# Generalized Symbolic Execution for Model Checking and Testing

Sarfraz Khurshid[1], Corina S. Păsăreanu[2], and Willem Visser[3]

[1] MIT Laboratory for Computer Science, Cambridge, MA 02139, USA
khurshid@lcs.mit.edu
[2] Kestrel Technology LLC and [3] RIACS/USRA,
NASA Ames Research Center, Moffett Field, CA 94035, USA
{pcorina,wvisser}@email.arc.nasa.gov

**Abstract.** Modern software systems, which often are concurrent and manipulate complex data structures must be extremely reliable. We present a novel framework based on symbolic execution, for automated checking of such systems. We provide a two-fold generalization of traditional symbolic execution based approaches. First, we define a source to source translation to instrument a program, which enables standard model checkers to perform symbolic execution of the program. Second, we give a novel symbolic execution algorithm that handles dynamically allocated structures (e.g., lists and trees), method preconditions (e.g., acyclicity), data (e.g., integers and strings) and concurrency. The program instrumentation enables a model checker to automatically explore different program heap configurations and manipulate logical formulae on program data (using a decision procedure). We illustrate two applications of our framework: checking correctness of multi-threaded programs that take inputs from unbounded domains with complex structure and generation of non-isomorphic test inputs that satisfy a testing criterion. Our implementation for Java uses the Java PathFinder model checker.

## 1 Introduction

Modern software systems, which often are concurrent and manipulate complex dynamically allocated data structures (e.g., linked lists or binary trees), must be extremely reliable and correct. Two commonly used techniques for checking correctness of such systems are testing and model checking. Testing is widely used but usually involves manual test input generation. Furthermore, testing is not good at finding errors related to concurrent behavior. Model checking, on the other hand, is automatic and particularly good at analyzing (concurrent) reactive systems. A drawback of model checking is that it suffers from the state-space explosion problem and typically requires a closed system, i.e., a system together with its environment, and a bound on input sizes [4, 6, 9, 19].

We present a novel framework based on *symbolic execution* [14], which automates test case generation, allows model checking concurrent programs that take inputs from unbounded domains with complex structure, and helps combat state-space explosion. Symbolic execution is a well known program analysis

technique, which represents values of program variables with *symbolic values* instead of concrete (initialized) data and manipulates expressions involving symbolic values. Symbolic execution traditionally arose in the context of checking sequential programs with a fixed number of integer variables. Several recent approaches [3, 5, 7] extend the traditional notion of symbolic execution to perform various program analyses; these approaches, however, require dedicated tools to perform the analyses and do not handle concurrent systems with complex inputs.

We provide a two-fold generalization of traditional symbolic execution. First, we define a source to source translation to instrument a program, which enables symbolic execution of the program to be performed using a standard model checker (for the underlying language) without having to build a dedicated tool. The instrumented program can be symbolically executed by any model checker that supports nondeterministic choice. The model checker checks the program by automatically exploring different program heap configurations and manipulating logical formulae on program data values (using a decision procedure).

Second, we give a novel symbolic execution algorithm that handles dynamically allocated structures (e.g., lists and trees), method preconditions (e.g., acyclicity of lists), data (e.g., integers and strings) and concurrency. To symbolically execute a method, the algorithm uses *lazy initialization*, i.e., it initializes the components of the method inputs on an "as-needed" basis, without requiring a priori bound on input sizes. We use method preconditions to initialize fields only with valid values and method postconditions as test oracles to check method's correctness; this builds on our previous work on the Korat framework [2] for specification-based testing. We also support partial correctness properties given as assertions in the program and temporal specifications.

The main contributions of our work are:

- Providing a two-fold generalization of symbolic execution: one, to enable a standard model checker to perform symbolic execution; two, to give a symbolic execution algorithm that handles advanced programming constructs;
- Performing symbolic execution of code during explicit state model checking
  - to address the state space explosion problem: we check the behavior of code using symbolic values that represent data from very large domains instead of enumerating and checking for a small set of concrete values;
  - to achieve modularity: checking programs with uninitialized variables allows checking of a compilation unit in isolation;
  - to check strong correctness properties of concurrent programs that take inputs from unbounded domains with complex structure;
  - to exploit the model checker's built-in capabilities, such as different search strategies (e.g., heuristic search), checking of temporal properties, and partial order and symmetry reductions;
- Automating non-isomorphic test input generation to satisfy a testing criterion for programs with complex inputs and preconditions;
- A series of examples and a prototype implementation in Java, that uses Java PathFinder [19] for model checking, Omega library [16] as a decision procedure, and Korat for program instrumentation; our approach extends to other languages, model checkers, and decision procedures.

```
class Node {
  int elem;
  Node next;

  Node swapNode() {
1: if(next!=null)
2:  if(elem-next.elem>0){
3:    Node t = next;
4:    next = t.next;
5:    t.next = this;
6:    return t;
    }
7: return this;
  }
}
```

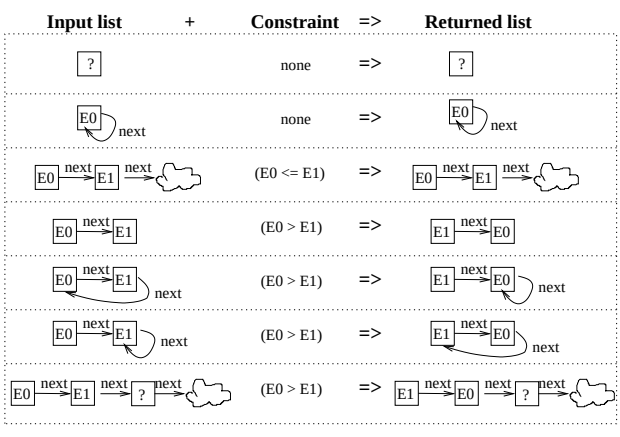


**Fig. 1.** Code to sort the first two nodes of a list (left) and an analysis of this code using our symbolic execution based approach (right)

Section 2 shows an example analysis in our framework. Section 3 describes traditional symbolic execution. Section 4 gives our algorithm for generalized symbolic execution. Section 5 describes our framework and Section 6 describes our implementation and instrumentation. Section 7 illustrates two applications of our implementation. We give related work in Section 8 and conclude in Section 9.

## 2 Example

This section presents an example to illustrate our approach. We check a method that destructively updates its input structure. The Java code in Figure 1 declares a class `Node` that implements singly-linked lists. The fields `elem` and `next` represent, respectively, the node's integer value and a reference to the next node. The method `swapNode` destructively updates its input list (referenced by the implicit parameter `this`) to sort its first two nodes and returns the resulting list.

We analyze `swapNode` using our prototype implementation (Section 6) and check that there are no unhandled runtime exceptions during any execution of `swapNode`. The analysis automatically verifies that this property holds.

The analysis checks seven symbolic executions of `swapNode` (Figure 1). These executions together represent all possible actual executions of `swapNode`. For each symbolic execution, the analysis produces an input structure, a constraint on the integer values in the input and the output structure. Thus for each row, any actual input list that has the given structure and has integer values that satisfy the given constraint, would result in the given output list. For an execution, the value "?" for an `elem` field indicates that the field is not accessed and the "cloud" indicates that the `next` field is not accessed.

Each input structure represents an isomorphism partition of the input space, e.g., the last row in the table shows an input that represents all (cyclic or acyclic)

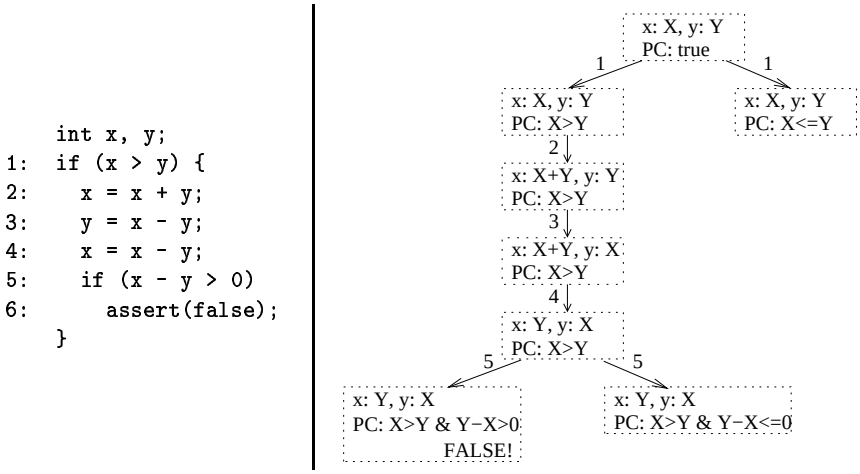


**Fig. 2.** Code that swaps two integers and the corresponding symbolic execution tree, where transitions are labeled with program control points

lists with at least three nodes such that the first element is greater than the second element; the list returned has the first two elements swapped.

If we comment out the check for `null` on line (1) in `swapNode`, the analysis reports that for the top most input in Figure 1, the method raises an unhandled `NullPointerException`. All other input/output pairs stay the same. The analysis, therefore, refutes the method's correctness by providing a counterexample.

The analysis supports method preconditions. For example, if we add to `swapNode` a precondition that the input list should be acyclic, the analysis does not consider the three executions (Figure 1), where the input has a cycle. The input structures and constraints can be used for test input generation.

## 3 Background: Symbolic execution

The main idea behind symbolic execution [14] is to use *symbolic values*, instead of actual data, as input values, and to represent the values of program variables as symbolic expressions. As a result, the output values computed by a program are expressed as a function of the input symbolic values.

The *state* of a symbolically executed program includes the (symbolic) values of program variables, a *path condition* (`PC`) and a program counter. The path condition is a (quantifier-free) boolean formula over the symbolic inputs; it accumulates constraints which the inputs must satisfy in order for an execution to follow the particular associated path. The program counter defines the next statement to be executed. A *symbolic execution tree* characterizes the execution paths followed during the symbolic execution of a program. The nodes represent program states and the arcs represent transitions between states.

Consider the code fragment in Figure 2, which swaps the values of integer variables `x` and `y`, when `x` is greater than `y`. Figure 2 also shows the corresponding symbolic execution tree. Initially, `PC` is *true* and `x` and `y` have symbolic values `X` and `Y`, respectively. At each branch point, `PC` is updated with assumptions about the inputs, in order to choose between alternative paths. For example, after the execution of the first statement, both `then` and `else` alternatives of the `if` statement are possible, and `PC` is updated accordingly. If the path condition becomes *false*, i.e., there is no set of inputs that satisfy it, this means that the symbolic state is not reachable, and symbolic execution does not continue for that path. For example, statement `(6)` is unreachable.

## 4 Algorithm

This section describes our algorithm for generalizing traditional symbolic execution to support advanced constructs of modern programming languages, such as Java and C++. We focus here on sequential programs. Section 5 presents the treatment of multithreaded programs.

### 4.1 Lazy initialization

The heart of our framework is a novel algorithm for symbolically executing a method that takes as inputs complex data structures with unbounded data. A key feature of the algorithm is that it starts execution of the method on inputs with *uninitialized* fields and uses *lazy initialization* to assign values to these fields, i.e., it initializes fields when they are first accessed during the method's symbolic execution. This allows symbolic execution of methods without requiring an a priori bound on the number of input objects.

We explain how the algorithm symbolically executes a method with one input object, i.e., the implicit input `this`. Methods with multiple parameters are treated similarly [2]. To execute a method `m` in class `C`, the algorithm first creates a new object `o` of class `C` with uninitialized fields. Next, the algorithm invokes `o.m()` and the execution proceeds following Java semantics for operations on reference fields and following traditional symbolic execution for operations on primitive fields, with the exception of the special treatment of accesses to uninitialized fields (Figure 3).

- When the execution accesses an uninitialized reference field, the algorithm nondeterministically initializes the field to the value `null`, to a reference to a new object with uninitialized fields, or to a reference of an object created during a prior field initialization; this systematically treats aliasing. When the execution accesses an uninitialized primitive (or string) field, the algorithm first initializes the field to a new symbolic value of the appropriate type and then the execution proceeds. Method preconditions are used to ensure that fields are initialized to values permitted by the precondition: when a reference field is initialized, the algorithm checks that the precondition does not fail for the structure and the path condition that currently constrain `o`;

```
if ( f is uninitialized ) {
   if ( f is reference field of type T ) {
      nondeterministically initialize f to
         1. null
         2. a new object of class T (with uninitialized field values)
         3. an object created during a prior initialization of a field of type T
      if ( method precondition is violated )
         backtrack();
   }
   if ( f is primitive (or string) field )
      initialize f to a new symbolic value of appropriate type
}
```

**Fig. 3.** Lazy initialization

– If the execution evaluates a branching condition on primitive fields, the algorithm nondeterministically adds the condition or its negation to the corresponding path condition and checks the path condition's satisfiability using a decision procedure. If the path condition becomes infeasible, the current execution terminates (i.e., the algorithm backtracks), otherwise the execution proceeds. This systematically updates path conditions on primitive fields.

**Input generation** To generate inputs that meet a given testing criterion, the algorithm symbolically executes the paths specified by the criterion. For every path, it generates an input structure and a path condition on the primitive input values, which together define a set of inputs that execute the path. To handle programs that perform destructive updates, the algorithm builds mappings between objects with uninitialized fields and objects that are created when those fields are initialized; these mappings are used to construct input structures.

**Isomorph breaking and structure generation** A nice consequence of lazy initialization of input fields is that for sequential programs, the algorithm only executes program paths on non-isomorphic[4] inputs. This can be used for systematic generation of inputs that have complex structural constraints by symbolically executing a predicate that checks the structural constraints, as in [2].

### 4.2 Illustration

We illustrate the algorithm using our running example from Figure 1. The symbolic execution tree in Figure 4 illustrates some of the paths that the algorithm explores while symbolically executing `swapNode`. Each node of the execution tree denotes a *state*, which consists of the state of the heap (including the symbolic values of the `elem` fields) and the path condition accumulated along the branch (path) in the tree. A transition of the execution tree connects two tree nodes

[4] This definition of isomorphism views structures as edge(node)-labeled graphs.

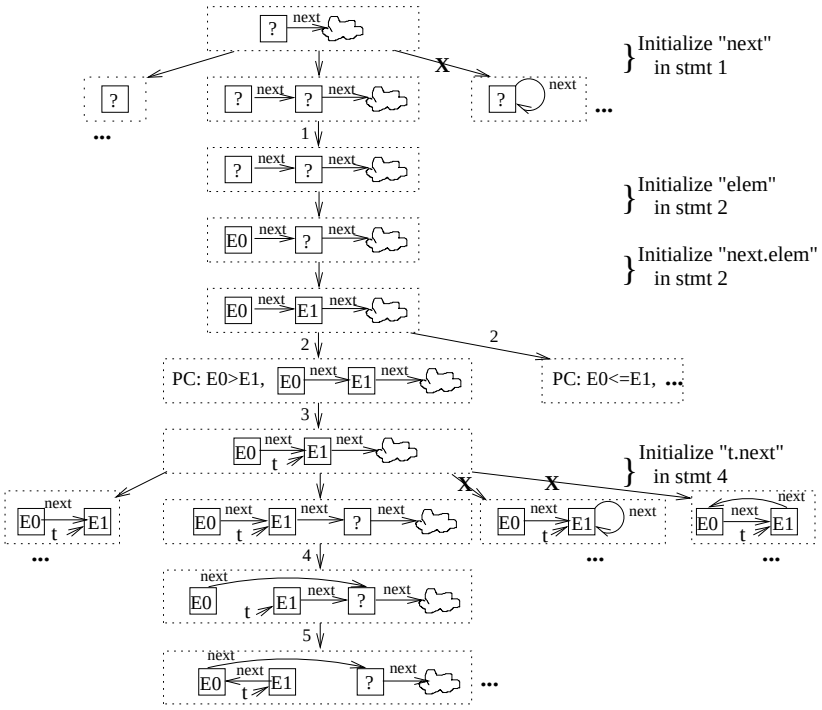


**Fig. 4.** Symbolic execution tree (excerpts), using notation described in Section 2

and corresponds to either execution of a statement of `swapNode` or to a lazy initialization step. Branching in the tree corresponds to a nondeterministic choice that is introduced to handle aliasing or build a path condition.

The algorithm creates a new node object and invokes `swapNode` on the object. Line (1) accesses the uninitialized `next` field and causes it to be initialized. The algorithm explores three possibilities: either the field is `null` or the field points to a new symbolic object or the field points to a previously created object of the same type (with the only option being itself). Intuitively, this means that, at this point in the execution, we make three different assumptions about the configuration of the input list, according to different aliasing possibilities. Another initialization happens during execution of statement (4), which results in four possibilities, as there are two `Node` objects at that point in the execution.

When a condition involving primitive fields is symbolically executed, e.g., statement (2), the execution tree has a branch corresponding to each possible outcome of the condition's evaluation. Evaluation of a condition involving reference fields does not cause branching unless uninitialized fields are accessed.

If `swapNode` has the precondition that its input should be acyclic, the algorithm does not explore the transitions marked with an "X". The input list corresponding to the output list pointed to by `t` in the bottom most tree node is shown on the bottom row of Figure 1.

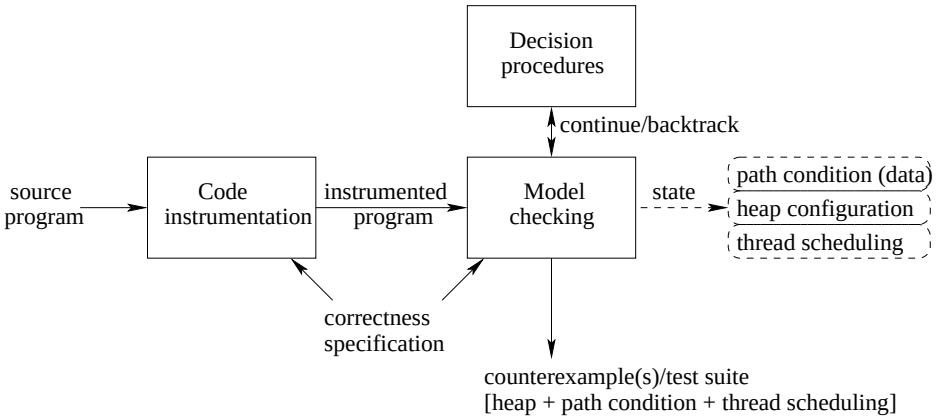


**Fig. 5.** General methodology

## 5 Framework

This section describes our symbolic execution based framework for checking correctness of software systems. Figure 5 illustrates our basic framework. To enable a model checker to perform symbolic execution (following the algorithm from Section 4), we instrument the original program by doing a source to source translation that adds nondeterminism and support for manipulating formulae that represent path conditions. The instrumentation allows any model checker that supports backtracking and nondeterministic choice to perform symbolic execution. Essentially, the model checker explores the symbolic execution tree of the program. Code instrumentation uses a correctness specification to add precondition checking (which is performed during field initialization) and postcondition checking (which is performed when an execution completes) to the original program. We describe some details of the instrumentation our prototype implementation performs in Section 6.

The model checker checks the instrumented program using its usual state space exploration techniques. A *state* includes a heap configuration, a path condition on primitive fields, and thread scheduling. Whenever a path condition is updated, it is checked for satisfiability using an appropriate decision procedure, such as the Omega library [16] for linear integer constraints. If the path condition is unsatisfiable, the model checker backtracks.

Correctness specifications can be given as preconditions and postconditions, assertions or more general *safety* properties. Safety properties can be written in the logical formalism recognized by the model checker or they can be specified with code instrumentation, as in [1]. The framework can be used for correctness checking and test input generation. While checking correctness, the model checker reports counterexample(s) that violate a correctness criterion. While generating test inputs, the model checker generates paths that are witnesses to a testing criterion encoded in the specification. Testing criteria can be encoded as

correctness specifications as in [8,13]. For every reported path, the model checker also reports the input heap configuration, the path condition for the primitive input fields thread scheduling, which can be used to reproduce the error.

**Multi-threaded and non-deterministic systems** Our framework allows a standard model checker to perform symbolic execution. We use the model checker also to systematically analyze thread interleavings and other forms of nondeterminism that might be present in the code. Our framework also exploits the model checker's built-in ability to combat state space explosion, e.g., by using partial order and symmetry reductions.

**Loops, recursion, method invocations** We exploit the model checker's search abilities to handle arbitrary program control flow. We do not require the model checker to perform state matching, since state matching is, in general, undecidable when states represent path conditions on unbounded data. Note also that performing (forward) symbolic execution on programs with loops can explore infinite execution trees. Therefore, for systematic state space exploration we use depth first search with iterative deepening (Section 7.1) or breadth first search (Section 7.2); our framework also supports heuristic based search [10]. Our framework can be used for finding counterexamples to safety properties; it can prove correctness for programs that have finite execution trees and have decidable data constraints.

## 6 Implementation

We have implemented our approach in Java to check Java programs. For code instrumentation, we build on the Korat tool [2]. We use Java PathFinder(JPF) [19] for model checking the instrumented programs. As a decision procedure, we use a Java implementation of the Omega library [16], that manipulates sets of linear constraints over integer variables. This section outlines the instrumentation, briefly describes JPF, and presents a critique of our approach.

### 6.1 Instrumentation

Conceptually, the instrumentation proceeds in two steps. First, the integer fields and operations are instrumented. The declared type of integer fields of input objects is changed to `Expression`, which is a library class we provide to support manipulation of symbolic integer expressions. A type analysis is used to determine which integer variables have their declared types changed to `Expression`. Operations involving these variables are replaced with method calls that implement "equivalent" operations that manipulate objects of type `Expression`. We have not yet automated the type analysis, but we could use for this the abstraction component of the Bandera toolset [6] that performs the same kind of analysis, but with a different purpose[5].

[5] Bandera performs a source to source translation to instrument a program, to reduce the cardinality of data sets associated with program variables.

```
class Node {
  Expression elem;
  Node next;
  boolean _next_is_initialized;
  boolean _elem_is_initialized;
  ...
  Node swapNode() {
1: if(_get_next() != null)
2:  if(Expression._pc._update_GT(
        _get_elem()._minus(
            _get_next()._get_elem()),
        new IntegerConstant(0)) {
3:   Node t = _get_next();
4:   _set_next(t._get_next());
5:   t._set_next(this);
6:   return t;
    }
7: return this; } }
```

```
class Expression { ...
 static PathCondition _pc;
 Expression _minus(Expression e){
   ...} }

class PathCondition { ...
 Constraints c;
 boolean _update_GT(Expression e1,
                    Expression e2){
  boolean result = choose_boolean();
  if (result)
      c.add_constraint_GT(e1,e2);
  else
      c.add_constraint_LE(e1,e2);
  if (!c.is_satisfiable())
      backtrack();
  return result;
  } }
```

**Fig. 6.** Instrumented code (left) and library classes (right)

Second, the field accesses are instrumented. Field reads are replaced by `get` methods that return a value based on whether the field is initialized or not (`get` methods implement the lazy initialization, as described in Section 4). Field updates are replaced by `set` methods which update the field's value. The `get` and `set` methods for a field also set a flag to indicate that the field is initialized.

As an illustration of the instrumentation, consider the code from Figure 1. Figure 6 gives part of the resulting code after instrumentation and the library classes that we provide. The `static` field `Expression._pc` stores the (numeric) path condition. Method `_update_GT` makes a nondeterministic choice (i.e., a call to `choose_boolean`) to add to the path condition the constraint or the negation of the constraint its invocation expresses and returns the corresponding `boolean`. Method `is_satisfiable` uses the Omega library to check if the path condition is infeasible (in which case, JPF will backtrack). Method `_minus` constructs a new `Expression` that represents the difference between its input parameters. `IntegerConstant` is a subclass of `Expression` and wraps concrete integer values. To keep track of uninitialized input fields we add a boolean field in the class declaration for each field in the original declaration, e.g., `_next_is_initialized` and `_elem_is_initialized`, which are set to true by get (set) methods.

To store the input objects that are created as a result of a lazy initialization, we use a variable of class `java.util.Vector`, for each class that is instrumented. The `get` methods use the elements in this vector to systematically initialize input reference fields. Our implementation also provides the library class `StringExpression` to symbolically manipulate strings.

### 6.2 Java PathFinder

Our current prototype uses the Java PathFinder model checker (JPF), an explicit-state model checker for Java programs that is built on top of a custom-made Java Virtual Machine (JVM). Since it is built on a JVM, it can handle all of the language features of Java, but in addition it also treats nondeterministic choice expressed in annotations of the program being analyzed. These features for adding nondeterminism are used to implement the updating of path conditions and the initialization of fields. JPF supports program annotations to cause the search to backtrack when a certain condition evaluates to true—this is used to stop the analysis of infeasible paths (when path conditions are found to be unsatisfiable). Lastly, JPF supports various heuristics [10], including ones based on increasing testing-related coverage (e.g., statement, branch and condition coverage), that can be used to guide the model checker's search.

### 6.3 Discussion

We use preconditions in initializing fields. In particular, a field is not initialized to a value that violates the precondition. Notice that we evaluate a precondition on a structure that still may have some uninitialized fields, therefore we require the precondition to be *conservative*, i.e., return `false` only if the initialized fields of the structure violate a constraint in the precondition. A conservative precondition or simply undecidability of path conditions may lead our analysis to explore infeasible program paths.

We have not provided here a treatment of arrays. Following [2], we could systematically initialize array length when an array field is first accessed, and then treat each array component as a field. We would like to extend our analysis to treat array length as a symbolic integer. Our algorithm handles subclassing: in step 3 in Figure 3 consider all objects created during a prior initialization of a field of type `T` or of a type `S`, where `S` is a subclass of `T`.

## 7 Applications

This section shows two applications of our framework: correctness checking of a distributed algorithm and test input generation for flight software.

### 7.1 Checking multithreaded programs with inputs

We illustrate an application of our symbolic execution framework on an example that (incorrectly) implements a distributed algorithm for sorting linked lists with integers in ascending order[6]. To sort an input list, the algorithm spawns a number of threads proportional to the number of nodes in the list. Each thread is assigned two adjacent list nodes and allowed a maximum number of swaps it can perform on elements in these nodes. This example illustrates our symbolic execution technique in the context of concurrency, structured data (linked lists), integer values as well as method preconditions and partial correctness criteria.

[6] We can correctly sort a list using this algorithm by controlling the thread scheduler.

```
class List {
  Node header;

  //@ precondition: acyclic();
  void distributedSort() {
    if (header == null) return;
    if (header.next == null) return;
    int i = 0;
    Node t = header;
    while (t.next != null) {
      new Swapper(t, ++i).start();
      t = t.next;
    }
  }
  ...
}

class Swapper extends java.lang.Thread {
  //can swap current.elem,current.next.elem
  Node current;
  int maxSwaps;

  Swapper(Node m, int n) {
    current = m; maxSwaps = n;
  }
  public void run() {
    int swapCount = 0;
    for (int i = 0; i < maxSwaps; i++)
      if (current.swapElem()) swapCount++;
    //@ assert: if (swapCount == maxSwaps)
    //@           current.inOrder();
  }
}
```

```
class List { ...
  boolean acyclic() {
    Set visited = new HashSet();
    Node current = header;
    while (current != null) {
      if (!visited.add(current))
        return false;
      current = current.next;
    }
    return true;
  }
}

class Node {
  int elem;
  Node next;

  synchronized boolean swapElem(){
    synchronized (next) {
      if (elem > next.elem) {
        // actual swap
        int t = elem;
        elem = next.elem;
        next.elem = t;
        return true;
      } }
    return false; // do nothing
  }
  synchronized boolean inOrder(){
    synchronized (next) {
      if (elem > next.elem) return false;
      return true;
    } } }
```

**Fig. 7.** A distributed sorting method for singly linked lists

The Java code in Figure 7 declares a singly linked list and defines a method for sorting lists. The method `distributedSort` takes an input list and spawns several threads to sort the list. For each adjacent pair of nodes in the list, `distributedSort` spawns a new thread that is responsible for swapping elements in these nodes. This method has a precondition that its input list should be acyclic, as specified by the `precondition` clause.

The `swapElem` method returns `true` or `false` based on whether the invocation actually swapped out of order elements or whether it was simply a no-op. Note that `swapElem` is different from `swapNode` in Figure 1, that performs destructive updating of the input list. We use synchronization to ensure that each list element is only accessed by one thread at a time. The `assert` clause declares a partial correctness property, which states that if a thread performs the allowed maximum number of actual swaps, then the element in node `current` is in order.

We used our implementation to symbolically execute `distributedSort` on acyclic lists and analyze the method's correctness. The analysis took 11 seconds (on a 2.2 GHz Pentium with 2GB of memory) and it produced a counterexample:

```
input list: [X] -> [Y] -> [Z] such that X > Y > Z
Thread-1: swaps X and Y
Thread-2: swaps X and Z
resulting list: [Y] -> [Z] -> [X]; Y and Z out of order
```

The input list consists of three symbolic integers `X`, `Y`, and `Z` such that `X > Y > Z`. `Thread-1` is allowed one swap and `Thread-2` is allowed two swaps. `Thread-1` performs its swap before `Thread-2` performs any swap. Now `Thread-2` performs a swap. The resulting list after these two swaps is `[Y] -> [Z] -> [X]` with `Y > Z`. Since `Thread-1` is not allowed any more swaps, it is not possible to bring `Y` and `Z` in order. Thus, the input list together with this thread scheduling give a counterexample to the specified correctness property. Note that to analyze `distributedSort` we did not a priori bound the size of the list (and therefore the number of threads to spawn).

### 7.2 Test input generation

We applied our framework to derive test inputs for code coverage, specifically condition coverage, of an Altitude Switch used in flight control software (1800 lines of Java code) [11]. The switch receives as input a sequence of time-stamped messages indicating the current altitude of an aircraft as well as an indication of whether this reading is considered accurate or not (represented by strings). The input sequence of messages was implemented as a linked list with undefined length. The program was instrumented to print out the input sequence as well as the integer and string constraints, whenever a new condition, i.e. one that was not covered before, was executed. This application presents a program that has as input a data structure and manipulates both integer and string constraints.

We used breadth-first search during model checking to generate test inputs that cover all the conditions within 22 minutes of running time (on a 2.2 GHz Pentium with 2GB of memory). In contrast, we also used traditional model checking with JPF, where we fixed the input sequence to have 3 messages and the range of altitude values to be picked nondeterministically from 0 to 20000 feet—the model checking did not finish, and as a consequence did not generate test inputs, for about a third of the conditions before memory was exhausted.

## 8 Related work

In previous work we developed Korat [2], a constraint solver for imperative predicates. Korat implements a novel search algorithm and performs a source to source translation to instrument a Java program and systematically handle aliasing and subclassing. Korat provides efficient generation of non-isomorphic inputs and was used to generate complex structures from preconditions for exhaustive black-box testing within a given input bound. The work we present here additionally provides input generation for white-box testing, supports symbolic manipulation of data values using a decision procedure, does not require bounds on input sizes, supports checking of multi-threaded programs and adapts Korat's instrumentation to enable any model checker to perform symbolic execution.

King [14] developed EFFIGY, a system for symbolic execution of programs with a fixed number of integer variables. EFFIGY supported various program analyses (e.g., test case generation) and is one of the earliest systems of its kind.

PREfix is a bug finding tool [3] based essentially on symbolic execution. PREfix has been used very successfully on large scale commercial applications. PREfix analyzes programs written in C/C++ and aims to detect defects in dynamic memory management. It does not check rich properties, such as invariants on data structures. PREfix may miss errors and it may report false alarms.

Several projects aim at developing static analyses for verifying program properties. The Extended Static Checker (ESC) [7] uses a theorem prover to verify partial correctness of classes annotated with JML specifications. ESC has been used to verify absence of such errors as null pointer dereferences, array bounds violations, and division by zero. However, tools like ESC cannot verify properties of complex linked data structures.

There are some recent research projects that attempt to address this issue. The Three-Valued-Logic Analyzer (TVLA) [17] is the first static analysis system to verify that the list structure is preserved in programs that perform list reversals via destructive updating of the input list. TVLA performs fixed point computations on shape graphs, which represent heap cells by shape nodes and sets of indistinguishable runtime locations by summary nodes. The lazy initialization of input fields in our framework is related to *materialization* of summary nodes in TVLA. We would like to explore this connection further.

The pointer assertion logic engine (PALE) [15] can verify a large class of data structures that can be represented by a spanning tree backbone, with possibly additional pointers that do not add extra information. These data structures include doubly linked lists, trees with parent pointers, and threaded trees. Shape analyses, such as TVLA and PALE, typically do not verify properties of programs that perform operations on primitive data values.

The Alloy constraint analyzer has been used for analyzing bounded segments of computation sequences manipulating linked structures by translating them into first order logic [18]. This approach requires a bound also on the input sizes and does not treat primitive data symbolically.

There has been a lot of recent interest in applying model checking to software. Java PathFinder [19] and VeriSoft [9] operate directly on a Java, respectively C program. Other projects, such as Bandera [6], translate Java programs into the input language of SPIN [12] and NuSMV [4]. They are whole program analysis (i.e., cannot analyze a procedure in isolation). Our source to source translation enables these tools to perform symbolic execution, and hence enables them to analyze systems with complex inputs and to analyze procedures in isolation.

The SLAM tool [1] focuses on checking sequential C code with static data, using well-engineered predicate abstraction and abstraction refinement tools. It does not handle dynamically allocated data structures. Symbolic execution is used to map abstract counterexamples on concrete executions and to refine the abstraction, by adding new predicates discovered during symbolic execution.

The Composite Symbolic Library [20] uses symbolic forward fixed point operations to compute the reachable states of a program. It uses widening to help termination but can analyze programs that manipulate lists with only a fixed number of integer fields and is a whole-program analysis.

## 9 Conclusion

We presented a novel framework based on symbolic execution, for automated checking of concurrent software systems that manipulate complex data structures. We provided a two-fold generalization of traditional symbolic execution based approaches. First, we defined a source to source translation to instrument a program, which enables standard model checkers to perform symbolic execution of the program. Second, we gave a novel symbolic execution algorithm that handles dynamically allocated structures, method preconditions, primitive data and concurrency. We illustrated two applications of our framework: checking correctness of multi-threaded programs that take inputs from unbounded domains with complex structure and generation of non-isomorphic test inputs that satisfy a testing criterion. Although we illustrated our framework in the context of Java programs, JPF, and the Omega library, our framework can be instantiated with other languages, model checkers and decision procedures.

In the future, we plan to investigate the application of widening and other abstraction techniques in the context of our framework. We also plan to integrate different (semi) decision procedures and constraint solvers that will allow us to handle floats and non-linear constraints. We believe performing symbolic execution during model checking is a powerful approach to analyze software. How well it scales to real applications remains to be seen.

## Acknowledgements

The authors would like to thank Doron Peled for useful discussions and Viktor Kuncak, Darko Marinov, Zhendong Su, and the anonymous referees for useful comments. The work of the first author was done partly while visiting NASA Ames Research Center under the NASA Ames/RIACS Summer Student Research Program and was also funded in part by NSF ITR grant #0086154.

## References

1. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proc. 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 36–5 of *ACM SIGPLAN Notices*, pages 203–213. ACM Press, June 2001.
2. C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, July 2002.
3. W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.
4. A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.
5. A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezze. Using symbolic execution for verifying safety-critical systems. In *Proc. 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 142–151. ACM Press, 2001.

6. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Păsăreanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In C. Ghezzi, M. Jazayeri, and A. Wolf, editors, *Proc. 22nd International Conference on Software Engineering (ICSE)*, pages 439–448. ACM, 2000.
7. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 1998.
8. A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proc. 7th European Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 146–162. Springer-Verlag, 1999.
9. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, pages 174–186, Paris, France, Jan. 1997.
10. A. Groce and W. Visser. Model checking java programs using structural heuristics. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, July 2002.
11. M. P. E. Heimdahl, Y. Choi, and M. Whalen. Deviation analysis through model checking. In *Proc. 17th IEEE International Conference on Automated Software Engineering (ASE)*, 2002.
12. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
13. H. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *Proc. 8th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Apr. 2002.
14. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
15. A. Moeller and M. I. Schwartzbach. The pointer assertion logic engine. In *Proc. SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, Snowbird, UT, June 2001.
16. W. Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *Communications of the ACM*, 35(8):102-114, 1992.
17. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, Jan. 1998.
18. M. Vaziri and D. Jackson. Checking properties of heap-manipulating procedures with a constraint solver. In *Proc. 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Poland, Apr. 2003.
19. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE)*, Grenoble, France, 2000.
20. T. Yavuz-Kahveci and T. Bultan. Automated verification of concurrent linked lists with counters. In G. P. M. Hermenegildo, editor, *Proc. 9th International Static Analysis Symposium (SAS)*, volume 2477 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.