Test Input Generation Using Dynamic Programming

Razieh Nokhbeh Zaeem and Sarfraz Khurshid
The University of Texas at Austin, Austin, TX, USA
nokhbeh@utexas.edu, khurshid@ece.utexas.edu

ABSTRACT
Constraint-based input generation is an effective technique for testing programs, such as compilers and web browsers, which have complex inputs. However, efficient generation of such inputs remains a challenging problem. We present a novel input generation technique that takes constraints written as recursive predicates in the underlying programming language and uses dynamic programming to solve the constraints efficiently. Our key insight is to leverage the recursive structure of desired inputs and partition the problem of generating an input into several sub-problems of generating smaller inputs that exhibit the same structure, and then to use dynamic programming — a well-known problem solving methodology designed to exploit common sub-problems — to combine them. A lazy initialization strategy and symbolic execution optimize our basic technique. Our technique provides not only bounded exhaustive input generation but also enables random input generation. We show the correctness of our technique. Furthermore, we present an experimental evaluation, which shows that our technique can provide over an order of magnitude performance improvement for input generation compared to Korat (an efficient solver for structural constraints) and Pex (a state-of-the-art tool for symbolic execution). Finally, we use our technique to effectively find bugs in production versions of Google Chrome and Apple Safari web browsers.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging—testing tools

Keywords
Recursive Test Generation, Dynamic Programming, Lazy Initialization, Symbolic Execution

1. INTRODUCTION
Test input generation is one of the most challenging tasks in automated testing. Generation is especially hard for programs, such as browsers or compilers, which take complex structures, e.g., HTML or Java programs, as inputs, because such inputs are hard to generate automatically or sample at random. Constraint-based testing [9, 21, 24, 29, 40] provides the basis for effective techniques [11, 35, 32, 16, 17, 25] for generation of such structurally complex inputs: an input constraint defines the structural properties of a class of desired inputs, a constraint solver enumerates solutions to the constraint, and the solutions are refined as test inputs. Advances in constraint solving technology [4, 13] have enabled solvers to handle more complex constraints. However, scaling constraint-based testing remains a challenging problem, particularly when used for systematic testing, i.e., bounded exhaustive testing, which tests against all inputs within a bound on the input size.

This paper introduces a novel constraint-based technique for efficient and scalable generation of structurally complex inputs. To write constraints, our technique supports recursive predicates that are written in the same language as the program under test. Imperative predicates that check constraints, especially class invariants, are traditionally called repOK methods [31]. Our key insight is that not only are recursive predicates more natural to write for recursive structures (which may also have some non-recursive elements) — e.g., a binary tree is either null, or a node whose left and right children are each a binary tree — but also recursive predicates naturally lend themselves to recursive input generation. Intuitively, since the predicate definition is recursive, it suffices to build the non-recursive elements of a new input, and use the same method to recursively build the recursive elements. For example, to build a binary tree which is not null, it suffices to assign one node as the root and recursively call the same procedure to build binary trees for the right and left children.

Our technique utilizes the recursive structure of desired inputs to divide the problem of generating an input into several sub-problems of generating smaller inputs that exhibit the same structure, and employs dynamic programming [10] — a well-known problem solving methodology designed to exploit common sub-problems — to combine them and generate inputs in a bottom-up manner. To illustrate, when exhaustively generating binary trees as inputs, we build all binary trees of sizes 0, 1, 2, and 3 and use them to construct binary trees of size 4. For random test generation, we randomly generate some binary trees of sizes 0, 1, 2, and 3, and then combine them to sample binary trees of size 4. Since many recursive test inputs exhibit the property of overlapping sub-problems, dynamic programming saves us a lot of computation.

We present three algorithms for input generation: (1) a basic algorithm that uses dynamic programming directly, (2) a lazy initialization strategy that optimizes dynamic programming, and (3) a further optimization using symbolic execution [27, 9]. Our first algorithm (DP) utilizes dynamic programming to generate test inputs up to a given size. To build a new test input, we take formerly generated test inputs and combine them to build the recursive parts, explore the state space to assign values to the non-recursive parts, and pass the resulting candidate object to the repOK predicate for
evaluation. If repOK returns true, this object is saved as a valid input and is then used to create larger inputs.

Our second algorithm (LazyDP) uses lazy initialization to optimize performance. To cope with the limitation of other resources beyond computational time, such as memory, we store a recursive test input concisely as an array of numbers, where each number recursively represents a formerly generated test. When combining inputs to build a candidate, we do not expand all the smaller test inputs in the hierarchy from their concise form to actual objects. Instead, we expand them lazily whenever repOK accesses them.

Our third algorithm (SymboLazyDP) further enhances the performance by skipping a systematic search for non-recursive fields when possible. For recursive fields, this search is improved by using the first two algorithms where we restrict the choice of smaller sub-problems to formerly generated solutions. To avoid the search for non-recursive fields, the third algorithm adds symbolic execution [27, 26, 46], which uses symbolic values for non-recursive fields, builds a path condition while executing repOK, and uses an in-house constraint solver to solve for the symbolic values.

Our technique not only enables efficient generation of inputs based on textbook data structures, but also a wide-range of string-based inputs that represent structured data, e.g., strings that belong to a context-free grammar, SQL queries, data in a database management system, Java programs, and HTML/CSS pages. We developed a prototype implementation of our three algorithms and evaluated it using microbenchmarks and real-world applications, including Google Chrome and Apple Safari web browsers.

This paper makes the following contributions. We introduce the use of recursive predicates in constraint-based generation of complex structures (that have recursive and non-recursive elements) to both facilitate predicate formulation and enable faster input generation. We present a novel technique for input generation using dynamic programming. To our knowledge, dynamic programming has not previously been used to generate structurally complex inputs. We prove the correctness of our technique. We utilize lazy initialization and symbolic execution to optimize the performance of dynamic programming for input generation. We apply our technique based on dynamic programming for both exhaustive and random generation of test inputs. We use our prototype implementation to evaluate our technique. Experimental results show our technique improves input generation performance and scalability for microbenchmarks over state-of-the-art testing tools Pex [46] and Korat [5], and effectively finds real bugs in production versions of two widely used applications.

2. EXAMPLE

In this section, we describe two examples: a binary tree, which we use as a running example, and an HTML input that we use alongside with a CSS input to test a web browser. The recursive definition of a binary tree is as follows: a binary tree is either null, or a node whose left and right children each point to a binary tree. The tree should not have any loops. Furthermore, in our binary tree, btSize is equal to the number of nodes in the tree. Listing 1 shows the recursive implementation of a binary tree. One can easily see that a recursive repOK is a natural way of describing the properties of a recursive data structure.

Listing 2 shows the repOK method of an HTML input. The backbone of such an input is a generic tree, a recursive data structure. In addition, other constraints are enforced on HTML tags. In Section 4.4, we demonstrate how to leverage this modeling of HTML files and list-modeling of CSS files to automatically generate bug revealing test inputs for the Chrome and Safari web browsers. Figure 1 models the nested structure of tags in Listing 7 (Section 4.4).

class BinaryTree {
    BinaryTree right, left;
    int btSize;
    
    boolean repOK() {
        if (!acyclic()) return false;
        return recursive_repOK();
    }
    
    boolean recursive_repOK() {
        int rightBtSize, leftBtSize;
        if (right == null) rightBtSize = 0;
        else {
            if (!right.recursive_repOK())
                return false;
            rightBtSize = right.btSize;
        }
        if (left == null) leftBtSize = 0;
        else {
            if (!left.recursive_repOK())
                return false;
            leftBtSize = left.btSize;
        }
        if (btSize == rightBtSize + leftBtSize + 1) return true;
        else return false;
    }
}

Listing 1: A recursive binary tree in Java.

class HTML {
    int tagIndex; HTML[] child;
    Attribute[] attr;
    String[] HTMLTags = {"html","head","link","body" ...};
    
    boolean repOK() {
        if (!acyclic()) return false;
        if (tagIndex != 0)="/html"
            return false;
        if (!child[0].repOKHead())
            return false;
        if (!child[1].repOKBody())
            return false;
        return true;
    }
    
    boolean repOKBody() {
        if (tagIndex != 3="/body"
            return false;
        if (!child[0].recursive_repOK())
            return false;
        ... return true; }
}

Listing 2: HTML repOK method.

Figure 1: A tree representation of an HTML input.

Note that we support repOK methods with both recursive and non-recursive parts. The btSize of each subtree in Listing 1 and the repOKHead and repOKBody checks in Listing 2 are examples of non-recursive parts of a repOK.

3. FRAMEWORK

In this section, we describe how we use recursive definitions of data structures for exhaustive and random test generation. We explain our three algorithms and prove a theorem on their correctness.

3.1 Recursive repOK Methods

Many test inputs, such as data structures, have an embedded recursive structure. Sets, trees, stacks, queues, arrays, heaps, and many other data structures have recursive definitions. Using recursion in repOK to identify correct instances of recursive loop-free data structures makes repOK easier to write, read, and debug.

Besides identifying correct instances of a data structure, a repOK method should be able to identify incorrect structures as well, but incorrect structures are not necessarily loop-free. To comply with this standard definition of repOK, we require the template shown in the repOK method of Listing 1. This repOK first checks for cycles, and then enters the recursion phase to avoid an infinite loop. Throughout this paper we use recursive repOK methods that assume acyclicity.
We process the source of a recursive repOK through a simple pattern matching to find all recursive calls. A field on which repOK is recursively called is identified as a recursive field. We could also identify recursive calls by processing the Abstract Syntax Tree of method calls.

### 3.2 Algorithms

Here we describe DP, LazyDP, and SymboLazyDP. The use of these algorithms is orthogonal to exhaustive or random test generation. In Section 3.2.2 we describe a variation of the algorithms that generates random tests.

#### 3.2.1 DP

Generation of test inputs can benefit from the recursive structure of repOK. Given a recursive repOK, the goal of exhaustive test generation is to generate all test inputs, in a given scope, for which repOK returns true. For example, the recursive repOK of a binary tree (Listing 1) checks whether its right and left children are correct binary trees and whether btSize is correctly set to the sum of the btSize fields of the children plus one. By observing the execution of repOK, we present our recursive method of generation: a new candidate test is generated by setting its recursive fields to formerly generated correct tests, and finding correct values for the non-recursive fields. Then, repOK is invoked on the candidates to filter out the incorrect ones. The repOK method directly rejects the candidates with loops and recursively calls itself to evaluate different parts of the loop-free candidates. This check includes assuring the correctness of recursive substructures. Because we provide previously generated valid test inputs as substructures, we can bypass these internal recursive calls and directly return true for them.

Besides breaking the problem into subproblems, our generation method demonstrates the other property necessary to a dynamic programming solution: overlapping subproblems. The same substructure is used multiple times to build new candidate test inputs.

For more efficient test generation, the DP algorithm avoids generating repetitive candidates. To this end, we generate test inputs in iterations and keep three sets of previously generated inputs: thisRoundTests contains correct tests generated in the current iteration, lastRoundTests includes correct tests generated in the last iteration (which have not yet been used to build other candidates), and pool contains all other correct tests generated so far. At the beginning of test generation, pool and lastRoundTests are empty, and lastRoundTests contains only null (line 6 of Listing 3). We assume that null is always a valid test input because we cannot call repOK on null (e.g., it throws an NullPointerException in a Java program), and for all common data structures null is a valid instance.

The DP algorithm proceeds as follows (Listing 3). If repOK makes r recursive calls, we need r (not necessarily different, but ordered) tests to build a new test. The outer while loop (lines 10 to 31) executes as long as it makes progress in generating new test inputs. At each iteration, the test inputs generated in the previous iterations are combined to form new candidates; then repOK is invoked to identify correct test inputs. More specifically, the recursives array selects a permutation of r recursive substructures by calling the nextPermutation method. This method iterates over pool and lastRoundTests and upon each invocation, provides the next permutation of r substructures from the set lastRoundTests & pool, such that at least one of the r substructures is selected from lastRoundTests. (Section 3.3 shows how this constraint assures that we would avoid repetitive candidates.) When all permutations are exhausted, this method returns null.

A challenge for DP is the exponential growth of the number of candidates at the outer boundaries of the scope. In fact, even creating those candidates can considerably affect the performance. To address this challenge, we keep the tests in each of the three sets sorted according to their sizes. We use bucket sort [10] (since the maximum size of a valid test is known from the scope) as we add new tests. In Listing 3, once recursives is selected, we first examine its size in line 15. If a candidate built with this permutation would be outside the scope, we throw this permutation away and also prune all other permutations with the same or bigger sizes by calling the nextPermutationPruning method. This method gives the next permutation that is inside the scope. Once we have such a permutation, we use the r substructures to build the recursive fields of a new candidate. In order to find the proper values for the non-recursive fields, we perform a systematic search by calling the nextValuation method, which, upon each invocation, returns one valuation for the non-recursive fields. A permutation for recursives together with a valuation of fields gives us a candidate (testcaseObj) which we send to repOK. If repOK returns true (ignoring randomIsOn and coinToss for now), the new test gets added to thisRoundTests with its appropriate size, otherwise it is discarded. At the end of the outer while loop, when all permutations are exhausted, the tests in lastRoundTests join pool (lines 28 and 29), the tests in thisRoundTests replace lastRoundTests (line 30), and we move on to the next round.

In the DP algorithm, size is defined recursively as follows. If a candidate is null, its size is 0. Otherwise, the size of a candidate is the sum of the sizes of its recursive substructures plus one. Note that in the example of a binary tree, btSize has the same meaning of size. The size concept built into the DP algorithm, however, does not necessarily correspond to a field of the data structure. The DP algorithm memoizes and uses the size of a test, whereas it treats btSize as a non-recursive field, and performs a systematic search to find its correct value for any candidate.

Consider the example of finding binary trees up to size 3 (Figure 2). In the first iteration, the invocation of nextPermutation

```
void testGeneration() {
    int r = getNumRecursiveChildren();
    int s = getNumNonRecursiveFields();
    pool = initialize();
    lastRoundTests = initialize();
    addTo(lastRoundTests, null, 0);//0 is the size of null
    totalExplored++;
    validCasesGenerated++;
    boolean progress = true;
    while ((progress)) {// test generation round loop
        progress = false;
        thisRoundTests = initialize();
        int[] recursives = nextPermutation(r);
        while (recursives != null) {
            if (size(recursives) >= maxNumRecursives) {
                recursives = nextPermutationPruning(r, size(recursives));
                continue;
            }
            int[] fields = nextValuation(s);
            while (fields != null) {
                totalExplored++;
                Object testcaseObj = buildCandidate(recursives, fields);
                if (testcaseObj != null) {
                    if (randomIsOn || coinToss(size(recursives))) {
                        progress = true;
                        validCasesGenerated++;
                        addTo(thisRoundTests, combine(recursives, fields), size(recursives));
                    }
                }
                fields = nextValuation(s);
                recursives = nextPermutation(r);
                for (int[] test : lastRoundTests) {
                    addTo(pool, test, size(test));
                }
                lastRoundTests = thisRoundTests;
            }//end test generation round loop
        }
    }
}
```

Listing 3: Test generation algorithm in Java.
returns different permutations of r (i.e., two) formerly generated binary trees such that at least one of them is from lastRoundTests. Here there is only one option: null for both right and left children. The total size of the binary trees in recursives is 0 + 0 = 0, which is still less than 3, so no pruning happens at this point. Furthermore, s = 1 implies that there is one field (btSize) for which we should systematically search all values in the scope. We build candidates by assigning null as both children and exploring different values for btSize ranging from 0 to 3 (these numbers come from the scope). Each of these candidates is sent to repOK, and the one with btSize correctly set to 1 returns true. Upon receiving true from repOK, the new test is saved in thisRoundTests and the algorithm continues until all permutations and valuations are exhausted. At the end of this round, null moves to pool and tree 1 moves to lastRoundTests. The algorithm continues in the same manner. In the third iteration, first, binary trees 2 and 3 are combined with binary tree 0 (null) to generate four binary trees of size 3. Then, an example of pruning occurs. Once the algorithm selects tree 2 from lastRoundTests and tree 1 from pool, the check on the size of recursives indicates that the resulting candidate would be outside the scope. Therefore, nextPermutationPruning is called to find the next permutation of recursives that is inside the scope. Because all other permutations are bigger in size, they are pruned. For brevity, we do not show iteration 4, wherein no new test is generated. The algorithm terminates at the beginning of iteration 5, when all valid inputs are in pool.

### 3.2.2 Random Generation

Our use of dynamic programming is orthogonal to random or exhaustive test generation and can be applied to both. In order to generate random tests, we introduce randomization into the process of saving valid inputs. For random test generation, randomIsOn() returns true on line 22 of Listing 3. Therefore, saving or discarding a correct test input depends on the value returned from coinTosses(size(recursives)). This method heuristically returns true for all small inputs (inputs with a size less than a threshold parameter) in order to save them all, and uses a random number generator (a coin toss) with a fixed probability of success to randomly save or discard other correct test inputs. As the algorithm continues to execute, discarded tests do not take part in test generation. At the end, the algorithm generates one or several random tests of a given size. Keeping all small inputs (for example the single-node binary tree) helps in reducing the chance of having repetitive patterns in one random test. In the case of generating multiple tests, if sharing structures between tests is undesirable, one could run the algorithm from scratch multiple times. As Section 4 shows, our algorithms are efficient enough to generate big inputs in a matter of seconds and one can run them several times.

#### 3.2.3 LazyDP

One problem that arises during the test generation is the limitation of other resources beyond computational time, such as memory. If we keep tests as objects, we run out of the heap memory space for bigger scopes. As Listing 3 shows, to optimize memory usage, tests are saved in the compact form of an array of integers. These integers are either indexes of smaller substructures, which in turn point to other arrays of integers, or values of non-recursive fields. For example, tree 5 is saved as (right child = 3, (left child = 0, (btSize = 3)) where tree 3 is in turn kept as [(right child = 1, (left child = 0, (btSize = 2)]. The value of a non-recursive field is saved as an index with respect to the scope. For example, if there is a field with primitive type boolean whose values are false and true in the scope, 0 represents false and 1 represents true.

Whenever we create a bigger candidate test input using smaller previously generated tests, we need to call repOK to examine the correctness of the candidate. Consequently, we need to retrieve the smaller tests, build their corresponding objects, and then utilize them to build the candidate. We build the smaller tests with the lazy initialization technique, which means that the substructures get initialized (i.e., expanded from arrays of integers to objects) only when repOK accesses a field from them.

For binary trees, since repOK of Listing 1 only accesses btSize
of the direct children of a node, we only expand the direct children to objects and keep their children as arrays. For example, the expansion of binary tree 5, is limited to expanding binary trees 3 and 0, and not binary tree 1, which is a child of binary tree 3.

3.2.4 Symbolic Lazy DP

While both dynamic programming and lazy initialization improve finding the recursive values of a candidate, in order to find the correct values for its non-recursive fields (e.g., btSize), we still need to search all valuations, which diminishes the efficiency and scalability of test generation. To avoid such a search, we observe that the values of many non-recursive fields of a test can be symbolically computed rather than searched for. Following previous work [27, 26, 46] we use symbolic execution for non-recursive fields. Using symbols (instead of concrete values) for fields, we build a path condition (a boolean formula over the symbols which represents the constraints that should be satisfied to follow a path) while executing repOK. At the end of the repOK execution, we use a constraint solver to solve the path condition and calculate concrete values for non-recursive fields.

In order to enable symbolic execution, we use a source to source instrumentation on the repOK method [26]. We replace each branch condition with a boolean variable which takes both true and false values. When it takes the true value, we add the original branch condition to the path condition. When it takes the false value, we add the negation of the original condition. All valuations of such boolean variables provide all execution paths. Listing 4 shows the corresponding instrumentation for btSize (replacing the last two lines of Listing 1). Upon reaching a return statement, we invoke a constraint solver to solve the path condition and consider each solution as an acceptable valuation for the non-recursive fields. For example, btSize of a candidate binary tree is assigned through solving the condition added on line 20 of Listing 4.

We save the path condition with each valid substructure, but do not save the solution; i.e., substructures are saved with symbolic non-recursive fields and constraints on them. After combining the substructures, we perform symbolic execution on the entire candidate, including the recursive calls, because solving the substructures separately does not necessarily give compatible results (e.g., consider solving the search constraint on integer elements of a binary search tree; it is possible to save valid integer elements in the right and left children that violate the search constraint at the root, making the constraint infeasible). If getBoolean()}
addCond("btSize", EQ, "rightBtSize+leftBtSize+1");
return true;
else
addCond("btSize", NOTEQ, "rightBtSize+leftBtSize+1");
return false;
Listing 4: Instrumenting BinaryTree for symbolic execution.

3.3 Theorem

In this section, we prove that the DP algorithm generates all valid tests in the scope, and it does not generated any valid or invalid candidate more than once.

Define $S_R$ as the value of set $S$ (e.g., $pool_R$) at the end of round $R$. Furthermore, define $discarded_R$ as the set of candidates discarded during round $R$, including those outside the scope.

**Definition 1.** Visited candidates: $visitedCands_R = pool_R \cup lastRoundTests_R \cup \bigcup_{i=1}^{R} discarded_i$.

**Definition 2.** $T(C)$: For a candidate $C$, let $T(C)$ be the set of all test inputs that $C$ uses as its recursive substructures.

**Lemma 1.** At the end of each round $R (R \geq 0)$, and for any candidate $C$, the following loop invariant holds:

$T(C) \subseteq pool_R \rightarrow C \in visitedCands_R$

**Proof by Induction.** The base case is the beginning of round one, were $pool_0 = \emptyset$. The only candidate that does not use any recursive substructures is null. Yet $null \in lastRoundTests_R$ and hence $null \notin visitedCands_R$, so the invariant holds.

For the induction step, assume that:

$T(C) \subseteq pool_R \rightarrow C \in visitedCands_R$ (1)

At the end of round $R+1$, the tests in lastRoundTests join pool, and then the tests in thisRoundTests replace lastRoundTests.

$pool_{R+1} = pool_R \cup lastRoundTests_R$ (2)

$lastRoundTests_{R+1} = thisRoundTests_{R+1}$ (3)

Suppose that the invariant does not hold at the end of round $R+1$. Let $A' = T(C') \subseteq pool_{R+1} \land C' \notin visitedCands_{R+1}$ (4)

Now, $T(C') \cap lastRoundTests_R$ is either (a) $\emptyset$ or (b) $\neq \emptyset$.

For case (a): $2 \land (4) \land (a) \rightarrow T(C') \subseteq pool_R$ (5)

$(1) \land (5) \rightarrow C' \in visitedCands_R$ (6)

$Definition1 \land (2) \land (6) \rightarrow C' \in visitedCands_{R+1}$ (7)

which contradicts with (4).

For case (b): $\exists t \in T(C') : t \in lastRoundTests_R$ (8)

Note that the algorithm generates all permutations of the tests belonging to lastRoundTests or pool that have at least one test from lastRoundTests, i.e.,

$(T(C) \subseteq pool_R \land lastRoundTests_R) \land \exists t \in T(C) : t \in lastRoundTests_R)$

$\leftrightarrow C \in thisRoundTests_{R+1} \cup discarded_{R+1}$ (9)

$(2) \land (4) \land (8) \land (9) \rightarrow C' \in thisRoundTests_{R+1} \cup discarded_{R+1}$ (10)

$Defintion1 \land (3) \land (10) \rightarrow C' \in visitedCands_{R+1}$ (11)

which again contradicts with (4).

**Theorem 1.** Part 1: for a given scope, the algorithm generates all valid tests. Part 2: the algorithm does not generate any (valid or invalid) candidate more than once.

**Proof of Part 1.** Let $C_{fin}$ be the smallest correct test that the algorithm fails to generate. If the algorithm terminates right after round $fin$:

$repOK(C_{fin}) = true \land C_{fin} \notin visitedCands_{fin}$ (12)

$\forall C : repOK(C) = true \land C \notin visitedCands_{fin}$

$\rightarrow size(C) \geq size(C_{fin})$ (13)

$\forall t \in T(C_{fin}) : size(t) < size(C_{fin})$ (14)

$\forall t \in T(C_{fin}) : repOK(t) = true$ (15)

$(13) \land (14) \land (15) \rightarrow \forall t \in T(C_{fin}) : t \notin visitedCands_{fin}$ (16)

Further, the termination of the algorithm at the end of round $fin$ means that no progress was made in this round. Accordingly, no test was added to $thisRoundTests_{fin}$. By using (3)³:

$lastRoundTests_{fin} = thisRoundTests_{fin} = \emptyset$ (17)

³$fin > 0$ since the test generation loop executes at least once.
Now, let us assume that $C''$ is inside the scope.

$$\text{size}(C'') \leq \text{scope.size} \quad (18)$$

$$(14) \land (18) \rightarrow \forall t \in T(C'') : \text{size}(t) \leq \text{scope.size} \quad (19)$$

From the definition of discarded, we have:

$$\forall C \in \text{discarded}_R \rightarrow \text{repOK}(C) \neq \text{true} \lor \text{size}(C) > \text{scope.size} \quad (20)$$

$$(15) \land (19) \land (20) \rightarrow \forall t \in T(C'') : t \notin \bigcup_{i=1}^{\text{fin}} \text{discarded}_i$$

$$(16) \land (17) \land (21) \rightarrow \forall t \in T(C'') : t \in \text{pool}_{\text{fin}}$$

$$\text{Lemma1} \land (22) \rightarrow C'' \in \text{visitedCands}_{\text{fin}} \quad (23)$$

But (23) contradicts with (12), which proves part 1. □

**Proof of part 2 by induction.** Consider $R = 0$ for the induction base. Because only one instance of null is generated before the first round, no repetition happens at $R = 0$.

For the induction step, assume that all candidates visited up to the end of round $R$ are distinct. We use $\not\equiv$ to show that two candidates are different instances, although they might be equal ($\equiv$).

$$\exists t, t' \in \text{visitedCands}_R : t \not\equiv t' \land t = t' \quad (24)$$

Suppose that the first repetitively candidate, named $C$, is generated during round $R + 1$.

$$C \in \text{thisRoundTests}_{R+1} \cup \text{discarded}_{R+1} \quad (25)$$

$C$ is repetitious, so another instance of it, named $C'$, is already generated at round $R^2$.

$$C' \in \text{thisRoundTests}_R \cup \text{discarded}_R \quad (26)$$

$$(9) \land (25) \rightarrow T(C) \subseteq \text{pool}_R \cup \text{lastRoundTests}_R \quad (27)$$

$$(9) \land (26) \rightarrow T(C') \subseteq \text{pool}_{R-1} \cup \text{lastRoundTests}_{R-1} \quad (28)$$

Either (a) $R^2 = R + 1$ or (b) $R^2 < R + 1$. For case (a), note that during one round of test generation, methods nextPermutation and nextPermutationPruning provide distinct permutations. So in order to have repetitive candidates, at least one of the substructures should have more than one instance.

$$\exists t \in T(C), t' \in T(C') : t \not\equiv t' \land t = t' \quad (29)$$

$$(27) \land (28) \land (29) \land (a) \rightarrow \exists t, t' \in \text{visitedCands}_R : t \not\equiv t' \land t = t'$$

$$(30)$$

which contradicts with (24).

For case (b), notice that at least one substructure is selected from lastRoundTests.

$$(9) \land (25) \land (c) = C' \rightarrow \exists t \in T(C') : t \in \text{lastRoundTests}_R$$

$$(28) \land (31) \rightarrow \exists t \in \text{lastRoundTests}_R \setminus \text{visitedCands}_{R-1}$$

$$(32)$$

$$(24) \land (32) \rightarrow R \leq R^2 - 1$$

$$(33)$$

which contradicts with (b). □

4. EVALUATION

In order to evaluate our test generation methods, we implemented a prototype of the algorithms and designed some experiments wherein we address two research questions:

- **RQ1** How efficient and scalable are our algorithms, compared to state-of-the-art test generation tools (Pex and Korat)?

The algorithm never generates null again, so $R^2 \neq 0$.

- **RQ2** How effective are the generated tests in finding bugs in real world applications (Chrome and Safari web browsers)?

In the first set of experiments, we used six microbenchmarks, which are complex data structures widely used in programs and as test inputs. Previous work has extensively used these benchmarks for evaluation [42, 39, 45, 41, 5, 16, 32]. In order to answer the first question, we need an alternative exploration method of the state space of possible test inputs. A naive exploration of the state space will give rather unacceptable results. Therefore, we compare our methods to Microsoft Pex [46] – a state-of-the-art test generation tool. Net – and Korat [5] – a well-known test generation method and an open source tool [37] for Java programs.

Pex is a white-box test generation tool that performs symbolic execution. In addition, it uses path-bounded model-checking to cover different paths in the program. Pex is an appropriate subject tool; it particularly addresses the effect of symbolic execution. Generation of test inputs considered in this paper is black-box with respect to the code under test, yet we allow Pex to explore different paths in repOK. We used the same repOK methods for Pex and Korat, except for minor changes to accommodate syntactic differences between C# and Java respectively.

The Korat algorithm monitors repOK executions to prune large portions of the bounded space of candidate structures. Korat is an appropriate subject tool too; previous work [43] shows that Korat is among the most efficient solvers for complex structural constraints.

In addition to exhaustively generating test inputs, we compare the efficiency and scalability of our algorithms with Pex and Korat, when sampling a few large test inputs.

In the second set of experiments, we show how to naturally model HTML and CSS3 files as acyclic data structures. Such files, which are test inputs to any web browser, are examples of practical and common, yet user-defined test inputs. By systematically generating HTML and CSS3 test inputs, our generation methods found real bugs in the latest versions of two well-known web browsers, Google Chrome and Apple Safari.

4.1 Experimental Settings

Throughout the evaluation, we ran each experiment five times and reported the averages. All experiments used a 2.50GHz Core 2 Duo processor with 4.00GB RAM running Windows 7. We used Sun’s Java SDK 1.6.0 JVM with our methods and Korat, and Microsoft Visual C# 2010 version 4.0.30319 RTMRel with Pex version 0.94.51006.1. Pex used Z3 theorem prover [13] version 2.0. In Section 4.4, we used Google Chrome version 13.0.782.220 m for Windows and Apple Safari 5.1 (7534.50), the latest versions as of the date of this paper.

For symbolic execution, we used our in-house constraint solver developed in Java. The source to source instrumentation for symbolic execution is currently manual, but is mechanical and can be automated [26]. For exhaustive test generation with Pex, we used the following setting to force Pex to generate all test inputs:

```
TestEmissionFilter = PexTestEmissionFilter.All
```

4.2 Microbenchmarks

To address RQ1 for exhaustive test generation, we considered six microbenchmarks. Table 1 shows the results for the biggest sizes considered. For all six microbenchmarks and all sizes considered (including those not shown), DP and LazyDP generate the same number of tests as Korat. SymboLazyDP and Pex generate the same number of tests, since they both use symbolic execution and report one solution for each path condition, instead of exploring all valuations from the state space. In addition to improving test generation performance and scalability, symbolic execution im-
Table 1: Exhaustive test generation for the biggest sizes. TO represents a time out of 1000s. Best performance highlighted.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Biggest Size</th>
<th>Valid Tests</th>
<th>Candidates</th>
<th>Generation Time (s)</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Korat/DP</td>
<td>LazyDP</td>
<td>SymboLazyDP</td>
<td>Pex</td>
</tr>
<tr>
<td>LinkedList</td>
<td>17</td>
<td>131072</td>
<td>18</td>
<td>18</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17825963</td>
<td>2228208</td>
<td>2228208</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td></td>
<td>33.188</td>
<td>7.089</td>
<td>3.491</td>
<td>0.025</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9.195</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>33.188</td>
<td>7.089</td>
<td>3.491</td>
<td>0.025</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.025</td>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>BinaryTree</td>
<td>12</td>
<td>290512</td>
<td>290512</td>
<td>TO</td>
<td>15770974</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2207699</td>
<td>61021746</td>
<td>445227</td>
<td>51.533</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8.728</td>
<td>962.483</td>
<td>7.947</td>
<td>0.075</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>41</td>
<td>41</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RedBlackTree</td>
<td>9</td>
<td>6753</td>
<td>271</td>
<td>TO</td>
<td>24900165</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1364398</td>
<td>2101957</td>
<td>3190</td>
<td>5.654</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.074</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.590</td>
<td>4.590</td>
<td>10.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>26</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FibonacciHeap</td>
<td>6</td>
<td>1125139</td>
<td>197</td>
<td>TO</td>
<td>1364398</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2207699</td>
<td>61021746</td>
<td>445227</td>
<td>8.728</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.654</td>
<td>962.483</td>
<td>7.947</td>
<td>0.075</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>14</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BinaryHeap</td>
<td>8</td>
<td>344571</td>
<td>9</td>
<td>TO</td>
<td>1364398</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2207699</td>
<td>61021746</td>
<td>445227</td>
<td>8.728</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.654</td>
<td>962.483</td>
<td>7.947</td>
<td>0.075</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>21</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HashTable</td>
<td>12</td>
<td>4083</td>
<td>233</td>
<td>TO</td>
<td>1364398</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2207699</td>
<td>61021746</td>
<td>445227</td>
<td>8.728</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.654</td>
<td>962.483</td>
<td>7.947</td>
<td>0.075</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>38</td>
<td>38</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: Performance comparison on microbenchmarks.

proves test execution by reporting less, yet representative test inputs. SymboLazyDP is the most efficient generation method for all the microbenchmarks on the biggest size.

Figure 3 shows the performance evaluation results. The first data structure is a sorted singly-linked list of integer elements. Here, DP and LazyDP constantly outperform Korat. Lazy initialization is effective here, especially because properties like being sorted are verified locally: it suffices to compare each node’s element with its neighbor’s. Pex outperforms DP and LazyDP on bigger sizes for two data structures: singly-linked list and binary heap. The reason is that Pex generates far fewer tests by symbolically executing repOK and representing all sorted lists of a given size with only one test, while Korat, DP, and LazyDP exhaust different valuations. SymboLazyDP, however, shows the best performance among all the methods. It generates the same number of tests and grows with the same pace as Pex, but it is multiple times faster. Notice that the vertical axis is logarithmic.

The next benchmark is a binary tree as described in Section 2. As Figure 3 displays, SymboLazyDP performs the best. Pex can only enumerate all binary trees up to size 7 before timing out. We also experimented with red-black trees with size, key and color. While DP and LazyDP are faster than Korat at first, Korat takes over them at some point because DP and LazyDP have to explore all valuations of key and color, but Korat prunes many of them. Notice that LazyDP takes slightly more time than DP on this benchmark. This is because repOK needs to explore down the tree to find minimum and maximum keys to evaluate the search tree property, and eventually expands many substructures, which undermines the usage of lazy initialization. Pex uses symbolic execution for key and color, but cannot generate all red-black trees with four nodes or
more. SymboLazyDP, using dynamic programming and symbolic execution, closely competes with Korat and eventually takes over.

Figure 4 compares the memory usage of our methods and Korat for red-black trees. DP and LazyDP generate the same number of tests, so they always use equal amounts of memory. SymboLazyDP generates less tests and hence uses less memory. Korat keeps only one candidate vector while we have to keep all correct tests (albeit in a compact format). Hence, the memory usage of our methods grows faster than Korat. However, since the Java heap space is usually in the order of a few GBs, 10MB of memory usage should not be a problem. The memory usage of the other benchmarks follows a very similar pattern.

The next benchmark is a Fibonacci heap. This is the only case where the memory usage is a concern for DP and LazyDP. In fact, for Fibonacci heaps with more than six nodes, DP and LazyDP run out of the heap space and Korat and Pex run out of time. Yet, SymboLazyDP does not time out and is the most efficient. The next benchmark, a binary heap, gives results similar to the sorted memory pattern. The memory usage of the other benchmarks follows a very similar pattern.

In the last benchmark, a hash table implemented using nested lists, LazyDP and DP are slightly better than Korat. The difference increases when a time-consuming hash function is used. Since we use previously generated tests, we avoid many calls to the hash function. The generation time for Pex starts off at a bigger value and increases at the same pace as SymboLazyDP. SymboLazyDP outperforms all the other methods.

Finally, it is worth mentioning that Pex and SymboLazyDP both require path bounds (e.g., for loop unrolling). Throughout the experiments, we used trial and error to find and set the smallest bounds that provide all tests. Similarly, Korat, DP, and LazyDP need a bound on primitive (e.g., integer) values. The number of primitives used usually has a relationship with the test size.

4.3 Random Test Generation

To address RQ1 for random test generation, we considered generating ten random tests of ninety to a hundred nodes (Table 2). For Korat and Pex, we took the first ten tests generated in the desired size range. For SymboLazyDP, all small tests (up to size 3) were saved and bigger tests were saved or discarded at random (Section 3.2.2). Except for the Fibonacci heap benchmark, SymboLazyDP is the most efficient and scalable.

4.4 Google Chrome and Apple Safari

To address RQ2 and showcase the ability of our methods in finding bugs in real world, well-tested2, commercial applications, we tested the support for rendering CSS3 3D effects by Chrome and Safari web browsers. CSS (Cascading Style Sheets) is a style sheet language that separates the presentation of a markup language document from its content, and is commonly used to style web pages written in HTML and XHTML. CSS3, the latest variation of CSS, enables web developers to add 3D effects to web pages, i.e., position and move elements in the three dimensional space.

Apple Safari and Google Chrome web browsers support CSS3 3D effects. As of the date of this paper, Chrome and Safari are the third and fourth most widely used desktop web browsers with 21.5% and 4.8% worldwide usage share respectively [3]. Safari is developed in C++ and Objective-C, and precedes Chrome in supporting 3D transforms. Chrome is developed in C++, Assembly, Python, and JavaScript. Both of these browsers use Webkit layout engine which introduced 3D transforms in CSS.

We directly tested 11 KLOC of C++ code (74 .cc/.h files) from Chrome. Safari is 37 MB compiled. Our test for Safari included 2.7 KLOC (19 .cpp/.h files) of its open source code plus its closed source implementation.

4.4.1 Modeling HTML and CSS Test Inputs

An HTML file is composed of a set of nested HTML elements. An HTML element includes a start tag (e.g., <h1>) and an end tag (e.g., </h1>). The start tag might also have some attributes (e.g., class="ClassName"). We modeled an instance of an HTML file as a tree. The whole document is contained between <html> start and end tags, which we consider as the root of the tree. Further, each tag is represented as a node that has some attributes and an ordered set of children, which are the tags immediately inside it. Listing 2 shows some parts of the HTML model. Figure 1 shows the tree representation of Listing 7.

Listing 5: Abstraction of a CSS rule.

A CSS file consists of a list of rules. A rule has a selector and a declaration block. Inside a block, each declaration has a property, followed by a list of values. Listing 5 shows an abstraction of a CSS rule. We modeled each CSS rule as a linked list of alphabetically sorted properties where each property has a linked list of values. As one could see, our HTML and CSS models are intuitive and easy to implement as recursive loop-free data structures. Indeed, we have already implemented both of them as microbenchmarks.

2For example, Chrome is extensively tested before release and claims to pass 99% of WebKit’s layout tests [2]. The CSS3 3D effects are among the WebKit’s layouts.

3Because the order of properties is irrelevant, we keep them sorted to avoid duplicates.
Listing 6: An automatically generated CSS test input (file.css).

```html
<html>
<head>
<link rel="stylesheet" type="text/css" href="file.css"></head>
<body>
<div class="ClassName4">
<h1>This is some text</h1>
</div>
</body></html>
```

Listing 7: An automatically generated HTML test input.

Listings 6 and 7 show bug-revealing examples of HTML and CSS inputs, automatically generated by our methods.

### 4.4.2 Experimental Results

Using the above models, we systematically generated all test inputs with up to eight tags (two `<div>` tags) inside an HTML file and two declarations inside a CSS declaration block. Five CSS properties were used: perspective, backface-visibility, transform, transform-origin, and transform-style. Also, various values for these properties were used including perspective, rotate, scale, skew, and translate for the transform property. Each HTML tag could have a CSS selector as its class attribute (See Listings 6 and 7 as an example). Consistency constraints between CSS and HTML files are maintained by first running the CSS input generator and then feeding the number of classes it generates to the HTML input generator to exhaustively cover all classes.

Table 3 shows a summary of the results. In total, 3081 test inputs (each including an HTML file with the corresponding CSS file) were generated. The size of the input space for the chosen bounds is $10^{10}$. The size bound in this experiment is too small to get LazyDP benefits. However, SymbolLazyDP gives a clear advantage. In our model, SymbolLazyDP is not applicable on CSS inputs for the lack of non-recursive fields that can be executed symbolically.

### 4.4.3 Differential Testing

So far, we have automatically generated the test inputs. But in order to test Chrome and Safari with these tests, we need an oracle that defines the correct rendered output for any given test input. Since no such oracle was available, we use differential testing [34], where the outputs of two implementations are checked against each other. Whenever the outputs are not the same, there likely is a bug in at least one of the implementations. We wrote a test harness in Java that automatically launches Chrome and Safari with each test input, and performs a basic image differencing algorithm to compare the screen shots taken from them. All test inputs were checked in less than 2 hours. Such time-consuming checks are specific to this application. Furthermore, improving the performance of launching the browsers and image differencing is possible, yet beyond the scope of this paper.

### 4.4.4 Bugs Found

Among the 3081 tests generated, 818 tests were rendered differently by Chrome and Safari. We semi-automatically investigated these tests. Out of these 818 failures, 148 cases were false positive due to the inaccuracy of our image differencing algorithm. We manually classified the rest of the failing tests (670 tests) based on the CSS properties used, and found at least three distinct bugs in the production code; stable releases of Chrome and Safari. The actual number of faults in the code, which produce these failures, in fact, may be greater. However, localizing the faults was not possible due to the proprietary code of Safari. We reported three bugs in Chrome. One of these bugs was regarding the hidden backface-visibility of an element. (Listings 6 and 7 reveal this bug.) Another bug involved the webkit-perspective property. Both of these bugs in Chrome were already reported and confirmed (issue 76947 in the Chromium project) and are fixed in the next Canary release. The last bug was due to a rotation direction inconsistency with the W3C editor’s draft (21 March 2011) [1] as the standardization in progress of CSS 3D transforms. This bug is not confirmed by Google Inc. as of the date of this paper.

Consider Listings 6 and 7. To reveal this bug, we need two nested classes where the outer one has a 180 degree rotation and the inner one has a hidden visibility as well as the webkit-perspective property. Invoking the hidden visibility by itself or in any other setting is not enough to show the problem.

### 4.4.5 Applying Symbolic Execution and Korat

We strove to use symbolic execution on the source code available from Chrome. The corresponding code, however, includes 74 .cc/.h files (11 KLOC of code) that collectively render a CSS 3D effect. We were unable to apply white box symbolic execution due to the code size and complexity. Symbolic execution is not feasible for testing closed source systems (Safari). Korat can, in principle, find the bugs if given enough time. Yet, as we showed, our technique outperforms Korat in all the cases of exhaustive test generation.

<table>
<thead>
<tr>
<th>Candidates</th>
<th>Gen. Time (s)</th>
<th>#Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSS HTML</td>
<td>10.231</td>
<td>0.140</td>
</tr>
<tr>
<td>LazyDP</td>
<td>10.231</td>
<td>0.140</td>
</tr>
<tr>
<td>SymbolLazyDP</td>
<td>N/A</td>
<td>1.628</td>
</tr>
</tbody>
</table>

### 4.5 Threats to Validity

#### Internal validity

(1) To implement our algorithms for test generation using dynamic programming, we strictly followed the original algorithms, used well-known libraries, and validated the number of inputs generated to match the numbers generated by other independently developed tools, namely Korat and Pex. (2) To compare with Korat, we used its open-source implementation that has been in the public domain for over five years, and used the repOK’s that are distributed with it. (3) To compare with Pex, we used its public distribution (version 0.94.51006.1) while setting the search depth bound to the smallest number required to complete the generation of all inputs within the chosen size in order to minimize the exploration time for Pex. We carefully performed a faithful translation of Java repOK’s used for Korat into C# to run Pex. We used our own in-house constraint solver developed in Java for symbolic execution with dynamic programming, which might give different performance results compared to Z3 [13] used by Pex. Although it is unlikely that our solver in Java is faster than the state-of-the-art Z3, to address this threat, we plan to use Z3 as a part of future work.

#### External validity

The main threat here involves using only two industrial programs (Chrome and Safari). To address this threat, we experimented with microbenchmarks that have previously been used by a number of other authors [42, 39, 45, 41, 5, 16, 32].

#### Construct validity

We used metrics commonly used in software testing research to compare test generation tools, and automated our entire test generation and execution process. Furthermore, we manually investigated the failures reported for the browser testing.
5. APPLICABILITY

Our work directly enables systematic (i.e., bounded exhaustive) testing to scale better for certain applications, e.g., refactoring engines, compilers, model checkers, and browsers, which clearly must be tested against larger inputs. The inputs to these applications are programs themselves, which can be modeled and generated at the Abstract Syntax Tree level – an acyclic structure – using structural constraints. E.g., Alloy programs were modeled and generated to find bugs in Alloy-alpha [32]; more recently, systematic testing found bugs in Eclipse, NetBeans, Sun javac, and JPF [16, 12].

More generally, our work can help systematic grammar-based testing techniques [33, 30]. Such techniques enumerate all strings, up to a given bound, that belong to a context-free grammar. Context-free grammars can describe various input types, such as XML schemas and programs. To illustrate, consider the work of Khalek et al. [25], which uses constraint-based testing to reveal bugs in Oracle 11g. It enumerates solutions for a subset of SQL grammar and a schema to provide queries and populate the database. Our technique has a direct application in generating strings that belong to the SQL grammar, and can also improve the generation of tabular test data.

Our use of dynamic programming is not limited to bounded exhaustive generation, rather our technique also facilitates random test generation, which complements systematic testing and has also been used successfully to find bugs [15, 8, 39, 19]. Most recently, Yang et al. used random test generation to find numerous bugs in mainstream C compilers [49]. As our work shows, dynamic programming can be used in synergy with random test generation.

While we describe algorithms for generating recursive structures without cycles, our approach can be used to generate cyclic structures as a part of a multi-step generation technique. For example, we can generate an acyclic backbone in the first step using dynamic programming and populate the remaining fields using constraint-based data structure repair [14] in the second step.

6. RELATED WORK

The importance of using specifications in testing has long been recognized [20]. Several projects automate test generation from specifications in various languages [23, 38]. The specific use of logical constraints to represent inputs dates back at least three decades [9, 21, 24, 29, 40]. But a focus of prior work has been to solve constraints on primitives, and not on complex structures – which require very different constraint solving techniques. Korat [5] and TestEra [32] are among the first frameworks to provide systematic generation of structurally complex tests from constraints. Following this spirit of systematic black-box testing, ASTGen [12] and UDITA [16] are two more recent frameworks, which have been used successfully to find bugs in real applications, including refactoring engines. ASTGen requires the user to write imperative test input generators, whose executions produce input programs for refactoring engines. ASTGen bears some similarities to our framework in composing test generators to build bigger inputs. However, it is limited to testing refactoring engines and requires the user to explicitly specify how to generate test inputs. UDITA provides a programming language to describe test inputs using a combination of declarative and imperative styles, where constraint solving is used in conjunction with partial generators.

Lava [44] and QuickCheck [8] can also provide generation of complex structures. Lava requires the user to describe inputs using a production grammar and generates strings in the grammar, but cannot handle complex constraints, such as those of a red-black tree. QuickCheck requires the user to write a generator for complex inputs and generates random inputs for testing functional programs in Haskell using a technique similar to recursion with lazy initialization. Similarly, Gast [28] generates tests for programs written in functional languages. However, QuickCheck and Gast use pure top-down recursion and not dynamic programming. While dynamic programming has been used for monitor generation from formal specifications [22], to our knowledge, there is no previous work on using dynamic programming for test input generation.

Several tools use method sequences for testing object-oriented programs, and can generate complex structures using systematic [48] or randomized exploration [39]. While these tools allow unit testing, they cannot feasibly generate inputs that are parsed from strings with semantic and syntactic constraints, e.g., XML files, which constraint-based test generation handles readily.

The recent advances in constraint solving technology [4, 13] have led to a rebirth of symbolic execution [27, 9] – a powerful program analysis technique that was traditionally used for checking small programs with primitive types. Generalized symbolic execution [26] implements Korat using the Java PathFinder model checker [47] and supports structural constraints using symbolic execution. Guiding symbolic execution using concrete executions is rapidly gaining popularity as a means of scaling it up in several recent frameworks, most notably DART [19], CUTe [42], EXE [7], and Pex [46]. While DART and EXE focus on properties of primitives and arrays to check for security bugs, such as buffer overflows, CUTe and Pex support the use of preconditions in white-box testing. Compositional techniques for symbolic execution, introduced by PReFix and PReFast [6], can handle larger code bases but they do not currently handle complex structural properties [18]. Our work provides a novel way to scale symbolic execution by applying it with dynamic programming in synergy.

7. CONCLUSIONS

We presented a novel technique for exhaustive and random generation of test inputs for programs that operate on structurally complex tests, e.g., recursive data structures. Our key insight is to leverage the recursive structure of desired inputs and partition the problem of generating an input into several sub-problems of generating smaller inputs that exhibit the same structure, and to use dynamic programming to combine them. We used a lazy initialization strategy as well as symbolic execution to optimize the technique. We formally proved the correctness of our algorithm. Experimental results show that our technique provides more efficient and scalable generation of structurally complex tests for a variety of subject programs, compared to state-of-the-art test generation tools Pex and Korat. Furthermore, our technique found real bugs in well-tested commercial applications Google Chrome and Apple Safari.

While in this paper we focus on generation of recursive data structures, our work paves the way for development of novel techniques for generating more general classes of inputs. To illustrate, our technique can be used as a sub-routine to populate a recursive component of a larger input and the other components could be populated using complementary techniques. Given the increasing use of constraint solving technology in software verification, we believe the time is ripe for dynamic programming to make a significant impact on our ability to find more bugs faster and to deploy more reliable software.

8. ACKNOWLEDGMENTS

We thank Mahdi Kefayati for helpful ideas. This work was funded in part by the NSF under Grant No. CCF-0845628 and AFOSR grant FA9550-09-1-0351.
9. REFERENCES


