Worst-Case Execution Time Testing via Evolutionary Symbolic Execution

Andrea Aquino[1], Giovanni Denaro[2], Pasquale Salza[1]
[1]USI Università della Svizzera italiana, Switzerland
[2]University of Milano-Bicocca, Italy
Email: andrea.aquino@usi.ch, denaro@disco.unimib.it, pasquale.salza@usi.ch

***Abstract*—Worst-case execution time testing amounts to constructing a test case triggering the worst-case execution time of a program, and has many important applications to identify, debug and fix performance bottlenecks and security holes of programs. We propose a novel technique for worst-case execution time testing combining symbolic execution and evolutionary algorithms, which we call "Evolutionary Symbolic Execution", that (i) considers the set of the feasible program paths as the search space, (ii) embraces the execution cost of the program paths as the fitness function to pursue the worst path, (iii) exploits symbolic execution with random path selection to collect an initial set of feasible program paths, (iv) incrementally evolves by steering symbolic execution to traverse new program paths that comply with execution conditions combined and refined from the currently collected program paths, and (v) periodically applies local optimizations to the worst currently identified program path to speed up the identification of the worst path. We report on a set of initial experiments indicating that our technique succeeds in generating good worst-case execution time test cases for programs with which existing approaches cannot cope.**

***Index Terms*—Symbolic Execution, Worst-Case Execution Time, Genetic Algorithms, Software Engineering**

I. Introduction

Worst-Case Execution Time (WCET) testing amounts to constructing a test case that triggers the worst execution time of a program, and has many important applications to identify, debug and fix performance bottlenecks and security holes of software programs. For example, WCET testing can reveal if a program matches the theoretical worst-case computational complexity of the algorithm that it implements, revealing a performance bug if it does not. More in general, knowing a WCET test case allows the developers to debug and profile the WCET behavior of the program, to spot and possibly fix performance issues or identify optimization opportunities. Also, WCET testing can highlight vulnerabilities to inputs that an adversary can exploit for denial of service attacks.

In this paper, we propose a WCET testing technique based on an unprecedented combination of symbolic execution and evolutionary algorithms, which we call Evolutionary Symbolic Execution (ESE), that crucially improves on all previous WCET testing techniques that rely only on symbolic execution.

Symbolic execution is a well-established program analysis technique to compute the execution conditions of sets of program paths [1], [2]. Symbolic execution computes the execution conditions in the form of propositional formulas, called "path conditions", over symbols that represent the possible values of the program inputs. Executing the program with actual values that satisfy a path condition results in executing through the corresponding path.

The information in the path conditions allows symbolic execution to discriminate between feasible (executable) and infeasible (not executable) program paths, based on whether it is possible to derive either actual assignments of the inputs that satisfy a path condition or a proof that the path condition is indeed unsatisfiable, respectively. Thus, symbolic executors integrate with constraint solvers that can automatically decide the satisfiability of the path condition formulas [3], [4], [5]. At state of the art there exist many symbolic execution tools that target a variety of programming languages and application domains [6], [7], [8], [9], [10], [11], [12], [13].

Traditional techniques for WCET analysis of hard real-time systems include steps based on symbolic execution to prune infeasible execution paths, but do not generally address the generation of WCET test cases [14], [15], [16], [17], [18], [19], [20]. These techniques exploit symbolic execution on restricted parts of the overall system, to tune higher lever static analysis procedures. More recently, WISE [21] and SPF-WCA [22] introduced WCET testing techniques that (i) exploit symbolic execution to discriminate between the feasible and infeasible execution paths of the program under test, (ii) identify the worst (feasible) path based on a cost function, e.g., the amount of executed instructions, measured during symbolic execution, and (iii) synthesize a WCET test case by solving the path condition of the worst identified program path.

However, since symbolic execution incurs unrealistic computational costs to exhaustively analyze large path spaces, which is a common case for even simple programs, WISE and SPF-WCA work in two phases: (i) they accomplish the exhaustive symbolic execution of the program under test, but only in the scope of a user-defined restriction of the program inputs, being the considered restriction suitable for limiting the number of explored paths to a practical amount; (ii) then, they extrapolate the information of the worst feasible path observed with the restricted inputs, to synthesize a path selection heuristic, called a "guidance policy", which they use to steer the symbolic execution of the program in the target (unrestricted) scope. If successful, the guidance policy allows them to identify the worst-case path by visiting a small subset of the feasible paths.

In Section II, we exemplify some programs for which WISE and SPF-WCA work well. In Section II, we provide examples of programs for which the worst-case path observed for the restricted inputs does not generalize, or even exhibits a radically different behavior with respect to the worst-case case of the program in the target scope. If so, the approaches of WISE and SPF-WCA results in either an ineffective guidance policy that selects an unmanageable amount of paths or even a misleading policy that may lead to reporting a wrong WCET test case.

The novel technique for WCET testing presented in this paper is able to analyze the program under test directly in its target scope, thus avoiding both the burden for the testers of having to identify any suitable input restriction, and the issues due to a possible mismatch between the behavior of the program with restricted and unrestricted inputs, respectively. Our technique embraces a meta-heuristic path selection strategy based on a "memetic" algorithm. Memetic algorithms are evolutionary algorithms that hybridize genetic and local search algorithms, such that the candidate solutions identified with a genetic algorithm can be improved with focused optimizations within the local search algorithm.

In particular, our algorithm exploits the synergy between symbolic execution, which contributes the ability to identify the execution conditions that characterize feasible program paths, and a memetic path selection heuristic, which contributes to the ability of heuristically (not exhaustively) exploring huge path spaces. The memetic heuristic is driven by a fitness function that measures the execution cost of the program paths and, in turn, include (i) a genetic algorithm that fosters the combination of the execution conditions from the already explored program paths, aiming to reveal sets of execution conditions that correspond to incrementally worse program paths, and (ii) a local search procedure, executed at regular intervals, that attempts atomic changes to the current path condition, aiming to further refine the decision sequences of the worst program path identified so far.

The paper is organized as follows. Section II motivates our research by exemplifying the existing techniques, WISE and SPF-WCA, and their limitations on a set of sample programs. Section III presents our novel technique for WCET testing in detail. Section IV discusses a set of experiments that evaluate our technique both in absolute terms and in comparison with WISE and SPF-WCA. Section V acknowledges the related work in the field. Finally, Section VI summarizes our conclusions and outlines directions of further research.

II. Motivation

This section discusses a set of examples that highlight the limitations of the state-of-the-art WCET testing techniques, thus motivating the new technique proposed in this paper.

The two phases symbolic execution approach that we surveyed in the introduction was first proposed in the technique WISE [21]. We illustrate WISE with reference to the program `is_palindrome` in Listing 1 that inspects an input list to answer whether it is palindromic. For the input lists of a fixed size, a WCET test case amounts to executing the program with a list that is indeed palindromic, thus causing the maximum number of iterations of the loop in the program. To find

Listing 1
`is_palindrome`

```
"""
Decides if the given list is a palindrome.

Worst case: a palindrome list.
"""
def is_palindrome(l):
    for i in range(len(l)):
        if l[i] != l[len(l) - 1 - i)]:
            return False
    return True
```

a WCET test case for this program, WISE starts with the exhaustive symbolic execution of the program in a restricted input scope that suffices to limit the feasible program paths to a manageable amount. The way to express the restriction tightly depends on the characteristics of the inputs of the program under analysis and must be thus indicated by a test analyst in the general case. If the input is a list of primitive values, as in the case of the program in Listing 1, WISE can work by bounding the size of the input list to few items.

For the program in Listing 1, considering only lists with 5 items limits the number of possible execution paths to 3 feasible (and 3 infeasible) paths. The feasible paths correspond to the executions in which the program exits the loop during the 1st or the 2nd iteration, or completes all the 5 iterations, respectively. Symbolic execution captures these paths with the path conditions (i) $\alpha_1! = \alpha_5$, (ii) $\alpha_1 = \alpha_5 \wedge \alpha_2! = \alpha_4$, and (iii) $\alpha_1 = \alpha_5 \wedge \alpha_2 = \alpha_4$, respectively, being $\alpha_{1..5}$ symbolic values that represent the 5 items in the input list. The last of these path conditions characterizes the palindromic lists, e.g., $[1, 2, 0, 2, 1]$. The infeasible paths correspond to the paths in which the program would exit the loop during the 3rd, the 4th or the 5th iteration, which is impossible in all 3 cases because the 3rd list item cannot be different from itself, and the conditions to exit at the 4th, or the 5th iteration contrast with the assumptions made at earlier iterations. Symbolic execution identifies that these 3 paths are infeasible because they map to unsatisfiable path conditions, for instance, the path condition of the path that exits the loop at the 4th iteration is $\alpha_1 = \alpha_5 \wedge \alpha_2 = \alpha_4 \wedge \alpha_3 = \alpha_3 \wedge \alpha_4! = \alpha_2$, in which the 2nd and the 4th conditions are mutually contradictory.

For the program of Listing 1, WISE would essentially work as follows. It symbolically executes the program with respect to a small input list, e.g., with the input list bounded to 5 items as above mentioned. Then, for each feasible path yielded by symbolic execution, it measures the number of instructions traversed in the path and selects the path that executes the highest amount of instructions. For the lists with 5 items, it obtains that the worst path is the one that completes 5 iterations of the loop in the program. Next, it observes that symbolic execution consistently selected the *false* branch of the if-statement inside the loop, and thus it infers that forcing symbolic execution to select this branch consistently guides the analysis throughout the worst path of the program. Indeed, using this *guidance policy*, WISE can efficiently analyze the program of Listing 1 for arbitrarily large input lists, because the

Listing 2
`alternate_0`

```python
"""
Iterates an expensive task based on the values of the
list.

Worst case: a list that alternates zeros and non-zeros
 values.
"""
def alternate_0(l):
    for i in range(len(l)):
        if l[i] == 0:
            check = 1 - (i % 2)
        else:
            check = i % 2

        if check != 0:
            execute_expensive_task()
```

guidance policy steers symbolic execution to explore only one path, and exactly the worst path of the program. For instance, considering input lists with 100 items, the above guidance policy leads WISE to explore the single path whose path condition is $\alpha_1 = \alpha_{100} \wedge \alpha_2 = \alpha_{99} \wedge \alpha_3 = \alpha_{98} \wedge \cdots \wedge \alpha_{50} = \alpha_{51}$, and then generates the WCET test case by solving this condition to actual values.

The SPF-WCA approach [22] extends WISE, synthesizing sophisticated guidance policies in which the decisions to make at the relevant decision points must not be consistently the same (as in the above example), but may vary depending on the history of decisions made during symbolic execution. Listing 2 exemplifies the program `alternate_0` for which SPF-WCA improves on WISE. For this program, the WCET test cases amount to executing the program with lists that contain zero and non-zero numbers at even and odd locations, respectively, thus making function `execute_expensive_task()` execute at all iterations of the loop in the program. To execute the worst case path of the program in Listing 2, symbolic execution must proceed in strict alternation through the *true* and *false* branches of the first if-statement inside the loop. In this case, WISE would synthesize an ineffective guidance policy, since it just observes that both decisions shall be allowed at that if-statement, and this results in providing no actual guidance. Instead, SPF-WCA is able to compute an effective policy that guides symbolic execution to select the *true* branch if it selected the *false* branch at the previous traversal of the if-statement, and the vice-versa. SPF-WCA can be instantiated to use decision histories of any size, but for this example, it suffices to inspect only one previous decision.

In this paper, we question the general validity of the fundamental assumption that underlies both WISE and SPF-WCA, i.e., we question that the analysis of the program with restricted inputs can always unveil a regular worst case behavior. We argue that in the general case the worst program path can either be irregular across increasingly large input bounds or even correspond to different program behaviors for different boundings. Below we provide examples of these observations.

Listing 3 shows the `depth_first_search` program for which the branch sequences of the worst path cannot be captured with WISE and SPF-WCA. The program implements

Listing 3
`depth_first_search`

```python
"""
Determines if the vertex "finish" is reachable from
the vertex "start" in the given graph.

Worst case: a fully connected directed graph, except
for the vertex "finish" that is reachable only from
the "start" vertex of which it is the last neighbur.
"""
def depth_first_search(graph, start, finish):
    visited = [start]

    while len(visited) > 0:
        current = visited.pop()

        if current == finish:
            return True

        visited.append(current)

        for neighbor in graph.neighbors(current):
            if not neighbor in visited:
                visited.append(neighbor)

    return False
```

a depth-first visit to find if there is a path that connects two vertexes “start” and “finish” in a graph. The worst case happens with a directed graph in which all vertexes but *finish* are fully interconnected, and the vertex *finish* is connected only to *start* and listed as its last neighbor. A graph with this structure requires the program to visit all the vertexes in the graph, and the maximum amount of edges for each vertex, while requiring to backtrack the entire worklist *visited* to find the wanted path. For this program, in any input restriction in which the input graph has a fixed amount of vertexes, the worst case graph leads symbolic execution through a sequence of decisions such that (i) the worst path includes both at least a *true* and at least a *false* outcome for all decision points in the program, thus making WISE be unable to identify an effective guidance policy, and (ii) the decision history needed to identify the right decision at each point of the sequence is a decision history of different size for each considered input restriction, which makes SPF-WCA be unable to identify a guidance policy. For instance, with reference to graphs of 4 vertexes, the worst case path traverses the first if-statement in the program with the sequence of decisions $\langle \mathit{false}, \mathit{false}, \mathit{false}, \mathit{true} \rangle$, where the first true decision happens after 3 *false* decisions, while with reference to graphs of 5 vertexes the sequence should be $\langle \mathit{false}, \mathit{false}, \mathit{false}, \mathit{false}, \mathit{true} \rangle$, where the first *true* decision happens after 4 *false* decisions.

Listing 4 shows the `memory_fill` program for which the worst path cannot be observed in a suitable input restriction. The program copies a list of inputs into a fixed size buffer with 16 cells, aiming at copying the largest set of non-zero values that fit in the buffer. The program handles two cases: (i) if the amount of inputs is less than the size of the buffer, the program optimizes its performance by making a straight copy of all inputs; (ii) otherwise, it inspects the value of each input, and copies only the largest fitting set of non-zero values. The WCET test case of this program is an input larger than

Listing 4
`memory_fill`

```python
"""
Copy non-zero values in a buffer of 16 cells.

Worst case: a list of non-zero values.
"""
def memory_fill(l):
    memory = [0] * 16
    free = 16

    if len(l) <= free:
        for i in range(len(l)):
            memory[i] = l[i]
        return

    for i in range(len(l)):
        if l[i] != 0:
            free -= 1
            if free >= 0:
                memory[free] = l[i]
```

the buffer and comprised of non-zero values only since each non-zero value makes the program execute the longest block of instructions. Interestingly, this WCET test case reveals a performance bug, since the program wastes time to scan inputs beyond the last one that fits in the buffer. This turns out being also a security vulnerability since an attacker might feed the program with enormous inputs to cause a denial of service.

The worst path of the program showed in Listing 4 cannot be observed when bounding the input list to less than 17 items, since with these inputs the program executes only the part of the algorithm that makes a straight copy of all inputs. Similarly, a scope with 17 input items leads to 2^{17} possible execution paths, which cannot be exhaustively analyzed with symbolic execution in reasonable time. Indeed, for this program, both WISE and SPF-WCA would limit the restricted analysis to at most 16 input items, thus failing to analyze the part of the code that corresponds to the worst case path.

Another similar example is reported in the seminal paper of WISE [21], in which the authors discuss a third-party implementation of `quicksort` (taken from the Java Development Kit (JDK) 1.5) that they considered as an experimental subject. They notice that the JDK-1.5 quicksort included the optimization of using a median-of-9 pivot when sorting arrays with more than 40 items, and a median-of-3 pivot when sorting smaller arrays. Thus, they downgraded the implementation to always use a median-of-3 pivot, acknowledging that otherwise the guidance policy computed in the small scope would be inconsistent with the behavior of the program with more than 40 inputs.

The next section presents our technique for WCET testing whose core novelty is to search for the worst case behavior of a program by heuristically analyzing it directly in the target scope, thus not suffering the above issues.

III. Evolutionary Symbolic Execution

This section presents our novel technique for WCET testing that works by combining symbolic execution with an evolutionary path selection strategy based on a memetic search algorithm. We refer to this technique as Evolutionary Symbolic Execution (ESE). ESE considers the set of the feasible program paths as the search space, through which it steers symbolic execution to explore a sample of incrementally selected program paths, up to ultimately identifying a program path that reveals the worst case execution time of the program. Then, it generates the WCET test case by exploiting the symbolic representation of this path. The core of ESE is a memetic algorithm that incrementally selects the program paths to explore during the search, with the aim of identifying the WCET behavior as quickly as possible.

Algorithm 1: The evolutionary symbolic execution algorithm

```
1  population ← RandomPathSampling(conf:popsize)
2  generations ← 0
3  while TerminationCriterion() do
4      parentpairs ← Selection(population, conf:popsize)
5      for parent_1, parent_2 in parentpairs do
6          child_1, child_2 ← Crossover(parent_1, parent_2)
7          offspring ← offspring ∪ {child_1} ∪ {child_2}

8      population ← population ∪ offspring
9      elite ← Elitism(population, conf:elitesize)
10     population ← elite ∪ Survivals(population − elite)

11     generations ← generations + 1
12     if generations % conf:lsperiod = 0 then
13         worst ← WorstIndividual(population)
14         worst' ← LocalSearch(worst)
15         population ← population − {worst} ∪ {worst'}

16 return WorstIndividual(population)
```

A. Overview of the ESE Algorithm

Algorithm 1 outlines the work flow of our ESE technique in pseudocode. The algorithm starts with generating an initial set (referred to as *population*) of randomly selected feasible paths (line 1). The details of the algorithm RandomSamplingOfPaths, which underlies this step, are presented in Section III-B. In a nutshell, this step steers symbolic execution with a random path selection strategy, stopping after visiting a predefined amount of program paths. We refer to each symbolically analyzed path as an "individual" of the current population. An individual stores the path condition computed for the corresponding path, and the counting of the instructions traversed while symbolically executing that path. As a large majority of techniques for WCET analysis [21], [22], we assume that the amount of instructions that comprise a program path is a good proxy of the execution time of the program when executed along that path. In our search-based algorithm, the function that associates the explored program paths, i.e., the individuals, with the corresponding amount of executed instructions plays the role of the "fitness function" that the algorithm aims to maximize.

Next, the algorithm continues with symbolically executing additional program paths, building on the knowledge of the path costs observed in the current population, to select program paths that might improve on the fitness of the current ones. Working in the style of genetic algorithms [23], we specify the path selection strategy that we use in this step in the form of a crossover mechanism (lines 4–7) that exploits the path conditions of (pairs of) already explored paths (selected as the *parents* at line 4) to steer symbolic execution to traverse new feasible program paths (the *children* at line 6) that might improve on the fitness of the parents. The SELECTION at line 4 corresponds to the classical selection operator of genetic algorithms that picks random individuals from the current population, with the probability of picking each individual being proportional to its current fitness. Section III-C presents the details of the algorithm CROSSOVER (line 6) that comprises the core of the crossover mechanism. It consists in enforcing symbolic execution to comply with subsets of execution conditions selected and combined from the parents' path conditions. We build on the intuition that the execution conditions of the parents may convey costly subpaths to propagate in the children, while the children may still include other (possibly newly explored) subpaths that are not constrained by those execution conditions.

Then, our algorithm consolidates the results of the crossover by shaping a new population that includes a small set of the current fittest individuals (line 9) and a fitness-biased random selection of the others (line 10), and iterates through this process within a predefined timeout (line 3).

At regular intervals, i.e., when the number of iterations is multiple of a predefined *conf:lsperiod* period value (line 12), the algorithm accomplishes a local search, trying to further optimize the worst program path computed so far (lines 13–15). This step indeed classifies the algorithm as "memetic", and not simply as "genetic". Section III-D presents the algorithm LOCALSEARCH (line 14) in detail. It consists in a hill-climbing strategy that incrementally negates a single random condition out of the ones in the path condition of the current worst individual, in the attempt to reveal suboptimal decisions that may further worsen the execution cost when inverted. After a fixed amount of attempts, it returns the worst individual identified along the process, or the initial individual unchanged if all attempts failed, which replaces the previous worst individual thereon (line 15).

The algorithm relies on some parameters that must be configured before invoking it. In Algorithm 1 we indicated these parameters with the notation *conf:⟨parameter_name⟩*, e.g., the timeout at line 3, and the local search period at line 12. Below, we continue to refer to this notation while explaining the other algorithms in detail. Section IV-B summarizes all the configuration parameters of the algorithm and their concrete values in the context of our experiments.

B. Exploring and Representing Program Paths

Our search-based algorithm computes the initial population of candidate solutions by exploring a random sampling of feasible program paths with symbolic execution. Algorithm 2 and Algorithm 3 specify this computation in pseudocode.

Algorithm 2 initializes a population with *npaths* individuals by iterating (Algorithm 2, line 2) as follows. It analyzes the program under test with symbolic execution, to compute the path condition and the number of instructions of a randomly chosen path (Algorithm 2, line 3), and then encodes these results as an individual, i.e., a candidate solution, of the search algorithm (Algorithm 2, line 4). The conjunctive formula that comprises the path condition is the "chromosome" representation of the individual, whereas the atomic constraints represent the "genes". The number of instructions in the program path is the measurement of the fitness of the individual.

Algorithm 2: RANDOMPATHSAMPLING($npaths$)

```
1 population ← ∅
2 for i ← 1 to npaths do
3     pc, instrs ← SymbolicExecution(program)
4     individual ← Individual(pc, instrs)
5     population ← population ∪ {individual}
6 return population
```

Algorithm 3: SYMBOLICEXECUTION(*program*)

```
1 instrs ← 0
2 state ← InitialSymbolicStatecution(program)
3 while ¬IsEndState(state) do
4     successors ← SymbolicExec(state)
5     state ← RandomSelection(successors)
6     instrs ← instrs + 1
7 pc ← PathCondition(state)
8 return pc, instrs
```

Algorithm 3 specifies the exploration of a randomly chosen feasible program path with symbolic execution. It starts with building an initial symbolic state in which the program inputs are assigned to symbolic values (Algorithm 3, line 2), and then iteratively computes the possible successor states as in classic symbolic execution, i.e., by manipulating the symbolic representation of the current state according to the semantics of the current program statement, and updating the program counter to point to the next statement to be executed (Algorithm 3, line 4). A symbolic execution step may yield either a single successor state, when executing non-branching statements in the program, like the statements that correspond to assignments of variables, or two successor states, when executing branching statements, like the statements that evaluate conditions at the decision points in the program. We refer to the standard embodiment of symbolic execution that relies on a constraint solver to incrementally check the feasibility of the successor states, and thus generates only reachable successor states. When the solver confirms more than a successor state, our algorithm chooses a random state out of those and continues the analysis on that state only (Algorithm 3, line 5).

This procedure guarantees that each run of symbolic execution according to Algorithm 3 explores a single program path, which is feasible based on the outcomes of the constraint solver, and which is randomly selected at each decision point where the execution might proceed through multiple distinct feasible paths. As by product, the procedure measures the execution cost of the analyzed path as the number of steps executed while symbolically analyzing the path (Algorithm 3, line 6).

Algorithm 4: CROSSOVER(*parent*$_1$, *parent*$_2$)

```
1  $pc_i \leftarrow$ PATHCONDITION($parent_i$), $\forall\ i \in \{1,2\}$
2  $len_i \leftarrow$ COUNTCONDITIONS($pc_i$), $\forall\ i \in \{1,2\}$
3  $cut_i \leftarrow$ RANDOMINTEGEREXEC(1, $len_i$), $\forall\ i \in \{1,2\}$
4  $pre_1 \leftarrow pc_1[0 : cut_1] \wedge pc_2[cut_2 : len_2]$
5  $pre_2 \leftarrow pc_2[0 : cut_2] \wedge pc_1[cut_1 : len_1]$
6  $children \leftarrow \emptyset$
7  for $pre \in \{pre_1, pre_2\}$ do
8      if RANDOM() < $conf{:}muteprob$ then
9          $pre \leftarrow$ REMOVERANDOMCONDITIONS($pre$)
10     $pc$, $instrs \leftarrow$ SYMBOLICEXECUTION'($program$, $pre$)
11     $children \leftarrow children \cup$ INDIVIDUAL($pc$, $instrs$)
12 return $children$
```

Algorithm 5: SYMBOLICEXECUTION'(*program*, *pre*)

```
1  $instrs \leftarrow 0$
2  $state \leftarrow$ INITIALSYMBOLICSTATECUTION($program$)
3  while $\neg$ISENDSTATE($state$) do
4      $successors \leftarrow$ SYMBOLICEXEC($state$)
5      $successors \leftarrow$ PRUNEUNSATSTATES($successors$, $pre$)
6      if $successors = \emptyset$ then
7          throw $SymbolicExecutionException$
8      $state \leftarrow$ RANDOMSELECTION($successors$)
9      $instrs \leftarrow instrs + 1$
10 $pc \leftarrow$ PATHCONDITION($state$)
11 return $pc$, $instrs$
```

C. Genetic Operators

We bootstrap our search-based algorithm with the random sample of program paths collected as discussed in the previous section, and then proceed with steering symbolic execution to explore additional program paths, by alternating between global and local search phases. The global search phase is a genetic algorithm that exploits the information in the current population of candidate solutions. The local search phase focuses on the best currently identified solution only. This section describes our genetic algorithm, while we present the algorithm of the local search phase in the next section.

Our genetic algorithm fosters the exploration of program paths that may probabilistically include and combine both high-cost subpaths that were already observed in some current individuals, and other (possibly new) randomly explored subpaths of the program. The core of this computation is done by the crossover operator of the genetic algorithm.

Algorithm 4 specifies the crossover operator in pseudocode. The crossover works on pairs of individuals selected from the current population, here denoted as the inputs $parent_1$ and $parent_2$. As we already commented earlier in this section (Algorithm 1, line 4) the selection of $parent_1$ and $parent_2$ is accomplished as a random pick from the current according to a (non-uniform) distribution such that the individuals with higher fitness are selected with higher probability (therefore more often) than the ones with lower fitness. This type of selection mechanism is standard in genetic algorithms, and suites our algorithm with no particular adaptation, thus we do not discuss it further. We remark only that our selection operator enforces $parent_1$ and $parent_2$ to be different individuals of the current population.

Our crossover fosters symbolic execution to explore (at most 2) additional program paths, and enforces these paths to comply with partial sets of the execution conditions excerpted from the path conditions of the two parents, $parent_1$ and $parent_2$, thus possibly replicating subpaths of these individuals. The algorithm (i) synthesizes two new conditions pre_1 and pre_2 by combining the path conditions of the two parents (Algorithm 4, lines 1–5), (ii) mutates each condition $pre \in \{pre_1, pre_2\}$ with some probability (lines 8–9), and (iii) exploits each condition $pre \in \{pre_1, pre_2\}$ with symbolic execution to collect the offspring individuals that the crossover returns as result (lines 10–11).

To synthesize the new conditions pre_1 and pre_2, we cut the path conditions of the two parents at random cutpoints, and join the first part of the path condition of $parent_1$ with the second part of the path condition of $parent_2$ (Algorithm 4, lines 1–4), and the vice-versa (Algorithm 4, line 5). In this phase, the algorithm relies on the knowledge that symbolic execution yields conjunctive path conditions, and thus regards the path conditions simply as lists of conditions. Then, with predefined probability *conf:muteprob*, we may mutate the new conditions (both pre_1 and pre_2, either one, or none of them) by removing some inner conditions chosen at random (Algorithm 4, lines 8–9). The actual mutation algorithm (not shown) removes from a minimum of a single condition up to a maximum of $10\,\%$ of the inner conditions. After each removal, it decides with even probability either to stop or continue with removing a further condition; it stops necessarily after removing the maximum number of conditions.

Algorithm 5 specifies the symbolic exploration of a feasible program path that complies with a set of conditions *pre* synthesized in the crossover algorithm. The symbolic execution algorithm mimics all steps of Algorithm 3 with the only additional behavior of pruning the symbolic states that are incompatible with the precondition *pre* (Algorithm 5, lines 5–7). We rely on the constraint solver to decide whether the path condition of a current symbolic state is satisfiable in conjunction with *pre*, and prune the symbolic states for which the solver returns an unsatisfiability verdict (Algorithm 5, line 5). If all successor states happen to be incompatible with *pre* (Algorithm 5, line 6), the symbolic execution algorithm terminates with an exception (Algorithm 5, line 7) indicating that the precondition prevents the execution of any feasible path of the program. The crossover does not generate the individual in this latter case – in Algorithm 4, for simplicity, we do not show this exceptional behavior explicitly.

This procedure guarantees that each (unexceptional) run of symbolic execution according to Algorithm 5 explores a single feasible program path, which complies with subsets of the execution conditions of the parent individuals, thus likely contains subpaths that belong also to those individuals, and which is randomly selected at any decision points with multiple paths that are not constrained by the precondition *pre*.

Algorithm 6: LOCALSEARCH(*individual*)

```
1 for attempt ← 1 to conf:lsattempts do
2     pre ← PATHCONDITION(individual)
3     pre ← INVERTRANDOMCONDITION(pre)
4     pc, instrs ← SYMBOLICEXECUTION'(program, pre)
5     if instrs > individual.fitness then
6         individual ← INDIVIDUAL(pc, instrs)
7 return individual
```

D. Local Search

At regular intervals, our algorithm accomplishes a local search phase in which it tries to make small focused changes to the individual that has the maximum cost in the current population, trying to further optimize that candidate solution. In this phase, the algorithm proceeds in the style of the "hill climbing" algorithm, i.e., incrementally changing single elements of the solution and accepting those changes that result in better solutions. The changes consist in exploring program paths that differ from the current one for the outcome at a decision point.

Algorithm 6 specifies the local search algorithm in pseudocode. The input *individual* denotes the individual with maximum cost in the current population at the beginning of the local search. The algorithm accomplishes a fixed amount of iterations (line 1) in which it inverts a condition chosen at random out of the path condition of the current individual, i.e., it replaces that condition with its logical negation (lines 2–3). Then, it computes a new individual, by exploiting the modified path condition with symbolic execution in a similar fashion as we explained for the crossover operator (line 4). If the new individual has higher execution cost than the current one, the local search continues by focusing on the new individual (lines 5–6), since the new individual is closer to the optimal solution than the previous one, and iterates to further optimize this individual. Otherwise, the algorithm continues without changing the current individual. The algorithm returns the individual with the highest cost identified throughout this process.

The intuition that underlies the local search phase is that, when the global search computes some solution that is close to the optimum, likely it is a program path that mimics the worse program path except for a small set of branch decisions. In this situation, the crossover is generally ineffective to find the missing optimizations without altering other parts of the path. Conversely, trying a punctual exploration of the possible changes is more likely to succeed. The local search is also effective to identify subpaths with regular behavior, e.g., a subpath in which all decisions at an *if* statement must regularly take the same branch (or alternated branches) for a given amount of subsequent evaluations of that decision point. Although the global search may succeed to identify a majority of the needed decisions, the fully regular sequence may appear like a singularity in the search space. The local search may incrementally spot the suboptimal decisions, and fix them.

Table I
THE PROGRAMS CONSIDERED IN THE EXPERIMENTS.

Program	Category	Complexity
`alternate_0`	Sequence checking	$\mathcal{O}(n)$
`is_palindrome`	Sequence checking	$\mathcal{O}(n)$
`merge_sort`	Sequence sorting	$\mathcal{O}(n \log n)$
`quicksort_jdk`	Sequence sorting	$\mathcal{O}(n^2)$
`kmp`	Sequence search	$\mathcal{O}(n)$
`memory_fill`	Sequence operation	$\mathcal{O}(n)$
`dfs`	Graph search	$\mathcal{O}(n^2)$
`bfs`	Graph search	$\mathcal{O}(n^2)$

IV. EXPERIMENTS

We implemented a prototype of ESE for Python programs (ESE$_{py}$), and used it to investigate the effectiveness of ESE with respect to a set of sample programs for inputs of increasing size. This section discusses our experiments indicating that:

1) *ESE effectively steers symbolic execution towards generating WCET test cases*. We show that ESE significantly outperforms both the baseline strategies of analyzing the feasible program paths in depth-first or random order, respectively.
2) *ESE properly complements the state-of-the-art approaches WISE and SPF-WCA*. We show that these approaches work well for programs where the worst-case behavior generalizes with regularity for inputs of increasing size, but are significantly worse than ESE in the many cases in which there is no such regularity.

Below we introduce our prototype of ESE, and discuss the setting and the results of our experiments.

A. Evolutionary Symbolic Execution Prototype

Our prototype ESE$_{py}$ is implemented in Python, based on a purposely designed symbolic executor for Python programs. The symbolic executor relies on the Z3 constraint solver [3], and works by instrumenting that inputs data of the program under test, to make the Python interpreter handle the inputs as symbolic values (similarly to the work of Saxena et al. on symbolic execution of JavaScript programs [24]). It can currently handle input data that consist of integers and bounded collections of integers.

B. Experiment Setting

Our experiments challenge ESE$_{py}$ to generate WCET test cases for the sample programs in Table I. `alternate_0`, `is_palindrome`, `dfs`, and `memory_fill` are the programs that we already discussed in Section II, `merge_sort` and `quicksort_jdk` are the popular sorting algorithms (in particular `quicksort_jdk` is the JDK-1.5 quicksort algorithm that switches from a mid-array pivot for arrays with less than 7 items, to a median-of-3 pivot and then a median-of-9 pivot for larger arrays with less or more than 40 items, respectively), `kmp` (Knuth-Morris-Pratt) scans a sequence while searching for an occurrence of a given subsequence, `bfs` (breadth-first search) searches in a graph in breath-first order.

Table II
PARAMETERS OF ESE IN THE EXPERIMENTS.

Parameter	Description	Value
`conf:timeout`	The time budget for the search (min)	60
`conf:popsize`	The size of the population	50
`conf:elitesize`	The individuals retained as elite	5
`conf:muteprob`	The probability of mutation	0.2
`conf:lsperiod`	The generations before local search	10
`conf:lsattempts`	The changes during local search	25

We used ESE$_{py}$ for WCET testing of these programs instantiated with respect to input lists or graph adjacency matrices that consist of 10, 50, 75 or 100 symbolic integers, respectively. For `kmp` we fixed the length of the searched subsequence to 3 symbolic integers. In these experiments, ESE$_{py}$ assigns the parameters of the ESE algorithm as summarized in Table II.

For each program and input size, we evaluated ESE both in absolute terms, and in relative terms, with respect to the competing approaches at the state of the art. In absolute terms, we compared the execution cost measured by profiling the programs with (i) the worst case inputs identified with ESE$_{py}$ and (ii) manually identified worst case inputs. In relative terms, we compared the worst case inputs computed with ESE$_{py}$ with the worst case inputs obtained with symbolic execution equipped with either (i) the classical depth-first (DFS) or (ii) random path selection strategies (RAND), or yet with the guidance policies identified with either (iii) WISE [21] or (iv) SPF-WCA [22] approaches, respectively.

To be fair, we implemented the four competing approaches on top of our symbolic executor for Python. DFS$_{py}$ makes the symbolic executor visit the program paths in depth-first order. RAND$_{py}$ works according to the random path selection strategy that we illustrated in Algorithm 3. WISE$_{py}$ and SPF-WCA$_{py}$ implement the algorithms of WISE and SPF-WCA, respectively, whose core step is to train a guidance policy for steering symbolic execution to identify the worst case (recall Section II). WISE$_{py}$ trains the guidance policy by analyzing the target program with (initially unitary and then) increasingly larger input bounds, until a training time budget. It then selects the worst path identified for the largest bound for which it successfully completes the exhaustive analysis of the program, and maps this path to a guidance policy, *decision point* $\longrightarrow$ *decisions*: the guidance policy associates the program decision points with the corresponding decisions (*none*, *true*, *false* or both *true* and *false*) taken at least once along that path. SPF-WCA$_{py}$ trains the guidance policy in a similar way, but builds a finer-grained guidance policy specified as $\langle$*decision point*, *decision history*$\rangle$ $\longrightarrow$ *decisions*, for the possible decision histories of a given length. SPF-WCA$_{py}$ uses *decision histories* of length one.

We ran ESE$_{py}$, DFS$_{py}$, and RAND$_{py}$ against each program instance with a time budget of 60 min. For WISE$_{py}$ and SPF-WCA$_{py}$, we split the time budget in 2 tranches of 30 min each, for the training phase and the test generation phase, respectively, thus allowing for guidance policies trained on inputs of meaningful size, while still leaving adequate time to exhaustively analyze the decision points that the guidance policy does not constrain. Hereon, we omit the *py* subscript when referring to the prototypes.

C. Results

Table III reports the results of our experiments. For each program listed in column "program" and each input size listed in column "size", the six columns "WCET test case execution cost" report the execution cost, expressed as the number of executed instructions, which we obtained by profiling the program with the worst case inputs identified either manually (column "manual") or with the techniques ESE, RAND, DFS, WISE and SPF-WCA, respectively. The five columns *Analysis time* report the time spent by the techniques to identify the respective worst case inputs: It is the time that a technique took to identify a test input that reproduces the same execution cost as the manual test case, or the entire analysis budget (60 min) if only suboptimal worst inputs were identified. For the experiments with the techniques, ESE and RAND the data are the average values across 10 runs, to control for the randomness in these techniques.

With inputs bounded at size 10, all techniques successfully compute the WCET test case efficiently for all programs, the only exception being `quicksort_jdk`. For `quicksort_jdk` we observe that: (i) ESE is the only technique that identifies the WCET test case; (ii) the systematic DFS strategy fails to visit the worst-case path of this program in 60 min; (iii) both WISE and SPF-WCA synthesize ineffective guidance policies. The analysis of this latter finding reveals that it is the expected consequence of the change of behavior of the program with arrays smaller or greater than 7 items, respectively: the behavior of the program that WISE and SPF-WCA observe with inputs of small size (less than 7 items) does not generalize with inputs of size 10.

We study the effectiveness and the significance ESE by comparing it with the baseline strategies DFS and RAND. For inputs bounded at size 50, 75 and 100, ESE computes the same optimal WCET test case as DFS for the programs `is_palindrome` and `bfs`, and consistently outperforms DFS for all other programs. The dominance of ESE on DFS confirms that ESE successfully steers the search towards the worst-case program path, and successfully contrasts the path-explosion issues incurred with the systematic strategy of DFS.

ESE consistently outperforms the baseline random strategy RAND, except for the of experiments `merge_sort` and `quicksort_jdk`, in which ESE and RAND both compute comparable sub-optimal worst inputs. The experiments in which ESE outperforms RAND indicate the significance of the memetic algorithm ESE with respect to relying on a purely random path selection strategy.

We further inspected the results of `merge_sort` and `quicksort_jdk` in the cases in which ESE does not significantly differ from RAND. We found that, when these programs execute for large arrays, the length of their execution paths hampers ESE from completing the generation of the

Table III
RESULTS OBTAINED WITH ESE AND THE COMPETING TECHNIQUES FOR THE SUBJECT PROGRAMS FOR DIFFERENT SIZE OF THE INPUTS.

Program	Size	WCET test case execution cost (#instructions)						Analysis time (min)				
		Manual	ESE	RAND	DFS	WISE	SPF-WCA	ESE	RAND	DFS	WISE	SPF-WCA
`alternate_0`	10	61	61	61	61	61	61	< 1	60	< 1	< 1	< 1
	50	301	301	277	269	267	301	13	60	60	60	30
	75	451	451	413	393	391	451	11	60	60	60	30
	100	601	601	543	517	517	601	31	60	60	60	30
`is_palindrome`	10	22	22	22	22	22	22	< 1	60	< 1	< 1	< 1
	50	102	102	37	102	102	102	< 1	60	< 1	30	30
	75	152	152	37	152	152	152	7	60	< 1	30	30
	100	202	202	37	202	202	202	2	60	< 1	30	30
`merge_sort`	10	279	279	279	277	276	279	1	60	60	60	< 1
	50	2019	2005	2003	1939	1939	2008	20	60	60	60	1
	75	3249	3217	3218	3108	3108	3242	8	60	60	60	4
	100	4549	4495	4501	4341	4341	4527	44	60	60	60	7
`quicksort_jdk`	10	348	348	302	248	248	296	20	60	60	60	60
	50	1665	1298	1232	548	512	562	38	60	60	60	60
	75	2528	1777	1750	673	635	685	48	60	60	60	60
	100	3666	2204	2252	798	764	846	24	60	60	60	60
`kmp`	10	83	83	80	83	83	83	1	60	26	43	22
	50	363	334	306	306	306	306	29	60	60	60	60
	75	538	478	444	442	442	442	53	60	60	60	60
	100	713	606	587	581	578	578	56	60	60	60	60
`memory_fill`	10	25	25	25	25	25	25	< 1	60	< 1	< 1	< 1
	50	222	222	212	188	186	186	4	60	60	60	60
	75	322	322	301	263	261	261	10	60	60	60	60
	100	422	422	390	336	334	334	21	60	60	60	60
`dfs`	10	18	18	18	18	18	18	< 1	60	< 1	< 1	< 1
	50	94	94	82	78	78	85	1	60	60	60	60
	75	156	156	122	116	116	116	10	60	60	60	60
	100	193	193	143	133	133	138	7	60	60	60	60
`bfs`	10	26	26	26	26	26	26	< 1	60	< 1	< 1	< 1
	50	114	114	102	114	114	114	1	60	< 1	30	30
	75	182	182	146	182	182	182	21	60	< 1	30	30
	100	222	222	176	222	222	222	6	60	< 1	30	30

initial random population in 60 min, and thus ESE indeed falls back to behave exactly as RAND, confirming the observed data. This is mostly a weakness of the current prototype that performs badly to symbolically execute program paths with large amounts of instructions and branches. Nevertheless, these data pinpoint an intrinsic limitation of ESE, which may require long time to analyze programs with very high WCET figures. In the future, we aim to port ESE on a mature symbolic executor, and evaluate it with respect to a larger selection of programs and test budgets, to quantify the extent of this issue.

Finally, we compare ESE with the state-of-the-art techniques WISE and SPF-WCA. In particular, we elaborate on the comparison by restricting our attention to ESE and SPF-WCA, since WISE is always equivalent to or worse than SPF-WCA in our experiments. The data in Section IV-C reveal the mutually complementary strengths of these techniques. ESE outperforms SPF-WCA in the experiments with `quicksort_jdk`, `kmp`, `memory_fill` and `dfs`. It is equivalent to SPF-WCA for `alternate_0`, `is_palindrome` and `bfs`, and worse than SPF-WCA for `merge_sort`.

The experiments in which ESE outperforms SPF-WCA confirm the observations that we made in Section II about (i) programs for which SPF-WCA cannot compute a conclusive guidance policy (e.g., `dfs` and `kmp`) and thus falls back to the same effectiveness of DFS, and (ii) programs for which the worst-case behavior observed on small inputs does not generalise for large inputs (e.g., `memory_fill` and `quicksort_jdk`). For these programs, SPF-WCA ends up with analyzing sets of program paths that may not include the worst-case path, while ESE suites better by sampling the target program scope directly.

On the other side of the spectrum, `merge_sort` has a regular worst case, which consists of alternating decisions when merging the items of the sorted sublists. This type of worst-case behavior is similar to the one that we described in Section II with reference to the program `alternate_0`, and can be efficiently captured with a guidance policy in the same way as SPF-WCA, while the length of the execution paths of `merge_sort` challenges ESE as we commented above.

In general, both WISE and SPF-WCA assume the existence (and the availability) of a monotone relation between inputs of increasing size and increasing worst-case execution costs. For the programs considered in our experiments, which take inputs shaped as list and graph structures, it is natural to identify this relation based on the size of the data structures, e.g., devising a guidance policy for `merge_sort` by considering the sorting of lists of small size, and then using that guidance policy to analyze the worst-case execution cost when sorting large lists. However, for many practical programs, which may take multiple inputs, shaped as various types of interwoven data structures, satisfying this assumption may not be easy: each different input may or not participate in the worst-case behavior of the program, and the definition of the size increment related to the combination of the participating inputs can be peculiar. ESE dismisses this requirement by working directly on the target program scope, and can thus natively address this general case.

D. Threats and Limitations

The goal of these experiments was primarily to provide empirical evidence of the implications of the *regular* and *irregular* worst case phenomena in both ESE and the competing techniques, but, as the main threat, we cannot claim that the considered programs are representative of all the challenges that may emerge when applying our technique to general purpose industrial scale programs, and thus our result may not generalize. A larger selection of benchmarks is still needed to systematically assess the absolute and mutual benefits of ESE, WISE and SPF-WCA. Moreover, we are aware that ESE is intrinsically bound to the theoretical and practical limitations of symbolic execution with constraints that the constraint solver cannot cope with, low level (possibly unsafe) operations that can be hard to simulate symbolically, and the possible mismatch between the static counting of the traversed instructions and the actual execution time of the program.

V. Related Work

Many static WCET analyzers for hard real-time systems combine symbolic execution with timed automata and static analysis to compute WCET estimations [14], [15], [16], [17], [18], [19], [20]. In the context of these techniques, symbolic execution complements (mostly static) analysis procedures, assisting the pruning of infeasible execution paths that would jeopardize the precision of the estimations. Knoop et al. augment the r-TuBound WCET analyzer by symbolically executing the identified worst path to determine whether the path is infeasible; if so, they iterate the analyzer with refined constraints to find a different path [19]. Biere et al. use symbolic execution to enhance the ability of the analyzer to compute loop bounds [18]. Kebbal end Sainrat rely on symbolic execution to automatically extract information about the program semantics that they used to tighten the WCET estimates [16]. In general, these techniques pursue the goal of computing WCET estimations, but not the generation of WCET test cases, and can hardly cope with general purpose programs with large path spaces.

Our work is closely related to techniques that rely on symbolic execution to generate test data for worst case and load testing, security exploit synthesis and structural code coverage [25], [21], [22], [26], [9], [8], [27], [13], [12]. We have extensively commented on state of the art in WCET testing [21], [22] in Section II, and experimentally compared the existing techniques, WISE and SPF-WCA, with our approach in Section IV. Zhang et al. propose a technique for load testing iterating symbolic execution to explore increasingly longer program paths, with a greedy path selection heuristic that, at each iteration, continues to explore only the paths that yielded the highest workload at the previous iteration [25]. Avgerinos et al. use symbolic execution to find working exploits for security vulnerabilities, using path selection heuristics based on domain knowledge on the vulnerabilities [26]. In general, many symbolic-execution-based test generators embeds path selection heuristics that address structural code coverage [9], [8], [27], [13], [28], [12]. The approach discussed in this paper is the first to investigate an evolutive path selection heuristic for symbolic execution, where a cost model guides the search. In the future, we aim to study if ESE can be beneficial for other goals other than WCET testing.

The area of Search-Based Software Testing (SBST) encompasses techniques that use search-based optimization algorithms for the generation of test data [29]. So far a substantial SBST research effort has targeted structural coverage [30], [31], [32], [33], [34], [35], [36], [37], [38]. In particular, we briefly survey the techniques that investigated combinations of SBST and symbolic execution [39], [40], [28]. Other researchers exploit symbolic execution to optimize the evolutionary algorithms used in SBST [41], [42], [43], [44]. Xie et al. and Baluda investigate search-based path selection strategies to steer symbolic execution towards paths with higher chances to execute uncovered branches [45], [46]. None of these previous SBST techniques targeted WCET testing so far, and none of them investigated a search strategy based on combining the path conditions of the incrementally analyzed program paths, as we propose in this paper.

VI. Conclusions

In this paper, we presented a novel technique for Worst-Case Execution Time (WCET) testing of software programs. We provided compelling examples of the limitations of the state-of-the-art techniques based on guided symbolic execution, and moved on by proposing a search-based approach that combines symbolic execution with a memetic algorithm that steers the incremental analysis of increasingly worse program paths. In fact, the experiments presented in this paper support our hypothesis that there exists a class of programs for which WCET testing cannot be addressed with the guided symbolic execution approach, and indicate preliminary evidence that our Evolutionary Symbolic Execution (ESE) technique provides a viable solution for WCET testing of these programs.

We believe that the benefits of ESE over guided symbolic execution generalize to practical programs, which may take multiple inputs, shaped as various complex data structures, each impacting in its own specific way on the worst-case behavior of the program. ESE may conceptually address such programs directly, while guided symbolic execution cannot. Currently, we are porting ESE on a mature symbolic execution framework, i.e., JBSE [47], to enable experiments with ESE on more and more general programs than the ones considered in this paper.

References

[1] J. C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[2] L. A. Clarke, "A Program Testing System," in *ACM Annual Conference*, 1976, pp. 488–491.

[3] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2008, pp. 337–340.

[4] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The mathSAT5 SMT Solver," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2013, pp. 93–107.

[5] B. Dutertre, "Yices 2.2," in *International Conference on Computer Aided Verification (CAV)*, 2014, pp. 737–744.

[6] X. Deng, J. Lee, and Robby, "Bogor/Kiasan: A K-Bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2006, pp. 157–166.

[7] S. Anand, C. S. P˘as˘areanu, and W. Visser, "JPF-SE: A Symbolic Execution Extension to Java PathFinder," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2007, pp. 134–138.

[8] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs," in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008, pp. 209–224.

[9] N. Tillmann and J. de Halleux, "Pex: White Box Test Generation for .NET," in *International Conference on Tests and Proofs (TAP)*, 2008, pp. 134–153.

[10] G. Li, I. Ghosh, and S. Rajan, "KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs," in *International Conference on Computer Aided Verification (CAV)*, 2011, pp. 53–68.

[11] P. Braione, G. Denaro, and M. Pezzè, "Symbolic Execution of Programs with Heap Inputs," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2015, pp. 602–613.

[12] C. Cadar and K. Sen, "Symbolic Execution for Software Testing: Three Decades Later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013.

[13] P. Braione, G. Denaro, A. Mattavelli, M. Vivanti, and A. Muhammad, "Software Testing with Code-Based Test Generators: Data and Lessons Learned from a Case Study with an Industrial Software Component," *Software Quality Journal*, vol. 22, no. 2, pp. 311–333, Jun. 2014.

[14] T. Lundqvist and P. Stenström, "An Integrated Path and Timing Analysis Method Based on Cycle-Level Symbolic Execution," *Real-Time Systems*, vol. 17, no. 2-3, pp. 183–207, Dec. 1999.

[15] F. Stappert and P. Altenbernd, "Complete Worst-Case Execution Time Analysis of Straight-Line Hard Real-Time Programs," *Journal of Systems Architecture*, vol. 46, no. 4, pp. 339–355, 2000.

[16] D. Kebbal and P. Sainrat, "Combining Symbolic Execution and Path Enumeration in Worst-Case Execution Time Analysis," in *International Workshop on Worst-Case Execution Time Analysis (WCET)*, F. Mueller, Ed., 2006.

[17] B. Benhamamouch, B. Monsuez, and F. Védrine, "Computing WCET Using Symbolic Execution," in *International Conference on Verification and Evaluation of Computer and Communication Systems (VECoS)*, 2008, pp. 128–139.

[18] A. Biere, J. Knoop, L. Kovács, and J. Zwirchmayr, "The Auspicious Couple: Symbolic Execution and WCET Analysis," in *International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2013, pp. 53–63.

[19] J. Knoop, L. Kovács, and J. Zwirchmayr, "WCET Squeezing: On-Demand Feasibility Refinement for Proven Precise WCET-Bounds," in *International Conference on Real-Time Networks and Systems (RTNS)*, 2013, pp. 161–170.

[20] K. S. Luckow, C. S. P˘as˘areanu, and B. Thomsen, "Symbolic Execution and Timed Automata Model Checking for Timing Analysis of Java Real-Time Systems," *EURASIP Journal on Embedded Systems*, vol. 2015, no. 1, p. 2, Sep. 2015.

[21] J. Burnim, S. Juvekar, and K. Sen, "WISE: Automated Test Generation for Worst-Case Complexity," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2009, pp. 463–473.

[22] K. Luckow, R. Kersten, and C. Pasareanu, "Symbolic Complexity Analysis Using Context-Preserving Histories," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Mar. 2017, pp. 58–68.

[23] M. Roper, "Computer Aided Software Testing Using Genetic Algorithms," in *International Software Quality Week (QW)*, 1997.

[24] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A Symbolic Execution Framework for JavaScript," in *IEEE Symposium on Security and Privacy (SP)*, 2010, pp. 513–528.

[25] P. Zhang, S. Elbaum, and M. B. Dwyer, "Automatic Generation of Load Tests," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2011, pp. 43–52.

[26] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic Exploit Generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, Feb. 2014.

[27] J. Burnim and K. Sen, "Heuristics for Scalable Dynamic Test Generation," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2008, pp. 443–446.

[28] P. Braione, G. Denaro, A. Mattavelli, and M. Pezzè, "Combining Symbolic Execution and Search-Based Testing for Programs with Complex Heap Inputs," in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2017, pp. 90–101.

[29] P. McMinn, "Search-Based Software Test Data Generation: A Survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[30] G. Fraser and A. Arcuri, "Whole Test Suite Generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[31] G. Fraser, A. Arcuri, and P. McMinn, "A Memetic Algorithm for Whole Test Suite Generation," *Journal of Systems and Software*, vol. 103, no. C, pp. 311–327, 2015.

[32] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary Test Environment for Automatic Structural Testing," *Information and Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.

[33] G. Fraser and A. Zeller, "Mutation-Driven Generation of Unit Tests and Oracles," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 278–292, 2012.

[34] M. Vivanti, A. Mis, A. Gorla, and G. Fraser, "Search-Based Data-Flow Test Generation," in *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2013, pp. 370–379.

[35] P. Tonella, "Evolutionary Testing of Classes," in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2004, pp. 119–128.

[36] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware Test Suite Prioritization," in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2006, pp. 1–12.

[37] T. E. Colanzi, W. K. G. Assunção, S. R. Vergilio, and A. Pozo, "Integration Test of Classes and Aspects with a Multi-Evolutionary and Coupling-Based Approach," in *International Symposium on Search Based Software Engineering (SSBSE)*, 2011, pp. 188–203.

[38] F. Gross, G. Fraser, and A. Zeller, "Search-Based System Testing: High Coverage, No False Alarms," in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2012, pp. 67–77.

[39] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2005, pp. 365–381.

[40] K. Inkumsah and T. Xie, "Improving Structural Testing of Object-Oriented Programs Via Integrating Evolutionary Testing and Symbolic Execution," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2008, pp. 297–306.

[41] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos, "Symbolic Search-Based Testing," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2011, pp. 53–62.

[42] J. P. Galeotti, G. Fraser, and A. Arcuri, "Improving Search-Based Test Suite Generation with Dynamic Symbolic Execution," in *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2013.

[43] J. Malburg and G. Fraser, "Combining Search-Based and Constraint-Based Testing," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2011, pp. 436–439.

[44] K. Lakhotia, M. Harman, and P. McMinn, "Handling Dynamic Data Structures in Search Based Testing," in *Genetic and Evolutionary Computation Conference (GECCO)*, 2008, pp. 1759–1766.

[45] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte, “Fitness-Guided Path Exploration in Dynamic Symbolic Execution,” in *International Conference on Dependable Systems and Networks (DSN)*, 2009, pp. 359–368.

[46] M. Baluda, “EvoSE: Evolutionary Symbolic Execution,” in *International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST)*, 2015, pp. 16–19.

[47] P. Braione, G. Denaro, and M. Pezzè, “JBSE: A Symbolic Executor for Java Programs with Complex Heap Inputs,” in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2016, pp. 1018–1022.