

# survey of new trends in symbolic execution for software testing and analysis

Corina S. Păsăreanu · Willem Visser

**Abstract** Symbolic execution is a well-known program analysis technique which represents program inputs with symbolic values instead of concrete, initialized, data and executes the program by manipulating program expressions involving the symbolic values. Symbolic execution has been proposed over three decades ago but recently it has found renewed interest in the research community, due in part to the progress in decision procedures, availability of powerful computers and new algorithmic developments. We provide here a survey of some of the new research trends in symbolic execution, with particular emphasis on applications to test generation and program analysis. We first describe an approach that handles complex programming constructs such as input recursive data structures, arrays, as well as multithreading. Furthermore, we describe recent hybrid techniques that combine concrete and symbolic execution to overcome some of the inherent limitations of symbolic execution, such as handling native code or availability of decision procedures for the application domain. We follow with a discussion of techniques that can be used to limit the (possibly infinite) number of symbolic configurations that need to be analyzed for the symbolic execution of looping programs. Finally, we give a short survey of interesting new applications, such as predictive testing, invariant inference,

program repair, analysis of parallel numerical programs and differential symbolic execution.

## 1 Introduction

Modern software systems must be extremely reliable and correct. Automatic methods for ensuring software correctness range from static techniques, such as (software) model checking or static analysis, to dynamic techniques, such as testing. All these techniques have strengths and weaknesses: model checking (with abstraction) is automatic, exhaustive, but may suffer from scalability issues. Static analysis, on the other hand, scales to very large programs but may give too many spurious warnings, while testing alone may miss important errors, since it is inherently incomplete.

We survey here several recent research trends that combine the strengths of these different techniques while overcoming their weakness. In particular, we focus here on approaches to software testing and analysis that are based on (forward) symbolic execution. Symbolic execution [15,42] is a well known program analysis technique that allows execution of programs using *symbolic* input values, instead of actual data, and represents the values of program variables as symbolic expressions. As a result, the outputs computed by a program are expressed as a function of the symbolic inputs. Its applications range from automated test input generation to proving program partial correctness. Symbolic execution has been proposed over three decades ago but recently it has found renewed interest in the research community, due in part to the progress in decision procedures, availability of powerful computers and new algorithmic developments.

We begin with a description of our approach [41,47] to symbolic execution that uses a model checker to explore different symbolic execution paths (Sect. 2). This enables us

---

C. S. Păsăreanu (✉)  
NASA Ames Research Center, Carnegie Mellon University,  
Moffett Field, CA 94035, USA  
e-mail: Corina.S.Pasareanu@nasa.gov

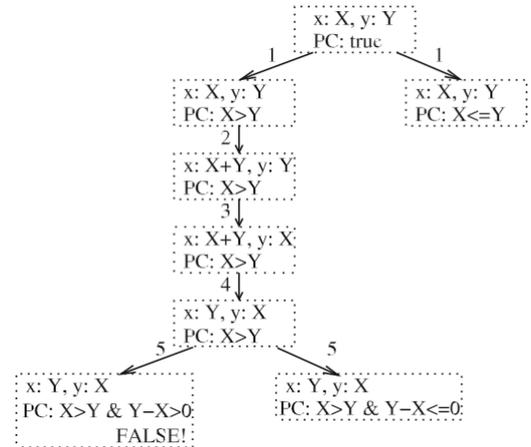
W. Visser  
Department of Computer Science,  
University of Stellenbosch, Stellenbosch, South Africa  
e-mail: willem@gmail.com

**Fig. 1** Code that swaps two integers and the corresponding symbolic execution tree (transitions are labeled with program control points)

```

int x, y;
1:  if (x > y) {
2:    x = x + y;
3:    y = x - y;
4:    x = x - y;
5:    if (x - y > 0)
6:      assert(false);
}

```



to take advantage of the model checker’s built-in capabilities, such as systematic analysis of thread interleavings, partial order reduction, different search strategies, etc. The approach applies to Java programs and it handles recursive input data structures, arrays, preconditions, as well as multithreading.

Furthermore, we discuss a popular recent technique, called “directed testing” [33] or “concolic execution” [53]. The technique combines symbolic with concrete execution [33, 53] to overcome some of the inherent limitations of symbolic execution, such as availability of decision procedures and handling calls to native libraries (Sect. 3). Other related hybrid approaches are discussed in the same section.

Performing symbolic execution on looping programs may result in a large (possibly unbounded) number of symbolic program configurations that need to be analyzed. Therefore symbolic execution might not terminate and in practice, one needs to put a limit on the number of such symbolic configurations. We also describe alternative techniques to better manage the symbolic space explored during symbolic execution (Sect. 4).

We follow with a description of various “classical” applications of symbolic execution, such as test input and sequence generation, proving program correctness, and static detection of runtime errors. We also describe some novel, “not so classical” applications, that use symbolic execution or its variants for predictive testing, dynamic invariant generation, data structure repair, analysis of parallel numerical programs, and differential symbolic execution (Sect. 5). Section 6 gives a short conclusion.

We give most of our presentation in terms of Java (because this was the context of our own work) but we believe that most of the presentation could also be generalized to other languages. The work related to the subject here is vast and it is simply impossible to cover it all in one article. However,

we hope that this survey (albeit very limited) will serve as a starting point for more new, exciting applications in this area.

## 2 Symbolic execution

### 2.1 Background

The main idea behind symbolic execution [15, 42] 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.

A *symbolic execution tree* characterizes the execution paths followed during the symbolic execution of a program. The tree nodes represent program states and they are connected by program transitions.

Consider the code fragment in Fig. 1 (left), which swaps the values of integer variables  $x$  and  $y$ , when  $x$  is greater than  $y$  [41]. Figure 1 (right) 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.

## 2.2 Exploring the symbolic execution tree using a model checking tool

Symbolic execution traditionally arose in the context of checking sequential programs with a fixed number of integer variables. Several recent approaches [12, 16, 25] implement dedicated tools to perform various program analyses based on some form of symbolic execution.

In our past work [41] we have defined a generalization of traditional symbolic execution that does not require a dedicated tool but instead enables a standard model checking tool (for the underlying language) to perform symbolic execution. Our approach targets Java programs and it handles complex input data structures and arrays (via “lazy initialization” as explained below) as well as concurrency. The Java PathFinder (JPF) model checking tool [38] is used to explore the symbolic execution tree of the analyzed program, as well as other forms of nondeterminism that might be present in the code. Thus, we take advantage of the model checker’s built-in state space exploration capabilities, such as different search strategies (e.g., heuristic search) as well as partial order and symmetry reductions. A similar tool [24] uses the Bogor model checking framework, instead of JPF, and a “lazier” treatment of initialization for input data structures.

In our approach, we defined a source-to-source translation that instruments a Java program by adding nondeterminism and support for manipulating formulae that represent path conditions in such a way that it enables JPF to perform symbolic execution of the program. The model checker checks the symbolic state space of the program using its usual state space exploration techniques. A *symbolic 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 off-the-shelf decision procedures, such as the Omega library [51] for linear integer constraints. If the path condition is unsatisfiable, the model checker backtracks. Preconditions are used to restrict the symbolic search space, to only enable exploration of inputs that satisfy the preconditions.

A specialized type-dependence analysis [2] can be used to minimize the instrumentation effort, by determining which parts of the code depend on the inputs and therefore needs to be instrumented, the rest of the code remaining unchanged. We describe some details of the instrumentation in Sect. 2.7 (in the context of handling input arrays).

Recently, we have implemented a new framework, Symbolic JPF [50], that does not require the code transformation, but instead it implements a non-standard interpreter of Java bytecodes on top of JPF, to enable symbolic execution of Java bytecodes.

## 2.3 Checking safety properties and generating test inputs

Our symbolic execution framework can be used for finding errors to safety properties and for test input generation. Safety properties can be written in the logical formalism recognized by the model checker or they can be specified with code instrumentation [9]. 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 as a safety property (see, e.g., [30, 36]). For a reported counterexample, the model checker also reports the input heap configuration, the path condition for the primitive input fields, and the thread scheduling, which can be used to reproduce the error.

## 2.4 Handling multithreaded and nondeterministic systems

As mentioned, our approach 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. Furthermore, we take advantage of the model checker built-in optimization techniques, such as partial order reduction for reducing the number of analyzed interleavings and different search heuristics, such as depth-first, breadth-first, heuristic, or random search.

## 2.5 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 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 put a limit on the search depth of the model checker or we limit the size of the constraints in the path condition. We discuss alternative techniques in Sect. 4.

The symbolic approach that we have just described 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. For proving properties of programs with unbounded loops, one would need to annotate the program with loop invariants (see discussion in Sect. 5.3).

## 2.6 Handling recursive input data structures

One of the challenges to symbolic execution is the handling of complex inputs, such as recursive data structures or arrays of

unspecified length. We use a *lazy initialization* algorithm for symbolically executing a method that takes as inputs complex data structures with unbounded data. The algorithm starts execution of the method on inputs with *uninitialized* fields and it assigns values to these fields “lazily”, i.e., 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.

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:

- When the execution accesses an uninitialized reference field, the algorithm nondeterministically initializes the field to `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 field, the algorithm first initializes the field to a new symbolic value of the appropriate type and then the execution proceeds according to the standard execution semantics.
- When 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).

### Example

We illustrate how lazy initialization works using the example from Fig. 2 (left). The example gives the Java declaration of 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 used symbolic execution to check that there are no unhandled runtime exceptions during any execution of `swapNode`. The result of the check is that the property holds; the analyzed executions are summarized in Fig. 2 (right). These executions together represent all possible actual executions of `swapNode`. For each execution, we show the

corresponding input structure, the 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. The value “?” for an `elem` field indicates that the field is not accessed and the “cloud” indicates that the `next` field is not accessed. Note that we do not depict the `null` values.

If we comment out the check for `null` on line (1) in `swapNode`, our framework reports that for the top most input in Fig. 2, the method raises an unhandled `NullPointerException`. All other input/output pairs stay the same.

The symbolic execution tree in Fig. 3 illustrates the (simplified) symbolic execution tree that results from the symbolic execution of `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 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.

Symbolic execution starts by first creating a new node object and invoking `swapNode` on the object. The first access to the uninitialized `next` field happens at line (1) and causes it to be initialized. Lazy initialization 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 field 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.

Assume now that `swapNode` has the precondition that its input should be acyclic; this can be written as a Java boolean method. Then symbolic execution does not explore the transitions marked with an “X”.

In order to keep track of the input data structures for programs with *destructive updating*, we build mappings between objects with uninitialized fields and objects that are created when those fields are initialized with our algorithm; these maps are used to re-construct the input structures, e.g., for test input generation.

```

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;
    }
}

```

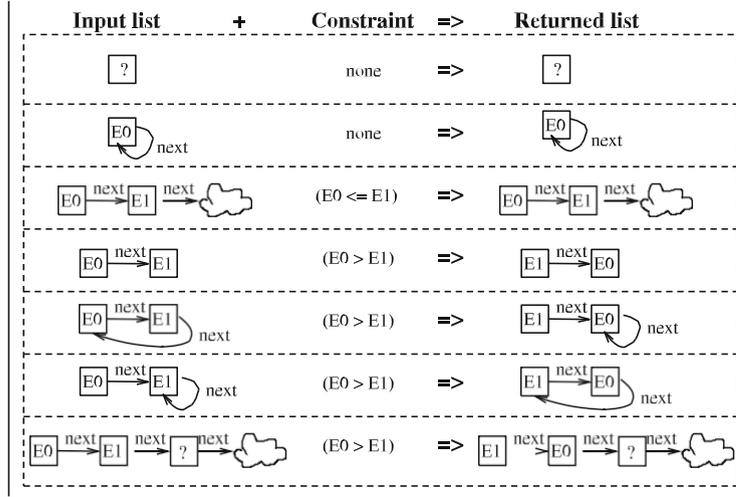
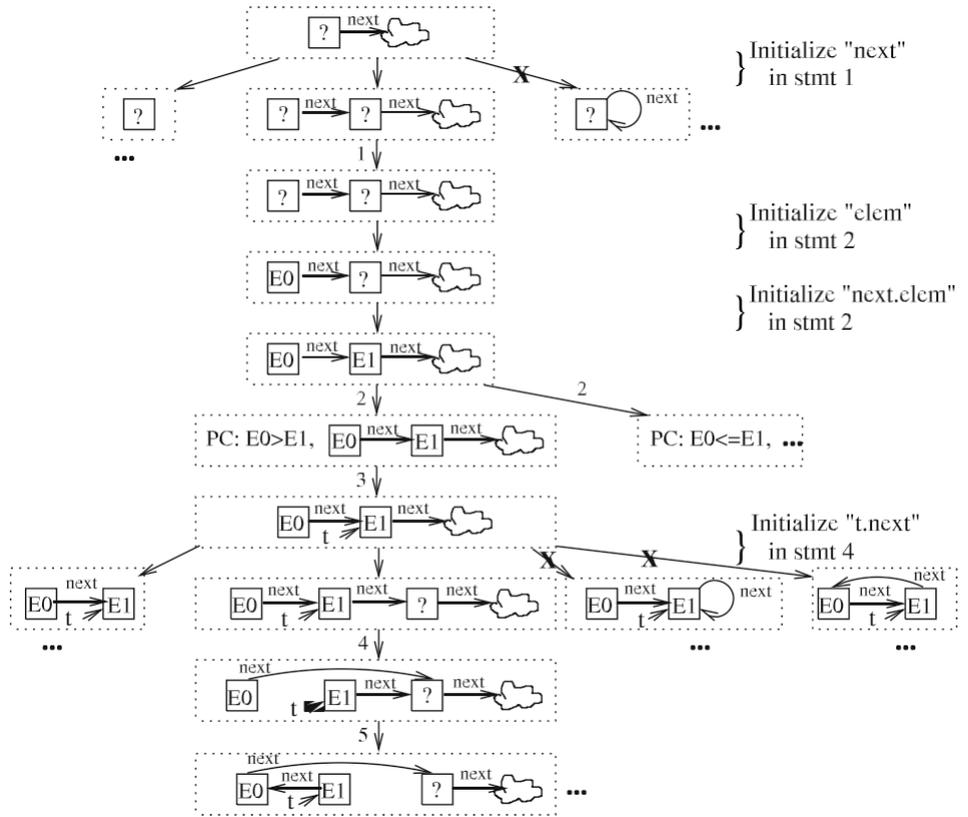


Fig. 2 Code to sort the first two nodes of a list (left) and an analysis of this code using our symbolic execution based approach (right)

Fig. 3 Symbolic execution tree (excerpts)



## 2.7 Handling input arrays

Symbolic execution for programs with input arrays of unspecified size one can also use lazy initialization [47].

Consider the code shown in Fig. 4 (left). This method takes as a parameter an array of integers *a* and it sets all the elements of *a* to zero. This method has a *precondition* that its input is not null. The *assert* clause declares a partial

correctness property that states that after the execution of the loop, the value of the first element in *a* is zero (we will describe in Sect. 5.3 how we can use symbolic execution and loop invariants to prove this property).

In order to symbolically execute the code we first instrument it to enable JPF to perform symbolic execution. The instrumented code and part of the library classes that we provide are illustrated in Fig. 4 (right) and Fig. 5, respectively.

**Fig. 4** Array example (*left*) and corresponding instrumented code (*right*)

```

// @ precondition: a != null;
void example(int[] a) {
1:  int i = 0;
2:  while (i < a.length) {
3:    a[i] = 0;
4:    i++;
  }
5:  assert a[0] == 0;
}

void example() {
  IntArrayStructure a = new IntArrayStructure();
  Expression i = new IntegerConstant(0);

  while(Expression.pc._update_LT(i,a.length)) {
    a._set(i,new IntegerConstant(0));
    i = i._plus(new IntegerConstant(1));
  }
  assert Expression.pc._update_EQ(
    a._get(new IntegerConstant(0)),0);
}

```

**Fig. 5** Library classes

```

class Expression { ...
  static PathCondition pc;
  Expression _plus(Expression e){
    ...} }

class PathCondition { ...
  Constraints c;
  boolean _update_LT(Expression l,
    Expression r){
    boolean result;
    result=Verify.choose_boolean();
    if (result)
      c.add_constraint_LT(e1,e2);
    else
      c.add_constraint_GE(e1,e2);
    Verify.ignoreIf(!c.is_sat());
    return result;
  } }

class IntArrayStructure {
  Vector _v;
  Expression length;
  ...
  ArrayCell _new_ArrayCell(Expression idx) {
    for(int i=0;i<_v.size();i++) {
      ArrayCell cell=(ArrayCell)_v.elementAt(i);
      if(Expression.pc._update_EQ(cell.idx,idx))
        return cell;
    }
    ArrayCell t=new ArrayCell(length,idx,name);
    _v.add(t);
    return t;
  }
  public Expression _get(Expression idx) {
    assert(Expression.pc._update_GE(idx, 0)&&
      Expression.pc._update_LT(idx,length));
    ArrayCell cell = _new_ArrayCell(idx);
    return cell.elem;
  } }

```

The interested reader is referred to [41,47] for a detailed description of code instrumentation, here we just highlight some key features.

The main idea is to replace concrete types with corresponding “symbolic types” (i.e., library classes that we provide) and concrete operations with method calls that implement “equivalent” operations on symbolic types. Classes `Expression` and `IntArrayStructure` support manipulation of symbolic integers and symbolic integer arrays, respectively. The static field `Expression.pc` stores the (numeric) path condition. Method `_update_LT` 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_sat` uses a decision procedure to check if the path condition is infeasible (in which case, JPF will backtrack). Method `_plus` constructs a new `Expression` that represents the sum of its input parameters. `IntegerConstant` is a subclass of `Expression` and wraps concrete integer values.

To store the input array elements that are created as a result of a lazy initialization, we use a variable of class `Vector`, for each input array. The `_get` and `_set` methods use the elements in this vector to systematically initialize input array elements. When the execution accesses a symbolic array cell,

the algorithm nondeterministically initializes it to a new cell or to a cell that was created during a prior cell initialization. The assertion checks in the `_get/_set` methods establish that there are no array out of bounds errors.

## 2.8 Integrating multiple decision procedures

Perhaps the main challenge to symbolic execution is the availability of the decision procedures for the application domain and the number of constraints that can be handled by the decision procedure/constraint solvers. This challenge can be addressed by first performing various simplifications of the path conditions (see, e.g., [53,56]), before sending them to the decision procedures.

Furthermore, a variety of powerful, fast decision procedures and constraint solvers are being developed and can be used. Of particular interest are the SMT (Satisfiability Modulo Theory) decision procedures for combinations of theories, such as the theory of real numbers, the theory of integers, and the theories of various data structures, such as lists, arrays, bit vectors and so on. The annual SMT competitions [57] and the associated SMT-lib public benchmarks (inspired by similar SAT competitions [52]) are a strong driver for new algorithmic developments and improvements in solver implementations for various theory combinations.

In order to take advantage of various available decision procedures, we equipped our symbolic execution framework with a generic interface to multiple decision procedures [4] (e.g., SMT solvers CVC3 [22], Yices [69], and STP [58]). More recently, we have also integrated two constraint solvers (Choco [14] and IASolver [37]) for handling constraints involving complex math functions, such as trigonometric functions (since none of the SMT solvers mentioned above can handle such constraints).

The user can choose between multiple decision procedures that interact in different modes with the symbolic execution framework (file, pipe, or native call interactions). If the decision procedure supports *incremental solving* (e.g., CVC3, Yices) then the path condition is not sent all at once to the decision procedure, but rather just the new constraint that needs to be added before checking satisfiability. The incremental solving of path conditions can be done only during a (bounded) depth-first search traversal of the symbolic execution tree.

We note that in our approach to symbolic execution we do not need a decision procedure for the theory of data structures or arrays, since we solve the constraints involving such structures explicitly, using lazy initialization. One advantage of this approach is that we can handle input data structures and complex math constraints at the same time. However, there are related symbolic execution tools (such as PEX [49]) that take a different approach: they treat the input structures completely symbolically and therefore require a decision procedure that is powerful enough to solve the resulting constraints.

## 2.9 Handling native code; strings

Other typical challenges to symbolic execution include handling common library classes and/or native code, i.e., code that can not be analyzed directly by symbolic execution. Such code needs to be modeled explicitly to be considered by the symbolic execution [50]. Section 3 describes an orthogonal technique that combines concrete and symbolic execution to address this problem.

A promising approach that targets Java *String* library classes is presented in [54]. In that work, the implementation details of strings are abstracted away using finite state automata, resulting in scaling of symbolic execution to complex string manipulating applications.

## 3 Combining concrete and symbolic execution

Several recent tools implement a hybrid analysis that performs a concrete execution along with symbolic execution for dynamic test generation, e.g., DART [33], CUTE [44, 53], EXE [13], PEX [49]. This popular approach has been applied

to finding errors in many challenging areas such as Web and database applications [7, 26, 64].

The idea [33] is to perform a concrete execution on random inputs and at the same time to collect the path constraints along the executed path; this is also called “concolic execution”. These path constraints are then used to compute new inputs that *drive* the program along alternative paths. More specifically, one can negate one constraint at a branch point to guide the test generation process towards executing the other branch. An off-the-shelf constraint solver is called to solve the path constraints and to obtain the test inputs. The program is executed on these new inputs, constraints are collected along the new program path and the process is repeated until all the execution paths are covered (therefore it may never terminate) or until the desired test coverage is achieved. The approach works by code instrumentation and does not use model checking; therefore it can not analyze multithreading easily. However, the main advantage of this hybrid approach is that the concrete execution can be used “to help” the symbolic execution in certain situations, e.g., when there are no available decision procedures or in the presence of native calls.

CUTE further extends this approach to handling input recursive data structures. The tool separates pointer constraints from numeric (integer) constraints. The pointer constraints are simplified to replace complex symbolic pointer expressions with simple symbolic pointer variables, resulting in some approximation.

## Example

As an example for dynamic test generation, consider the code in Fig. 6 [31]. Assume we have decision procedures/constraint solvers that can only reason about linear constraints. Initially the inputs that were randomly generated are  $x = 3$  and  $y = 7$ . The concrete value of  $z$  is 27, but the symbolic value is  $z = X * X * X$ , and the path condition (corresponding to the `else` branch) is  $Y \neq X * X * X$ ; therefore the decision procedures cannot handle it. However, instead of taking the symbolic value  $z = X * X * X$  in the path condition, one can take the concrete value (i.e.,  $z = 27$ ). The path condition then becomes  $Y \neq 27$  and the execution continues until the end of the procedure. In order to obtain inputs that guide the execution towards the `then` branch, one

```

1: void foo(int x,int y){
2:   int z = x*x*x; /* could be z = h(x) */
3:   if (y == z) {
4:     assert(false); /* error */
5:   }
6: }
```

Fig. 6 Code for illustrating concolic execution

needs to solve  $y = 27$  which can be done easily with the available constraint solver. The program is then re-executed with the new inputs:  $x = 3$  and  $y = 27$  and the error at line 4 is discovered.

Assume now that instead of `int z = x*x*x;`, statement 2 is `int z = h(x);`, where `h` is some library function. Alternatively assume its code is simply unavailable to symbolic execution, e.g., could not be instrumented. Then the same reasoning as above can be applied, therefore eliminating the need for explicit modeling of `h`. Of course, there may be some situations when such an approach would not be recommended, due to certain side-effects of method `h`, e.g., writing data to a file that is later read and affects the execution. In that case, some modeling would still be required.

### 3.1 Other combined analyses

In concolic execution the idea is to perform a concrete execution together with a symbolic analysis that is used to produce inputs to cover “new” behavior with the aim to uncover errors. One can also take the opposite approach by first doing a symbolic, imprecise analysis to find a possible error and then perform a concrete execution (i.e., run the program) to determine if it is real or not. The reason for this second step is that the symbolic execution can be imprecise (it might follow paths in the code that are not possible in reality); this may happen if the analysis is only intra-procedural (do not follow procedure calls) and just returns new unconstrained symbolic values for the returned values of the procedures that are not analyzed.

The Check&Crash system [20] uses ESC/Java [28] to do the symbolic analysis and then JCrasher to execute the test to see if it is a real test. In [61] a custom symbolic execution is used that allows inter-procedural analysis in which the degree of procedure nesting can be varied (see Sect. 5.4 for more details).

Other related hybrid techniques include the use of concrete execution to effectively “set-up” the environment for symbolic execution [50] and a combination of test case generation based on symbolic execution and runtime monitoring [6]; both these techniques have been applied in the context of NASA software systems. Furthermore, related approaches [34, 70] seek to combine abstraction techniques, with automatic abstraction refinement, and theorem proving for program analysis and testing.

## 4 Scaling symbolic execution

As mentioned, performing symbolic execution on programs that have loops or recursion may result in an infinite execution tree. Even in the absence of such infinite behavior, performing symbolic (or concolic) execution on large programs becomes quickly expensive, due to the large number

and also the size of paths that need to be explored. In this section we discuss several techniques that aim to alleviate these scalability problems.

### 4.1 Abstraction

Abstraction [18] is a well-known technique that reduces the large data domains of a program to smaller domains, that are more amenable for verification. Typically, abstraction in verification has been used to compute *over-approximations* of program behaviors. Such over-approximations are useful for *proving program properties*, e.g., if a safety property is found to be true in the abstracted program, then the property is also true in the original, unabstracted program.

We discuss here a complementary approach [3, 62], which uses *under-approximation* based abstraction for the purpose of *property falsification*. A related approach [67] combines symbolic execution with a particular under-approximation based abstraction that only keeps information about the *length* of the analyzed lists/buffers in the context of testing for buffer over-flows.

In particular here, we consider state matching techniques to limit the state space explored during symbolic execution. The work has been done in the context of using a model checker to explore the symbolic execution tree, as described in Sect. 2. The approach involves checking when a symbolic state ( $s_i$ ) is subsumed by another symbolic state ( $s_j$ ), i.e., the set of concrete states represented by  $s_i$  is included in the set of concrete states represented by  $s_j$ .

Subsumption is used to determine when a symbolic state is revisited, in which case the model checker backtracks, thus pruning the state space search. Even with subsumption, the number of symbolic states may still be unbounded. We therefore define abstraction mappings to be used during state matching. More precisely, for each explored state, the model checker computes and stores an abstract version of the state, as specified by the abstraction mappings. Subsumption checking then determines if an abstract state is being revisited. This effectively explores an *under-approximation* of the (feasible) paths through the program. Therefore the technique is still useful for finding safety errors or for test input generation (see also Sect. 5.2 for a discussion of applications of abstract subsumption in the context of test sequence generation).

### Example

In previous work [3] we defined abstract subsumption checking for singly linked lists and arrays, by reducing their representation to lists. The abstraction that we have implemented are inspired by the work in shape analysis [45, 68] and are based on the idea of summarizing all the nodes in a *maximally uninterrupted* list segment with a *summary* node. The main

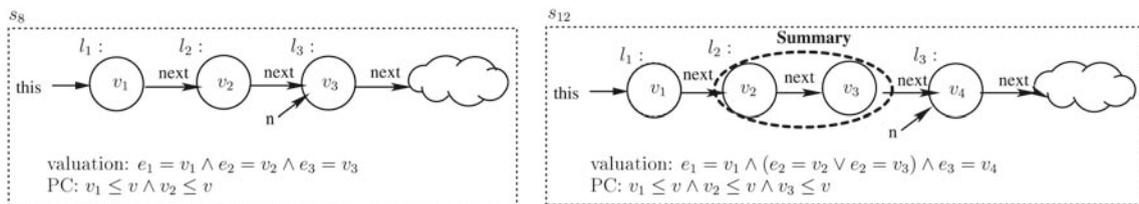


Fig. 7 Abstract subsumption between  $s_8$  and  $s_{12}$

difference between [45, 68] and our abstractions is that we also summarize the numeric data stored in the summarized nodes and we give special treatment to uninitialized nodes. The numeric data stored in the abstracted list is summarized by setting the valuation for the summary node to be a *disjunction* of the valuations of the summarized nodes. Intuitively, the numeric data stored in a summary node can be equal to that of any of the summarized nodes.

We illustrate abstract subsumption for singly-linked lists using the example in Fig. 7. For more details, please see the related paper [3].

Figure 7 depicts two symbolic states,  $s_8$  and  $s_{12}$  that resulted during the analysis of a list manipulating program [3]. These states can not be matched, since their “heap shape” is different. However, let us consider the abstract heap shape and the corresponding valuations for state  $s_{12}$ . The abstracted state is subsumed by state  $s_8$  since the corresponding heap shapes match (as illustrated by the common node labels  $l_1, l_2, l_3$ ). Furthermore, there is a valid logical implication between the normalized numeric constraints of the two states.

## 4.2 Compositional symbolic execution

Recent work [1, 32] proposes compositional reasoning as a means of scaling up symbolic execution. The work has been done in the context of “dynamic testing”, the hybrid concrete-symbolic execution described in Sect. 3, but we believe that it can also be extended to “classical” symbolic execution (as introduced in Sect. 2).

The idea [32] is to use logic “summaries” of individual functions (similar to inter-procedural static analysis). A summary consists of preconditions on the function’s inputs and post-conditions on the function’s output; they are computed “top down”, to take into account the proper calling context of the function under analysis. If  $f()$  calls  $g()$ , one can summarize  $g()$  and use  $g()$ ’s summaries when analyzing (or testing)  $f()$ ; thus, each method is analyzed separately and the over-all number of analyzed paths is smaller than in the case the two procedures are analyzed as a whole.

The work in [1] extends the compositional analysis with a demand-driven approach, which allows as few intra-procedural paths as possible to be symbolically executed in order to form an inter-procedural composed path leading to a specific

target branch or statement of interest (like an assertion). The approach uses first-order logic formulas with uninterpreted functions in order to represent function summaries and allow compositional symbolic execution using a SMT solver.

## 4.3 Path merging

Another scaling technique is path merging [5, 8, 43] – it comes from the hardware domain and it is closely related to abstraction. Path merging involves the definition of “merge points”—program points where the merging of symbolic paths should occur. Merge points are typically placed at the beginning of what is, semantically, a new algorithm or significant procedure in the program under analysis, or before loops and other computationally expensive code, to ensure that code is symbolically executed only once. The merging itself involves performing a logical disjunction on the symbolic states that reach the merging point. It has been shown that path merging may result in significant speed-up of symbolic execution, an order of magnitude for example for low level software [5].

## 5 Applications/analyses

Symbolic execution has many applications, most notably in testing and proving program correctness. We discuss them below, together with some exciting new applications.

### 5.1 Test case generation

The goal of testing is typically to achieve a high degree of code coverage, such as statement, branch, condition, MC/DC coverage. One traditional application of symbolic execution is the automated generation of test-cases that achieve a high degree of coverage. Symbolic execution lends itself particularly well to this task, since the path condition to reach a branch or statement in the code when solved, gives exactly the inputs to reach the statement or branch (i.e., the test inputs for the test case). We refer to this approach as test-case generation for white-box testing.

Alternatively, one can perform test generation in a black-box fashion by essentially using the same general technique,

but instead of symbolically executing the program under test, one executes a specification of the inputs, such as a Java predicate characterizing all valid input structures for the code under analysis. An example of a Java predicate is “the class invariant”, or, `repOk()` boolean method [11, 63] for data structures in object-oriented code. The objective here is to generate structures that satisfy the class invariant to form valid input for the program under test. This general approach, was initially proposed by the Korat tool [11] and it did not use symbolic execution. See [63] for a detailed description of using symbolic execution to generate test inputs in this fashion.

## 5.2 Test sequence generation

Both the white- and black-box techniques described above suffer from the issue that the generated inputs may not be actually possible during normal execution of the program. With the white-box technique this can happen since the analysis of one method in isolation does not take into account the implicit preconditions imposed by the method’s calling context. Similarly with the black-box technique it may be the case that although a certain input satisfies the class invariant, it can not be constructed using the public methods and fields allowed by the respective Java class.

To alleviate these concerns one can generate sequences of tests, rather than single tests [62, 66]. As a simple example, consider a class `BinTree` that provides a Java implementation of binary search trees.

```
public class BinTree {
    private Node root;
    ...
    public void add (int x) { ... }
    public boolean remove (int x) { ... }
}
```

A *test sequence* for this class is as follows:

```
BinTree t = new BinTree();
t.add(1); t.add(2); t.remove(1);
```

It contains a sequence of method calls in the class interface (e.g., `add` and `remove`), together with method arguments, that builds relevant object states and exercise the code in some desired fashion, e.g., to achieve statement or predicate coverage [62].

Test sequences are generated by enumerating all the possible method Sequences, up to some user specified sequence size. This can be done with the help of a model checker, for example [62], or with a dedicated tool [66].

Analyzing all combinations of method calls quickly becomes expensive (in terms of time and memory). One solution is to provide a mechanism for state-matching between method calls in this symbolic case. In particular, after each method call, the object state is examined to see if it can

be “matched” with a previously stored state, in which case that sequence is discarded; otherwise the search for new sequences continues with the next method call. Since symbolic states represent sets of concrete states, state “matching” involves checking subsumption between sets of states.

Although this problem is undecidable in general, if one only considers container classes storing integer data, the problem may become tractable. One can also match states using an abstraction of the state (as explained in Sect. 4), i.e., match abstract versions of symbolic states where the unabstracted states will not match. The trade-offs are obvious, match too liberally (i.e., using abstraction) and the coverage will not be obtained, and match too finely (i.e., check full subsumption on symbolic states), and run the risk of never terminating the search.

Using the shape of the container as the abstraction function was found to be particularly powerful [62]: for example, we could show that the shortest sequence of API calls on a Fibonacci Heap implementation to obtain statement coverage was 12. This is an interesting result in itself, since the code is only a few hundred lines long and the simplest form of coverage requires 12 calls.

For a detailed study of the various techniques for generating test sequences for container classes see [62] (all examples are made available though the JPF SourceForge website). We analyzed Java implementations for Binary Tree, Fibonacci Heap, Binomial Heap, Tree Map). We compared explicit state model checking, symbolic and concrete execution (with and without abstract matching) and random testing. We found that symbolic execution worked better than explicit model checking and that, not surprisingly, shape abstraction provides an accurate representation of containers. We found that random testing worked pretty well but it requires longer sequences to achieve good coverage.

## 5.3 Proving program properties

If there is an upper bound on the number of times each loop in the program may be executed, symbolic execution can be used for proving correctness, since the corresponding symbolic execution tree is finite.

However, for most programs, no fixed bound on the number of times each loop is executed exists and the corresponding execution trees are infinite. In order to prove the correctness of such programs, one needs to traverse the symbolic execution tree inductively rather than explicitly [35], using annotations in the form of loop invariants. Such annotations are provided by the user or may be discovered automatically, see e.g., [17, 19, 29, 46, 47, 59, 65]. Recent tools that implement such reasoning include ESC/Java [28] (it does not use traditional symbolic execution, but rather similar symbolic reasoning) and Bogor/Kiasan [24] for reasoning about properties of Java programs. Furthermore, Small-

<pre> init; while (C) {   B; } assert P; </pre>	<pre> init; assert I; /* base case */ make symbolic variables read in B; assume I; if (C) {   B;   assert I; /* induction step */ } else   assert P; </pre>
---	---

**Fig. 8** Single loop program (*left*) and instrumented program for proof (*right*)

foot [10] uses symbolic execution and separation logic for proving Hoare-style triples on heap-manipulating programs.

For simplicity of presentation, we illustrate the technique on a single-loop program such as the one in Fig. 8 (left); multiple loops can be treated similarly, see e.g., [65]. The program consists of some (loop-free) initialization code, a loop with condition  $C$  and (loop-free) body  $B$ , and post condition  $P$ .

To verify that  $P$  holds, it suffices to find a loop invariant  $I$ , i.e., a formula that is true when entering the loop, re-entering the loop during its iteration and exiting the loop [35]. Moreover,  $I$  must be strong enough to produce verifiable results (hence a loop invariant *true* is, in general, not sufficient). In a symbolic execution framework, this amounts to checking the three assertions in the modified program in Fig. 8 (right). Here, we replaced the `while` statement with an `if` statement; this is equivalent to placing a “cut” in the loop [35]. At this cut point, we consider all the variables that are modified in the loop body initialized to *new* symbolic values, and the path condition initialized to *true*. Note that a symbolic execution from this point on is representative of an arbitrary number of loop unrollings; the “input variables” at the cut point are the variables that are modified by the loop body and their new symbolic values represent all cases. Since the program loop has been cut, this symbolic execution will terminate and have a finite symbolic execution tree.

We then use symbolic execution to check three assertions:

- the assertion at line (4) is the *base case* of the inductive argument and checks that  $I$  holds when entering the loop
- the assertion at line (7) is the *induction step* and checks that, *assuming*  $I$  holds at the beginning of the loop,  $I$  also holds after the execution of the loop body (i.e.,  $I$  is inductive)
- the assertion at line (9) checks that  $I$  is strong enough for the property to hold (i.e.,  $I \wedge \neg C \rightarrow P$ )

If there are no assertion violations in the loop-free program of Fig. 8 (right), then the program of Fig. 8 (left) does not violate the property  $P$ .

## Example

As an example, consider again the code presented in Fig. 4. Using the loop invariant  $i \geq 0$ , symbolic execution can be used to automatically check that there are no array bounds violations. This is a simple invariant that can be stated without much effort. In order to prove that there are no assertion violations, a more complex loop invariant is needed, namely  $\neg(a[0] \mid= 0 \wedge i > 0)$ . In [47] we present a technique that generates such invariants automatically, by iterative approximation. The technique handles different types of constraints (e.g., boolean or numeric, constraints on dynamically allocated data and arrays) and it allows for checking universally quantified formulas. Such formulas are necessary for expressing properties of programs that manipulate unbounded data (such as the input array in Fig. 4)

## 5.4 Static detection of runtime errors

Using symbolic execution to find potential runtime errors is a well-known technique. The most famous example of this is the success of Intrinsic’s PREFIX tool [12] that ultimately led to a buy-out by Microsoft. More recent examples include the work of Engler et al. in [13] for detecting runtime errors in C code and Tomb et al. in [60] that detects errors in Java code.

The idea behind all these tools is to symbolically execute a program until a state is reached where a runtime violation is “possible”, for example a null-pointer dereference, division by zero, etc., and a potential error is reported. Unfortunately, due to mostly scalability issues, one can often not execute programs from their inputs, thus it is common to only analyze public or API methods and often times only intra-procedurally. This means the analysis can report errors that are not possible, so-called spurious errors.

One approach to reduce the false positives is to use the “variably inter-procedural” analysis described in [60]. As the name suggests the idea here is to allow one to vary the level of the inter-procedural analysis to follow calls  $n$  levels deep. Furthermore the approach proposes to solve the input constraints that are associated with a possible error and to form a test case; the analysis reports the error only if the test case actually produces the expected error (similar to Check-n-Crash [20]).

## Examples

As an illustration of some of the advantages of variably inter-procedural analysis, consider the program in Fig. 9 and the problem of detecting null pointer dereferences. Lets first assume we use an intra-procedural analysis where we don’t follow the calls to the `Integer.toHexString` method (as is done in [20]); a possible null pointer dereference will be flagged at line 8, with no constraints on the value of  $x$ .

```

1: class Example {
2:   public String hexAbs(int x) {
3:     String result = null;
4:     if(x > 0)
5:       result = Integer.toHexString(x);
6:     else if (x < 0)
7:       result = Integer.toHexString(-x);
8:     return result.toUpperCase();
9:   }
10:}

```

**Fig. 9** A simple Java program that illustrates some benefits of symbolic execution

```

1: int target = ...;
2: int delta = ...;
3: foo(int i) {
4:   if (similar(i,target)) {
5:     y = 10/i; // possible error
6:   }
7: }
8: ...
9: boolean similar (int i, int target) {
10:  if (((target - delta) <= i) &&
11:      (target + delta) >= i)
12:    return true;
13:  return false;
14:}

```

**Fig. 10** An example where intra-procedural analysis is sufficient

Using variably inter-procedural symbolic execution, we can do better. If we set the analysis to evaluate all method calls up to a depth of 1, it can follow the calls to `Integer.toHexString`, and determine that they never return null values. Then, because it is a path-sensitive analysis, it can determine that a null pointer dereference can only happen (and must happen) if  $x = 0$ . Thus, the analysis has ruled out the false positives (the assignments on lines 5 and 7), and has given more information about the true error (the missing case for  $x = 0$ ). Given the constraint on  $x$ , it is then straightforward to construct a test case that will trigger the bug.

Varying the level of inter-procedural analysis can have some interesting consequences, for example in [60] it was found that going from an intra-procedural to an inter-procedural analysis might not find more errors but will reduce the number of possible errors (also referred to as *warnings* below) the symbolic analysis discovers (and thus will lead to test cases to run to see if it is a real error). The code in Fig. 10 illustrates the intuition for this behavior. Note that depending on the value of *target* and *delta* there could be a division by zero in this code. Let’s assume we pick *target* = 100 and *delta* = 10, in which case there is no division by zero. The result of an intra-procedural analysis is one warning, but no

```

1: foo(int m) {
2:   m = answer(m);
3:   m = m/(1-m);
4: }
5: ...
6: int answer(int v) {
7:   return v == 42 ? 1: 0;
8: }

```

**Fig. 11** An example where inter-procedural analysis is required

error (since the warning corresponds to the case when  $i = 0$  and that would make the division unreachable). The reason for this behavior is that during the intra-procedural analysis the call to *similar* is ignored and a fresh symbolic variable is created to hold the result of the call.

However, an inter-procedural analysis results in no warnings (and therefore no errors) since the constraints on *similar* combined with the fact that  $i$  is 0 makes the division unreachable.

The interesting case here is if we pick the values to expose the problem (e.g. change *target* to 1). Now both an intra- and an inter-procedural analysis expose the error. Note that an intra-procedural analysis also finds the error, since it still only returns the constraint that  $i$  should be 0, but now *similar* returns *true* so the division is reachable.

One can also create an example to show the opposite effect where obtaining additional constraints actually exposes errors that would otherwise not have been found—this happens when analyzing the code in Fig. 11. Here an intra-procedural analysis has no additional constraints on the input value  $m$  and thus the chances of the test generation to randomly pick 42 is almost zero. However during an inter-procedural analysis the constraint that  $m$  should be 42 is recorded and that would make picking  $m$  trivial to expose the division by zero error.

In general a statement that is potentially buggy can be reached in many more ways that do not expose the error than in ways that will expose the error—if this is not true then the error will be found and fixed quickly anyways. Therefore the additional constraints one obtains by doing an inter-procedural analysis will mostly reduce the number of infeasible paths (of an intra-procedural analysis) that reach a potentially buggy statement but it will not necessarily increase the likelihood of generating a test to reach the error.

An enhancement to the general approach of symbolic execution for finding runtime errors is suggested in [27] where it is pointed out that the analysis can be optimized by taking the unconstrained inputs to a program and then constraining them by the negation of the path conditions corresponding to paths that lead to errors. The intuition here is to reduce the importance of errors due to unconstrained inputs and rather to report deeper, and possibly more hard to find errors. For example, consider the following code:

```
public void foo(Object o) {
    o.x = 5;
    ...
}
```

Assume  $o$  is unconstrained; a possible null-pointer exception will be flagged on the dereference in the first line. However since  $o$  is unconstrained this error is not reported and one adds the constraint that from now on  $o$  is non-null. This technique eliminates false positives and in addition constrains executions which allows better scaling. Note that this technique is best used as a heuristic to rank errors when shown to the user, since errors due to unconstrained inputs can also be real errors and should be reported (if only at a lower importance).

### 5.5 Other applications

Symbolic execution has many applications and it is impossible to enumerate them all. We can only list here a few new “non-standard” applications of symbolic execution (and related hybrid approaches):

- *Predictive testing* [39] attempts to predict errors from correct traces. The idea is to perform a “concolic execution” along concrete traces generated by running an existing test suite and to check for assertion violations and other types of errors along these executions: the assertions that hold along a concrete execution do not necessarily hold along the corresponding symbolic execution (since the latter characterizes multiple concrete executions).
- *Invariant inference* [21] generates “likely” program invariants in the form of method pre- and post-conditions and class invariants that hold for a given set of tests; the technique is similar in spirit to Daikon [23] but uses the constraints collected during a symbolic execution to come up with the invariants, instead of the invariant patterns used by Daikon.
- *Program and Data Structure Repair* can be done using symbolic execution; e.g., given an assertion that represents desired structural integrity constraints and a structure that violates them, the algorithm from [40] can “mutate” the given structure to satisfy the constraints.
- *Parallel numerical program analysis* [55] involves combining model checking and symbolic execution to establish the equivalence of a sequential and a parallel program. The sequential program acts as the “specification” for the parallel one. The symbolic execution is particularly tailored to handling floating point arithmetic.
- *Differential symbolic execution* [48] computes the “logical” differences between two versions of a program; such differences can be used to automate software

evolution tasks such as regression test maintenance, reducing re-certification activities or checking behavioral equivalence of two programs after software re-factoring.

## 6 Conclusions and future directions

In this paper, we surveyed new techniques based on symbolic execution and we discussed some of their “traditional” applications, such as test generation and program analysis, as well as some new, interesting applications. The work related to the subject here is vast and it is simply impossible to cover it all in one article. However, we hope that this survey (albeit very limited) will serve as a starting point for more new, exciting applications in this area.

Scalability is still the main obstacle against the widespread application of symbolic execution techniques. We believe that parallelizing the analyses discussed in this article, as well as extending the abstraction and compositional presented here, should lead to future fruitful research. The investigation of new heuristic searches that guide the symbolic execution towards “interesting” program states will also be promising. Furthermore, despite the emergence of powerful decision procedures there is still a lack of (semi-)decision procedures for combinations of theories that are useful for symbolic execution applications, such as handling both strings and numeric constraints – useful for Web applications. We’ve only sketched here a few future directions. We are sure that there are many others waiting to be explored.

## References

1. Anand, S., Godefroid, P., Tillmann, N.: Demand-driven compositional symbolic execution. In: Proceedings of TACAS (2008)
2. Anand, S., Orso, A., Harrold, M.J.: Type-dependence analysis and program transformation for symbolic execution. In: Proceedings of TACAS (2007)
3. Anand, S., Păsăreanu, C.S., Visser, W.: Symbolic execution with abstract subsumption checking. In: Proceedings of SPIN (2006)
4. Anand, S., Păsăreanu, C.S., Visser, W.: JPF-SE: A symbolic execution extension to Java PathFinder. In: Proceedings of TACAS (2007)
5. Arons, T., Elster E., Ozer S., Shalev J., Singerman, E.: Efficient symbolic simulation of low level software. In: Proceedings of DATE (2008)
6. Artho, C., Barringer, H., Goldberg, A., Havelund, K., Khurshid, S., Lowry, M.R., Păsăreanu, C.S., Rosu, G., Sen, K., Visser, W., Washington, R.: Combining test case generation and runtime verification. *Theor. Comput. Sci.* **336**(2–3), 209–234 (2005)
7. Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., Ernst, M.D.: Finding bugs in dynamic web applications. In: Proceedings of ISSTA (2008)
8. Babic, D.: Exploiting Structure for Scalable Software Verification. Ph.D. thesis, University of British Columbia, Vancouver, Canada, Aug (2008)

9. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.: Automatic predicate abstraction of C programs. In: Proceedings of PLDI (2001)
10. Berdine, J., Calcagno, C., O'Hearn, P.: Symbolic execution with separation logic. In: Proceedings of Third Asian Symposium (2005)
11. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on Java predicates. In: Proceedings of ISSTA (2002)
12. Bush, W.R., Pincus, J.D., Sielaff, D.J.: A static analyzer for finding dynamic programming errors. *Softw. Pract. Experience* **30**(7), 775–802 (2000)
13. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: Proceedings of ACM Conference on Computer and Communications Security (2006)
14. The Choco Constraint Solver: <http://choco.sourceforge.net/>
15. Clarke, L.A.: A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.* **2**(3), 215–222 (1976)
16. Coen-Porisini, A., Denaro, G., Ghezzi, C., Pezze, M.: Using symbolic execution for verifying safety-critical systems. In: Proceedings of ESEC/FSE (2001)
17. Colon, M., Sankaranarayanan, S., Sipma, S.: Linear invariant generation using non-linear constraint solving. In: Proceedings of CAV (2003)
18. Cousot, P.: The role of abstract interpretation in formal methods. In: Proceedings of SEFM (2007)
19. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of POPL (1978)
20. Csallner, C., Smaragdakis, Y.: Check 'n' crash: Combining static checking and testing. In: Proceedings of ICSE (2005)
21. Csallner, C., Tillmann, N., Smaragdakis, Y.: DySy: Dynamic symbolic execution for invariant inference. In: Proceedings of ICSE (2008)
22. CVC3: <http://www.cs.nyu.edu/acsys/cvc3/>
23. The Daikon invariant detector: <http://groups.csail.mit.edu/pag/daikon/>
24. Deng, X., Lee, J., Robby: Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In: Proceedings of ASE (2006)
25. Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended static checking. Research Report 159, Compaq Systems Research Center (1998)
26. Emmi, M., Majumdar, R., Sen, K.: Dynamic test input generation for database applications. In: Proceedings of ISSTA (2007)
27. Engler, D., Dunbar, D.: Under-constrained execution: making automatic code destruction easy and scalable. In: Proceedings of ISSTA (2007)
28. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proceedings of PLDI (2002)
29. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: Proceedings of POPL (2002)
30. Gargantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specifications. In: Proceedings of ESEC/FSE (1999)
31. Godefroid, P.: Software model checking via static and dynamic program analysis. In: MOVEP (2006)
32. Godefroid, P.: Compositional dynamic test generation. In: Proceedings of POPL (2007)
33. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proceedings of PLDI (2005)
34. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: a new algorithm for property checking. In: Proceedings of SIGSOFT FSE (2006)
35. Hantler S.L., King, J.C.: An introduction to proving the correctness of programs. *ACM Comput. Surv.* **8**(3), 331–353 (1976)
36. Hong, H., Lee, I., Sokolsky, O., Ural, H.: A temporal logic based theory of test coverage and generation. In: Proceedings of TACAS, April (2002)
37. IASolver (The Brandeis Interval Arithmetic Constraint Solver): <http://www.cs.brandeis.edu/~tim/Applets/IASolver.html/>
38. Java PathFinder: <http://javapathfinder.sourceforge.net>
39. Joshi, P., Sen, K., Shlimovich, M.: Predictive testing: Amplifying the effectiveness of software testing (short paper). In: Proceedings of ESEC/FSE (2007)
40. Khurshid, S., Garcia, I., Suen, Y.: Repairing structurally complex data. In: Proceedings of SPIN (2005)
41. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Proceedings of TACAS (2003)
42. King, J.C.: Symbolic execution and program testing. *Commun. ACM* **19**(7), 385–394 (1976)
43. Koelbl, A., Pixley, C.: Constructing efficient formal models from high-level descriptions using symbolic simulation. *Int. J. Parallel Programm.* **33**(6), 645–666 (2005)
44. Majumdar, R., Sen, K.: Hybrid concolic testing. In: Proceedings of ICSE (2007)
45. Manevich, R., Yahav, E., Ramalingam, G., Sagiv, M.: Predicate abstraction and canonical abstraction for singly-linked lists. In: Proceedings of VMCAI, LNCS, vol. 3385, Paris (2005)
46. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification (1992)
47. Păsăreanu, C.S., Visser, W.: Verification of java programs using symbolic execution and invariant generation. In: Proceedings of SPIN (2004)
48. Person, S., Dwyer, M.B., Elbaum, S., Păsăreanu, C.S.: Differential symbolic execution. In: Proceedings of FSE (2008)
49. PEX: Automated Exploratory Testing for .NET: <http://research.microsoft.com/Pex/>
50. Păsăreanu, C.S., Mehrlitz, P., Bushnell, D., Gundy-Burlet, K., Lowry, M., Person, S., Pape, M.: Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In: Proceedings of ISSTA (2008)
51. Pugh, W.: The Omega test: A fast and practical integer programming algorithm for dependence analysis. In: Conference on High Performance Networking and Computing archive. Proceedings of the 1991 ACM/IEEE Conference on Supercomputing table of contents Albuquerque, New Mexico, pp. 4–13 (1991)
52. SAT Competitions: <http://www.satcompetition.org/>
53. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Proceedings of ESEC/FSE (2005)
54. Shannon, D., Hajra, S., Lee, A., Zhan, D., Khurshid, S.: Abstracting symbolic execution with string analysis. In: Proceedings of TAIC-PART (2007)
55. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Using model checking with symbolic execution to verify parallel numerical programs. In: Proceedings of ISSTA (2006)
56. Sinha, N.: Symbolic program analysis using term rewriting and generalization. In: Proceedings of FMCAD, Nov. (2008)
57. SMT Competitions: <http://www.smtcomp.org/>
58. STP (Simple Theorem Prover): <http://sourceforge.net/projects/stp-fast-prover>
59. Tiwari, A., Rues, H., Saidi, H., Shankar, N.: A technique for invariant generation. In: Proceedings of TACAS (2001)
60. Tomb, A., Brat, G., Visser, W.: Variably interprocedural program analysis for runtime error detection. In: Proceedings of ISSTA (2007)
61. Tomb, A., Brat, G.P., Visser, W.: Variably interprocedural program analysis for runtime error detection. In: Proceedings of ISSTA (2007)

62. Visser, W., Păsăreanu, C.S., Pelanek, R.: Test input generation for java containers using state matching. In: Proceedings of ISSTA (2006)
63. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test input generation in Java Pathfinder. In: Proceedings of ISSTA (2004)
64. Wassermann, G., Yu, D., Chander, A., Dhurjati, D., Inamura, H., Su, Z.: Dynamic test input generation for web applications. In: Proceedings of ISSTA (2008)
65. Wegbreit, B.: The synthesis of loop predicates. *Commun. ACM* **17**(2), 102–112 (1974)
66. Xie, T., Marinov, D., Schulte, W., Notkin, D.: Symstra: A framework for generating object-oriented unit tests using symbolic execution. In: Proceedings of TACAS (2005)
67. Xu, R.-G., Godefroid, P., Majumdar, R.: Testing for buffer overflows with length abstraction. In: Proceedings of ISSTA (2008)
68. Yavuz-Kahveci, T., Bultan, T.: Automated verification of concurrent linked lists with counters. In: Hermenegildo, G.P.M. (ed.) Proceedings of SAS (2002)
69. Yices: An SMT Solver <http://yices.csl.sri.com/>
70. Yorsh, G., Ball, T., Sagiv, M.: Testing, abstraction, theorem proving: better together!. In: Proceedings of ISSTA (2006)