Universal Symbolic Execution and its Application to Likely Data Structure Invariant Generation

Yamini Kannan
EECS, UC Berkeley
yamini@eecs.berkeley.edu

Koushik Sen
EECS, UC Berkeley
ksen@cs.berkeley.edu

ABSTRACT

Local data structure invariants are asserted over a bounded fragment of a data structure around a distinguished node M of the data structure. An example of such an invariant for a sorted doubly linked list is "for all nodes M of the list, if $M \neq$ `null` and $M.next \neq$ `null`, then $M.next.prev = M$ and $M.value \leq M.next.value$." It has been shown that such local invariants are both natural and sufficient for describing a large class of data structures. This paper explores a novel technique, called KRYSTAL, to infer likely local data structure invariants using a variant of symbolic execution, called universal symbolic execution. Universal symbolic execution is like traditional symbolic execution except the fact that we create a fresh symbolic variable for every read of a lvalue that has no mapping in the symbolic state rather than creating a symbolic variable only for inputs. This helps universal symbolic execution to symbolically track data flow for all memory locations along an execution even if input values do not flow directly into those memory locations. We have implemented our algorithm and applied it to several data structure implementations in Java. Our experimental results show that we can infer many interesting local invariants for these data structures.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification, D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Reliability, Design, Documentation

Keywords: Program invariants, symbolic execution, dynamic analysis, execution traces, logical inference.

1. INTRODUCTION

Local data structure invariants are an important class of invariants that can express interesting properties of data structures. An example of such an invariant for a sorted doubly-linked list is "for all nodes M of the list, if $M \neq$ `null` and $M.next \neq$ `null`, then $M.next.prev = M$ and $M.value \leq M.next.value$." Such properties are asserted over a bounded fragment of a data structure around a distinguished node M of the data structure. It has been shown that such local invariants are both natural and sufficient for describing a large class of data structures [19]. This paper explores a novel technique, called KRYSTAL, to infer likely local data structure invariants using a variant of symbolic execution [15, 3, 4, 27, 14, 26, 16, 29], called *universal symbolic execution*.

There are three important contributions of this work. First, we propose *universal symbolic execution*, that helps us to symbolically track data flow for all memory locations along an execution even if input values do not flow directly into those memory locations. In universal symbolic execution, we execute a program concretely on test inputs, while simultaneously performing symbolic execution of the program. However, unlike traditional symbolic execution, *we create a fresh symbolic variable for every read of a lvalue that has no mapping in the symbolic state* rather than creating a symbolic variable only for inputs. For example, universal symbolic execution of the following code

$x = 1;$
$y = x + 1;$
$z = x - 1;$

creates a symbolic memory where x maps to a fresh symbolic variable, say x_0, y maps to $x_0 + 1$, and z maps to $x_0 - 1$. Note that we have introduced a symbolic variable x_0 despite the fact that the code has no input. In contrast, traditional symbolic execution of the code creates a memory map where x maps to 1, y maps to 2, and z maps to 0. The advantage of universal symbolic execution is that now we can relate y and z by simply looking at the symbolic memory, i.e. we can infer $y - 1 = z + 1$. This crucial observation is central to our invariant generation algorithm. Specifically, we use symbolic memory maps and symbolic path conditions and equate the symbolic variables present in them to generate symbolic relations among various memory cells.

Second, we show how the results of universal symbolic execution can be refined and generalized to generate likely local data structure invariants. Specifically, universal symbolic execution of a program on a test input generates a *symbolic memory* and a *set of path conditions*, where the symbolic memory is a map from program addresses to symbolic expressions and each path condition is a symbolic predicate generated by the execution of a conditional statement. The symbolic memory and the path condition set are then used to generate a set of symbolic predicates that hold along the test execution. Subsequently, local predicates, i.e. predicates involving a single symbolic variable, are derived from the set of predicates through variable elimination. These local predicates are then generalized and simplified across all test executions to infer a set of likely local invariant templates. The templates that hold on all nodes of the data structure along all test executions are then output as likely local data structure invariants.

Third, we evaluate KRYSTAL on a number of data structure implementations in Java. Our experimental results are encouraging.

```
class Node {
   int key;  // data field
   int depth; // depth of the node from the root
   Node left, right, parent;
}
class BinarySearchTree {
   Node root;
   void insert(int data) {
ℓ0:     Node node = new Node();
ℓ1:     node.key = data;
ℓ2:     node.left = node.right = node.parent= null;
ℓ3:     if (root == null) {
ℓ4:        root = node; node.depth = 0; return;}
ℓ5:     Node current = root;
ℓ6:     if (current.key < data)
ℓ7:        if (current.right == null) {
ℓ8:           current.right = node; goto l15; }
          else {  // loop again
ℓ9:           current = current.right; goto l6; }
ℓ10:    if (current.key > data)
ℓ11:       if (current.left == null) {
ℓ12:          current.left = node; goto l15; }
          else { // loop again
ℓ13:           current = current.left; goto l6;}
ℓ14:    return;  //data already present in the tree
ℓ15:    node.parent = current;
ℓ16:    node.depth = current.depth + 1;
     }
}
```

Figure 1: Binary search tree and insert

```
class BSTHarness {
   static BinarySearchTree bst
                     = new BinarySearchTree();
   public static void testBST(int input) {
      bst.insert(input);
   }
}
```

Figure 2: Test harness for the binary search tree

We show that KRYSTAL can infer interesting local invariants for these data structures in a few seconds. We also argue that many of these invariants cannot be inferred by existing techniques.

2. OVERVIEW

We informally present KRYSTAL on a binary search tree example shown in Figure 1. The example defines two classes NODE and BINARYSEARCHTREE and the method INSERT. An instance of the class BINARYSEARCHTREE holds a reference to the *root* of a binary search tree. The nodes of the tree are instances of the class NODE. The INSERT method takes as input an integer *data*, creates a new *node* for *data*, and computes the correct position for *node* in the tree. If *data* is absent from the tree, INSERT adds the node into the tree by setting fields in appropriate nodes. The class NODE defines a field *depth* that holds the current depth of the node from the root node in the tree. We use **goto**s in the body of the method INSERT to explicitly represent the control flow.

The interesting local invariants that hold for any node M in a binary search tree are the following.

$$(M \neq \texttt{null} \wedge M.left \neq \texttt{null}) \Rightarrow (M.left.parent = M)$$
$$(M \neq \texttt{null} \wedge M.right \neq \texttt{null}) \Rightarrow (M.right.parent = M)$$
$$(M \neq \texttt{null} \wedge M.parent \neq \texttt{null}) \Rightarrow (M.depth = M.parent.depth + 1)$$
$$(M \neq \texttt{null} \wedge M.left \neq \texttt{null}) \Rightarrow (M.depth + 1 = M.left.depth)$$
$$(M \neq \texttt{null} \wedge M.right \neq \texttt{null}) \Rightarrow (M.depth + 1 = M.right.depth)$$
$$(M \neq \texttt{null} \wedge M.left \neq \texttt{null}) \Rightarrow (M.key > M.left.key)$$
$$(M \neq \texttt{null} \wedge M.right \neq \texttt{null}) \Rightarrow (M.key < M.right.key)$$
$$(M \neq \texttt{null} \wedge M.parent \neq \texttt{null}) \Rightarrow (M.key \neq M.parent.key)$$
$$(M \neq \texttt{null} \wedge M.parent \neq \texttt{null}) \Rightarrow ((M.parent.left = M) \vee (M.parent.right = M))$$

The goal of KRYSTAL is to infer these local invariants by running the test harness in Figure 2 on a set of test inputs. KRYSTAL infers likely local data structure invariants using the following steps.

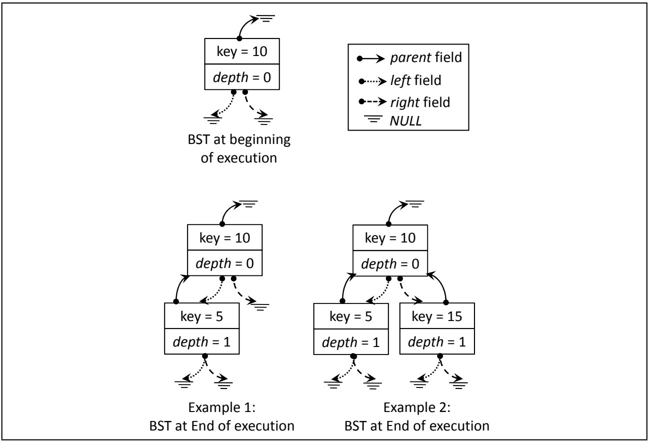

Figure 3: Structure of the binary search tree at the beginning and end of a test execution

1. KRYSTAL assumes that there is a test harness and a set of test inputs for the given data structure. KRYSTAL performs a variant of symbolic execution, called *universal symbolic execution*, along each test execution path to compute a symbolic memory and a symbolic path condition set.
2. KRYSTAL uses the symbolic memory and the symbolic path condition set to generate a set of local symbolic predicates, i.e. a set of predicates each of which is an expression over a single symbolic variable. A local symbolic predicate in the set represents a local property that holds over a specific data structure node along the test execution.
3. KRYSTAL simplifies "similar" (defined in Section 3.4) local predicates in the set by taking conjunctions of similar local predicates. The simplified local predicates are then used to generate local invariant templates. A local invariant template denotes a predicate that could potentially hold on any node in the data structure.
4. KRYSTAL collects local invariant templates across test executions and further simplifies them by taking disjunctions of "similar" (defined in Section 3.4) templates. The resultant templates are then relaxed by incorporating implicit constraints (such as dereferencing a `null` object is illegal.) Collecting templates across executions helps to ensure that the set of templates contains all potential local data structure invariants.
5. KRYSTAL then validates each template, by checking if it holds on every node of the data structure at the end of each test execution. The templates that hold for all nodes along all test executions are output as likely local data structure invariants.

Consider a test execution where we insert the integer 5 into a binary search tree which has been pre-populated to contain a single node with *key* 10. The statements that get executed are ℓ_0, ℓ_1, ℓ_2, ℓ_3, ℓ_5, ℓ_6, ℓ_{10}, ℓ_{11}, ℓ_{12}, ℓ_{15}, ℓ_{16}. The structure of the binary search tree before and after the execution of the method INSERT is given in Figure 3. We next describe the various steps that KRYSTAL uses to generate local data structure invariants for the binary search tree example.

2.1 Universal Symbolic Execution

Along a test execution path, KRYSTAL uses a variant of symbolic execution, called *universal symbolic execution*, to compute a symbolic memory and a set of symbolic path conditions. Universal symbolic execution is similar to symbolic execution as done in concolic execution in several aspects—during universal symbolic execution, each expression in the execution trace is evaluated symbolically in the context of the symbolic memory, the symbolic memory is updated on every assignment, and a symbolic constraint is generated on the execution of every conditional expression. However, unlike concolic symbolic execution, universal symbolic execution creates a new symbolic variable whenever a lvalue is read and the lvalue has no mapping in the symbolic memory. A new symbolic variable is created and an entry mapping the lvalue to the newly created symbolic variable is added to the symbolic memory. Furthermore, in universal symbolic execution, an assignment statement can update the symbolic memory in two distinct ways. Universal symbolic execution evaluates the right hand expression in the assignment in the context of the symbolic memory. If the expression evaluates to a constant, universal symbolic execution removes from the symbolic memory any previous mapping associated with the address computed for the lvalue. Otherwise, universal symbolic execution updates the symbolic memory to map the appropriate address to the evaluated symbolic expression. This special treatment of unmapped lvalues in the symbolic memory and assignment statements is crucial in universal symbolic execution as it helps to establish symbolic relations among various nodes and fields of the nodes of a data structure.

We describe universal symbolic execution along the execution path ℓ_0, ℓ_1, ℓ_2, ℓ_3, ℓ_5, ℓ_6, ℓ_{10}, ℓ_{11}, ℓ_{12}, ℓ_{15}, ℓ_{16}. At the beginning of the execution the symbolic memory map, say $\mathcal{S}$, is initialized to an empty map and the set of symbolic path constraints, say Φ, is initialized to an empty set. During universal symbolic execution, the statements in the execution trace update the symbolic memory and symbolic path constraints set. Figure 4 illustrates the universal symbolic execution along this path.

Universal symbolic execution of statement ℓ_0 leaves the symbolic memory $\mathcal{S}$ and the path condition set Φ unmodified. The universal symbolic execution of the statement labeled ℓ_1 creates two new symbolic variables x_0 and x_1 as the lvalues that are being read—*node* and *data*, have no mapping in the symbolic memory $\mathcal{S}$. Consequently, the symbolic memory gets updated as shown in Figure 4. Note that in the new symbolic memory, x_0.*key* denotes the symbolic address of the lvalue *node.key*.

The following statement ℓ_2 has no effect on $\mathcal{S}$ and Φ. The execution of the conditional labeled ℓ_3 creates a new symbolic variable x_2 as the lvalue *root* has no mapping in $\mathcal{S}$, updates the symbolic memory to include the mapping ($root \mapsto x_2$), and adds the symbolic path constraint ($x_2 \neq \texttt{null}$) to the set Φ.

We continue universal symbolic execution along the rest of the path in the same way. Figure 4 shows the updated symbolic memory and path condition set that are obtained after the execution of the corresponding statement.

At the end of the execution, we have symbolic memory $\mathcal{S} = [node \mapsto x_0, data \mapsto x_1, x_0.key \mapsto x_1, root \mapsto x_2, current \mapsto x_2, x_2.key \mapsto x_3, x_2.left \mapsto x_0, x_0.parent \mapsto x_2, x_2.depth \mapsto x_5, x_0.depth \mapsto x_5 + 1]$ and path constraint set $\Phi = \{x_2 \neq \texttt{null}, x_3 \geq x_1, x_3 > x_1, x_4 = \texttt{null}\}$.

2.2 Local predicate generation

The symbolic memory and the path condition set are then used to generate the set of predicates $\varphi = \{node = x_0, data = x_1, x_0.key = x_1, root = x_2, current = x_2, x_2.key = x_3, x_2.left = x_0, x_0.parent = x_2, x_2.depth = x_5, x_0.depth = x_5 + 1, x_2 \neq \texttt{null}, x_3 \geq x_1, x_3 > x_1, x_4 = \texttt{null}\}$. This is obtained by adding to the set Φ a predicate $t_1 = t_2$ for each mapping $t_1 \mapsto t_2$ in $\mathcal{S}$. The predicates in the set φ represent the symbolic constraints over the symbolic variables that hold for the current execution.

ID	Stmt	Symbolic Memory $\mathcal{S}$	Path Cond. Set Φ
ℓ_0	NODE *node* = new NODE();	$[]$	$\{\}$
ℓ_1	*node.key* = *data*;	$\underline{\boldsymbol{[node \mapsto x_0, data \mapsto x_1,}}$ $\underline{\boldsymbol{x_0.key \mapsto x_1}}]$	$\{\}$
ℓ_2	*node.left* = *node.right* = *node.parent* = `null`;	$[node \mapsto x_0, data \mapsto x_1,$ $x_0.key \mapsto x_1]$	$\{\}$
ℓ_3	`if` (*root* == `null`)	$[node \mapsto x_0, data \mapsto x_1,$ $x_0.key \mapsto x_1, \underline{\boldsymbol{root \mapsto x_2}}]$	$\underline{\boldsymbol{\{x_2 \neq \texttt{null}\}}}$
ℓ_5	NODE *current* = *root*;	$[node \mapsto x_0, data \mapsto x_1,$ $x_0.key \mapsto x_1, root \mapsto x_2,$ $\underline{\boldsymbol{current \mapsto x_2}}]$	$\{x_2 \neq \texttt{null}\}$
ℓ_6	`if` (*current.key* < *data*)	$[node \mapsto x_0, data \mapsto x_1,$ $x_0.key \mapsto x_1, root \mapsto x_2,$ $current \mapsto x_2, \underline{\boldsymbol{x_2.key \mapsto x_3}}]$	$\{x_2 \neq \texttt{null},$ $\underline{\boldsymbol{x_3 \geq x_1}}\}$
ℓ_{10}	`if` (*current.key* > *data*)	$[node \mapsto x_0, data \mapsto x_1,$ $x_0.key \mapsto x_1, root \mapsto x_2,$ $current \mapsto x_2, x_2.key \mapsto x_3]$	$\{x_2 \neq \texttt{null},$ $x_3 \geq x_1,$ $\underline{\boldsymbol{x_3 > x_1}}\}$
ℓ_{11}	`if` (*current.left* == `null`)	$[node \mapsto x_0, data \mapsto x_1,$ $x_0.key \mapsto x_1, root \mapsto x_2,$ $current \mapsto x_2, x_2.key \mapsto x_3,$ $\underline{\boldsymbol{x_2.left \mapsto x_4}}]$	$\{x_2 \neq \texttt{null},$ $x_3 \geq x_1,$ $x_3 > x_1,$ $\underline{\boldsymbol{x_4 = \texttt{null}}}\}$
ℓ_{12}	*current.left* = *node*;	$[node \mapsto x_0, data \mapsto x_1,$ $x_0.key \mapsto x_1, root \mapsto x_2,$ $current \mapsto x_2, x_2.key \mapsto x_3,$ $\underline{\boldsymbol{x_2.left \mapsto x_0}}]$	$\{x_2 \neq \texttt{null},$ $x_3 \geq x_1,$ $x_3 > x_1,$ $x_4 = \texttt{null}\}$
ℓ_{15}	*node.parent* = *current*;	$[node \mapsto x_0, data \mapsto x_1,$ $x_0.key \mapsto x_1, root \mapsto x_2,$ $current \mapsto x_2, x_2.key \mapsto x_3,$ $x_2.left \mapsto x_0, \underline{\boldsymbol{x_0.parent \mapsto x_2}}]$	$\{x_2 \neq \texttt{null},$ $x_3 \geq x_1,$ $x_3 > x_1,$ $x_4 = \texttt{null}\}$
ℓ_{16}	*node.depth* = *current.depth* $+1$;	$[node \mapsto x_0, data \mapsto x_1,$ $x_0.key \mapsto x_1, root \mapsto x_2,$ $current \mapsto x_2, x_2.key \mapsto x_3,$ $x_2.left \mapsto x_0, x_0.parent \mapsto x_2,$ $\underline{\boldsymbol{x_2.depth \mapsto x_5}},$ $\underline{\boldsymbol{x_0.depth \mapsto x_5 + 1}}]$	$\{x_2 \neq \texttt{null},$ $x_3 \geq x_1,$ $x_3 > x_1,$ $x_4 = \texttt{null}\}$

Figure 4: Universal symbolic execution of INSERT(5) in a BINARY SEARCH TREE containing 10. Updates made at each step are underlined for emphasis.

For local invariant generation, we need predicates that are over a single symbolic variable and do not contain variable names, also called *local predicates*. In local predicates, we can then replace the symbolic variable by a generic symbolic variable M to generate a likely local template. However, the predicates in φ, in general, are not over a single symbolic variable or contain variable names (such as *node*, *data*.) KRYSTAL tries to derive new predicates containing a single symbolic variable from the set φ through variable substitution. Specifically, KRYSTAL uses predicates of the form $x_i = e$ (or $e = x_i$) in φ (where x_i is a symbolic variable and e is a symbolic expression) to derive more predicates. This is done by replacing the variable x_i with the expression e in each predicate in φ that contains x_i.

KRYSTAL performs variable substitution on the predicates in φ repeatedly to generate the complete set of local predicates that can be derived from φ. Note that due to possible circular dependency among the predicates, we can perform an unbounded number of substitutions. Therefore, we restrict the number of times KRYSTAL applies the variable substitutions to two. Note that if we increase this bound further, we will get more complex, but rich set of predicates. However, our experiments showed that only two substitutions are sufficient to infer the most interesting local invariants.

For example, using the predicate $x_2.left = x_0$, we derive the local predicate $x_2.left.parent = x_2$ by replacing x_0 with $x_2.left$ in the predicate $x_0.parent = x_2$. The complete set of local predicates containing a single symbolic variable derived from φ is $\psi = \{x_0.\mathit{parent.left} = x_0, x_2.\mathit{left.parent} = x_2, x_0.\mathit{depth} = x_0.\mathit{parent.depth} + 1, x_2.\mathit{left.depth} = x_2.\mathit{depth} + 1, x_0.\mathit{parent} \neq \mathtt{null}, x_2 \neq \mathtt{null}, x_2.\mathit{key} \geq x_2.\mathit{left.key}, x_0.\mathit{parent.key} \geq x_0.\mathit{key}, x_2.\mathit{key} > x_2.\mathit{left.key}, x_0.\mathit{parent.key} > x_0.\mathit{key}, x_4 = \mathtt{null}\}$.

2.3 Local invariant template generation from local predicates

In this step, we simplify sets of *similar* local predicates in the set ψ by taking their conjunction. In informal terms, local predicates that are expressions over the same set of lvalues are similar (a formal definition is given in Section 3.4.) To understand why we take a conjunction to simplify sets of local predicates, recall that the local predicates in the set ψ are generated from a single execution trace, and hence hold true over the corresponding symbolic variables simultaneously. It follows that a conjunction of local predicates also holds true along the execution trace. For example, we simplify the predicates in the set $\{x_2.\mathit{key} \geq x_2.\mathit{left.key}, x_2.\mathit{key} > x_2.\mathit{left.key}\}$ to $x_2.\mathit{key} > x_2.\mathit{left.key}\}$. After simplification, the set of local predicates become $\{x_0.\mathit{parent.left} = x_0, x_2.\mathit{left.parent} = x_2, x_0.\mathit{depth} = x_0.\mathit{parent.depth} + 1, x_2.\mathit{left.depth} = x_2.\mathit{depth} + 1, x_2 \neq \mathtt{null}, x_0.\mathit{parent} \neq \mathtt{null}, x_2.\mathit{key} > x_2.\mathit{left.key}, x_0.\mathit{parent.key} > x_0.\mathit{key}, x_4 = \mathtt{null}\}$. The set represents local predicates that hold along the test execution.

Subsequently, we replace any symbolic variable in any local predicate by the generic symbolic variable M. The resultant set is a local invariant template set $\{M.\mathit{parent.left} = M, M.\mathit{left.parent} = M, M.\mathit{depth} = M.\mathit{parent.depth} + 1, M.\mathit{left.depth} = M.\mathit{depth} + 1, M \neq \mathtt{null}, M.\mathit{parent} \neq \mathtt{null}, M.\mathit{key} > M.\mathit{left.key}, M.\mathit{parent.key} > M.\mathit{key}, M = \mathtt{null}\}$. The templates in the set denote predicates that could potentially hold on any node in the binary search tree.

2.4 Collecting local invariant templates across executions

So far, we have generated a set of local invariant templates from a single test execution. Since a single execution path in a program cannot cover all possible behaviors of the program, the set of templates may not be complete (i.e. some interesting templates may be missing) or sound (i.e. some templates that may not hold over all executions may be present.) For example, the template $M.\mathit{key} < M.\mathit{right.key}$ is missing from the set, while the template $M.\mathit{parent.key} > M.\mathit{key}$ is not true for all nodes in the binary search tree. In order to minimize the incompleteness, we take the union of all sets of templates generated on all test inputs. Note that the set of templates after taking union could still be incomplete as our test suite might not be good enough. We can overcome this shortcoming partly by ensuring that the test suite has a high path coverage, by using a systematic automated test generation tool such as CUTE [21]. In the above example, consider inserting another datum, say 15, in the binary search tree. After taking union, the set of templates becomes: $\{M.\mathit{parent.left} = M, M.\mathit{parent.right} = M, M.\mathit{left.parent} = M, M.\mathit{right.parent} = M, M.\mathit{depth} = M.\mathit{parent.depth} + 1, M.\mathit{left.depth} = M.\mathit{depth} + 1, M.\mathit{right.depth} = M.\mathit{depth} + 1, M \neq \mathtt{null}, M.\mathit{parent} \neq \mathtt{null}, M.\mathit{key} > M.\mathit{left.key}, M.\mathit{key} < M.\mathit{right.key}, M.\mathit{parent.key} > M.\mathit{key}, M.\mathit{parent.key} < M.\mathit{key}, M = \mathtt{null}\}$.

We again perform simplifications on sets of similar local invariant templates, where similarity is defined in the same manner as similarity of local predicates. We simplify sets of similar local invariant templates by taking disjunction. To understand the reasoning behind this, note that local invariant templates are generated from local predicates over different symbolic variables or from different execution traces. By taking disjunction, we are able to generate simplified invariant templates that potentially hold over all nodes in the data structure, across all execution paths. For example, we replace the templates $M.\mathit{parent.key} > M.\mathit{key}$ and $M.\mathit{parent.key} < M.\mathit{key}$ by their disjunction $M.\mathit{parent.key} \neq M.\mathit{key}$. The resultant set becomes $\{M.\mathit{parent.left} = M, M.\mathit{parent.right} = M, M.\mathit{left.parent} = M, M.\mathit{right.parent} = M, M.\mathit{depth} = M.\mathit{parent.depth} + 1, M.\mathit{left.depth} = M.\mathit{depth} + 1, M.\mathit{right.depth} = M.\mathit{depth} + 1, M.\mathit{parent} \neq \mathtt{null}, M.\mathit{key} > M.\mathit{left.key}, M.\mathit{key} < M.\mathit{right.key}, M.\mathit{parent.key} \neq M.\mathit{key}\}$.

The templates that we have generated so far could potentially hold over all nodes in the tree. However, these templates also assume some implicit constraints, such as one cannot access the field of a null object. Therefore, an invariant template $M.\mathit{parent.left} = M$ is true only if $M \neq \mathtt{null}$ and $M.\mathit{parent} \neq \mathtt{null}$. We incorporate these implicit constraints in each template. For example, we modify the template $M.\mathit{parent.left} = M$ to $(M \neq \mathtt{null} \wedge M.\mathit{parent} \neq \mathtt{null}) \Rightarrow (M.\mathit{parent.left} = M)$. We perform the modification for each template in the set of templates.

In this phase, we have collected templates across multiple executions so that the set of templates is complete. We next try to remove unsoundness by validating the templates on all nodes across all executions.

2.5 From local invariant templates to likely local data structure invariants

In the final validation phase, we run our test harness on all test inputs and check at the end of each execution whether each template obtained from the previous phase holds on each node of the binary tree at the end of the execution. We output those templates that hold on all nodes on all executions as local data structure invariants. In our example, we get the following set of local invariants.

$\{(M \neq \mathtt{null} \wedge M.\mathit{left} \neq \mathtt{null}) \Rightarrow (M.\mathit{left.parent} = M),$
$(M \neq \mathtt{null} \wedge M.\mathit{right} \neq \mathtt{null}) \Rightarrow (M.\mathit{right.parent} = M),$
$(M \neq \mathtt{null} \wedge M.\mathit{parent} \neq \mathtt{null}) \Rightarrow (M.\mathit{depth} = M.\mathit{parent.depth} + 1),$
$(M \neq \mathtt{null} \wedge M.\mathit{left} \neq \mathtt{null}) \Rightarrow (M.\mathit{depth} + 1 = M.\mathit{left.depth}),$
$(M \neq \mathtt{null} \wedge M.\mathit{right} \neq \mathtt{null}) \Rightarrow (M.\mathit{depth} + 1 = M.\mathit{right.depth}),$
$(M \neq \mathtt{null} \wedge M.\mathit{left} \neq \mathtt{null}) \Rightarrow (M.\mathit{key} > M.\mathit{left.key}),$
$(M \neq \mathtt{null} \wedge M.\mathit{right} \neq \mathtt{null}) \Rightarrow (M.\mathit{key} < M.\mathit{right.key}),$
$(M \neq \mathtt{null} \wedge M.\mathit{parent} \neq \mathtt{null}) \Rightarrow (M.\mathit{key} \neq M.\mathit{parent.key})\}$

Note that in the validation phase we remove the spurious templates $(M \neq \mathtt{null} \wedge M.\mathit{parent} \neq \mathtt{null}) \Rightarrow (M.\mathit{parent.left} = M)$, $(M \neq \mathtt{null} \wedge M.\mathit{parent} \neq \mathtt{null}) \Rightarrow (M.\mathit{parent.right} = M)$, and $(M \neq \mathtt{null}) \Rightarrow (M.\mathit{parent} \neq \mathtt{null})$.

3. ALGORITHM

In this section, we formally describe the KRYSTAL algorithm.

3.1 Programs and concrete semantics

We describe the KRYSTAL algorithm on a simple imperative language. The operations of the language consist of labeled statements $\ell : s$. Labels denote statement addresses. A statement in the language is either (1) the HALT statement denoting the normal termination of the program, (2) an assignment statement $m := e$, where e is a side-effect free expression and m is an lvalue of the form v or $v.f$ denoting a variable or a field f of the object referenced by v, respectively, and (3) a conditional statement of the form `if` (e) `goto` ℓ, where e is a side-effect free expression and ℓ is a statement label. The language allows a special kind of object creation expression of the form `new` T, where T is an object type (or class.) For simplicity of exposition, we do not include function calls in the language; function calls can be handled in the standard way as described in [22].

The set of *data values* consists of objects, integer values, and boolean values. The concrete semantics of a program is given using a *memory* consisting of a mapping from program addresses to values. Execution starts from the initial memory $\mathcal{M}_0$ which maps addresses to some default value in their domain. Given a memory $\mathcal{M}$, we write $\mathcal{M}[a \mapsto d]$ to denote the memory that maps the address a to the value d and maps all other addresses a' to $\mathcal{M}(a')$.

Statements update the memory. The concrete semantics of a program is given in the usual way as a relation from program location and memory to an updated program location (corresponding to the next statement to be executed) and an updated memory [20]. For an assignment statement $\ell : m := e$, this relation calculates, possibly involving address dereferencing, the address a of the lvalue m. The expression e is evaluated to a concrete value d in the context of the current memory $\mathcal{M}$, the memory is updated to $\mathcal{M}[a \mapsto d]$, and the new program location is $\ell + 1$. For a conditional ℓ : `if` (e) `goto` ℓ', the expression e is evaluated in the current memory $\mathcal{M}$. If the evaluated value is true, the new program location is ℓ', and if the value is false, the new location is $\ell + 1$. In either case, the new memory is identical to the old one. Execution terminates normally if the current statement is HALT. For an object creation statement $m :=$ `new` T, the address a of the lvalue m is calculated, an object o is allocated in the heap, and the memory is updated to $\mathcal{M}[a \mapsto o]$.

3.2 Universal Symbolic Execution

In order to infer likely local data structure invariants, we use a variant of symbolic execution called *universal symbolic execution*. Universal symbolic execution performs symbolic execution along a concrete execution path as in concolic execution. However, universal symbolic execution differs from concolic execution in two important ways. Universal symbolic execution introduces a fresh symbolic variable during symbolic execution whenever the execution reads the value at an address, and the address has no mapping in the symbolic memory (i.e. a map that represents the program memory symbolically.) This is in contrast to concolic execution where we only introduce a symbolic variable whenever the program executes an input statement. Additionally, in universal symbolic execution, an assignment statement $m := e$ can update the symbolic memory in one of two different ways. Universal symbolic execution computes the symbolic address of m, and evaluates the expression e in the context of the symbolic memory. If the symbolic evaluation of e results in a constant, universal symbolic execution removes from the symbolic memory any previous mapping associated with the computed symbolic address for m. Otherwise, the symbolic memory is updated to map the symbolic address to the symbolic value of e. This is in contrast to concolic execution (and classic symbolic execution) where assignment statements update the symbolic memory uniformly to map the address of m to the symbolic expression for e. This is described in more detail in the rest of the section. The introduction of a symbolic variable for any read to an unmapped address helps KRYSTAL to track relations among various memory locations, rather than memory locations that are only data-dependent on inputs. This is important to establish relations among various data structure nodes all of which are not necessarily data-dependent on inputs. For example, in a red-black tree we can infer the invariants that the color field of any node is either red or black despite the fact the values red and black are not provided in inputs. We provide the details in the rest of the section.

```
execute_program($P$, TestInp)
    $pc = \ell_0; i = 0;$
    $\mathcal{M} = \mathcal{S} = [\,];$
    $\Phi = [\,];$
    while (true)
        s = statement_at($P, pc$);
        match (s)
            case ($m := e$):
                $\mathcal{M} = \mathcal{M}[eval\_concrete(m, \mathcal{M}) \mapsto eval\_concrete(e, \mathcal{M})];$
                $(\mathcal{S}, i) = exec\_symbolic(m, e, \mathcal{S}, i);$
                $pc = pc + 1;$
            case (if ($e$) goto $\ell'$):
                $b = eval\_concrete(e, \mathcal{M});$
                $(c, \mathcal{S}, i) = eval\_symbolic(e, \mathcal{S}, i);$
                if $b$ then
                    $\Phi = \Phi \cup c; pc = \ell';$
                else
                    $\Phi = \Phi \cup \neg c; pc = pc + 1;$
            case HALT:
                return $\Phi$;
```

Figure 5: Universal symbolic execution algorithm of KRYSTAL

The second difference between concolic execution and universal symbolic execution is that the symbolic memory in universal symbolic execution maps symbolic addresses to symbolic expressions, whereas in concolic execution it maps concrete addresses to symbolic expressions. The fact that the domain of the symbolic memory in universal symbolic execution is symbolic rather than being concrete helps KRYSTAL to derive pointer predicates (such as `M.next.prev = M`) as described in the rest of the section.

Universal symbolic execution maintains two data structures: (1) A symbolic memory $\mathcal{S}$ that maintains mapping from symbolic addresses to symbolic expressions over symbolic variables, and (2) a path condition set Φ which is a set of symbolic constraints over symbolic variables that arise from execution of conditional statements.

We now formally describe the universal symbolic execution algorithm of KRYSTAL. Let $\mathcal{X}$ be the domain of symbolic variables. Let $\text{EXP}(\mathcal{X})$ and $\text{PRED}(\mathcal{X})$ be the domain of symbolic expressions and symbolic predicates over the symbolic variables in $\mathcal{X}$, respectively. Let $\mathcal{A}$ be the domain of symbolic addresses—any $a \in \mathcal{A}$ is always of the form v or $x.f$, where v and f are names and $x \in \mathcal{X}$. The symbolic memory $\mathcal{S}$ is then the map $\mathcal{A} \to \text{EXP}(\mathcal{X})$ and $\Phi \subseteq \text{PRED}(\mathcal{X})$.

The pseudo-code of the symbolic execution of KRYSTAL is given in Figure 5. The symbolic execution populates the symbolic memory $\mathcal{S}$ and the path condition set Φ, which are later used by KRYSTAL for local invariant generation. The function *eval_concrete* evaluates an expression according to the standard semantics of the language in the context of the concrete memory $\mathcal{M}$. The function *eval_symbolic*, which is described in Figure 7, evaluates an expression symbolically in the context of the symbolic memory $\mathcal{S}$. At every assignment statement $m := e$, KRYSTAL executes the assignment statement both concretely and symbolically. In the concrete execution, the address of the lvalue m is calculated and mapped to the value obtained by concretely evaluating the expression e. In the symbolic execution, the statement is executed symbolically (see Figure 6) by invoking *exec_symbolic*$(m, e, \mathcal{S}, i)$.

At every conditional statement `if` (e) `goto` ℓ, KRYSTAL evaluates e both concretely and symbolically to obtain the concrete boolean value b and the symbolic predicate c, respectively. If b is true, then c is added to the set Φ ; otherwise, if b is false, then $\neg c$ is added to the set.

We next describe *exec_symbolic* and *eval_symbolic* functions (see Figure 6 and Figure 7) which are different from standard sym-

```
$exec\_symbolic(m, e, \mathcal{S}, i)$
    $(r, \mathcal{S}, i) = eval\_symbolic(e, \mathcal{S}, i)$;
    match $m$
        case $v$: // the variable named $v$
            if $r$ is not a constant then
                $\mathcal{S} = \mathcal{S}[v \mapsto r]$;
            else // remove any mapping associated with $v$
                $\mathcal{S} = \mathcal{S} - v$;
        case $v.f$: // the field named $f$ of the variable $v$
            if $v \notin$ domain($\mathcal{S}$) then
                // introduce a new symbolic variable $x_i$
                $\mathcal{S} = \mathcal{S}[v \mapsto x_i]; i = i + 1$;
            if $r$ is not a constant then
                $\mathcal{S} = \mathcal{S}[\mathcal{S}(v).f \mapsto r]$;
            else // remove any mapping associated with $\mathcal{S}(v).f$
                $\mathcal{S} = \mathcal{S} - \mathcal{S}(v).f$;return $(\mathcal{S}, i)$;
```

Figure 6: Symbolic execution

```
$eval\_symbolic(e, \mathcal{S}, i)$
    match $e$
        case $c$: // $c$ is a constant
            return $(c, \mathcal{S}, i)$;
        case $e_1$ $op$ $e_2$:
            $(r_1, \mathcal{S}, i) = eval\_symbolic(e_1, \mathcal{S}, i)$;
            $(r_2, \mathcal{S}, i) = eval\_symbolic(e_2, \mathcal{S}, i)$;
            return $(r_1\ op\ r_2, \mathcal{S}, i)$; // symbolically apply $op$ on $r_1$ and $r_2$
        case new $T$:
            return $(0, \mathcal{S}, i)$; // return a dummy constant say 0
        case $v$: // the variable named $v$
            if $v \notin$ domain($\mathcal{S}$) then
                // introduce a new symbolic variable $x_i$
                $\mathcal{S} = \mathcal{S}[v \mapsto x_i]; i = i + 1$;
            return $(\mathcal{S}(v), \mathcal{S}, i)$;
        case $v.f$: // the field named $f$ of the variable $v$
            if $v \notin$ domain($\mathcal{S}$) then
                // introduce a new symbolic variable $x_i$
                $\mathcal{S} = \mathcal{S}[v \mapsto x_i]; i = i + 1$;
            if $\mathcal{S}(v).f \notin$ domain($\mathcal{S}$) then
                // introduce a new symbolic variable $x_i$
                $\mathcal{S} = \mathcal{S}[\mathcal{S}(v).f \mapsto x_i]; i = i + 1$;
            return $(\mathcal{S}(\mathcal{S}(v).f), \mathcal{S}, i)$;
```

Figure 7: Symbolic evaluation

bolic evaluation of concolic execution in three ways: (1) If an lvalue is of the form v, then we take the name v as the symbolic address of the lvalue v; if $v.f$ is the form of an lvalue, then we take $\mathcal{S}(v).f$ as the symbolic address of the lvalue, (2) if the right hand side of an assignment evaluates to a constant, then we remove any mapping associated with the symbolic address of the lvalue, and (3) if the symbolic address of a lvalue that is accessed has no mapping in the symbolic memory, then we create an entry in $\mathcal{S}$ by mapping the address to a fresh symbolic variable.

Rather than using the concrete address of $v.f$ (i.e. $\&(v.f)$) in $\mathcal{S}$, we use $\mathcal{S}(v).f$ as the symbolic address of the lvalue $v.f$. By doing so, the symbolic memory $\mathcal{S}$ maintains the logical structure of the heap: if $x.f$ maps to x' in $\mathcal{S}$, where x and x' are symbolic variables and the field f is of reference type, then we know that in the current state there is an edge via the field named f from the node denoted by x to the node denoted by x' in the heap.

The rationale behind creating a symbolic variable for every unmapped lvalue in $\mathcal{S}$ is that it helps us to maintain the relation among the various nodes in a data structure. In order to illustrate this, consider the following program, where the type or the class T has a field *next*.

```
$\ell_0 : a = $ new $T$;
$\ell_1 : b = $ new $T$;
$\ell_2 : a.next = b$;
$\ell_3 : b.next = a$;
```

Initially, $\mathcal{S} = [\ ]$ (and Φ remains the empty set throughout the symbolic execution.) After the execution of the statement labeled ℓ_0 and ℓ_1, $\mathcal{S}$ and Φ remain unchanged. The statement labeled ℓ_2 reads two lvalues a and b and writes to the lvalue *a.next*. However, both a and b have no mapping in $\mathcal{S}$. Therefore, KRYSTAL creates two fresh symbolic variables x_0 and x_1 and maps b to x_0 and a to x_1 in $\mathcal{S}$. The symbolic execution of the assignment statement ℓ_2 then maps $x_1.next$ to x_0. $\mathcal{S}$, therefore, becomes $[a \mapsto x_1, b \mapsto x_0, x_1.next \mapsto x_0]$. Similarly, after the execution of the statement labeled ℓ_3, $\mathcal{S}$ becomes $[a \mapsto x_1, b \mapsto x_0, x_1.next \mapsto x_0, x_0.next \mapsto x_1]$. This final symbolic memory completely characterizes the structure of the heap at the end of the execution: each symbolic variable represents a node, and each entry in $\mathcal{S}$ represents a directed edge. A close examination of the map also allows us to derive the local invariants $x_0.next.next = x_0$ and $x_1.next.next = x_1$. We formalize the technique of deriving these local invariants in the next section.

$exec\_symbolic(m, e, \mathcal{S}, i)$ symbolically evaluates e in the context of the symbolic memory $\mathcal{S}$. If m is of the form $v.f$ and if the lvalue v has no mapping in $\mathcal{S}$, then a fresh symbolic variable x_i is created and a mapping from v to x_i is added to $\mathcal{S}$. (The index i keeps track of the number of symbolic variables created so far.) A map from the symbolic address of m to the symbolic value of e is added to $\mathcal{S}$ if the symbolic evaluation of e is not a constant. If the symbolic evaluation of e is constant, then we remove from $\mathcal{S}$ any mapping associated with the symbolic address of m. This reflects the fact that we have written the symbolic address of m with a constant and any subsequent dereference of the address should not return a stale symbolic expression.

eval_symbolic recursively evaluates its first argument e. For example, if e is of the form e_1 *op* e_2, then *eval_symbolic* recursively evaluates e_1 and e_2 and then symbolically applies *op* to the results with applicable simplifications. If e is a constant, then symbolic evaluation of e trivially results in the constant. The above cases are straightforward. However, if e is an lvalue of the form v, then two situations may arise: the name v may not or may have a mapping in $\mathcal{S}$. In the former case, a new symbolic variable x_i is created and an entry $v \mapsto x_i$ is added to $\mathcal{S}$. The symbolic evaluation of v then results in $\mathcal{S}(v)$. If e is an lvalue of the form $v.f$, then as before if v has no mapping in $\mathcal{S}$, then an entry $v \mapsto x_i$ is added to $\mathcal{S}$. Similarly, if $\mathcal{S}(v).f$ has no mapping in $\mathcal{S}$, then an entry $\mathcal{S}(v).f \mapsto x_i$ is added to $\mathcal{S}$. The above two cases for the symbolic evaluation of an lvalue ensures that every lvalue that is read during the execution has a mapping in $\mathcal{S}$. If e is of the form new T, then a dummy constant, say 0, is returned.

At the end of the symbolic execution, we get a path condition set Φ and a symbolic memory $\mathcal{S}$. We use these two data structures to generate local axioms.

3.3 Local predicate generation

In this phase, KRYSTAL uses the path condition set Φ and the symbolic memory $\mathcal{S}$ to generate local predicates. We say that a predicate over $\mathcal{X}$ is a *local predicate* if the set of symbolic variables in the predicate is a singleton and the predicate contains no variable names. The goal of this phase is to generate a set of local predicates over $\mathcal{X}$ that are consistent with Φ and $\mathcal{S}$. Once we have such a set of local predicates, we can generalize the predicates to generate data structure invariants. We describe this generalization phase in the next section.

The key insights behind the current phase are the following. (1) If $a \mapsto d$ is an entry in $\mathcal{S}$, then the predicate $a = d$ is true for the symbolic execution. Let φ be the set $\{a = d \mid \mathcal{S}(a) = d\} \cup \Phi$, i.e. the set of all predicates that are true for the symbolic execution.

(2) One can generate local invariants by treating the set φ as a set of equations and by performing variable elimination on the set of equations. We next describe the above step.

Let us call predicates of the form $x = d$ or $d = x$, where $x \in \mathcal{X}$, *definition predicates*. Given a definition predicate $x = d$ or $d = x$, let $p[x \backslash d]$ be the predicate obtained by replacing any occurrence of the symbolic variable x in p by the symbolic expression d. We use the set φ (i.e. the set of predicates that are true for the current symbolic execution) to compute the set $\textsc{Derive}(\varphi)$ by replacing x by d in each predicate $p \in \varphi$ where $x = d$ or $d = x$ is also a predicate in φ. Formally,

$$\textsc{Derive}(\varphi) = \varphi \cup \{p'[x \backslash d] \mid p \in \varphi \text{ and } p' \in \varphi \text{ and } p \text{ is a definition predicate and } p \text{ is of the form } x = d \text{ or } d = x\}$$

Note that the predicates in $\textsc{Derive}(\varphi)$ are implied by the predicates in φ. Moreover, due to variable substitution, we get some predicates in $\textsc{Derive}(\varphi)$ that are local predicates. Subsequent applications of $\textsc{Derive}$ gives rise to more complex local predicates. Let ψ be the set of all local predicates in $\textsc{Derive}(\textsc{Derive}(\varphi))$. We restrict the number of applications of $\textsc{Derive}$ to two in order to get small, but rich enough, set of local predicates. Note that if we apply $\textsc{Derive}$ more times , we will get more complex, but rich set of predicates. However, our experiments showed that only two applications of $\textsc{Derive}$ are sufficient to infer the most interesting local invariants.

For example, if at the end of symbolic execution along a path we generate a symbolic memory $\mathcal{S} = \{x_0.\mathit{next} \mapsto x_1, x_1.\mathit{prev} \mapsto x_0, x_0.\mathit{val} \mapsto x_2, x_1.\mathit{val} \mapsto x_3\}$ and a path condition set $\Phi = \{x_2 \leq x_3, x_2 \neq x_3\}$, then $\varphi = \{x_0.\mathit{next} = x_1, x_1.\mathit{prev} = x_0, x_0.\mathit{val} = x_2, x_1.\mathit{val} = x_3, x_2 \leq x_3, x_2 \neq x_3\}$ and the local predicates in $\textsc{Derive}(\textsc{Derive}(\varphi))$ forms the set $\psi = \{x_1.\mathit{prev.next} = x_1, x_0.\mathit{next.prev} = x_0, x_1.\mathit{prev.val} \leq x_1.\mathit{val}, x_0.\mathit{val} \leq x_0.\mathit{next.val}, x_1.\mathit{prev.val} \neq x_1.\mathit{val}, x_0.\mathit{val} \neq x_0.\mathit{next.val}\}$.

3.4 Generating local invariant templates from local predicates along an execution

The local predicates generated in the previous phase are over various symbolic variables. In this phase, we generalize them to predicates over a single generic symbolic variable. We call such predicates as *templates*. Templates are predicates that could potentially hold on any node of a data structure. The template generation takes place in two steps. First, we simplify sets of local predicates by using simple theorem proving. For example, we simplify the local predicates $x_0.\mathit{val} \leq x_0.\mathit{next.val}$ and $x_0.\mathit{val} \neq x_0.\mathit{next.val}$ to the local predicate $x_0.\mathit{val} < x_0.\mathit{next.val}$. Second, we replace any symbolic variable in any local predicate by a generic symbolic variable M to generate a set of templates for local invariants. For example, the set $\{x_1.\mathit{prev.next} = x_1, x_0.\mathit{next.prev} = x_0, x_1.\mathit{prev.val} < x_1.\mathit{val}, x_0.\mathit{val} < x_0.\mathit{next.val}\}$ obtained from the set $\psi = \{x_1.\mathit{prev.next} = x_1, x_0.\mathit{next.prev} = x_0, x_1.\mathit{prev.val} \leq x_1.\mathit{val}, x_0.\mathit{val} \leq x_0.\mathit{next.val}, x_1.\mathit{prev.val} \neq x_1.\mathit{val}, x_0.\mathit{val} \neq x_0.\mathit{next.val}\}$ through simplification generates the set of templates $\{M.\mathit{prev.next} = M, M.\mathit{next.prev} = M, M.\mathit{prev.val} < M.\mathit{val}, M.\mathit{val} < M.\mathit{next.val}\}$.

The templates generated using the above two steps represent potential local invariants that are true for any node of the given data structure along the execution path.

The templates that we generate are similar in functionality to those that Daikon uses to generate its invariants. The crucial difference between Daikon and KRYSTAL is that we generate invariant templates in KRYSTAL through universal symbolic execution whereas in Daikon the templates are fixed and provided by the user. Therefore, unlike Daikon, KRYSTAL can discover complex templates involving field accesses.

We next formally describe the two steps of template generation.

Step 1: Simplification.

In the simplification step, we perform a very limited form of simplification by replacing every set of *similar* local predicates in ψ by their conjunction. We say that two local predicates are *similar* if both of them have the same symbolic variable and they can be written in the normalized forms $\sum c_i e_i + d \bowtie 0$ and $\sum c'_i e_i + d' \bowtie 0$, respectively where c_i's, c'_i's, d and d' are constant literals, e_i's are the non-constant parts, and $\bowtie \in \{<, >, \leq, \geq, =, \neq\}$. For example, $x_0.\mathit{val} > 1$ and $x_0.\mathit{val} \leq 8$ are similar. Given two similar predicates $\sum c_i e_i + d \bowtie 0$ and $\sum c'_i e_i + d' \bowtie 0$ in ψ, we replace them in ψ by their conjunction $\sum c_i e_i + d \bowtie 0 \wedge \sum c'_i e_i + d' \bowtie 0$. This simplification step helps KRYSTAL to generate more precise and compact local data structure invariants.

The rationale behind taking a conjunction of the similar local predicates is the following. Any symbolic variable in an execution has a corresponding concrete value in the concrete execution. A local predicate over a symbolic variable constrains the possible values that the symbolic variable (and its fields) can assume while executing the same execution path. Therefore, a conjunction of all the local predicates in ψ over a given symbolic variable also constrains the possible values that the symbolic variable and related fields can assume along the execution path.

Step 2: Template Generation.

Let ψ' be the set of local predicates after simplification of the set ψ. In this step, we first remove any local predicate whose symbolic variable does not correspond to type T, where the class T represents a node in the data structure. We then replace any symbolic variable in any predicate in the modified set ψ' by a generic symbolic variable M.

3.5 Collecting local invariant templates across executions

All the phases of the algorithm discussed so far are applied to a single execution trace. After running these phases along a single execution path, we get a set of local invariant templates. In the current phase, we accumulate the templates generated from multiple execution traces, simplify them, and relax them to infer the likely local invariant templates for the data structure. We next describe these three steps.

Step 1: Accumulating templates across executions.

KRYSTAL performs universal symbolic execution on the execution trace for each test input and computes the set of local invariant templates for each execution path. KRYSTAL then takes the union of these sets of local invariant templates. Let Γ be the set obtained after taking union. The templates in the set Γ then represent local data structure invariants that could potentially hold over any data structure node.

Step 2: Simplification.

After constructing the set Γ, KRYSTAL performs further simplification by taking disjunction of similar templates, where similarity is defined in the same way as similarity of local predicates (see Section 3.4.) To understand the rationale behind taking disjunction, recall that, in the previous phase, sets of similar local predicates were simplified to a single local predicate by taking a conjunction. It follows that distinct templates are generated from local predicates over dissimilar symbolic variables or from different execution traces. Therefore, a disjunction of all the local invariant templates in Γ represents the constraint that can potentially hold for any node in the data structure across execution traces.

The simplification step helps to reduce the number of templates and prevent eliminating some relevant invariants in the next phase, which is the validation phase. For example, if the templates $M.color = $ 'R' and $M.color = $ 'B' are generated along different execution paths, respectively, then we definitely know that either of the templates do not hold for all execution paths. Therefore, in the validation phase, which is described next, we will eliminate both of them. However, if we take their disjunction $M.color = $ 'R' $\vee$ $M.color = $ 'B', then the resultant invariant holds over all execution paths and we keep it in the final set of local invariants that we infer.

Step 3: Relaxation.

After simplification we get a set of local invariant templates, say Γ', that contains likely local data structure invariants. We relax each template in the set Γ' by incorporating some implicit constraints. For example, if $M.next.prev = M$, then we know from the pointer dereference semantics that the template is true if $M \neq \texttt{null}$ and $M.next \neq \texttt{null}$. We, therefore, incorporate these implicit constraints in the template by changing it to $(M \neq \texttt{null} \wedge M.next \neq \texttt{null}) \Rightarrow (M.next.prev = M)$. This is similar to guarded implications added by Daikon [8].

After the relaxation step, let the modified set of local invariant templates be Δ. In the final phase, we check all the templates in the set Δ against all execution paths and only keep those invariants that hold along all execution paths.

3.6 Using local invariant templates to generate likely local data structure invariants

In this final phase, we run our program on all test inputs and check at the end of each test run that each template in Δ holds on each node of the data structure. We retain only those templates that hold on all data structure nodes on all test executions. These templates represent the likely local data structure invariants inferred by KRYSTAL.

4. IMPLEMENTATION AND EVALUATION

We have implemented the KRYSTAL algorithm in programming language Java. The tool takes as input the source code for a Java program. The front-end uses the Soot compiler framework [25] to perform instrumentation. Instrumentation inserts various method calls into Java byte-code. The inserted method calls log the trace of an execution at runtime. The backend parses a trace generated from an execution and performs universal symbolic execution on the trace. We also use the YICES theorem prover [6] to solve satisfiability and validity queries. Such queries are used in the simplification of predicates during the conjunction and disjunction operations.

We evaluate KRYSTAL on a set of data structure implementations. We assume that each implementation provides a set of API methods to manipulate the data structure.

4.1 Experimental Setup

To setup the evaluation process, we write a test harness for each data structure. The test harness is a Java main method that creates an instance object of the data structure class and calls the object's API methods in a random sequence. The arguments to the different API methods are also randomly generated from a finite domain. The test harness is combined with the instrumented program (data structure implementation) to form a self-executable program, which can be compiled and executed to generate the execution traces. The backend of KRYSTAL performs universal symbolic execution for each API method invocation along the execution to generate the invariants for the data structure.

We ran KRYSTAL over ten data structures. We give a high-level description of these data structures below. The data structures SORTED SINGLY LINKED LIST, SORTED DOUBLY LINKED LIST, and SORTED SKIP LIST implement list-based data structures and they maintain their elements in an increasing order of the values in their *key* fields. BINARY SEARCH TREE maintains its elements in a binary tree, such that the value of *key* at a node is larger than any of the *key* values in the left sub-tree and is at least as small as the values of the *key* fields in the right sub-tree. AVL TREE and RED BLACK TREE implement self-balancing binary search trees, performing various tree restructuring operations to balance the height property. Two implementations of the red-black tree are used, one that uses the conventional *sentinel* node, and the other, implemented without the *sentinel* node. SPLAY TREE also implements a self-balancing binary search tree with an additional property that recently accessed elements are quick to access again. Moreover, unlike other types of self balancing trees, splay trees preserve the order of the nodes with identical keys within the tree. MAX HEAP is a priority queue implemented with a binary tree. It maintains the property that the *key* at a node, is at least as large as the keys in its left and right subtrees. N-ARY TREE implements a tree, where each node can have an arbitrary number of children. Every node has a pointer to its parent node and a pointer to its first child, while the nodes in the same level are connected by sibling pointers.

4.2 Experimental Results

Figure 8 summarizes our experimental results. We list all the invariants that KRYSTAL finally reports to the user for each data structure. The second column in the table lists the time taken by KRYSTAL to generate the final set of invariants. For each invariant that KRYSTAL generates, for comparison, we also indicate if it was inferred by a closely related work for dynamic invariant inference—Daikon.

The Daikon invariant detector included in the Daikon distribution computes invariants over scalars and arrays. Therefore, for Daikon, we rewrote the programs to linearize collections into arrays at the relevant program points. This was done in accordance with the description in [9]. For every data structure collection, explicit variables in the program are selected as roots. For each field that leads from an object to another object of the same type, we create an array and output the non-recursive type fields of the two objects as successive elements in the same array.

We manually generated the set of invariants for each data structure and compared the hand-generated output with the set of invariants that is generated by KRYSTAL. Based on this, we classify the invariants into three classes–(1) Interesting: relevant invariants that are reported by KRYSTAL, (2) Irrelevant: invariants that are reported by KRYSTAL, which hold true for the data structure, but are not interesting from the programmer/user point of view, and (3) Spurious: incorrect invariants that are reported by KRYSTAL. The class of invariant in each case is also indicated in Figure 8.

We analyze the quality of the output generated by KRYSTAL using three metrics—soundness (i.e. we do not generate spurious invariants), completeness (i.e. we generate all possible local data structure invariants), and relevance (i.e. we generate only interesting/useful invariants.)

Note that we are dependent on the quality of the test suite for both the generation of invariants and for checking the validity of the invariants. Therefore, our technique is neither sound (spurious invariants may be reported) nor complete (we may miss invariants.) However, for all the given data structures, we were able to get good results.

We will now take a closer look at the invariants that KRYSTAL generated. For each test program, the test harness instantiates the

Data Structure	Time	Data Structure Invariants	Invariant Type[†]	Daikon [‡]
Sorted Singly Linked List	1.31s	$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.next} \neq \mathtt{null}) \Rightarrow (\mathtt{M.key} \leq \mathtt{M.next.key})$	Interesting	Yes
Sorted Doubly Linked List	1.22s	$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.prev} \neq \mathtt{null}) \Rightarrow (\mathtt{M.prev.next} = \mathtt{M})$	Interesting	No
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.next} \neq \mathtt{null}) \Rightarrow (\mathtt{M.next.prev} = \mathtt{M})$	Interesting	No
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.prev} \neq \mathtt{null}) \Rightarrow (\mathtt{M.key} \geq \mathtt{M.prev.key})$	Interesting	No
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.next} \neq \mathtt{null}) \Rightarrow (\mathtt{M.next.key} \geq \mathtt{M.key})$	Interesting	Yes
Sorted Skip List	3.14s	$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.next3} \neq \mathtt{null}) \Rightarrow (\mathtt{M.next3.key} \geq \mathtt{M.key})$	Interesting	Yes
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.next2} \neq \mathtt{null}) \Rightarrow (\mathtt{M.next2.key} \geq \mathtt{M.key})$	Interesting	Yes
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.next1} \neq \mathtt{null}) \Rightarrow (\mathtt{M.next1.key} \geq \mathtt{M.key})$	Interesting	Yes
Binary Search Tree	1.52s	$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.left} \neq \mathtt{null}) \Rightarrow (\mathtt{M.left.parent} = \mathtt{M})$	Interesting	No
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.right} \neq \mathtt{null}) \Rightarrow (\mathtt{M.right.parent} = \mathtt{M})$	Interesting	No
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.left} \neq \mathtt{null}) \Rightarrow (\mathtt{M.key} \geq \mathtt{M.left.key})$	Interesting	Yes
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.right} \neq \mathtt{null}) \Rightarrow (\mathtt{M.key} < \mathtt{M.right.key})$	Interesting	Yes
Max Heap	2.28s	$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.left} \neq \mathtt{null}) \Rightarrow (\mathtt{M.left.parent} = \mathtt{M})$	Interesting	No
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.right} \neq \mathtt{null}) \Rightarrow (\mathtt{M.right.parent} = \mathtt{M})$	Interesting	No
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.left} \neq \mathtt{null}) \Rightarrow (\mathtt{M.key} \geq \mathtt{M.left.key})$	Interesting	Yes
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.parent} \neq \mathtt{null}) \Rightarrow (\mathtt{M.parent.key} \geq \mathtt{M.key})$	Interesting	No
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.right} \neq \mathtt{null}) \Rightarrow (\mathtt{M.key} \geq \mathtt{M.right.key})$	Interesting	Yes
N-ary Tree	1.31s	$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.firstChild} \neq \mathtt{null}) \Rightarrow (\mathtt{M.firstChild.parent} = \mathtt{M})$	Interesting	No
AVL Tree	19.47s	$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.left} \neq \mathtt{null}) \Rightarrow (\mathtt{M.left.parent} = \mathtt{M})$	Interesting	No
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.right} \neq \mathtt{null}) \Rightarrow (\mathtt{M.right.parent} = \mathtt{M})$	Interesting	No
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.left} \neq \mathtt{null}) \Rightarrow ((\mathtt{M.left.key} \leq \mathtt{M.key}) \wedge (\mathtt{M.left.key} \neq \mathtt{M.key}))$	Interesting	Yes
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.right} \neq \mathtt{null}) \Rightarrow ((\mathtt{M.right.key} \geq \mathtt{M.key}) \wedge (\mathtt{M.right.key} \neq \mathtt{M.key}))$	Interesting	Yes
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.parent} \neq \mathtt{null}) \Rightarrow (\mathtt{M.parent.key} \neq \mathtt{M.key})$	Interesting	No
Red Black Tree (implemented using *sentinel* node	4.32s	$(\mathtt{M} \neq \mathtt{null}) \Rightarrow (\mathtt{M} \neq \mathtt{M.right})$	Irrelevant	No
		$(\mathtt{M} \neq \mathtt{null}) \Rightarrow (\mathtt{M.key} \geq \mathtt{M.left.key})$	Interesting	Yes
		$(\mathtt{M} \neq \mathtt{null}) \Rightarrow ((\mathtt{M.color} = \mathtt{1}) \vee (\mathtt{M.color} = \mathtt{0}))$	Interesting	Yes
		$(\mathtt{M} \neq \mathtt{null}) \Rightarrow (\mathtt{M} \neq \mathtt{M.left})$	Irrelevant	No
		$(\mathtt{M} \neq \mathtt{null}) \Rightarrow (\mathtt{M} \neq \mathtt{M.parent})$	Irrelevant	No
Red Black Tree (implemented without *sentinel* node	3.42s	$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.right} \neq \mathtt{null}) \Rightarrow (\mathtt{M.right.key} \geq \mathtt{M.key})$	Interesting	Yes
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.right} \neq \mathtt{null}) \Rightarrow (\mathtt{M.right.parent} = \mathtt{M})$	Interesting	No
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.left} \neq \mathtt{null}) \Rightarrow (\mathtt{M.left.parent} = \mathtt{M})$	Interesting	No
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.left} \neq \mathtt{null}) \Rightarrow (\mathtt{M.key} \geq \mathtt{M.left.key})$	Interesting	Yes
		$(\mathtt{M} \neq \mathtt{null}) \Rightarrow ((\mathtt{M.color} = \mathtt{1}) \vee (\mathtt{M.color} = \mathtt{0}))$	Interesting	Yes
Splay Tree	15.5s	$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.left} \neq \mathtt{null}) \Rightarrow (\mathtt{M.left.parent} = \mathtt{M})$	Interesting	No
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.right} \neq \mathtt{null}) \Rightarrow (\mathtt{M.right.parent} = \mathtt{M})$	Interesting	No
		$(\mathtt{M} \neq \mathtt{null}) \Rightarrow (\mathtt{M.parent} \neq \mathtt{M})$	Irrelevant	No
		$(\mathtt{M} \neq \mathtt{null}) \Rightarrow (\mathtt{M} \neq \mathtt{M.left})$	Irrelevant	No
		$(\mathtt{M} \neq \mathtt{null}) \Rightarrow (\mathtt{M} \neq \mathtt{M.right})$	Irrelevant	No
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.left} \neq \mathtt{null}) \Rightarrow (\mathtt{M.left.key} < \mathtt{M.key})$	Interesting	Yes
		$(\mathtt{M} \neq \mathtt{null} \wedge \mathtt{M.right} \neq \mathtt{null}) \Rightarrow (\mathtt{M.right.key} \geq \mathtt{M.key})$	Interesting	Yes

Figure 8: Experimental Results ([†]Each invariant is marked as Interesting, Irrelevant or Spurious—see Section 4.2 for details. [‡]Indicates whether Daikon generates the specified invariant.)

data structure and then makes calls to the API methods in a random sequence. It uses randomly generated input values for method arguments. In each case, a total of 50 API method executions were analyzed. Even with this limited test suite, all the invariants that we finally report to the user are correct (i.e. sound.)

Moreover, most of the invariants that KRYSTAL generates are invariants that are relevant to the programmer. On the contrary, Daikon produces several irrelevant invariants that either reflect properties of the test suite (for eg., $M.key \geq 0$) or are spurious (for eg., $size(left.key[]) \geq size(right.key[])$.)

There are a few relevant data structure invariants that KRYSTAL fails to generate. These are listed in Figure 9. Again, for comparison, we indicate whether these invariants were inferred by Daikon. Note that none of the relevant invariants that KRYSTAL failed to generate, was reported by Daikon. This reinforces our belief that a vast majority of interesting invariants are encoded in the program.

One class of invariants that KRYSTAL fails to generate is conditional invariants, that is, invariants that are true subject to a specified condition. For instance, in case of the implementation of red-black tree using a *sentinel* node, the invariants are expected to hold over all nodes in the data structure, excepting the *sentinel* node. Since we do not make this distinction, choosing instead to represent all nodes in the data structure with the generic symbolic variable M, we end up generating several invariants that are eliminated in the validation phase, and hence are not reported to the user.

On closer inspection of the invariants that KRYSTAL failed to generate, we discovered that, in most cases, the missing invariants were present in incorrect forms among the invariants that were eliminated in the final validation phase. For example, in case of a binary search tree, we fail to generate the invariant $(M \neq \mathtt{null} \wedge M.parent \neq \mathtt{null}) \Rightarrow ((M.parent.left = M) \vee (M.parent.right = M))$. However, we eliminate the invariants $(M \neq \mathtt{null} \wedge M.parent \neq \mathtt{null}) \Rightarrow (M.parent.left = M)$ and $(M \neq \mathtt{null} \wedge M.parent \neq \mathtt{null}) \Rightarrow (M.parent.right = M)$ in the final phase. This possibly indicates that our definition of *similar* invariants, used during simplification of invariants is not generic enough.

5. RELATED WORK

There is a rich literature on invariant generation [8, 7, 11, 2, 28, 24, 30, 1, 18, 23, 5]. KRYSTAL has several advantages and disadvantages over existing techniques for invariant generation. We next position our work by comparing KRYSTAL with several closely related work.

Data Structure	Data Structure Invariants	Daikon
BINARY SEARCH TREE	$(\texttt{M} \neq \texttt{null} \wedge \texttt{M.parent} \neq \texttt{null}) \Rightarrow (\texttt{M.parent.left} = \texttt{M}) \vee (\texttt{M.parent.right} = \texttt{M})$	No
MAX HEAP	$(\texttt{M} \neq \texttt{null} \wedge \texttt{M.parent} \neq \texttt{null}) \Rightarrow (\texttt{M.parent.left} = \texttt{M}) \vee (\texttt{M.parent.right} = \texttt{M})$	No
N-ARY TREE	$(\texttt{M} \neq \texttt{null} \wedge \texttt{M.rightSibling} \neq \texttt{null}) \Rightarrow (\texttt{M.parent} = \texttt{M.rightSibling.parent})$	No
AVL TREE	$(\texttt{M} \neq \texttt{null} \wedge \texttt{M.parent} \neq \texttt{null}) \Rightarrow (\texttt{M.parent.left} = \texttt{M}) \vee (\texttt{M.parent.right} = \texttt{M})$	No
	$(\texttt{M} \neq \texttt{null}) \Rightarrow (\ (\texttt{M.left} \neq \texttt{null}) \Rightarrow (\texttt{M.height} = \texttt{M.left.height} + 1) \vee$ $(\texttt{M.right} \neq \texttt{null}) \Rightarrow (\texttt{M.height} = \texttt{M.right.height} + 1)\ )$	No
RED BLACK TREE (impl. using *sentinel* node)	$(\texttt{M} \neq \texttt{null} \wedge \texttt{M.parent} \neq \texttt{null} \wedge \texttt{M.parent} \neq \textit{sentinel}) \Rightarrow (\texttt{M.parent.left} = \texttt{M}) \vee (\texttt{M.parent.right} = \texttt{M})$	No
	$(\texttt{M} \neq \texttt{null} \wedge \texttt{M.right} \neq \texttt{null} \wedge \texttt{M} \neq \textit{sentinel} \wedge \texttt{M.right} \neq \textit{sentinel}) \Rightarrow \texttt{M.right.key} \geq \texttt{M.key}$	No
RED BLACK TREE (impl. without *sentinel* node)	$(\texttt{M} \neq \texttt{null} \wedge \texttt{M.parent} \neq \texttt{null}) \Rightarrow (\texttt{M.parent.left} = \texttt{M}) \vee (\texttt{M.parent.right} = \texttt{M})$	No
SPLAY TREE	$(\texttt{M} \neq \texttt{null} \wedge \texttt{M.parent} \neq \texttt{null}) \Rightarrow (\texttt{M.parent.left} = \texttt{M}) \vee (\texttt{M.parent.right} = \texttt{M})$	No

Figure 9: Relevant invariants that KRYSTAL failed to generate

Comparison with DySy.

Csallner et al. [5] propose an approach, called DySy, that combines symbolic execution with dynamic testing to infer preconditions and postconditions for program methods. This work is closely related to our approach—both approaches collect predicates from the statements executed in the program and extract invariants by performing variable substitutions.

We differ from DySy mainly in the way we perform our symbolic execution and in the way we infer our invariants from symbolic execution. We use universal symbolic execution, which we argue is more powerful in deriving complex invariants over heaps. In universal symbolic execution we introduce a fresh symbolic variable whenever the symbolic execution reads the value at an address that has no mapping in the symbolic memory. Subsequently, in contrast with DySy, which always expresses the predicates in terms of the program (or method) inputs, we can generate symbolic predicates over arbitrary heap memory which may not be data dependent on program inputs. This helps us generate invariants that might be missed by DySy. For example, in Section 2, we show that we can generate the invariant $((M \neq \texttt{null} \wedge M.\textit{parent} \neq \texttt{null}) \Rightarrow (M.\textit{depth} = M.\textit{parent.depth} + 1))$ for a binary search tree. DySy fails to infer this invariant since the values of neither $M.\textit{depth}$ nor $M.\textit{parent.depth}$ are derived from program/method inputs.

DySy uses the symbolic path condition set and the symbolic memory obtained from symbolic execution to generate pre and post-conditions of methods, respectively. Since our goal is to generate invariants over heap memories, method pre and post conditions are not useful for our purpose. We, therefore, use a richer symbolic memory and symbolic path conditions obtained from universal symbolic execution to derive likely invariant templates. These templates are subsequently used to generate the likely local invariants using Daikon [8] like dynamic program analysis.

DySy uses a combination of precondition and postcondition that enables it to infer conditional invariants in post-conditions of methods. For instance, in case of red-black implementation with *sentinel* node, DySy infers post-condition invariants of the form: $(\textit{root} = \textit{sentinel}) \Rightarrow (\textit{return new}$ NODE$)$. On the contrary, KRYSTAL is currently not capable of inferring conditional invariants.

Comparison with Daikon and DIDUCE.

Other approaches based on dynamic analysis include Daikon and DIDUCE. Hangal and Lam [13] propose DIDUCE, which uses online analysis to discover simple invariants over the values of program variables. Daikon [8, 9, 7] presents a more complex form of dynamic invariant discovery technique. Daikon takes as input a fixed set of templates that encode patterns for equalities, inequalities, and affine relationships that can hold among program variables. At specific program points such as method entries, method exits, and loop heads, Daikon instantiates the templates with the values of variables in scope and checks for invariants that hold true for all executions on the test suite. A big advantage of Daikon is that it can discover complex and implicit program invariants. Such invariants not only help to identify the programmer's intentions, but also to establish the quality of the test suite. One disadvantage of these tools is that they cannot often generate complex invariants related to data structure shapes. Moreover, the quality of the invariants inferred by Daikon depends on the pre-set invariant patterns and test-suite.

Like these tools, we are dependent on the quality of the test suite for generating the invariants. However, since we do not use a fixed set of templates, choosing instead to extract them from the program text by performing symbolic analysis of the program along test executions, it is less likely that we generate an invariant that is a property of the test suite as opposed to the program. For example, consider an example of a program that carries out insertion sort using a linked list. For Daikon to generate the correct set of invariants in this case, the test suite has to include test cases containing duplicate elements. On the other hand, even with few test cases, we identify the correct relation as $M.\textit{element} \leq M.\textit{next.element}$, using the information from the program conditionals, as opposed to $M.\textit{element} < M.\textit{next.element}$, that might be reported by Daikon with limited test suite.

An advantage of Daikon over the symbolic technique DySy is that Daikon can identify relationships that hold between program variables that are not directly encoded in the program. For example, affine relationships like those constraining the sum/difference of variables, may be implicit in the program. Even though such cases exist, we believe that very few interesting invariants are left unspecified in some way in the program.

Comparison with Deryaft.

Deryaft [18] is a tool that specializes in generating constraints of complex data structures. Deryaft takes as input a handful of concrete data structures of small sizes and generates a predicate that represents their structural integrity constraints. Deryaft has a preset list of templates, which are written to encode common properties that hold between objects in a data structure. In contrast, KRYSTAL generates these templates through symbolic execution; therefore, KRYSTAL can discover new templates.

Comparison with static tools.

Logozzo [17] proposed a static approach that derives invariants for a class as a solution of a set of equations derived from the program source. Houdini [11] is an annotation assistant for ESC/Java [10]. It generates a large number of candidate invariants and repeatedly invokes the ESC/Java checker to remove unprovable annotations, until no more annotations are refuted. The initial candidate annotation set is generated from the program text using a set of checker-specific heuristics. Houdini's initial candidate invariants are all possible arithmetic and (in)equality comparisons among fields and "interesting constants" (such as `null`, `true` or other constants extracted from program text.) The usefulness of the inferred annotations in Houdini's output is dependent on preset heuristics or user intervention.

In contrast, KRYSTAL is a fully automated technique, that extracts templates from the execution of the program, thereby, making it possible to infer complex relationships without any user help.

Comparison with concolic execution.

As in concolic execution [12, 22], KRYSTAL intertwines concrete execution with symbolic execution. However, KRYSTAL uses universal symbolic execution, which differs from classic symbolic execution and concolic execution in two key aspects.

In concolic execution, new symbolic variables are generated only at input statements. In contrast, universal symbolic execution generates a fresh symbolic variable whenever the execution reads the value at an address that has no mapping in the symbolic memory.

The second difference between concolic and universal symbolic execution is in the format of the symbolic memory. In concolic execution, the symbolic memory maps concrete memory addresses to symbolic expressions; whereas, in universal symbolic execution, it maps symbolic addresses (expressions over program variables, symbolic variables, and field references) to symbolic expressions.

This is ideally suited for inference of data structure invariants, since it helps us derive pointer predicates and infer complex relationships among field references.

6. CONCLUSION

We have introduced a novel variant of symbolic execution—universal symbolic execution. We have demonstrated its utility by using it to track complex relationships among nearby heap cells, which are subsequently used in the generation of data structure invariants. Our experiments on common data structures suggest that KRYSTAL is effective in inferring the relevant local invariants.

Despite the advantages, KRYSTAL is still dependent on the quality of the test suite (though to a lesser extent compared to previous techniques like Daikon) for completeness and soundness. Another limitation of KRYSTAL is that it cannot generate global data structure invariants such as the non-circularity of a linked list.

Acknowledgments

We are grateful to Nikolai Tillman and Christoph Csallner for clarifying how DySy differs from the current work, and helping us set up the tool for comparison studies. We would like to thank Jacob Burnim, Pallavi Joshi, and A. J. Shankar for providing valuable comments on a draft of this paper. This work is supported in part by the NSF Grant CNS-0720906 and a gift from Toyota.

7. REFERENCES

[1] R. Alur, P. Cerny, G. Gupta, P. Madhusudan, W. Nam, and A. Srivastava. Synthesis of Interface Specifications for Java Classes. In *Proceedings of POPL'05 (32nd ACM Symposium on Principles of Programming Languages)*, 2005.

[2] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, 2002.

[3] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2:215–222, 1976.

[4] P. D. Coward. Symbolic execution systems-a review. *Software Engineering Journal*, 3(6):229–239, 1988.

[5] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *30th ACM/IEEE International Conference on Software Engineering (ICSE)*, May 2008. To appear.

[6] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for dpll(t). In *Computer Aided Verification*, volume 4144 of *LNCS*, pages 81–94, 2006.

[7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb. 2001.

[8] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, June 2000.

[9] M. D. Ernst, W. G. Griswold, Y. Kataokay, and D. Notkin. Dynamically discovering program invariants involving collections. In *TR UW-CSE-99-11-02, University of Washington*, 2000.

[10] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. of the ACM SIGPLAN Conference on Programming language design and implementation (PLDI'02)*, pages 234–245, 2002.

[11] C. Flanagan and R. M. Leino. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe (FME)*, pages 500–517, March 2001.

[12] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[13] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, May 2002.

[14] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th Int. Conf. on TACAS*, pages 553–568, 2003.

[15] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[16] G. Lee, J. Morris, K. Parker, G. A. Bundell, and P. Lam. Using symbolic execution to guide test generation: Research articles. *Softw. Test. Verif. Reliab.*, 15(1):41–61, 2005.

[17] F. Logozzo. Automatic inference of class invariants. In *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, January 2004.

[18] M. Z. Malik, A. Pervaiz, , and S. Khurshid. Generating representation invariants of structurally complex data. In *TACAS*, pages 34–49, 2007.

[19] S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *Proceedings of Computer Aided Verification (CAV)*, pages 476–490, 2005.

[20] J. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[21] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. Technical Report UIUCDCS-R-2005-2597, UIUC, 2005.

[22] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*. ACM, 2005.

[23] M. Taghdiri and D. Jackson. Inferring specifications to detect errors in code. *Automated Software Engg.*, 14(1):87–121, 2007.

[24] N. Tillmann, F. Chen, and W. Schulte. Discovering likely method specifications. In *ICFEM*, pages 717–736, 2006.

[25] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON 1999*, pages 125–135, 1999.

[26] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, 2004.

[27] S. Visvanathan and N. Gupta. Generating test data for functions with pointer inputs. In *17th IEEE International Conference on Automated Software Engineering*, 2002.

[28] J. Whaley, M. C. Martin, and M. S. Lam. Automatic Extraction of Object-Oriented Component Interfaces. In *Proceedings of ACM SIGSOFT ISSTA'02 (International Symposium on Software Testing and Analysis)*, 2002.

[29] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Procs. of TACAS*, 2005.

[30] J. Yang and D. Evans. Dynamically inferring temporal properties. In *PASTE '04*, pages 23–28. ACM, 2004.