PKorat: Parallel Generation of Structurally Complex Test Inputs

JUNAID HAROON SIDDIQUI
SARFRAZ KHURSHID

THE UNIVERSITY OF TEXAS AT AUSTIN

ICST 2009
2 APRIL 2009

Overview

- PKorat builds on the Korat search algorithm [ISSTA'02] for constraint-based test generation
- Korat focuses on structurally complex test inputs, e.g., red-black trees, DAGs, XML documents
  - Manual generation of such inputs is tedious and error-prone
- PKorat parallelizes the Korat search to optimize test generation
- Experimental results show PKorat provides significant speed-ups over sequential Korat

Korat Overview

- Korat is a solver for imperative predicates
  - Predicates describe properties of desired test inputs
  - Each solution represents a desired test input
- User manually provides
  - Predicates
  - Bound for input size (e.g., number of nodes)
- Korat automatically generates all inputs within given bound
  - Bounded-exhaustive testing

Problem: Large test generation time

- Korat explores very large input spaces
  - Example: Binary Tree up to 13 nodes
    - State space is $>10^{32}$
    - >61 million inputs considered
    - >1 million inputs generated
- How to reduce test generation time?
  - Idea: distribute test generation in parallel workers
    - Previously explored in the context of a Google app [FSE'07]
- Our solution
  - PKorat: Parallel generation of structurally complex test inputs

Outline

- Overview
- Background: Korat
- PKorat
- Evaluation
- Conclusions

Korat: Input

- Representation for test inputs

```
class BinaryTree {
  class Node {
    Node* left;
    Node* right;
  };
  Node* root;
  int size;
};
```

- Imperative predicate method to identify valid test inputs (repOK, class invariant)
- Finitization defines search bounds

Korat: Imperative Predicate (repOK)

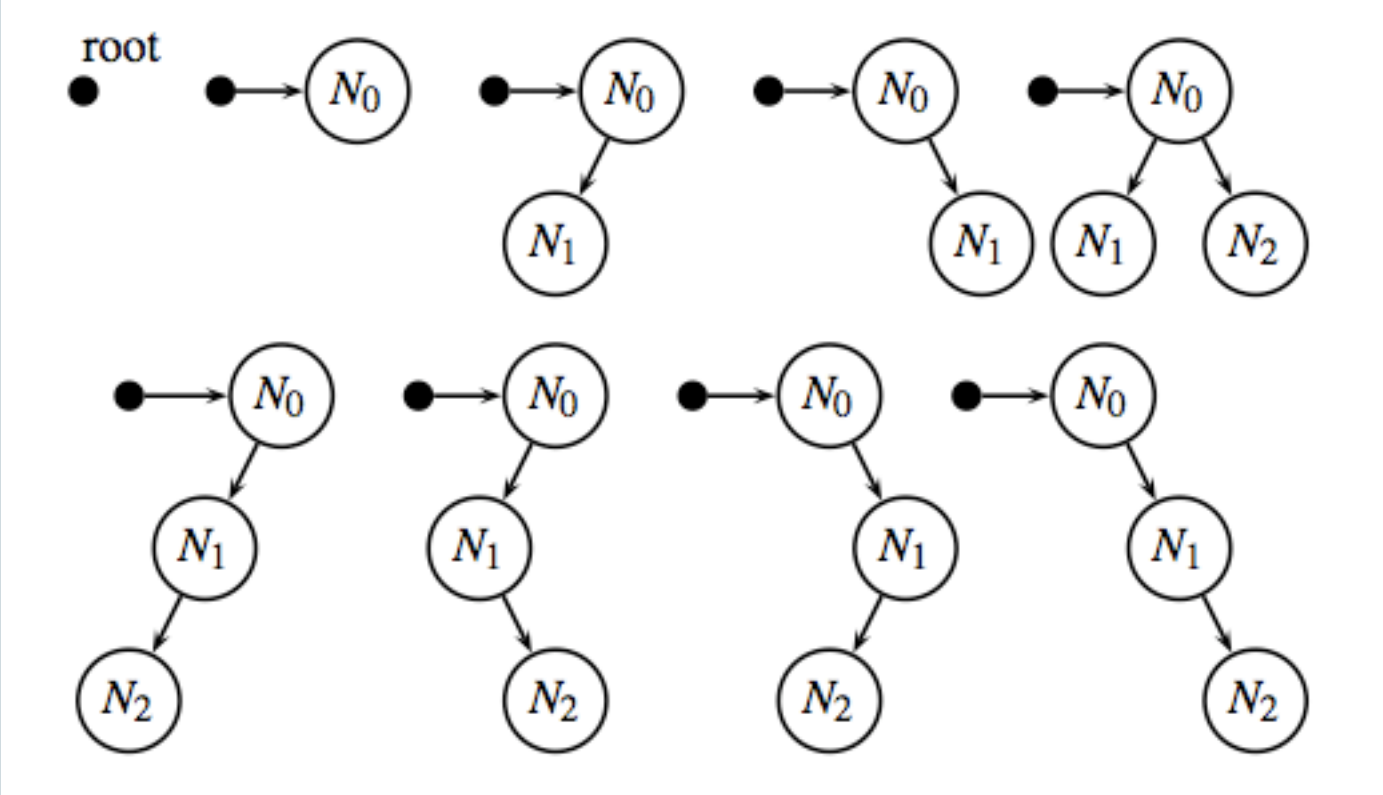
```
bool BinaryTree::repOK() {
   std::set<Node*> visited;
   std::stack<Node*> worklist;
   if( root ) {
     worklist.push( root );
     visited.insert( root ); }
   while( !worklist.empty() ) {
     Node* current = worklist.top();
     worklist.pop();
     if( current->left ) {
       if( !visited.insert( current->left ).second )
         return false;
       worklist.push( current->left ); }
     if( current->right ) {
       if( !visited.insert( current->right ).second )
         return false;
       worklist.push( current->right ); }
   }
   return visited.size() == size;
 }
```

Korat: Finitization

- Bounds search space
- Example
  - Number of objects
    - 1 BinaryTree object (T_0)
    - S BinaryTree::Node objects (N_0, N_1, … N_{S-1})
  - Values for fields
    - 0..S for `size`
    - Null, N_0 … N_{S-1} for `root`
    - Null, N_0 … N_{S-1} for `left` and `right` children
    - Each child is one of S nodes

Korat: Output

- Generates structurally complex data
  - Example: Binary Trees up to 3 nodes (S=3)



Korat: Input Space

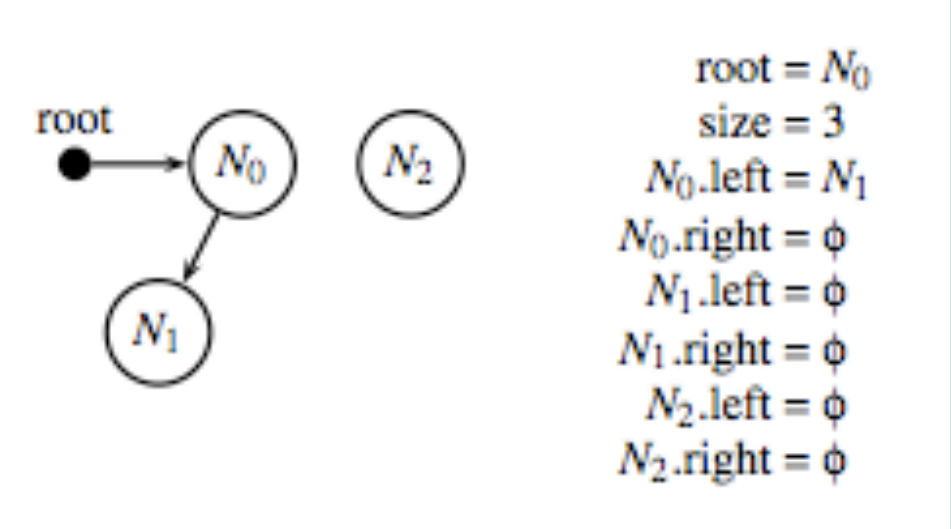
- Korat exhaustively explores a bounded input space
- Finitization describes all possible inputs
  - Example for size 3 (S=3)

T_0.root	T_0.size	N_0.left	N_0.right	N_1.left	N_1.right	N_2.left	N_2.right
NULL	0	NULL	NULL	NULL	NULL	NULL	NULL
N_0	1	N_0	N_0	N_0	N_0	N_0	N_0
N_1	2	N_1	N_1	N_1	N_1	N_1	N_1
N_2	3	N_2	N_2	N_2	N_2	N_2	N_2

- Size of input space is 4^8

Korat: Candidate Vector

- Sequence of indices into possible values
- Encodes 1 object graph, valid or invalid
- Example



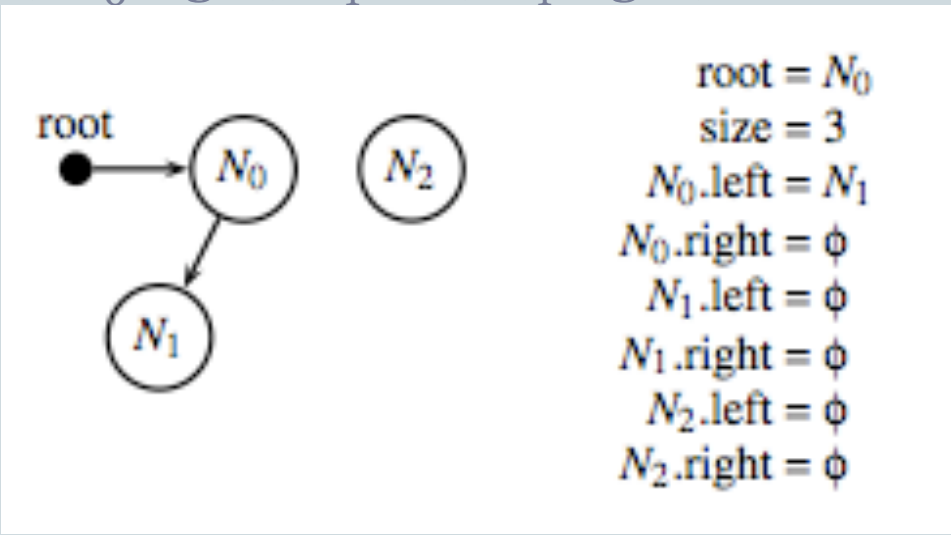
- Candidate Vector is <1, 3, 2, 0, 0, 0, 0, 0>

Korat: Search

- Starts from candidate vector with all 0's
- Generates candidate vectors in a loop until the entire space is explored
  - For each vector, executes repOK to find
    - whether the candidate is valid or not
    - what next candidate vector to try out
  - Field-access stack
    - Korat monitors field accesses during execution of repOK
    - Backtracks on last accessed field on stack, pruning large portions of the search space

Korat: next Candidate Vector

- Example: Backtracking on size and N_1.right
  - Candidate Vector is <1, 3, 2, 0, 0, 0, -, ->
  - Access Stack is <root, N_0.left, N_0.right, N_1.left, N_1.right, size>



- Produces next candidate
  - Candidate Vector is <1, 0, 2, 0, 0, 1, -, ->

Korat: Key Concepts

- repOK
  - User provides predicate that check properties of valid inputs
- Candidate vector
  - Used in Korat search
  - Next vector computed from previous by executing repOK

Outline

- Overview
- Background: Korat
- PKorat
- Evaluation
- Conclusions

Korat: parallelization concerns

- Candidate vector compactly encodes the entire search state, both
  - Space that has been explored
  - Space that is yet to be explored
- Korat pruning
  - Makes search efficient
  - But mostly sequential—pruning based on current execution
    - Next candidate vector depends on the execution of `repOK` on current candidate vector

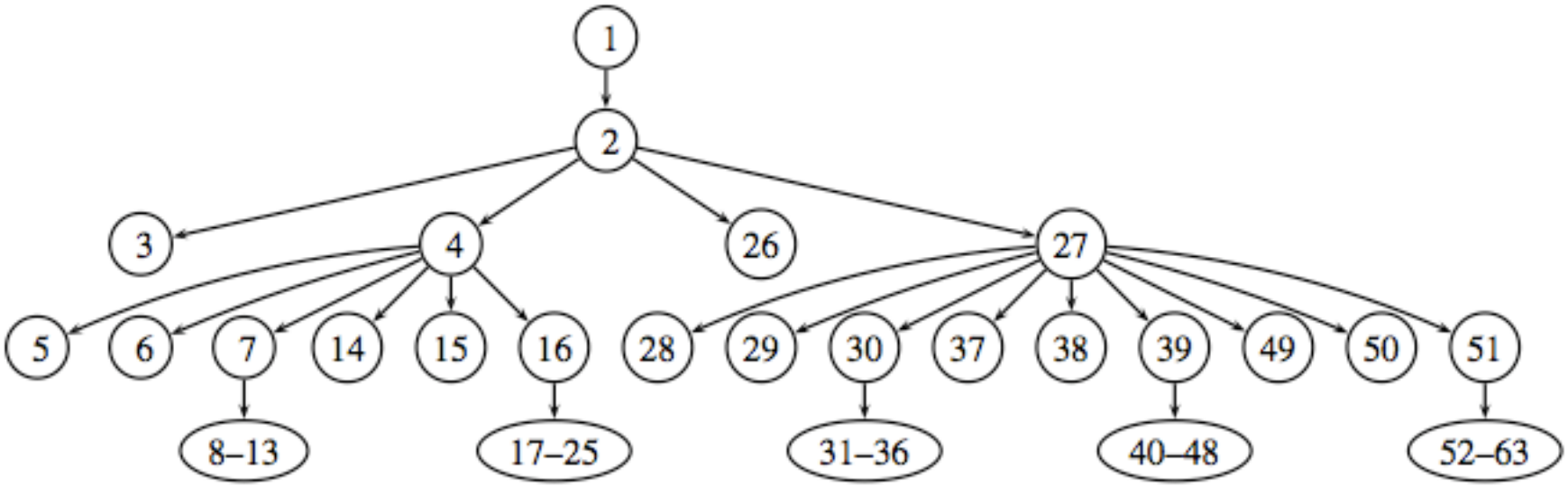
PKorat: Main Idea

- Whenever repOK accesses a new field, Korat will backtrack and explore all values for this field

- Example: New fields accessed A=0, B=0, C=0
- Next candidate generated A=0, B=0, C=1
- Candidates that can be generated
  - A=0, B=0, C=1..max
  - A=0, B=1..max, C=0
  - A=1..max, B=0, C=0

PKorat: illustration

- Tree of explored candidates with generation order
  - Nodes identify candidates that are not pruned by Korat search
  - Numbers indicate order in which Korat explores them
  - An edge $a \rightarrow b$ indicates candidate b created based on executing repOk on a



PKorat: Master Slave Algorithm

- Master slave algorithm
  - Master keeps a work queue of candidate vectors
  - Work Queue is initialized with candidate vector of all zeroes
- Slave gets a candidate, runs repOk, finds multiple next candidates and sends back to master
- Key question: how to find multiple next candidates?

PKorat: Finding Multiple Candidates

- Loop over new field(s) accessed by last repOK execution
  - Generate candidates for all non-zero valid indices for each field
  - Give zero value to remaining new fields
- Example: New fields accessed A=0, B=0, C=0
- Candidates generated
  - A=0, B=0, C=1..max
  - A=0, B=1..max, C=0
  - A=1..max, B=0, C=0
- If no new field accessed, no new candidate generated

PKorat: Finding New Accessed Fields

- Problem: How to find new fields accessed
  - Are the most recently accessed ones
  - Have a zero index
  - Are preceded by a field with a non-zero index
- First field with non-zero index must be accessed during testing of some previous candidate
  - Otherwise it has an identical prefix to the current candidate and the next field accessed should be common to both
- Solution: Pop indices from accessed fields stack until the first non-zero index

Outline

- Overview
- Background: Korat
- PKorat
- Evaluation
- Conclusions

Evaluation

- Subjects
  - Sorted Singly Linked List
  - Binary Tree
  - Red Black Tree
  - Directed Acyclic Graphs
  - Java Class Hierarchies
- Implemented in C++ and MPI
- Tested with 1 core, 4 cores, and 16 cores (TACC)
  - 4 cores with 1 master and 3 slaves
  - 16 cores with 1 master and 15 slaves

Experimental Results

Subject	Serial Time (s)	4 Core Time (s)	4 Core Speedup	16 Core Time (s)	16 Core Speedup
Singly Linked List (500)	1387	523	2.6x	108	12.8x
Binary Tree (13)	480	250	1.9x	105	4.6x
Red Black Tree (10)	347	136	2.6x	77	4.5x
Directed Acyclic Graph(7)	1163	544	2.1x	284	4.1x
Java Class Hierarchies (8)	149	71	2.1x	39	3.8x

Evaluation: Insight

- Quad core performance is best (2.6x speedup with 3 workers)
- More workers are useful for structures having more nodes
- For many more processors, communication costs start to have an effect

Outline

- Overview
- Background: Korat
- PKorat
- Evaluation
- Conclusions

Conclusions & Future Work

- PKorat is a natural parallel extension to the Korat algorithm
  - Previous work developed a randomization approach [FSE'07]
- PKorat explores the same candidates as Korat in parallel
- Experimental results show significant improvement
  - Larger inputs enable more parallelism
- Our in-progress work (aka my PhD thesis)
  - Develop a generic framework (language and tool-support) for specifying dynamic analyses that are automatically parallelized
    - E.g., korat, symbolic execution, model checking, …

Questions!