during the execution of a `repOK` and if some properties are removed, we can simply return the memoized valid instances captured prior to the removed properties. Other transformations to code, e.g., refactorings [29]–[32], adding a field, or removing a field, can also impact the generated instances. iGen already supports cases when a field is added or removed. We plan to explore other potential transformations of properties in the future.

Benefits. Clearly, the benefits of iGen, like any incremental technique, depend on the differences between the base subject and extended subject; if the candidate space is substantially larger for the extended subject, the speedup is limited. Our evaluation showed both cases when iGen provides substantial or limited speedup. Given that the overhead of iGen is negligible, it should be used as the default test generation engine.

Future work. iGen is the first system to introduce incremental constraint solving for Java (Python) constraints. While incremental solving using SAT is commonplace, its use requires translating imperative code to declarative propositional logic formulas, which has high overhead and does not scale. Incremental solving for Java can substantially enhance various constraint-based analyses, including test generation (this paper's focus), symbolic execution, synthesis, and repair.

VI. Related Work

BET has its basis in the spirit of model checking [33], which introduced the idea of exhaustive exploration of large state spaces, and various optimization techniques for effective pruning. The specific form of BET using imperative predicates, which is our focus, is one of many test generation techniques [34]. Our focus in this section is on most closely related work to our approach.

Parallel Korat Misailovic et al. [8] introduced the first approach for parallel test generation and execution using Korat, which we discussed in prior sections. PKorat [9] introduced an alternative parallel approach based on a work list that consists of work items that Korat search must explore. Dini et al. [10], [35] built on Parallel Korat [8] and introduced *invalid ranges* that optimize re-execution of the Korat search on the *same* search problem by skipping known ranges of consecutive invalid candidates. Invalid ranges provide the basis for our work. The key difference is that previous work optimized Korat when the exact same search problem is re-solved whereas iGen allows memoizing and re-using intermediate results of the BET search even when the search space is changed. Recent work [7] utilized GPUs to speed up test generation.

Ranged analysis was also used for ranged symbolic execution [36] using KLEE [37], ranged model checking using JPF [36], [38], and ranged declarative constraint solving using Alloy [25], [39]. Qiu [40] defined feasible ranges, i.e., sequences of feasible paths, for Symbolic PathFinder to summarize path condition satisfaction to enable memoization.

Generation of complex data structures has received much attention for BET. TestEra [15] was among the first to generate tests up to the given bounds based on declarative predicates written as Alloy formulas. Korat [13] enabled the user to write the predicates as executable checks in an imperative language. Generalized symbolic execution [41] combined lazy initialization of reference fields with symbolic analysis of primitive fields to more efficiently handle predicates that contain constraints on primitives using off-the-shelf decision procedures, e.g., SMT solvers [42]. UDITA [11] supports both declarative predicates and imperative generators. HyTeK [43] allowed predicates to be written in a combination of declarative predicates and imperative checks. More recently, Kuraj et al. [5] introduced SciFe that uses an algebra of enumerators to make the generation incremental and parallelizable. iGen focuses on re-using the key results of the search during execution of properties to enhance generation of structurally complex tests as the imperative predicates are extended.

Incremental analysis for systematic bug finding was used for efficient generation of complex inputs [44], symbolic execution [45]–[47], and model checking [48]–[51]. Uzuncaova [44] introduced a technique based on Alloy for incremental test generation for software product lines where each product consists of a base feature and may have some combination of additional features; this technique employed off-the-shelf SAT and SMT solvers. Bagheri and Malek's recent work addressed evolving Alloy specifications by using the set of solutions enumerated by SAT in its previous run to refine the search space bounds for SAT's next run [20]. In contrast, iGen integrates memoization *within* the solver – for imperative constraints.

KLEE [37] introduced memoization of constraint solving results during one run of symbolic execution of the program. Green [46] enabled memoization across different runs, even for different programs. Memoise [52] introduced a trie structure to efficiently summarize symbolic execution results for re-use when the program evolves. Such re-uses of constraint solving results has a basic difference from our approach: iGen memoizes key *intermediate* results during constraint solving, not just its end result(s).

Test-suite augmentation techniques generate new tests that target changed code. Santelices et al. [53] introduced MaTRIX, an augmentation technique based on dependency analysis and symbolic execution. Xu et al. [54] explored cost and effectiveness of several augmentation techniques. Qi et al. [55] introduced a technique based on dynamic symbolic execution. Kim et al. [56] proposed a hybrid framework that combines several test generation techniques. Alshahwan and Harman [57] augment test-suites with the goal to increase output diversity. Mixed ranges are conceptually similar to augmentation techniques because we reason about the changes between subjects and properties to speed up generation of new tests.

Propositional satisfiability (SAT) solvers [58], [59] implement numerous techniques for efficient solving. Mixed ranges in iGen share the spirit of some of these techniques. The key difference is the very different level iGen works at – Java predicates have much more complex structures and semantics than CNF formulas. Moreover, while structural properties can be translated to SAT, e.g., using the Alloy tool-set, a straightforward application of incremental SAT for generating complex structures like iGen is not feasible.

VII. Conclusions

We presented iGen, a novel approach to optimize bounded exhaustive testing based on imperative predicates. iGen memoizes intermediate results of a test generation and reuses the results in a future search even when the new search space differs from the old search space. We instantiated iGen for two programming languages (Java and Python) and evaluated these implementations with several data structure pairs, including two pairs from the Standard Java Library. Our results show that iGen speeds up test generation up to 46.59× and 49.47× (over extension-unaware techniques) for Java and Python, respectively. Additionally, we show that the speedup increases for larger test instances. iGen also complements the existing work on parallel test generation and the work on using BET techniques for backend model counting, and the combination of these techniques is a promising future direction.

Acknowledgments

This research was partially supported by the US National Science Foundation under Grants Nos. CCF-1566363, CCF-1652517, and CCF-1704790.

REFERENCES

[1] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *International Conference on Software Engineering*, 2007, pp. 75–84.

[2] P. Tonella, "Evolutionary testing of classes," in *International Symposium on Software Testing and Analysis*, 2004, pp. 119–128.

[3] G. Fraser and A. Arcuri, "Whole test suite generation," *Trans. Softw. Eng.*, vol. 39, no. 2, pp. 276–291, 2013.

[4] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov, "BALLERINA: Automatic generation and clustering of efficient random unit tests for multithreaded code," in *International Conference on Software Engineering*, 2012, pp. 727–737.

[5] I. Kuraj, V. Kuncak, and D. Jackson, "Programming with enumerable sets of structures," in *Conference on Object-oriented Programing, Systems, Languages, and Applications*, 2015, pp. 37–56.

[6] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *International Symposium on the Foundations of Software Engineering*, 2007, pp. 185–194.

[7] A. Celik, S. Pai, S. Khurshid, and M. Gligoric, "Bounded exhaustive test-input generation on GPUs," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2017, pp. 94:1–94:25.

[8] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov, "Parallel test generation and execution with Korat," in *International Symposium on Foundations of Software Engineering*, 2007, pp. 135–144.

[9] J. H. Siddiqui and S. Khurshid, "PKorat: Parallel generation of structurally complex test inputs," in *International Conference on Software Testing Verification and Validation*, 2009, pp. 250–259.

[10] N. Dini, C. Yelen, and S. Khurshid, "Optimizing parallel Korat using invalid ranges," in *International Symposium on Model Checking of Software*, 2017, pp. 182–191.

[11] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in UDITA," in *International Conference on Software Engineering*, 2010, pp. 225–234.

[12] P. Ponzio, N. Aguirre, M. F. Frias, and W. Visser, "Field-exhaustive testing," in *International Symposium on Foundations of Software Engineering*, 2016, pp. 908–919.

[13] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on Java predicates," in *International Symposium on Software Testing and Analysis*, 2002, pp. 123–133.

[14] D. Marinov, "Automatic testing of software with structurally complex inputs," Ph.D. dissertation, Massachusetts Institute of Technology, 2004.

[15] D. Marinov and S. Khurshid, "TestEra: A novel framework for automated testing of Java programs," in *International Conference on Automated Software Engineering*, 2001, pp. 22–31.

[16] "Korat web page," 2017, http://korat.sourceforge.net/index.html.

[17] B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[18] "Adding null key to empty TreeMap without Comparator should throw NPE," 2018, https://bugs.java.com/bugdatabase/view_bug.do?bug_id=5045147.

[19] "Inputgen (PyIG) web page," 2017, https://pypi.python.org/pypi/inputgen/0.7.1.

[20] H. Bagheri and S. Malek, "Titanium: efficient analysis of evolving Alloy specifications," in *International Symposium on Foundations of Software Engineering*, 2016, pp. 27–38.

[21] A. Filieri, C. S. Păsăreanu, and W. Visser, "Reliability analysis in symbolic Pathfinder," in *International Conference on Software Engineering*, 2013, pp. 622–631.

[22] A. Filieri, M. F. Frias, C. S. Păsăreanu, and W. Visser, "Model counting for complex data structures," in *International Symposium on Model Checking of Software*, 2015, pp. 222–241.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. Cambridge, MA, USA: The MIT Press, 2009.

[24] N. Dini, C. Yelen, Z. Alrmaih, A. Kulkarni, and S. Khurshid, "Korat-API: a framework to enhance Korat to better support testing and reliability techniques," in *Symposium on Applied Computing*, 2018, pp. 1934–1943.

[25] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. Cambridge, MA: The MIT Press, 2006.

[26] S. Khurshid, "Generating structurally complex tests from declarative constraints," Ph.D. dissertation, 2004.

[27] "Titanium web page," 2017, http://www.ics.uci.edu/~seal/projects/titanium/index.html.

[28] "Korat-API heap-PC problems GitHub page," 2018, https://git.io/fxJsR.

[29] W. F. Opdyke and R. E. Johnson, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems," in *Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990, pp. 145–161.

[30] W. F. Opdyke, "Refactoring Object-Oriented Frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[31] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Adison-Wesley, 1999.

[32] D. Silva, N. Tsantalis, and M. T. Valente, "Why We Refactor? Confessions of GitHub Contributors," in *International Symposium on the Foundations of Software Engineering*, 2016, pp. 858–870.

[33] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[34] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, 2013.

[35] N. Dini, "MKorat: A novel approach for memoizing the Korat search and some potential applications," Master's thesis, The University of Texas at Austin, 2016.

[36] J. H. Siddiqui and S. Khurshid, "Scaling symbolic execution using ranged analysis," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2012, pp. 523–536.

[37] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *USENIX Conference on Operating Systems Design and Implementation*, 2008, pp. 209–224.

[38] W. Visser, K. Havelund, G. P. Brat, and S. Park, "Model checking programs," in *International Conference on Automated Software Engineering*, 2000, pp. 3–12.

[39] N. Rosner, J. H. Siddiqui, N. Aguirre, S. Khurshid, and M. F. Frias, "Ranger: Parallel analysis of Alloy models by range partitioning," in *International Conference on Automated Software Engineering*, 2013, pp. 147–157.

[40] R. Qiu, "Scaling and certifying symbolic execution," Ph.D. dissertation, The University of Texas at Austin, 2016.

[41] S. Khurshid, C. S. Pasareanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2003, pp. 553–568.

[42] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.

[43] N. Rosner, V. Bengolea, P. Ponzio, S. A. Khalek, N. Aguirre, M. F. Frias, and S. Khurshid, "Bounded exhaustive test input generation from hybrid invariants," in *International Conference on Object Oriented Programming Systems Languages & Applications*, 2014, pp. 655–674.

[44] E. Uzuncaova, "Efficient specification-based testing using incremental techniques," Ph.D. dissertation, The University of Texas at Austin, 2008.

[45] P. Godefroid, "Compositional dynamic test generation," in *Symposium on Principles of Programming Languages*, 2007, pp. 47–54.

[46] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: Reducing, rusing and recycling constraints in program analysis," in *International Symposium on Foundations of Software Engineering*, 2012, pp. 58:1–58:11.

[47] G. Yang, C. S. Păsăreanu, and S. Khurshid, "Memoized symbolic execution," in *International Symposium on Software Testing and Analysis*, 2012, pp. 144–154.

[48] O. Sokolsky and S. A. Smolka, "Incremental model checking in the modal mu-calculus," in *International Conference on Computer Aided Verification*, 1994, pp. 351–363.

[49] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan, "Incremental state-space exploration for programs with dynamically allocated data," in *International Conference on Software Engineering*, 2008, pp. 291–300.

[50] G. Yang, M. B. Dwyer, and G. Rothermel, "Regression model checking," in *International Conference on Software Maintenance*, 2009, pp. 115–124.

[51] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler, "Conditional model checking: A technique to pass information between verifiers," in *International Symposium on the Foundations of Software Engineering*, 2012, pp. 57:1–57:11.

[52] G. Yang, S. Khurshid, and C. S. Păsăreanu, "Memoise: A tool for memoized symbolic execution," in *International Conference on Software Engineering*, 2013, pp. 1343–1346.

[53] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *International Conference on Automated Software Engineering*, 2008, pp. 218–227.

[54] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: Techniques and tradeoffs," in *International Symposium on the Foundations of Software Engineering*, 2010, pp. 257–266.

[55] D. Qi, A. Roychoudhury, and Z. Liang, "Test generation to expose changes in evolving programs," in *International Conference on Automated Software Engineering*, 2010, pp. 397–406.

[56] Y. Kim, Z. Xu, M. Kim, M. B. Cohen, and G. Rothermel, "Hybrid directed test suite augmentation: An interleaving framework," in *International Conference on Software Testing, Verification, and Validation*, 2014, pp. 263–272.

[57] N. Alshahwan and M. Harman, "Augmenting test suites effectiveness by increasing output diversity," in *International Conference on Software Engineering*, 2012, pp. 1345–1348.

[58] N. Een and N. Sorensson, "An extensible SAT-solver," in *International conference on theory and applications of satisfiability testing*, 2003, pp. 502–518.

[59] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Design Automation Conference*, 2001, pp. 530–535.