Parallel Test Generation and Execution with Korat

Saša Misailović
University of Belgrade
Belgrade, Serbia
sasa.misailovic@gmail.com

Aleksandar Milićević
University of Belgrade
Belgrade, Serbia
aca.milicevic@gmail.com

Nemanja Petrovic
Google, Inc.
New York, NY
nemanja@gmail.com

Sarfraz Khurshid
University of Texas
Austin, TX
khurshid@ece.utexas.edu

Darko Marinov
University of Illinois
Urbana-Champaign, IL
marinov@cs.uiuc.edu

ABSTRACT

We present novel algorithms for parallel testing of code that takes structurally complex test inputs. The algorithms build on the Korat algorithm for constraint-based generation of structurally complex test inputs. Given an imperative predicate that specifies the desired structural constraints and a finitization that bounds the desired input size, Korat performs a systematic search to generate all test inputs (within the bounds) that satisfy the constraints. We present how to generate test inputs with a parallel search in Korat and how to execute test inputs in parallel, both off-line (when the inputs are saved on disk) and on-line (when execution immediately follows generation).

The inputs that Korat generates enable bounded-exhaustive testing that checks the code under test exhaustively for all inputs within the given bounds. We also describe a novel methodology for reducing the number of equivalent inputs that Korat can generate. Our development of parallel Korat and the methodology for reducing equivalent inputs are motivated by testing an application developed at Google. The experimental results on running parallel Korat across up to 1024 machines on the Google's infrastructure show that parallel test generation and execution can achieve significant speedup, up to 543.55 times.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

General Terms: Algorithms, Performance, Reliability

Keywords: Parallel testing, Korat, bounded-exhaustive testing, test data generation

1. INTRODUCTION

Software testing is an important part of software development and can account for more than 50% of the development cost [3]. Two main activities in testing are *test generation*, which creates tests to be executed, and *text execution*, which executes the tests to check the code under test. While test execution is often automated and can easily handle a large number of tests, test generation is typically manual and thus tedious and error-prone when generating a large number of tests.

Manual generation of test inputs is particularly hard for code that takes structurally complex inputs. For example, code that manipulates XML documents—e.g., an XPath compiler [40]—effectively takes as input XML parse trees that need to satisfy complex syntactic and semantics constraints for valid XML documents. As another example, a code unit that operates on complex data structures—e.g., a set implemented as a red-black tree—takes inputs that are structural (consisting of objects linked in a tree) and need to satisfy complex invariants (prescribed by the red-black coloring of the tree nodes).

We have previously developed two approaches, TestEra [19, 21, 41] and Korat [4, 29, 30, 33], for automated generation of structurally complex test inputs. Both approaches are constraint-based: they allow the users to manually write only the constraints that describe the desired properties of test inputs, and a tool then automatically generates a large number of such inputs. TestEra uses constraints written in a declarative language [15] and has been applied only by academic researchers [19, 21, 41]. Korat uses constraints written in an imperative language, such as Java or C#, and has been applied in industry [40, 43].

Both TestEra and Korat can generate all test inputs within given small bounds. Such test inputs enable *bounded-exhaustive testing* that checks the code under test exhaustively within given bounds. Such testing previously revealed faults in several real-world applications, including a protocol for dynamic networks [20], a constraint analyzer for a declarative language [21], an XPath compiler [40], and a fault-tree analyzer [41].

This paper focuses on Korat. Given an imperative predicate that specifies the desired structural integrity constraints and a finitization that bounds the desired test input size, Korat generates all test inputs (within the bounds) for which the predicate returns true. Korat performs a systematic search of the predicate's input space and attempts to prune from the search and generation equivalent inputs, which only increase the testing time but do not increase the chance to reveal a fault [46]. The Korat tool for Java is publicly available at `http://mir.cs.uiuc.edu/korat`. (The FSE group at Microsoft Research implemented the Korat algorithm for the AsmL and .NET languages as a part of the SpecExplorer tool publicly available at `http://research.microsoft.com/specexplorer`).

This paper presents two significant improvements of Korat: (1) a set of algorithms for parallel testing and (2) a methodology for gen-

ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.
Copyright 2007 ACM 978-1-59593-811-4/07/0009 ...$5.00.

erating fewer equivalent inputs. The direct motivation for our improvements was testing an application developed at Google. At a high level, the application takes as input a graph that represents links between entities and requires each input graph to be a directed acyclic graph (DAG). To generate test inputs for this application with the Korat tool [33] requires representing DAGs and specifying their properties in Java. A direct use of the previous version of Korat results in sequential testing and generates a large number of equivalent test inputs.

We describe four algorithms that enable parallel test generation and execution using Korat. The algorithms consider the cases when the generation and execution are performed either *off-line* (the generation writes the inputs to the disk, and the execution reads them from the disk) or *on-line* (the generation produces the inputs that the execution immediately consumes). The ability of Korat to efficiently enumerate a lot of inputs requires the user to consider whether (and how) to store the inputs to the disk: even if abundant disk space is available, reading inputs back into the memory may take more time than just re-running Korat and re-generating the inputs. The algorithms also consider the cases when the initial generation is either sequential or parallel, while the subsequent generations and executions are always parallel. When storing all inputs to the disk is impractical, the algorithms only store a small fraction of inputs to speed up subsequent runs of test generation.

We developed our algorithms targeting the Google's computing infrastructure (consisting of large clusters of commodity machines) and its parallel programming model [7]. An important design decision was to minimize the communication between machines, which are called *workers* [7]. As a result, our algorithms communicate only the simple messages to start or stop generation (and execution), notify completion, or initialize state. This approach can improve the overall performance by removing any expensive message passing and also makes the algorithms easily portable. For example, by writing a simple wrapper script for the Google's infrastructure, we directly ported our single-machine implementation to a massively distributed computing environment that prohibits complex messages.

This paper makes the following contributions:

- **Parallel generation:** We present new algorithms for parallel search and test generation in Korat. Intuitively, our algorithms partition the predicate's input space such that each worker explores only its assigned partition.

- **Parallel execution:** We present new algorithms for parallel test execution, in particular when the execution immediately follows Korat's test generation, with no storing of the generated test inputs on the disk. Our algorithms attempt to achieve *load-balancing* by evenly distributing the test execution across all the workers.

- **Methodology for fewer equivalent inputs:** We present a novel methodology that enables Korat to generate a smaller number of equivalent test inputs. Our methodology shows how to encode equivalence of inputs in the input constraints to significantly prune the search space and generate mostly non-equivalent test inputs. This methodology builds on our previous work with TestEra [22].

- **Case study:** We evaluate parallel Korat by testing the Google application in a bounded-exhaustive manner with hundreds of millions of test inputs based on DAGs. Our experiments use the Google's infrastructure to generate and execute test inputs. The results on up to 1024 workers show that parallel Korat can achieve significant speedup, of up to 543.55 times.

```
class DAG {
  DAGNode[] nodes;
  int size;

  static class DAGNode {
    DAGNode[] children;

    boolean repOK(Stack<DAGNode> path, Set<DAGNode> visited) {
      path.push(this);
      for (int i = 0; i < children.length; i++) {
        DAGNode child = children[i];
        // all children must be different
        for (int j = 0; j < i; j++)
          if (child == children[j]) return false;
        // there should be no directed cycle
        if (path.search(child) != -1) return false;
        // no need to visit already visited child
        if (!visited.add(child)) continue;
        // visit all unvisited children
        if (!child.repOK(path, visited)) return false;
      }
      path.pop();
      return true;
    }
  }

  boolean repOK() {
    Set<DAGNode> visited = new HashSet<DAGNode>();
    Stack<DAGNode> path = new Stack<DAGNode>();
    for (DAGNode node : nodes)
      if (visited.add(node))
        if (!node.repOK(path, visited)) return false;
    return size == visited.size();
  }
}
```

Figure 1: Code that represents DAGs and checks their structural constraints

2. BACKGROUND: KORAT

We first describe *what* Korat does and how to apply it to generate and execute test inputs. We then describe *how* Korat works, in particular how it performs sequential search for test generation. (Section 3 presents our new algorithms for parallel search and test execution.) We also discuss equivalent test inputs. (Section 4 presents how to reduce the number of equivalent inputs.)

As our running example, we use generation of directed acyclic graphs (DAGs). Each DAG has a set of nodes and a set of directed edges such that there is no directed cycle along those edges. A DAG can have undirected cycles obtained by viewing edges as undirected. (The term *DAG* is commonly used, but the more precise term would be acyclic directed graphs, i.e., *ADG*.) These seemingly simple structures form the core of test inputs for several real-world applications, including the Google application from our case study and a fault-tree analyzer developed for NASA [34, 41]. To generate actual test inputs, the user can write code that appropriately labels nodes or edges of the DAGs [21, 41] or can directly model test inputs instead of modeling DAGs.

To generate DAGs with Korat, the user first needs to write a representation for DAGs and code that checks structural constraints on this representation. Figure 1 shows one possibility in Java. Each object of the class `DAG` represents a DAG, and each object of the class `DAGNode` represents a node. The field `nodes` stores all nodes of a DAG, and each node has the field `children` that stores the destination nodes of outgoing edges. These fields effectively represent sets, and Section 4 discusses the issues that arise in representing a set with an array.

The two `repOK` methods check the structural constraints of DAGs. Following Liskov [28], we use the name `repOK` for the methods that check the *representation invariant* of data structures used to implement abstract data types. We refer to these methods as *imperative*

```
public static IFinitization finDAG(int numNodes) {
  IFinitization f = new Finitization(DAG.class);
  f.set("size", f.createIntSet(numNodes, numNodes));
  IObjSet nodes = f.createObjSet(DAGNode.class, numNodes);
  f.addAll("nodes", nodes);
  IIntSet arrLen = f.createIntSet(0, numNodes - 1);
  IArraySet childrenArray = f.createArraySet(DAGNode[].class,
                               arrLen, nodes, numNodes);
  f.set("DAGNode.children", childrenArray);
  return f;
}
```

Figure 2: Finitization that bounds the size of generated DAGs

predicates [29]: they are written in an imperative language and return a boolean value.

In our example, the methods take as input an object graph consisting of `DAGNode` objects and check the absence of (directed) cycles. These `repOK` methods use the Tarjan's algorithm for strongly connected components [42] to traverse the graph and return `true` if a given object graph indeed represents a DAG or return `false` otherwise. Writing `repOK` methods is usually easy. Two undergraduate students (the first two authors of this paper) wrote in a matter of hours `repOK` methods for several data structures, including various versions of DAGs: connected or unconnected, labeled or unlabeled, with one root or multiple roots, etc. The specific `repOK` methods in Figure 1 allow unconnected DAGs with multiple roots.

The user also provides a *finitization* that bounds the size of the test inputs that Korat generates. Figure 2 shows sample code that specifies bounds for the `DAG` class. Each finitization bounds the number of objects of a given class (for example, `numNodes` objects of the class `DAGNode`) and the values of fields for the objects (for example, `size` is set to `numNodes`, and the `children` array has length between 0 and `numNodes - 1` and has elements that are from the set of nodes). To specify these bounds, the code uses the classes from the Korat library [33].

2.1 Korat's output

Korat is effectively a constraint-solver for imperative predicates: given an imperative predicate (`repOK`) and a finitization (`finDAG`), Korat generates *all* inputs (within the bounds given in the finitization) for which the predicate returns `true`. We refer to such inputs as *valid inputs* (even though the code might be expected to generate an error message for those inputs). Executing all valid inputs on the code under test is called *bounded-exhaustive testing* and provides a strong guarantee that there is no fault within the given bounds. When time limits prevent bounded-exhaustive testing, one can consider taking a subset of all test inputs that Korat generates.

Korat can save the generated objects on disk or display them graphically. For example, Korat generates four DAGs with exactly three nodes, and Figure 3 shows the visualization for two of these four DAGs. Our current Korat implementation [33] uses the Alloy Analyzer [15] for visualization; Korat automatically translates object graphs into the Alloy representation.

2.2 Non-equivalent inputs

Korat does not actually generate *all* valid object graphs but only *non-isomorphic* object graphs. Two object graphs are isomorphic if they differ only in the identity of the objects in the graphs [4,14,29]: isomorphic object graphs have the same branching structure (same shape) and the same values for primitive fields. Arrays are viewed as objects with one field labeled `length` and the other "fields" for array elements labeled with array indexes. For example, we obtain isomorphic graphs if we swap the identity of some nodes in the left DAG in Figure 3, say put `DAGNode2` in `children[0]` and `DAGNode1` in `children[1]`. However, the four graphs (two of which are shown) are themselves all non-isomorphic as they have different shapes.

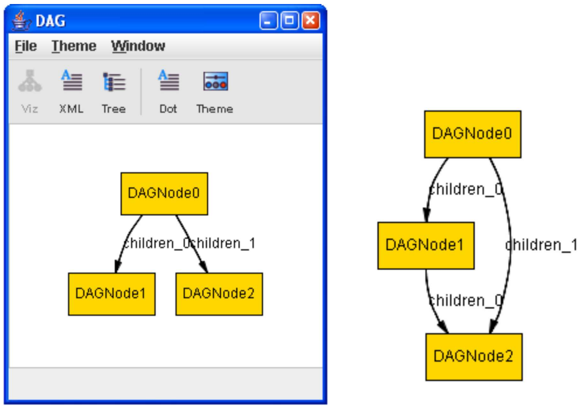

Figure 3: Two DAGs of four that Korat generates for three nodes; the left DAG shows an entire visualization window with several customization options

Isomorphic object graphs form equivalent test inputs, i.e., two isomorphic object graphs either both reveal a fault or none reveals a fault [45]. Testing code with more than one representative from an equivalence class only increases the testing time but cannot reveal more faults. Hence, we want to avoid isomorphic object graphs among the generated graphs. Korat can efficiently generate all non-isomorphic objects graphs (i.e., exactly one representative from each isomorphism class) at the concrete level [4, 29]. For DAGs, however, the arrays used at the concrete level represent sets at the abstract level, and several object graphs that Korat generates can be non-isomorphic at the concrete level but correspond to the same DAG at the abstract level. Section 4 presents a methodology for reducing the number of generated equivalent structures in such cases.

2.3 Korat's search

We briefly describe how Korat generates non-isomorphic valid inputs. We present only the parts of Korat necessary to introduce parallel Korat and our methodology for generating fewer equivalent inputs; more details of Korat are available elsewhere [4, 29]. Korat implements a backtracking search algorithm that systematically explores the space of the predicate's inputs, bounded by the finitization. Korat efficiently prunes search through the space to generate only the non-isomorphic valid inputs.

The main entity in Korat's search is *candidate vector* (which we also call simply *candidate* or *vector*). Each candidate vector encodes an object graph that may be valid or invalid (i.e., for which `repOK` may return `true` or `false`). Korat represents a candidate vector as a sequence of integers. These integers are indexes into the possible values for each field in the finitization. For example, consider the (implicit) field `length` of the `children` arrays in `DAGNode` objects. The `finDAG` finitization specifies that the possible values for the length can vary between 0 and `numNodes - 1`. It also specifies that there are `numNodes` such objects. Thus, each candidate vector encodes specific length value for each array. When `numNodes=`N, there are N arrays, each of which can have N values for length, so the space of all combinations of array lengths is N^N. (The candidate vectors for DAGs encode not only array lengths but also array elements and the value of the field `size`.)

Korat starts its search from the initial candidate vector that has all zeroes. This vector encodes that all fields take their first possible value. In our example with array lengths, it would represent that all arrays have length 0.

Korat generates candidate vectors in a loop until it traverses the entire space of the predicate's inputs. The basic idea is to execute the `repOK` predicate on a candidate to determine (1) whether the candidate is valid or not, and (2) what next candidate to try out. Korat executes the actual `repOK` code on concrete object graphs. (In particular, Korat does not use symbolic execution [23].) Therefore, for each candidate vector, Korat first builds a corresponding object graph, i.e., links the objects (in our example nodes and arrays) and sets the values of their fields. (A candidate vector is a sequence of integers, but the code can execute only on object graphs.) If `repOK` returns `true`, the object graph is valid and Korat outputs it. Regardless of whether `repOK` returns `true` or `false`, Korat continues its search with the next candidate.

During the execution of `repOK`, Korat monitors field accesses and builds a *field-access stack* [4]. Korat builds this stack to prune the search. Indeed, the key idea in Korat is to generate the next candidate vector based on the execution of the previous candidate vector. The intuition is that if `repOK` returns `false` by accessing only some fields of the object graph, then it would return `false` for any values of the non-accessed fields. Therefore, Korat backtracks the search based on the last accessed field on the stack. Korat increments the integer value of the appropriate index to generate the next candidate vector. To eliminate exploration of isomorphic structures, Korat increments an index by more than one when possible [4]. We show in Section 3.2.3 how one of our algorithms for parallelization makes an even bigger "jump" on the field-access stack.

3. PARALLEL KORAT

We describe four algorithms that enable parallel test generation and execution using Korat. Our algorithms target clusters of commodity machines and attempt to minimize inter-machine communication. This approach not only can improve the overall performance by removing any expensive message passing but also makes the code easily portable, for example to a massively distributed computing environment that prohibits complex messages such as the Google's computing infrastructure [7].

We first discuss the properties of Korat's search that make it easy or hard to parallelize. We then present the basic versions of the algorithms, which store candidate vectors regardless of their validity. We finally present some potential optimizations, which can for example store only those candidate vectors that are valid.

3.1 Properties

The key property of Korat's search that enables its parallelization is that the current candidate vector *compactly* encodes the *entire* search state, i.e., both the part of the state space that has been explored and the part of the state space that has yet to be explored. We can thus parallelize the search by using candidate vectors as the bounds for the ranges that split the state space. This is unlike say explicit-state model checking [8, 27, 39] where the current search state is encoded in a large graph of all visited states, and it is unclear how to parallelize the work among several workers such that they do not communicate and do not do any overlapping work. The fact that Korat's search state is encoded in the candidate vector also allows for a great deal of fault tolerance. If some worker machine goes down after exploring the state up to some candidate vector, another worker machine can finish the search by simply starting from that candidate vector.

The main challenge in parallelizing Korat's search is due to the pruning that search performs. While pruning makes the search more efficient, it also makes it mostly sequential: to determine the next candidate vector for the search, Korat needs to execute `repOK` on the current candidate vector. In other words, given an arbitrary candidate vector, we cannot tell a-priori (before actually performing the search) whether the search would explore that vector or not. The vector may not be explored because `repOK` returns `false` for some previous vector or because the search explores another isomorphic vector. This means that we cannot purely randomly choose candidate vectors to partition the state space, since a randomly chosen vector may not be explored at all.

3.2 Algorithms

The ability of Korat to efficiently enumerate large numbers of test inputs requires a consideration of whether (and how) to store these inputs to disk. Even if abundant disk space is available, serializing all inputs to disk during generation and then deserializing them back into the memory during execution may take more time than just re-running Korat and re-generating the inputs.

When storing all inputs to disk is infeasible, an algorithm may still feasibly store a small fraction of them, especially if doing so speeds up the future runs of (the same) test generation. Such an algorithm, which amortizes its cost, may start with a sequential run before switching over to a parallel mode for future runs, or it may have all of its runs including the first one in a parallel mode.

Two of our four algorithms assume that storing all inputs is advantageous, while the other two assume that it is not. We refer to the algorithms that store the inputs as *off-line* (in notation *OFF*) since storing decouples test generation and execution into separate phases. We refer to the algorithms that do not store the inputs as *on-line* (in notation *ON*) since they run test generation immediately followed by execution of the code under test.

Two of our four algorithms have the initial generation sequential, while the other two have the initial generation parallel. We use the notation *SEQ* and *PAR* for the algorithms with the initial sequential and parallel run, respectively. All four algorithms have subsequent generations parallel.

3.2.1 Algorithm SEQ-OFF

This algorithm runs test generation sequentially, storing the generated inputs to disk, and then parallelizes test execution: (1) run Korat on one machine to generate all inputs and store them to disk; (2) distribute the inputs evenly across several worker machines to execute the code under test.

The algorithm attempts to load-balance the work across all workers during test execution. However, test generation remains sequential. Although test generation does not have to fully complete before test execution can begin, this algorithm is mostly suitable for situations when test generation and execution are separated. For example, test generation may produce a small number of inputs or may require a lot of search to produce the inputs (so it is preferred to save them), and the `repOK` methods do not change often (so it is not necessary to re-run Korat).

3.2.2 Algorithm SEQ-ON

This algorithm assumes that storing all inputs is not preferred, starts test generation with a sequential run and then switches over to a parallel mode for future runs. Test execution immediately follows test generation.

The design goal for this algorithm is that it stores sufficient information during the first, sequential run of test generation such that all future runs of test generation can be done in parallel and load-balanced. More precisely, for n workers, we want to obtain a sequence of candidate vectors $\langle v_1, \ldots, v_n \rangle$ that are *equi-distant*, i.e., Korat explores the same number of candidate vectors for any range

```
// input: m is the maximum number of workers
// output: an array of equi-distant candidates,
//         with the array length between m and 2 * m
Candidate[] equiDistantCandidates(int m) {
  Candidate[] candidates = new Candidate[2 * m];
  int distance = 1;
  int index = 0;
  while (Korat.hasNext()) {
    for (int i = 0; i < distance; i++) {
      candidates[index] = Korat.next();
      if (!Korat.hasNext()) break;
    }
    index++;
    if (index == candidates.length) {
      // half the array and double the distance
      for (int j = 0; j < candidates.length / 2; j++)
        candidates[j] = candidates[2 * j + 1];
      distance = distance * 2;
      index = m;
    }
  }
  // resize the output length to valid indexes
  Candidate[] result = new Candidate[index];
  for (int i = 0; i < index; i++)
    result[i] = candidates[i];
  return result;
}

// inputs: w is the actual number of workers;
//         candidates is from equiDistantCandidates
// output: an array of (mostly) equi-distant candidates
//         selected from the given input,
//         with the array length exactly w
Candidate[] selectCandidates(int w, Candidate[] candidates) {
  Candidate[] result = new Candidate[w];
  for (int i = 0; i < w; i++)
    result[i] = candidates[(i + 1) * candidates.length / w - 1];
  return result;
}
```

Figure 4: Equi-distancing for SEQ-ON

$[v_i, v_{i+1})$ (for $i < n$), and the union of all ranges (i.e., $[v_1, v_n]$) covers the entire search space. Note that the algorithm does not need to know in advance the exact number of workers on which the Korat will be run in parallel; it suffices to know the upper bound for this number. For example, building 1024 equi-distant candidates (for a maximum of 1024 workers) allows running the search on say 16 workers by giving each to explore 64 ranges.

A challenge in this algorithm is that we do not know a-priori (before the search) the total number of candidate vectors that Korat will explore. Call this number C. A simple two-pass algorithm can compute C and then the desired sequence of candidates: first run Korat once to count C, and then run Korat again to store to disk every k^{th} candidate, where $k = \lfloor \frac{C}{n} \rfloor$. However, this simple algorithm requires two complete executions of Korat, each of which can be expensive. We have thus developed the SEQ-ON algorithm that requires only one sequential pass.

Figure 4 shows the pseudo-code of the algorithm. The function `equiDistantCandidates` maintains an array of candidate vectors, with the array length being twice the number of maximum workers. Since the number of candidates is not known a-priori, the algorithm initially stores in the array every candidate that Korat explores. Once the array reaches its capacity, the algorithm (1) moves the candidates at even indexes in the array to its lower half (while preserving the relative ordering among candidates); and (2) starts storing every second candidate in the upper half of the array. The next time the array reaches its capacity, the algorithm performs the previous steps (halves the array and doubles the distance) and starts storing every fourth candidate, and so on. At the end, the function returns the candidates to be stored for future parallel runs.

```
// input: w is the actual number of workers
// params: MAX_STEPS is maximum number of Korat steps to perform
//         MAX_FIELDS is maximum number of fields to truncate
Candidate[] randomFastForward(int w) {
  Candidate[] candidates = new Candidate[w];
  int currW = 0;
  while (currW < w) { // restart search from the beginning
    Candidate candidate = initialCandidate(); // all 0's
    while (currW < w) { // fast-forward within one search
      // action 1: perform a number of actual Korat steps
      int steps = random.nextInt(MAX_STEPS);
      while (Korat.hasNext() && steps > 0) {
        candidate = Korat.next();
        steps--;
      }
      if (!Korat.hasNext()) break;
      // action 2: jump ahead in the search
      Korat.shortenAccessStack(random.nextInt(MAX_FIELDS));
      candidate = Korat.next();
      candidates[currW++] = candidate;
    }
  }
  // sort them since they could be from various restarts
  return sort(candidates);
}
```

Figure 5: Fast-forwarding for PAR-OFF

3.2.3 Algorithm PAR-OFF

This algorithm parallelizes the initial run of Korat and writes all generated inputs to disk. It parallelizes test execution the same way that SEQ-OFF does.

An optimal parallelization of Korat's search would partition the space of all candidate vectors into several ranges such that they together cover the entire search space and the exploration of each range takes about the same time. Recall that Korat prunes its search based on the accessed fields, as determined dynamically by the executions of `repOK`. Therefore, it is hard to estimate before the search how many candidates Korat will explore between two given candidates: this number depends on pruning, which in turns requires reasoning about the (dynamic) behavior of `repOK`.

Our algorithm uses randomization. It randomly *fast-forwards* the Korat search on one machine to select a given number of initial candidate vectors and writes these vectors onto the disk. It then parallelizes the search for these randomly selected candidates. The design goal for fast-forwarding is that it executes much faster than the full search and yet produces mostly equi-distant candidates. Figure 5 shows the pseudo-code of our fast-forwarding algorithm. It starts the search from the initial Korat candidate vector (which is all zeroes). It then repeats two actions—(1) performing a randomly selected number of actual steps of Korat search (which build field-access stack as explained in Section 2.3) followed by (2) "jumping" in the search space by truncating the field-access stack for a randomly selected number of fields—until it builds the required number of candidates or fast-forwards through the entire search space. If it finishes the entire space, it starts the search over.

This algorithm always generates candidates that the Korat search would explore (i.e., not prune due to accesses or isomorphism): the action 2 of this algorithm is a step that Korat would perform during the backtracking. Moreover, this algorithm guarantees that the workers explore each candidate exactly once (there is no overlap of work between workers and together they cover the entire search space). However, the load-balancing of test generation depends on the initial set of randomly chosen candidate vectors. While it is unlikely that they are evenly apart, it is also unlikely that they cause only one worker to do almost all of the generation. Test execution can be load-balanced by evenly dividing the generated inputs to all available workers as for SEQ-OFF.

3.2.4 *Algorithm PAR-ON*

This algorithm assumes that storing all inputs to disk is not preferred and makes all its runs in the parallel mode. It parallelizes test execution in the online mode, the same way that SEQ-ON does.

This algorithm combines the advantage of PAR-OFF(using randomly chosen candidate vectors to run the initial generation in parallel) with the advantage of SEQ-ON(using equi-distant candidate vectors to load-balance the subsequent generation). PAR-ON proceeds as follows. Like PAR-OFF, PAR-ON first randomly selects n candidate vectors $\langle r_1, \ldots, r_n \rangle$ and then runs in parallel search over the ranges between the consecutive candidates. Additionally, during this first generation run, PAR-ON creates n equi-distant candidates $v_1^i, \ldots, v_n^i$ for each range $[r_i, r_{i+1})$, effectively splitting each range into n subranges. In the post-processing of the first run, PAR-ON appropriately merges the subranges generated for different ranges. Therefore, in the subsequent generation runs, PAR-ON operates like SEQ-ON, achieving better load-balancing among workers than in the first generation run.

3.3 Potential optimizations

We have so far described the basic parallel algorithms for online execution (ON). These basic algorithms store the appropriate candidate vectors, either equi-distant or randomly selected. However, it is not necessary to store the candidate vectors for the portions of the state space with no valid test inputs. For example, consider a range of candidate vectors $[v_i, v_{i+1})$ and suppose that the first and the last *valid* candidate vectors are v_i' and v_i'', respectively. It holds that $v_i < v_i' < v_i'' < v_{i+1}$. Therefore, the algorithms need only to store the range $[v_i', v_i'']$ and search through it in the subsequent searches; it is known that there are no valid candidates in $[v_i, v_i')$ or $(v_i'', v_{i+1}]$.

The algorithms could use the number of *valid* candidate vectors, instead of all candidate vectors, to partition the input search, effectively using "*equi-valid-distant*" ranges. However, valid vectors are often clustered around certain points in the search space. Therefore, searches that generate the same number of valid vectors could explore vastly different number of candidate vectors, which could result in reduced load-balancing among the workers.

Moreover, the algorithms do not need to use the number of candidate vectors or the number of valid candidate vectors as a measure to partition the space. Ideally, the partitioning should be done based on the actual running time. When the workers on which the algorithms run have nearly uniform speed (which is not often the case when using large clusters of commodity machines such as at Google), the algorithms could measure the actual run time in one run and use it to improve load-balancing of the subsequent runs.

Finally, we can consider a parallel environment with cheaper inter-worker communication, which would allow the workers to exchange more complex messages and to dynamically partition the search space. We plan to investigate this option in the future.

4. FEWER EQUIVALENT INPUTS

We next present a methodology for rewriting `repOK` methods such that Korat generates fewer equivalent test inputs. The definition of equivalent inputs is up to the user of Korat: the user effectively deems that two inputs either can both reveal a fault or none of them reveals a fault and thus wants to execute the code under test for only one of such inputs. In our case study with the Google application, we consider equivalence of DAGs. Our methodology is important for this case study but is independent of parallel Korat. The methodology requires changing `repOK` methods, but the same parallel algorithms can handle changed `repOK` methods. Also, the methodology is applicable even for sequential Korat. We first present an overview of the methodology, then show an example, and finally discuss limitations of the methodology.

```
class DAG {
  DAGNode[] nodes;
  int size;

  static class DAGNode {
    DAGNode[] children;

    public int repOKD(Stack<DAGNode> path, Set<DAGNode> visited,
                      HashMap<DAGNode, Integer> descendants) {
      path.push(this);
      int maxD = Integer.MAX_VALUE;
      int thisD = 1;
      for (int i = 0; i < children.length; i++) {
        DAGNode child = children[i];
        for (int j = 0; j < i; j++)
          if (child == children[j]) return -1;
        if (path.search(child) != -1) return -1;
        int childD;
        if (!visited.add(child)) {
          childD = descendants.get(child);
        } else {
          childD = child.repOKD(path, visited, descendants);
          if (childD == -1) return -1;
        }
        if (childD > maxD) return -1;
        maxD = childD;
        thisD += childD;
      }
      path.pop();
      descendants.put(this, thisD);
      return thisD;
    }
  }

  boolean repOKD() {
    Set<DAGNode> visited = new HashSet<DAGNode>();
    Stack<DAGNode> path = new Stack<DAGNode>();
    HashMap<DAGNode, Integer> descendants =
                             new HashMap<DAGNode, Integer>();
    for (DAGNode node : nodes)
      if (visited.add(node))
        if (node.repOKD(path, visited, descendants) == -1)
          return false;
    return size == visited.size();
  }
}
```

Figure 6: Modified repOK method that enables pruning based on the number of descendants

4.1 Overview

The main idea of our methodology is to *add more checks* to `repOK` such that Korat prunes some inputs that it would not prune by default. This idea originates from the *symmetry-breaking predicates* [1,6,22] developed for search problems over declarative predicates. Recall, however, that Korat uses imperative predicates; our anecdotal experience suggests that practicing testers find it easier to write imperative predicates than declarative predicates. Our methodology suggests how users should write additional checks to make test generation with Korat faster and to produce fewer inputs, thus also making subsequent test execution faster.

We adapt symmetry-breaking predicates for Korat as follows. The user needs to *identify a partial order* among candidate vectors such that vectors "larger" according to the order do not need to be considered by Korat since they are not the representatives of their equivalence classes. The default order that Korat uses is based on the lexicographic order of vectors and breaks isomorphisms at the concrete level of object graphs [4]. We proved that this pruning is optimal; Korat generates exactly one representative for each isomorphic class [29]. However, the user may want to consider equivalences other than isomorphism of concrete object graphs.

	`repOKB`			`repOKC`			`repOKD`			
size	time [s]	explored	generated	time [s]	explored	generated	time [s]	explored	generated	non-isom.
1	0.61	1	1	0.61	1	1	0.59	1	1	1
2	0.60	5	2	0.60	5	2	0.60	5	2	2
3	0.60	68	8	0.61	68	7	0.60	68	7	6
4	0.68	1,518	95	0.67	1,418	57	0.67	1,394	48	31
5	1.39	74,902	4,858	1.15	54,113	1,541	1.07	44,888	691	302
6	213.26	16,650,503	1,336,729	75.07	6,255,988	185,569	30.48	2,628,140	21,430	5,984
7	> 3 hrs	-	-	> 3 hrs	-	-	4,593.13	313,006,096	1,468,397	243,668

Figure 7: Comparison of Korat's generation for different predicates; for size 7, it takes more than 3 hours for repOKB and repOKC

4.2 Example

We illustrate the methodology by showing how to reduce the number of equivalent inputs for DAGs shown in Figure 1 and discussed in Section 2. We refer to the `repOK` method from Figure 1 as `repOKB` (`B` for "Basic"). We want to generate only non-isomorphic DAGs; we consider that the code under test has equivalent behavior for any two isomorphic DAGs. Recall that `DAG` and `DAGNode` objects represent sets of children with arrays. Korat performs its search at the concrete level of arrays, not at the abstract level of sets. Thus, Korat can generate two or more different arrays even when they represent the same set, and so Korat can generate two or more DAGs that differ at the concrete level but represent isomorphic DAGs at the abstract level. To reduce the number of such arrays that represent the same set (and the number of concrete DAGs that represent isomorphic abstract DAGs), we can modify `repOK` to require that the arrays be ordered such that "larger" arrays are definitely not representatives of their equivalence classes.

The first order that we describe for arrays is by the number of (immediate) *children* of the array elements. This requires a new method `repOKC` (`C` for "Children") that makes a small change in the `repOKB`, replacing the line

```
if (child == children[j]) return false;
```

with

```
if (child == children[j] ||
    child.children.length < children[j].children.length)
  return false;
```

The extra check for the length/size of the children eliminates from the Korat search (and thus from generation) those arrays where "later" elements have more children than "earlier" elements. For example, consider two arrays, each with two elements, where the lengths of children in the first array are 0 and 1, and the lengths of children in the second array are 1 and 0. At the concrete level, these arrays are different. However, they both represent the same set (more precisely, belong to the same equivalence class) and thus only one need to be explored to generate all non-equivalent DAGs.

The second order that we describe for arrays is by the number of (total) *descendants* of the array elements. Figure 6 shows the new `repOKD` method (`D` for "Descendants"), which differs from the original `repOKB` method in three points. First, this method takes an extra argument `descendants` that maps each node to its number of descendants. Second, this method returns an integer that is the number of descendants, not just a boolean value. (The integer -1 encodes that the structural constraints were not satisfied and correspond to `false`.) Third, this method computes the number of descendants while checking the structural constraints. Specifically, `thisD` keeps the number of descendants for the current node, and `childD` is the number of descendants for the current child. If the values of `childD` are not sorted, i.e., `childD > maxD`, the method prunes the candidate.

Figure 7 shows the experimental results that compare the three methods: `repOKB` (basic), `repOKC` (with additional checks based on children), and `repOKD` (with additional checks based on descendants). More complex methods such as `repOKD` clearly require more work from the user. However, they significantly reduce the number of explored and generated structures, the time for search and generation, and subsequently the time for test execution.

We used the tool called *nauty* [31] to count the number of non-isomorphic DAGs among those generated by Korat for the three `repOK` methods. All three methods give the same number of non-isomorphic DAGs, and these numbers appear in the Sloane's Encyclopedia of Integer Sequences [37], increasing our confidence in the correctness of these `repOK` methods. (Since `repOK` methods are manually written, they could have errors themselves, especially when adding symmetry-breaking predicates.) The DAGs actually used for our testing at Google have additional properties, including that each component must have at least two nodes, and their numbers do not appear in the Sloane's Encyclopedia.

4.3 Limitations

In theory, our methodology for generating fewer equivalent inputs should be general enough to support any equivalence relation. In practice, all the examples we have worked on use equivalences based on graph isomorphism. In particular, our example with DAGs uses the isomorphism of DAGs viewed at the abstract level as pairs of nodes and edges. While this equivalence is coarser then the Korat's default equivalence based on the isomorphism of concrete object graphs, they are still both based on graph isomorphism.

The question is how well would the methodology work if we had a completely different equivalence relation. For example, for testing some other code, we could define two DAGs to be equivalent if they had the same number of edges, regardless of the graph structure. In such case, reducing the number of generated inputs would require more extensive changes to the `repOK` methods. Indeed, the main question is how hard it is to manually add symmetry-breaking predicates. We plan to investigate this further in the future. We point out, however, that it is not necessary to generate only non-equivalent inputs: even `repOKD` generates some equivalent inputs, e.g., for size 7 it generates over 1.2 million equivalent inputs, six times more than the number of non-equivalent inputs. The equivalent inputs increase test generation and execution times.

5. EVALUATION

We present an evaluation of our two base algorithms for parallel search in Korat, SEQ-ON and PAR-OFF. (SEQ-OFF runs Korat sequentially and then parallelizes only test execution, while PAR-ON combines SEQ-ON and PAR-OFF.) Our evaluation uses a Google application that takes as input a graph whose nodes are a set of entities and whose edges are links among entities. The application

num. of	total	worker time [s]			speedup
workers	time [s]	max.	avg.	min.	ratio
1	569	555	555	555	1.00
2	322	308	293	279	1.77
4	166	151	144	134	3.42
8	95	80	72	66	5.99
16	56	40	37	34	10.16
32	38	21	20	17	14.97
64	32	12	10	9	17.78
128	28	7	6	5	20.32
256	32	5	3	3	17.78
512	47	5	2	2	12.11
1024	74	4	2	1	7.69

Figure 8: Speedup of running time for DAGs of size 7

num. of	total	worker time [s]			speedup
workers	time [s]	max.	avg.	min.	ratio
1	129383	129365	129365	129365	1.00
2	67301	67282	66593	65904	1.92
4	34239	34220	34115	31874	3.78
8	18199	18180	16519	15080	7.11
16	9534	9514	8384	7820	13.57
32	5062	5041	4248	3875	25.55
64	2383	2359	2100	1894	54.27
128	1263	1238	1056	916	102.42
256	658	627	529	451	196.60
512	366	320	266	228	353.46
1024	238	164	132	111	543.55

Figure 9: Speedup of running time for DAGs of size 8

requires that the input graph be a directed acyclic graph (DAG). We therefore perform our evaluation using Korat for generation of DAGs. Figure 1 shows example classes that represent DAGs, and Section 4 discusses various `repOK` methods for that representation.

Our experimental setup uses *bounded-exhaustive generation* of test inputs and *differential testing* [32] as a test oracle. The application computes certain reachability properties among the nodes of the input DAG. The actual code under test is written in C++ and implements a highly optimized computation of the properties. We additionally use a simple but slow implementation of the same computation written in Java. We provide each input to both implementations and compare their outputs. Moreover, we do this for all inputs that Korat generates within the given bounds.

5.1 Equi-distancing in SEQ-ON

We measured the running time of parallel Korat using SEQ-ON on the Google's infrastructure [7]. We first prepared candidate vectors using the equi-distancing algorithm (Figure 4) with $m = 8192$ and then ran generation and execution of test inputs on $w = 1, 2, 4, \ldots, 512, 1024$ worker machines on a moderately loaded cluster. Our experiments assess the speedup that parallel Korat provides as a function of the number of workers.

We consider the total running time, which includes overhead time for transferring data and files to the workers, scheduling, resource allocation, and fetching the results from the workers. To account for resource allocation time, we separately operated the scheduling system in two modes: in the first mode, we scheduled all workers to start their tasks simultaneously by waiting for the slowest worker to have its resources allocated; in the second mode, each worker started its task as soon as its resources had been allocated. In theory, the second mode should finish the overall execution faster, but in practice we did not observe any significant improvement over the first mode. Hence, resource allocation time was near constant across the workers.

Our experimental setup for transferring files to and from the workers was as follows. All inputs files (C++ binary under test, Java jar files for Korat and differential implementation, and input candidate vectors) were first transferred to Google File System (GFS) [9] that served as the data store and as the distributor of files to the workers. This transfer was computationally inexpensive one-time operation, independent of the number of workers. Then, a launcher program scheduled the tasks and allocated the resources for the workers. As mentioned, relatively small resource allocation on moderately used cluster was performed in time almost independent of number of machines. Next, each worker was fetching input files from GFS, and to minimize the load on GFS master, workers were reading from GFS replicated files. Also, to take advantage of locality, input files were stored on the machines in the same cluster. In our experiments, transfer time from GFS to workers experienced linear increase in the number of workers. Finally, workers' output files and logs (basically stating that the code did not fail any test) were transferred back to GFS, and the time for this transfer was negligible. It is worth noting that for certain problem sizes, the total overhead time is non-trivial, and increasing the number of workers actually hurts the performance.

Figures 8 and 9 show the experimental results for DAGs of size 7 and 8 nodes, respectively. For a range of the number of worker machines, we tabulate the total running time, the actual test generation and execution time for workers, and the speedup obtained by using a different number of workers. Total running time includes overhead time in addition to the (maximum) actual worker time to (1) generate test inputs (DAGs), (2) execute each of them on both implementations of the computation under test (the highly optimized C++ one and the non-optimized Java one), and (3) compare the results from both implementations. We list maximum, average, and minimum worker time. This time varies among the workers (sometimes almost 50%) due to the different complexities of generation and execution of test inputs among the workers.

Figure 8 shows the results for size 7. While per-worker time experienced steady, almost linear decrease in the number of workers, the overhead cost soon became the dominant term in the total running time and for large numbers of workers can reverse the benefit of increasing the number of workers. The overhead cost was mainly due to slower distribution of input files from GFS to workers as the number of workers was increasing. Using GFS replication alleviated slower distribution but only to some extent. As in many other parallel applications, the benefits of using too many workers can become overshadowed by the high overhead costs. For our application and size 7, for example, running parallel Korat on 1024 workers is slower than running it on 16 workers.

Figure 9 shows the results for size 8. This is a non-trivial problem for a single machine as 3.4 GHz desktop would take almost two days to generate and test more than 200 million input DAGs. (During the generation of this many DAGs, Korat explores more than 4 billion candidate vectors.) For this size, the benefits of using up to 1024 workers far outweigh the overhead costs. We repeated experiments for number of machines varying from 1 to 1024 and recorded the relative speedup with parallelization over 1 machine taken as the baseline.

5.2 Fast-forwarding in PAR-OFF

We also performed an experiment to measure quality of random selection in PAR-OFF. Recall that fast-forwarding randomly selects a given number of candidate vectors to perform the initial parallel

num. of	range ratio		
workers	min.	avg.	max.
1	1.00	1.00	1.00
2	1.00	1.00	1.00
4	1.00	1.01	1.01
8	1.03	1.15	1.20
16	1.25	1.41	1.79
32	2.40	3.65	6.89
64	7.90	7.93	8.35
128	7.90	7.94	8.34
256	7.90	7.98	9.14
512	7.90	8.04	9.20
1024	7.91	8.08	10.89

Figure 10: Potential speedup with fast-forwarding

search in PAR-OFF. Like our other algorithms, PAR-OFF guarantees that the workers explore each candidate vector exactly once, but the speedup in PAR-OFF depends on the initial set of randomly chosen candidate vectors.

Figure 10 summarizes the results of our experiment for DAGs of size 7. We ran fast-forwarding for 50 different seeds. For each seed, we randomly select a number of candidates (equal to the number of workers), measure the range of candidate vectors between these selected candidate vectors, and compute the maximum of these ranges. We then compute the potential speedup of the initial PAR-OFF run over a sequential run by dividing the total number of explored candidate vectors with the maximum range: while a sequential run needs to explore all candidates to finish, PAR-OFF finishes when the largest range is explored. Note that this computation only measures the number of candidate vectors and not the actual running time.

We tabulate the results for a number of workers. The potential speedup increases as the number of workers increase up to 64 and then plateaus after that. (Notice that, for the same size 7, PAR-OFF plateaus for the number of workers similar as the number of workers for which SEQ-ON plateaus.) The variance of potential speedup across random seeds is fairly small for any of workers (except 32), showing that fast-forwarding can relatively well choose random candidate vectors for the initial parallel run of PAR-OFF.

6. RELATED WORK

While there is a plethora of research on parallel search algorithms [11, 16, 18], their use in state-space search for model checking and testing of programs is relatively new. Stern and Dill's parallel Murϕ [38] was among the first tools to parallelize a general purpose model checker. The parallel Murϕ search algorithm maintains a shared set of visited states to prevent workers from exploring the same states. Maintaining this set requires expensive inter-worker communication and influences the scalability of the algorithm. Lerda and Visser [27] presented a similar technique for parallelizing the Java PathFinder model checker (JPF) [44].

Lerda and Sisto [26] implemented a parallel version of the SPIN model checker [12] to check for safety properties. Barnat at al. [2] extended this work to support model checking for linear temporal logic (LTL) but require the nested depth-first search to be sequentially scheduled.

Kumar and Mercer [25] proposed an algorithm for dynamic load-balancing to achieve even workload among workers. However, the inter-worker communication overhead still remains significant. Jones and Sorber [17] proposed a randomized algorithm for checking LTL violations, implemented as an extension to Murϕ [38]. This extension improved the time to find violations and is suited to run in an environment with non-dedicated workers, but it does not exhaustively explore the state space. Palmer and Gopalakrishnan [35] apply partial order reduction to parallel model-checking; by reducing the size of the state space, they also reduce communication among workers.

Dwyer et al. [8] have recently developed the Parallel Randomized State-space Search (PRSS) for the JPF [44]. PRSS requires minimal communication among workers. It runs the model checker on different workers using different randomization seeds, which allows the workers to explore the state space in different orders. Experimental results show that PRSS can give significant speedups in time to find first error. However, when no errors are found, each worker in PRSS explores the entire state space. A key difference between PRSS and parallel Korat is that our algorithms have no overlap among the explorations done by distinct workers, while still maintaining low inter-worker communication. All four of our algorithms also ensure that test execution on distinct workers execute disjoint sets of inputs.

The idea of using constraints to represent inputs dates back at least three decades [5, 13, 23, 36] and was implemented in various tools, including EFFIGY [23], TEGTGEN [24], and INKA [10]. But the focus of prior work was on solving constraints on primitive data and not on solving constraints on complex structures, which requires very different constraint solving techniques. Korat [4, 29] and TestEra [19, 21] are among the first frameworks to support constraint-based generation of complex structures.

7. CONCLUSIONS

We have presented two significant improvements of Korat: (1) a set of algorithms for parallel testing, and (2) a methodology for generating fewer equivalent inputs. Korat is a constraint-based test-generation tool that performs a systematic search to generate all valid test inputs (within the bounds). Our algorithms for parallel testing split generation and execution across a number of worker machines, both off-line (when the inputs are saved on disk) and on-line (when execution immediately follows generation). We also describe a novel methodology for reducing the number of equivalent inputs that Korat generates. These improvements were motivated by testing an application developed at Google. The experimental results on running parallel Korat across up to 1024 computers on the Google's infrastructure show that test generation and execution can achieve significant speedup.

In the future, we plan to investigate parallelization of Korat for systems that have cheaper inter-worker communication (e.g., multi-core processors or dedicated clusters) and to evaluate parallel Korat on more applications. We also plan to investigate parallel versions of other testing algorithms. With the increasingly available multi-core processors and large clusters of machines (such as the Google's distributed computing infrastructure), we believe it is important to study how to exploit that computational power to improve testing tools, not only for testing parallel programs but also for testing sequential programs.

Acknowledgments

We thank Jeff Erickson for his comments on the algorithm for equidistancing; Dragan Milićev for discussions on the initial design of the current Korat implementation; and Brett Daniel, Jesus DeLaTorre, and Choonghwan Lee for their comments on Korat. This material is based upon work partially supported by the NSF under Grant Nos. IIS-0438967, CNS-0613665, and CNS-0615372. We also acknowledge support from Microsoft Research.

8. REFERENCES

[1] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Solving difficult SAT instances in the presence of symmetry. In *Proc. of the 39th Conference on Design Automation*, 2002.

[2] J. Barnat, L. Brim, and J. Chaloupka. Parallel breadth-first search ltl model-checking. In *Proc. of the 18th IEEE International Conference on Automated Software Engineering*, 2003.

[3] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, July 2002.

[5] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, Sept. 1976.

[6] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proc. Fifth International Conference on Principles of Knowledge Representation and Reasoning*, 1996.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, Dec. 2004.

[8] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *Proc. of the 29th International Conference on Software Engineering (ICSE)*, Minneapolis, MN, May 2007.

[9] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *19th Symposium on Operating Systems Principles*, Lake George, NY, 2003.

[10] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, Clearwater Beach, FL, 1998.

[11] A. Grama and V. Kumar. State of the art in parallel search techniques for discrete optimization problems. *IEEE Trans. Knowl. Data Eng.*, 11(1):28–35, 1999.

[12] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[13] J. C. Huang. An approach to program testing. *ACM Computing Surveys*, 7(3), 1975.

[14] R. Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *Proc. of the 16th Conference on Automated Software Engineering (ASE)*, Nov. 2001.

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

[16] V. K. Janakiram, D. P. Agrawal, and R. Mehrotra. A randomized parallel backtracking algorithm. *IEEE Trans. Comput.*, 37(12):1665–1676, 1988.

[17] M. D. Jones and J. Sorber. Parallel search for ltl violations. *Int. J. Softw. Tools Technol. Transf.*, 7(1):31–42, 2005.

[18] R. M. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *J. ACM*, 40(3):765–789, 1993.

[19] S. Khurshid. *Generating Structurally Complex Tests from Declarative Constraints*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2003.

[20] S. Khurshid and D. Marinov. Checking Java implementation of a naming architecture using TestEra. In S. D. Stoller and W. Visser, editors, *Electronic Notes in Theoretical Computer Science (ENTCS)*, volume 55. Elsevier Science Publishers, 2001.

[21] S. Khurshid and D. Marinov. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering Journal*, 11(4):403–434, Oct. 2004.

[22] S. Khurshid, D. Marinov, I. Shlyakhter, and D. Jackson. A case for efficient solution enumeration. In *Proc. of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, Santa Margherita Ligure, Italy, May 2003.

[23] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

[24] B. Korel. Automated test data generation for programs with procedures. In *Proc. of the International Symposium on Software Testing and Analysis*, San Diego, CA, 1996.

[25] R. Kumar and E. G. Mercer. Load balancing parallel explicit state model checking. *Electr. Notes Theor. Comput. Sci.*, 128(3):19–34, 2005.

[26] F. Lerda and R. Sisto. Distributed-memory model checking with spin. In *Proc. of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 22–39, 1999.

[27] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. *Lecture Notes in Computer Science*, 2057:80–102, 2001.

[28] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[29] D. Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2004.

[30] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical Report MIT-LCS-TR-921, MIT Computer Science and Artificial Intelligence Lab, Cambridge, MA, Sept. 2003.

[31] B. D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 1(30):45–87, 1981. `http://cs.anu.edu.au/~bdm/nauty/`.

[32] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1), 1998.

[33] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. Korat: A tool for generating structurally complex test inputs. In *Proc. of the International Conference on Software Engineering, Demo Papers (ICSE Demo 2007)*, pages 771–774, Minneapolis, MN, May 2007.

[34] S. Misailovic, A. Milicevic, S. Khurshid, and D. Marinov. Generating test inputs for fault-tree analyzers using imperative predicates. In *the Workshop on Advances and Innovations in Systems Testing (STEP 2007)*, Memphis, TN, May 2007.

[35] R. Palmer and G. Gopalakrishnan. A distributed partial order reduction algorithm. In *FORTE '02: Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems*, pages 370–379, 2002.

[36] C. V. Ramamoorthy, S.-B. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4), 1976.

[37] N. J. A. Sloane, S. Plouffe, J. M. Borwein, and R. M. Corless. The encyclopedia of integer sequences. *SIAM Review*, 38(2), 1996. `http://www.research.att.com/~njas/sequences/Seis.html`.

[38] U. Stern and D. L. Dill. Parallelizing the murphi verifier. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 256–278, London, UK, 1997. Springer-Verlag.

[39] U. Stern and D. L. Dill. Parallelizing the Murϕ verifier. *Formal Methods in System Design*, 2001.

[40] K. Stobie. Advanced modeling, model based test generation, and Abstract state machine Language (AsmL). Seattle Area Software Quality Assurance Group, `http://www.sasqag.org/pastmeetings/asml.ppt`, Jan. 2003.

[41] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[42] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[43] M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 273–282, New York, NY, 2005. ACM Press.

[44] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. of the 15th Conference on Automated Software Engineering (ASE)*, Grenoble, France, 2000.

[45] E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6(3), May 1980.

[46] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. of the 19th IEEE International Conference on Automated Software Engineering*, Sept. 2004.