Symbolic Execution with Existential Second-Order Constraints

Sergey Mechtaev
National University of Singapore
Singapore
mechtaev@comp.nus.edu.sg

Alberto Griggio
Fondazione Bruno Kessler
Italy
griggio@fbk.eu

Alessandro Cimatti
Fondazione Bruno Kessler
Italy
cimatti@fbk.eu

Abhik Roychoudhury
National University of Singapore
Singapore
abhik@comp.nus.edu.sg

ABSTRACT

Symbolic execution systematically explores program paths by solving path conditions — formulas over symbolic variables. Typically, the symbolic variables range over numbers, arrays and strings. We introduce symbolic execution with existential second-order constraints — an extension of traditional symbolic execution that allows symbolic variables to range over functions whose interpretations are restricted by a user-defined language. The aims of this new technique are twofold. First, it offers a general analysis framework that can be applied in multiple domains such as program repair and library modelling. Secondly, it addresses the path explosion problem of traditional first-order symbolic execution in certain applications. To realize this technique, we integrate symbolic execution with program synthesis. Specifically, we propose a method of second-order constraint solving that provides efficient proofs of unsatisfiability, which is critical for the performance of symbolic execution. Our evaluation shows that the proposed technique (1) helps to repair programs with loops by mitigating the path explosion, (2) can enable analysis of applications written against unavailable libraries by modelling these libraries from the usage context.

CCS CONCEPTS

• **Software and its engineering** → **Automatic programming**; **Software testing and debugging**;

KEYWORDS

Program synthesis, Program repair, Library modelling

ACM Reference Format:
Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti, and Abhik Roychoudhury. 2018. Symbolic Execution with Existential Second-Order Constraints. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3236024.3236049

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236049

1 INTRODUCTION

Symbolic execution (SE) is a widely used program analysis technique that has found many applications including automated test generation [18], software verification [14], input filtering [10], program debugging [30] and program repair [26, 28]. In symbolic execution, program inputs are assigned symbolic variables instead of concrete values. The result of executing a program with symbolic inputs is a set of constraints over these symbolic variables called path conditions. Path condition of a program path captures all inputs that would drive the execution along this program path. Symbolic variables used in existing symbolic execution systems typically range over numbers, arrays and strings.

We introduce symbolic execution with existential second-order constraints (SE-ESOC), that extends traditional symbolic execution by allowing symbolic variables to range over functions. A function in a program can be marked as "symbolic" via a second-order symbolic variable. Then, the goal of SE-ESOC is to *synthesize* an interpretation of this function that satisfies certain reachability properties of the analyzed program (the properties depend on the application). SE-ESOC collects constraints on second-order variables and solves them through program synthesis.

Example 1.1. Assume that `search(data, pred)` returns the index of an element of the array `data` that satisfies the predicate `pred`. Consider the question "What predicate would make `search` return 2 given the array `[0, 1, 2]`?". SE-ESOC can answer this question by executing `search([0, 1, 2],` ρ`)` symbolically with a second-order variable ρ, and synthesizing e.g. $\rho := \lambda x.\ x > 1$.

Contrary to works [13] utilizing the theory of uninterpreted functions, SE-ESOC aims to discover *implementations* of symbolic functions. Thus, SE-ESOC takes in a *language of interpretations* for second-order variables. Similar to the syntax-guided program synthesis approach [3], a language of interpretations is defined in our approach via a context-free grammar, and a size bound.

Applications. In this work, we describe two applications of SE-ESOC: an application to program repair, and a novel application to library modelling from the usage context. In the context of program repair, suspicious statements in the buggy program can be replaced with second-order symbolic variables. Thus, a fragment of code in a program can be abstracted as a second-order symbolic variable. Instantiations of the second-order variable then amount to alternate code fragments to replace the current one, thereby bringing out the connection between SE-ESOC and program repair. SE-ESOC

```
size_t search(int data[],
              size_t len,
              int (*pred)(int)) {
  size_t i;
  for (i = 0; i < len; i++)
    if (pred(data[i]))
      return i;
  return len;
}
```

(a) Search function.

Inputs $\alpha_1, \alpha_2, \alpha_3$ are integer variables:

`int pos(int x) { return x > 0; }`
`search((int[]){`$\alpha_1, \alpha_2, \alpha_3$`}, 3, pos);`

Symbolic execution results:

Path condition	Input	Output
$\alpha_1 > 0$	$\{1, 0, 0\}$	0
$\alpha_1 \leq 0 \land \alpha_2 > 0$	$\{0, 1, 0\}$	1
$\alpha_1 \leq 0 \land \alpha_2 \leq 0 \land \alpha_3 > 0$	$\{0, 0, 1\}$	2
$\alpha_1 \leq 0 \land \alpha_2 \leq 0 \land \alpha_3 \leq 0$	$\{0, 0, 0\}$	3

(b) Traditional SE.

Input ρ is a functional (second-order) variable:

`search((int[]){0, 1, 2}, 3, `ρ`);`

Symbolic execution results:

Path condition	Input	Output
$\rho(0)$	$\lambda x.\ true$	0
$\neg\rho(0) \land \rho(1)$	$\lambda x.\ x > 0$	1
$\neg\rho(0) \land \neg\rho(1) \land \rho(2)$	$\lambda x.\ x > 1$	2
$\neg\rho(0) \land \neg\rho(1) \land \neg\rho(2)$	$\lambda x.\ false$	3

(c) SE-ESOC.

Figure 1: Testing search function via traditional SE and SE-ESOC.

can directly synthesize a patch by finding interpretations of the symbolic functions. In the context of library modelling, SE-ESOC can enable symbolic analysis of applications written against unavailable libraries (this might be useful, for example, for analyzing partially released applications with proprietary software components). Specifically, unknown library functions can be replaced with second-order symbolic variables, which can be complemented with library sketches. Then, SE-ESOC can synthesize a model of the library by analyzing how the library is used inside the application. In both of these applications, SE-ESOC synthesizes interpretations of second-order variables that enable the reachability of test assertions, which is equivalent to passing the tests.

Addressing path explosion. The proposed technique alleviates the path explosion problem in program repair algorithms relying on first-order symbolic execution. Program repair techniques such as SemFix [28] and Angelix [26] split patch generation into two steps. First, they replace suspicious statements with first-order symbolic variables and infer specification via symbolic execution. As a second step, they synthesize patches that satisfy the inferred specification. An important limitation of these approaches is that they have to potentially explore an infinite number of paths e.g. if the suspicious statements are inside a loop. However, by raising the order of path constraints, we can efficiently prune irrelevant paths. The pruning is achieved by avoiding paths that are infeasible in the context of considered language of interpretations (the space of patches).

Technical challenges and solutions. To implement SE-ESOC, it is sufficient, in principle, to apply a syntax-guided synthesizer [3] for solving queries with second-order variables. However, existing synthesis algorithms are not suitable for this application. SMT solvers used in symbolic execution engines [8, 36] cannot solve the considered kind of second-order constraints, however our goal was to support second-order variables without switching to a specialized solver. The reason for not switching to specialized solvers is that we might have second-order variables as well as first-order variables in various theories in a single path condition. Then, a suitable approach to support second-order variables is to encode second-order formulas through first-order formulas as proposed by Jha et al. [15]. However, our experiments demonstrated that the mentioned encoding provides highly inefficient unsatisfiability proofs. Meanwhile, the performance of symbolic execution critically depends on the performance of unsatisfiable queries for on-the-fly pruning of infeasible paths. To address the above challenges, we introduce a new method of second-order constraint solving that relies on propositional encoding, which substantially improves the efficiency of unsatisfiability proofs compared with previous techniques.

Contributions. The contributions of this work are the following. First, we introduce SE-ESOC — an extension of symbolic execution that allows symbolic variables to range over functions whose interpretations are restricted by a user-defined language. Secondly, we propose a method of second-order constraint solving based on propositional encoding that provides efficient unsatisfiability proofs. Finally,we conduct an evaluation that demonstrates that (1) in the context of program repair, SE-ESOC helps to repair programs with loops by alleviating the path explosion, and (2) in the context of library modelling, SE-ESOC can enable analysis of applications written against unavailable libraries by modelling them from the usage context.

2 OVERVIEW

This section describes (1) second-order formulas considered in this work (2) the difference between traditional SE and SE-ESOC, (3) an application of SE-ESOC to program repair and (4) an application of SE-ESOC to library modelling.

2.1 Second-order Formulas

We view second-order constraint solving as an instance of program synthesis [11]. Formally, we consider second-order formulas with existentially quantified second-order variables, and a Henkin (non-standard) semantics [27] of satisfiability (Definition 4.1). Specifically, each second-order variable is associated with a domain of interpretations defined via a user-provided language.

Example 2.1. Assume that ρ is a second-order variable whose domain is restricted by the language LIA defined as follows:

$$\langle Term \rangle ::= \langle Var \rangle \mid \langle Constant \rangle$$
$$\mid \quad \langle Term \rangle \text{ `+' } \langle Term \rangle \mid \langle Term \rangle \text{ `-' } \langle Term \rangle \mid \langle Constant \rangle \text{ `*' } \langle Term \rangle$$

Then, $\rho(0) > 0 \land \rho(1) \leq 0$ is satisfiable by $\rho := \lambda x.\ 1 - x$, while $\rho(0) > 0 \land \rho(1) \leq 0 \land \rho(2) > 0$ is unsatisfiable since all functions in LIA are monotonic.

Since we rely on a non-standard semantics of satisfiability, we cannot use the theory of uninterpreted functions supported by most SMT solvers as in previous works [13]. Thus, we have to add support for this semantics in an existing SMT solver. To make the problem more tractable, we bound the size of interpretations by a user-defined constant D. However, even with this restriction,

```
scanf("%d",&x);
for (i=0;i<10;i++) {
  int t = x - i;
  if (t>0)
    printf("1");
  else
    printf("0");
}
```

(a) Buggy program.

```
scanf("%d",&x);
for (i=0;i<10;i++) {
  int t = α;
  if (t>0)
    printf("1");
  else
    printf("0");
}
```

(b) Symbolic state.

```
scanf("%d",&x);
for (i=0;i<10;i++) {
  int t = ρ(i, x);
  if (t>0)
    printf("1");
  else
    printf("0");
}
```

(c) Symb. function.

$\pi_1 \coloneqq \alpha_1 > 0 \land \alpha_2 > 0 \land \ldots$

$\pi_2 \coloneqq \alpha_1 \leq 0 \land \alpha_2 > 0 \land \ldots$

$\pi_3 \coloneqq \alpha_1 > 0 \land \alpha_2 \leq 0 \land \ldots$

(d) First-order PCs.

$\pi_1 \coloneqq \rho(0,5) > 0 \land \rho(1,5) > 0 \land \ldots$

$\pi_2 \coloneqq \rho(0,5) \leq 0 \land \rho(1,5) > 0 \land \ldots$

$\pi_3 \coloneqq \rho(0,5) \leq 0 \land \rho(1,5) \leq 0 \land \ldots$

(e) Second-order PCs.

Figure 2: Repairing program using different approaches.

an integration of second-order solving with symbolic execution remains challenging, since existing synthesis algorithms are not optimized for unsatisfiable queries [11]. This motivated us to design a new SMT-based synthesis method described in Section 4.1.

2.2 Comparing SE-ESOC with Traditional SE

Consider the function `search` in Figure 1a. This function takes an array `data`, a value `len` representing its length, a pointer to a predicate function `pred`, and returns the index of the first element of the array that satisfies the predicate.

In traditional symbolic execution, numeric inputs are replaced with logical variable as shown for the elements $\alpha_1, \alpha_2, \alpha_3$ of the array in Figure 1b. Assume that the predicate `pred` is a function `pos` that checks if a given value is positive. In this context, symbolic execution explores four paths as shown in the table in Figure 1b, in which the path conditions are constraints over the variables $\alpha_1, \alpha_2, \alpha_3$. The corresponding test inputs are concrete values of the elements of the array: $\{1,0,0\}$, $\{0,1,0\}$, $\{0,0,1\}$ and $\{0,0,0\}$.

In contrast to traditional symbolic execution, SE-ESOC enables us to explore possible program executions depending on the definition of the predicate `pred`. Assume that `pred` is represented by a variable ρ, for which the language of interpretations is as follows:

$\langle \mathit{BoolTerm} \rangle$ ::= $\langle \mathit{Term} \rangle$ ‘>’ $\langle \mathit{Term} \rangle$ | $\langle \mathit{Term} \rangle$ ‘>=’ $\langle \mathit{Term} \rangle$
 | $\langle \mathit{Term} \rangle$ ‘=’ $\langle \mathit{Term} \rangle$ | ‘`true`’ | ‘`false`’

where `Term` is defined in Example 2.1. Then, the path conditions are constraints over ρ as shown in the table in Figure 1c. The corresponding test inputs are interpretations of ρ: $\lambda x.\, \mathit{true}$, $\lambda x.\, x > 0$, $\lambda x.\, x > 1$ and $\lambda x.\, \mathit{false}$.

Note that it is possible to combine first-order and second-order symbolic variables in the same symbolic execution session by executing `search((int[])`$\{\alpha_1, \alpha_2, \alpha_3\}, 3, \rho)$. Then, the synthesized predicates ρ will be parameterized by the variables $\alpha_1, \alpha_2, \alpha_3$.

2.3 Application to Program Repair

The goal of program repair is to modify a buggy program to eliminate the observable failures. Its important subtask is to fill a hole in the program (e.g. replace a buggy statement) to enable the program to satisfy the requirements (e.g. to pass the tests). We review existing approaches to solve this subtask relying on traditional SE, and show how SE-ESOC addresses their limitations.

```
void main(int argc, char *argv[]) {
  int a = atoi(argv[1]);
  printf("%d\n", 16 / a);
}
```

(a) Program P using `atoi`.

$P(\text{"4"}) \rightarrow \text{"4"}$

$P(\text{"16"}) \rightarrow \text{"1"}$

(b) Tests.

```
int accumulation(char *arr) {
  int acc, i;
  for(i = 0; i < strlen(arr); i++)
    acc = ρ(acc, arr[i]);
  return acc;
}
```

(c) Sketch of `atoi`.

$\rho \coloneqq \lambda xy.$

$10x + y - 48$

(d) Model.

Figure 3: Library modelling from usage context.

Consider a program P in Figure 2a that reads a number, performs 10 loops iterations and, at each iteration, prints “0” or “1” depending on the sign of the variable `t`. For instance, for the input “5”, it prints “1111100000”. Assume that the correct output should be “1111111000”, and our goal is to repair the program by replacing `x - i` with an expression from LIA (defined in Example 2.1) that would enable the program to pass the test (e.g. `x - i + 2`).

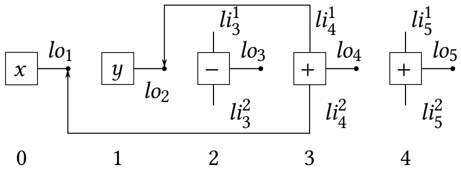
Semantics-based repair approaches [26, 28] infer a specification using symbolic execution, and synthesize a patch based on this specification. First, they replace the identified buggy expression with a symbolic variable α as shown in Figure 2b. Then, they symbolically execute the program with the input “5" and infer path conditions $\pi_1, \pi_2, \ldots, \pi_{1024}$ shown in Figure 2d. Finally, a patch is synthesized by solving the following second-order formula:

$$\exists e \in \mathit{Term}.\ (\bigvee_i \pi_i[\alpha \mapsto e]) \land \mathit{stdout} = \text{“1111111000"}$$

where stdout is a variable that captures the standard output of the application, $\pi_i[\alpha \mapsto e]$ is a formula obtained from π_i by substituting α with the term e. Such techniques suffer from the path explosion problem. For instance, there are 10 loop iterations and therefore the algorithm has to explore 1024 paths, as shown in Figure 2d.

We now demonstrate how SE-ESOC can be used to address the aforementioned limitation of previous techniques. Instead of using first-order variables α to infer synthesis specification, we replace the buggy statement with a symbolic function ρ as shown in Figure 2c. Then, SE-ESOC is applied to directly synthesize a patch by finding an interpretation of ρ that satisfies a test-passing path. The key benefit of this approach is that it substantially reduces the number of explored paths. For the described example, it will explore at most 20 execution paths as shown in Figure 2e, and the rest of the paths are infeasible, which can be non-constructively proven as follows.

Proof. There are totally 1024 possible execution paths. Among them, 20 paths consist of clauses $\rho(1,5) > 0, \ldots, \rho(i,5) > 0, \rho(i+1,5) \leq 0, \ldots, \rho(10,5) \leq 0$ or $\rho(1,5) \leq 0, \ldots, \rho(i,5) \leq 0, \rho(i+1,5) > 0, \ldots, \rho(10,5) > 0$ for some i, that is ρ changes its sign once with the increase of its first argument. Meanwhile, the other 1004 paths contain the clauses $\rho(l,5) > 0, \rho(n,5) \leq 0, \rho(m,5) > 0$ or the clauses $\rho(l,5) \leq 0, \rho(n,5) > 0, \rho(m,5) \leq 0$ for some $l < n < m$, that it ρ changes its sign at least twice with the increase of its first argument. All these path conditions are unsatisfiable since for any $\rho \in \mathit{Term}$, $\lambda x.\rho(x,5)$ is monotonic. □



$lo_1, lo_2 = 0, 1$

$lo_3, lo_4 = 2, 3$

$li_4^2, li_4^1 = 0, 1$

(a) Circuit of term $x + y$ (below are locations). **(b) L-variables.**

Figure 4: Encoding via integer location variables.

Note that monotonic functions in this example are given for clarity. Our approach does not rely on monotonicity, and is effective in more general cases as shown experimentally in Section 5.2.

SE-ESOC enables a reduction of the number of explored paths since it takes the language of symbolic function interpretations into account, i.e. it is *syntax-guided.* The reduction of the number of explored paths has important implications, since it might increase the efficiency of program repair or increase the probability of finding a patch when repairing programs with loops.

2.4 Application to Library Modelling

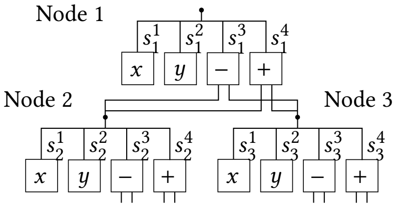
Real-world applications rely on libraries and frameworks, which complicates analysis of these applications. We consider the scenario when a library used in the analyzed application is unavailable (e.g., proprietary) as in [2, 6]. SE-ESOC can be applied to synthesize an approximate model of the library from the usage context, i.e. by analyzing how the library should behave to satisfy certain properties of the application (e.g. pass given tests). Then, this model can be used to perform symbolic analysis of the application.

Consider a program P in Figure 3a. This program uses a function `atoi` from the standard library to parse the command-line argument. Assume that the standard library is not available, then it is impossible to symbolically analyze the program, since `atoi` affects the input-output relationship. SE-ESOC can address this problem by synthesizing a model of the function `atoi` from the usage context using existing tests given in Figure 3b.

To synthesize a model, it is sufficient to replace the function call with a second-order functional variable and find an interpretation as explained in Section 2.3. However, this approach scales to only relatively simple models. To make the technique more practical, we complement it with sketches of unknown functions. For example, since `atoi` takes an array and returns a number, we provide a generic accumulator sketch that iterates through the array and applies a symbolic function ρ at each iteration (Figure 3c). Then, SE-ESOC can synthesize an interpretation of this function (the body of the loop) that is sufficient to pass the tests, e.g. the function in Figure 3d (48 is the ASCII code of ’0’). This model can be used to perform symbolic analysis of the application, e.g. to generate a new input “0" that triggers a division-by-zero error.

3 BACKGROUND

Jha et al. [15] proposed to semantically encode a space of terms using linear integer arithmetic constraints. In this approach, terms are represented as circuits built from user-provided components such as addition, subtraction, etc. Connections between components are captured using integer *location variables.*



$s_1^1 \mapsto x$

$s_1^3 \wedge s_2^1 \wedge s_3^2 \mapsto x - y$

$s_1^4 \wedge s_2^1 \mapsto \{x + T\}_{T \in Term}$

(a) Tree with “abstract” nodes. **(b) Selectors to terms.**

Figure 5: Encoding via propositional selector variables.

Assume that $\texttt{out}_i$ is the output of i-th component, lo_i is the location of the output of the i-th component, $\texttt{in}_i^j$ is the j-th input if the i-th component, li_i^j is the locations of the j-th input if the i-th component, C is the number of components, N_i is the number of inputs of the i-th component, F_j is the semantics if the j-th component (e.g. $\lambda xy.\ x + y$ for addition). The set of well-formed terms is encoded using $\phi_{wpf} := \phi_{range} \wedge \phi_{cons} \wedge \phi_{acyc}$, such that

$$\phi_{range} := \bigwedge_{i \in [1,C]} \Big(0 \leq lo_i < C \ \ \wedge \bigwedge_{j \in [1,N_i]} 0 \leq li_i^j < C \Big)$$

$$\phi_{cons} := \bigwedge_{i,j \in [1,C], i \neq j} lo_i \neq lo_j$$

$$\phi_{acyc} := \bigwedge_{i \in [1,C], j \in [1,N_i]} lo_i > li_i^j$$

where range constraints ϕ_{range} allocate inputs and outputs within a legal range, consistency constraints ϕ_{cons} ensure that all outputs have unique locations, and acyclicity constraints ϕ_{acyc} forbid loops. Besides, connection constraints ϕ_{conn} bind location variables and connections between components, and semantic constraints ϕ_{sem} define the relation between components’ inputs and outputs:

$$\phi_{conn} := \bigwedge_{i,j \in [1,C], k \in [1..N_i]} lo_i = li_j^k \Rightarrow \texttt{out}_i = \texttt{in}_j^k$$

$$\phi_{sem} := \bigwedge_{i \in [1,C]} \texttt{out}_i = F_i(\texttt{in}_i^1, \texttt{in}_i^2, ..., \texttt{in}_i^{N_i})$$

A term is constructed from an assignment of locations variables that satisfies $\phi_{wpf} \wedge \phi_{conn} \wedge \phi_{sem}$ using a function `Lval2Term`. This function connects inputs and outputs of components that have the same location. For example, $\lambda xy.\ x + y$ is constructed from the assignment in Figure 4b (as in the circuit in Figure 4a).

4 METHODOLOGY

In this section, we first formally describe second-order constraints used in our approach and a method of solving these constraints. Secondly, we demonstrate how second-order solving is integrated with symbolic execution, and describe implemented constraint optimizations. Finally, we show how the resulting technique can be applied for program repair and environment modelling.

4.1 Second-order Solving

As is usual in SMT literature [5], we consider formulas and terms built from predicate and function symbols (e.g. “+", “–", “>") from a given signature Σ. We denote the set of all such formulas and terms as L_Σ. We also consider a background theory $\mathcal{T}$ that fixes the interpretations of the symbols in Σ. In this work, we are interested in an extended set of formulas and terms $L_{\Sigma \cup \mathrm{P}}$ constructed from the symbols in Σ and an additional set of predicate and function symbols $\mathrm{P} := \{\rho_1, ..., \rho_n\}$ without interpretations in $\mathcal{T}$, that we refer to as *second-order variables* (or symbolic functions).

For a formula ϕ over a second-order variable ρ and a term $t \in L_\Sigma$ with a designated set of variables $x_1, ..., x_n$, we say that a first-order formula $\phi[\rho \mapsto t]$ is a *substitution* of ρ with t, if it is obtained by replacing each sub-term $\rho(t_1, ..., t_n)$ of ϕ (for some terms t_1, ..., t_n) with a term computed as the result of beta-reduction of the lambda expression $(\lambda x_1 ... x_n.\ t)\ t_1\ ...\ t_n$. For instance, let ϕ be $\rho(a, 1) > 0$ and t be $x_1 + x_2$, then $\phi[\rho \mapsto t]$ is defined as $a + 1 > 0$.

Definition 4.1 (Second-order satisfiability). Let $\phi \in L_{\Sigma \cup \mathrm{P}}$ be a second-order formula, $\mathcal{L} : \mathrm{P} \rightarrow 2^{L_\Sigma}$ — a mapping from second-order variables to sub-languages of L_Σ — be domains of interpretations. Then, ϕ is satisfiable iff, for some terms $t_1 \in \mathcal{L}(\rho_1), ..., t_n \in \mathcal{L}(\rho_n)$, the first-order formula $\phi[\rho_1 \mapsto t_1, ..., \rho_n \mapsto t_n]$ is satisfiable w.r.t. $\mathcal{T}$.

The key part of this definition is the domains of interpretations $\mathcal{L}$ that are sub-languages of L_Σ for each second-order variable. In our approach, the sub-languages are either provided by the user or by a tool/algorithm that relies on SE-ESOC. Particularly, a sub-language is defined as a pair (G, D) of a context-free grammar G with the symbols from Σ as terminals (as in SyGuS format [3]) and an integer value D that describes the maximum depth of considered terms (i.e. the maximum number of nodes in a path from the root of a term to its leaf).

Similar to the prior approach described in Section 3, we rely on a library of components to encode a space of terms from a given language of interpretations. Note that for a synthesis problem with a language defined via a pair (G, D), it is straightforward to encode it as a component-based synthesis problem by considering each grammar rule $N \rightarrow F(N_1, ..., N_n)$ (for non-terminals $N, N_1, ..., N_n$) of G as a component F with inputs $N_1, ..., N_n$. Thus, without the loss of generality, we assume later that, instead of a grammar G, our language is defined through a set of components $F_1, ..., F_C$.

One way to implement a solver for the considered kind of second-order formulas is to encode them through first-order formulas using e.g. the approach described in Section 3. However, this approach relies on linear integer arithmetic to encode a space of terms, which results in inefficient proofs of unsatisfiability. On the other side, SE-ESOC critically depends on the performance of unsatisfiable queries to avoid infeasible paths, as shown in Section 2.3.

In order to optimize unsatisfiable queries, we introduce an new encoding of second-order formulas through propositional *selector variables* instead of integer location variables. Intuitively, this increases the effectiveness of conflict clause learning in CDCL-based [33] SMT solvers [9, 12] and therefore significantly improves the performance on unsatisfiable queries, which is shown experimentally in Section 5.

The key idea of the introduced *propositional synthesis encoding* is to represent the space of terms constructed from a given library of components via a tree with "abstract" nodes as shown in Figure 5a. Specifically, each intermediate node of the tree has as many subnodes as the maximal number of inputs of a component in a given component library. Each leaf of the tree corresponds to components that have no inputs. The semantics of each node is defined through the semantics of a component activated via selector variables.

Assume that s_i^j is the j-th selector of the i-th node, $\mathtt{out}_i$ is the output of i-th node, C is the number of components, F_j is the semantics if the j-th component (e.g. $\lambda xy.\ x + y$ for addition). For each node i with subnodes $i_1, i_2, ..., i_k$, a set of terms is encoded as $\psi_i := \psi_{node} \wedge \psi_{choice}$, such that

$$\psi_{node} := \bigwedge_{j \in [1,C]} s_i^j \Rightarrow \mathtt{out}_i = F_j(\mathtt{out}_{i_1}, \mathtt{out}_{i_2}, ..., \mathtt{out}_{i_k})$$

$$\psi_{choice} := exactlyOne(s_i^1, s_i^2, ..., s_i^C)$$

In this encoding, ψ_{node} describes how the output value of a node depends on the values of its subnodes, ψ_{choice} ensures that exactly one of the components is selected inside each node (the cardinality constraint *exactlyOne* is implemented using sorting networks [1]). A term is constructed from an assignment of selector variables using a function `Sval2Term` that at each node picks a component that is activated by the corresponding selector variable as shown in Figure 5b. For instance, the term $x - y$ is constructed by enabling the component $-$ of the node 1 (via the selector s_1^3), the component x of the node 2 (via the selector s_2^1), and the component y of the node 2 (via the selector s_3^2).

Using the above encoding, a second-order constraint solver can be implemented on top of a first-order solver. Specifically, for a given formula ϕ over a second-order variable ρ, we define the procedure `Encode` as follows:

- each occurrence of a subterm $\rho(t_1, ..., t_n)$ in ϕ (for some terms $t_1, ... t_n$) is assigned a unique index i;
- for each occurrence of a subterm $\rho(t_1, ..., t_n)$ in ϕ with index i (for some terms $t_1, ... t_n$), the formula ϕ is conjoined with $\psi_1^i \wedge ... \wedge \psi_m^i$, where m is the number of tree nodes and the terms $t_1, ..., t_n$ are treated as components without inputs;
- each occurrence of a subterm $\rho(t_1, ..., t_n)$ in ϕ with index i (for some terms $t_1, ... t_n$) is replaced with the variable $\mathtt{out}_1^i$ representing the root of the i-th tree in the encoding.

Using this procedure, a second-order formula is transformed into a first-order formula over selector variables, which can be solved using an off-the-shelf SMT solver. From any satiafying assignment of the selector variables, an interpretation of ρ that satisfies ϕ can be reconstructed using `Sval2Term`, as stated formally below:

Proposition 4.2. *For any assignment of selector variables $S := \{s_1 \mapsto b_1, ..., s_n \mapsto b_n\}$ that satisfies $\phi' :=$ `Encode`(ϕ), the assignment $\{\rho \mapsto$ `Sval2Term`$(S)\}$ satisfies ϕ.*

4.2 Extension of Symbolic Execution

Algorithm 1 describes the overall workflow of symbolic execution with our extension. The function `exeSymbolic` takes an instruction pointer, a program state (a mapping from variables to concrete value or logical terms) and a path condition, performs symbolic execution of the corresponding instruction, and recursively continues execution. For an assignment `v := e`, this function updates the state by replacing the value of `v` with `e` evaluated in the context S (denoted as $[\![e]\!]_S$). Then, the execution continues from the next instruction. For a conditional `if e then` $\mathtt{C}_1$ `else` $\mathtt{C}_2$, this function checks whether the if-condition is consistent with the current path

ALGORITHM 1: Extension of symbolic execution

```
Procedure exeSymbolic(instruction pointer IP, program state S, path condition PC)
    I := getInstruction(IP);
    switch I do
        case Assignment v := e
            $S' := S[v \mapsto [\![e]\!]_S]$;
            IP' = increment(IP);
            exeSymbolic(IP', S', PC);
        end
        case Conditional if e then C1 else C2
            $\phi := PC \wedge [\![e]\!]_S$;
            if isSatisfiable($\phi$) then
                IP' = getPointer(C1);
                exeSymbolic(IP', S, $PC \wedge [\![e]\!]_S$);
            end
            ... // check the other branch
        end
        otherwise
            ... // handle other instructions
        end
    endsw
Procedure isSatisfiable(formula $\phi$)
    $\phi'$ := Encode(IP);
    return underlyingSolver($\phi'$);
```

condition and whether the negation of the if-condition is consistent with the path condition (the later case is omitted). If the constraint is satisfiable, the algorithm continues execution of the corresponding branch with an augmented path condition.

Compared with traditional SE, SE-ESOC modifies the function `isSatiafiable` highlighted in Algorithm 1. Specifically, it adds support for second-order constraints by implementing the approach described in Section 4.1. The function `isSatiafiable` encodes each query ϕ using `Encode` before passing it to the underlying SMT solver. Later, a model of ϕ can be reconstructed from the model computed by the underlying SMT solver using `Sval2Term`.

Definition 4.3 (Second-order infeasible paths). Let P be a program taking a function as an input, ρ be a second-order variable, and L be a sub-language of L_Σ. Then, a path along which SE-ESOC computes a path condition π by executing P with the symbolic input ρ is *infeasible* iff the second-order formula π is unsatisfiable w.r.t. the domain of interpretations $\{\rho \mapsto L\}$.

This definition of infeasible path depends on the syntax of the language of interpretations L. This property is crucial for mitigating the path explosion as will be shown in Section 5.2.

4.3 Program Repair and Library Modelling

SE-ESOC can be used, among others, to synthesize patches for program defects or models for unavailable libraries. Similarly to prior works [26, 28], the workflow of both these applications consists of three steps: injecting second-order symbolic variables, performing specification inference, and synthesizing patches/models.

Symbolic variable injection. In the context of program repair, suspicious program statements are substituted with applications of symbolic functions to local program variables. Suspicious program statements can be identified using, for instance, statistical fault localization [16]. For each of the identified suspicious statements, we iteratively apply the following transformation schemas parameterized with second-order variable ρ:

- changing the right-hand side of an assignment:

  $$\texttt{x := E;} \quad \mapsto \quad \texttt{x := } \rho(\texttt{v}_1,\ldots,\texttt{v}_n)\texttt{;}$$

- changing a condition:

  $$\texttt{if (E) \{...\}} \quad \mapsto \quad \texttt{if (}\rho(\texttt{v}_1,\ldots,\texttt{v}_n)\texttt{) \{...\}}$$

- adding an if-guard:

  $$\texttt{S;} \quad \mapsto \quad \texttt{if (}\rho(\texttt{v}_1,\ldots,\texttt{v}_n)\texttt{) S;}$$

where `S` is a statement, `E` is an expression, and $\texttt{v}_1,\ldots\texttt{v}_n$ are visible program variables. Specifically, we adopted a recently proposed heuristics [38] to select up to 10 local program variables whose definitions are the closest to the considered suspicious location.

In the context of library modelling, symbolic functions are used to replace calls of unknown library functions. Specifically, we replace all calls of an unknown function either with a fresh second-order variable ρ or a generic sketch parameterized with ρ, depending on the signature of this function:

- `func` has integer inputs and an integer output:

  $$\texttt{func(E}_1\texttt{,...,E}_n\texttt{)} \quad \mapsto \quad \rho(\texttt{E}_1\texttt{,...,E}_n)\texttt{;}$$

- `func` has integer and array inputs and an integer output:

  $$\texttt{func(E}_1\texttt{,...,E}_n\texttt{)} \quad \mapsto \quad \texttt{accumulation(E}_1\texttt{,...,E}_n\texttt{);}$$

- `func` has integer and array inputs and an array output:

  $$\texttt{func(E}_1\texttt{,...,E}_n\texttt{)} \quad \mapsto \quad \texttt{transformation(E}_1\texttt{,...,E}_n\texttt{);}$$

where $\texttt{E}_1,\ldots\texttt{E}_n$ are expression, and the sketches `accumulation` and `transformation` are defined in the following way:

```
int accumulation(char *arr, ...) {
  int acc, i;
  for(i = 0; i < strlen(arr); i++)
    acc = ρ(acc, arr[i], ...);
  return acc;
}
void transformation(char *arr, char *out, ...) {
  int i;
  for(i = 0; i < strlen(arr); i++)
    out[i] = ρ(acc, arr[i], ...);
}
```

In principle, an arbitrary C function parameterized with first-order and second-order variables can be used as a sketch.

Specification inference. The purpose of specification inference is to collect constraints over the injected second-order variables such that any interpretation that satisfies these constraint would meet our requirements. Specifically, our goal is to find interpretations of the symbolic functions that would enable the program to pass given tests. Assume that P is the original program, and P' is a program obtained by injecting a second-order variable ρ into P. Assume also that $\{in_i, \phi_i\}_{i\in[0,n]}$ is a set of tests, where in_i is the input of the i-th test and ϕ_i is the test assertion. For each test i, we execute the program P' using SE-ESOC with the concrete input in_i and obtain a set of path conditions $\pi_1^i, \ldots, \pi_k^i$, which are constraints over the variable ρ. Then, the specification is defined as follows:

$$\bigwedge_{i=0}^{n}\Big(\bigvee_{j=0}^{k}\pi_j^i\Big) \wedge \phi_i \qquad (1)$$

The above second-order formula captures the property that for each test with index i there should be at least one path π_j^i along which the test assertion ϕ_i holds.

Patch/model synthesis. To synthesize interpretations of symbolic functions that satisfies the formula (1), we apply the second-order constraint solving method described in Section 4.1. For program repair, these interpretations constitute patches, and for library modelling, these interpretations constitute models of the unknown library functions.

In the context of program repair (and also library modeling), tests are typically insuffient to guarantee the correctness of patches, which causes the test-overfitting problem [34]. The proposed approach is orthogonal to the problem of test-overfitting, however it is straightforward to integrate it with existing techniques for alleviating overfitting such as synthesizing minimal change via maximum satiafiability [25] or applying anti-pattens [35] or applying correctness assertions — by conjoining the encoding with additional constraints over selector variables.

4.4 Implementation

We implemented SE-ESOC as an extension[1] of KLEE [8], a widely used symbolic execution engine for C programs. Firstly, we extended KLEE to support second-order variables and implemented the generation of second-order path conditions in the symbolic execution runtime. Secondly, we implemented the encodings described in Section 3 and Section 4.1 on top of the underlying SMT solver.

KLEE provides an intrinsic function `klee_make_symbolic` for injecting symbolic variables. For example, the following call marks the memory corresponding to the variable `foo` as symbolic.

```
klee_make_symbolic(&foo, sizeof(foo), "foo");
```

To let users introduce second-order variables, we added an intrinsic function `klee_apply_symbolic`. This function applies a symbolic function to program expressions. For instance, the following code can be used to inject a call of ρ in Example 2c:

```
int t = klee_apply_symbolic("rho", 2, (int[]){i, x});
```

where `"rho"` is the name of the second-order variable, 2 is the number of arguments, and `(int[]){i, x}` is the array of arguments.

5 EVALUATION

This evaluation addresses the following research questions:

(RQ1) Does SE-ESOC reduce the number of explored paths compared with program repair techniques relying in first-order symbolic execution? Does it improve the effectiveness of program repair?

(RQ2) Can SE-ESOC synthesize library models from the usage context that improve symbolic execution-based test generation?

(RQ3) Does the introduced second-order solving method based on propositional encoding improve SE-ESOC performance compared to previous encodings?

5.1 Experimental Setup

To address the research questions, we conducted experiments with programs from GNU Coreutils[2], GNU Findutils[3] and GNU Grep[4], mature and widely-used implementations of UNIX utilities included in the majority of Linux distributions, that have been also employed in previous symbolic execution studies [8, 29].

Table 1: Subjects of DBGBench dataset

Program	Description	Defects
find	Search for files in directory hierarchy	14
grep	Search for lines containing match to specified pattern	13

Table 2: Modelled library functions

Function	Description	Sketch
read	Read bytes from file with specified descriptor	Transformation
write	Write bytes to file with specified descriptor	Transformation
stat	Return information about file	-
seek	Change read/write position of file descriptor	-
chmod	Change permissions of file	-
chown	Change ownership of file	-
malloc	Allocate block of memory	-
strtol	Convert string to int	Accumulation
strlen	Return string length	Accumulation
strcmp	Compare two strings	Accumulation

Table 3: Usage of library functions.

Program	Description	Used functions
base64	Compute Base64	read, write
cat	Concatenate files	strcmp, stat, read, malloc
cp	Copy file	read, write, chown, chmod, stat, malloc
head	Print beginning of file	read, write, seek, malloc, strcmp
ls	List files in directory	write, stat, malloc, strcmp, strlen
pr	Convert text files	strtol
pwd	Print current directory	malloc, stat
sort	Sort lines	write, malloc, strcmp
test	Evaluate expression	stat
wc	Count words	strtol, read

In the context of program repair, we used a recently introduced DBGBench dataset [7]. DBGBench is a collection of 27 bugs from GNU Findutils and GNU Grep shown in Table 1. We chose this benchmark because it contains real error in widely used software, and because this dataset was designed for evaluating, among others, program repair techniques.

To evaluate our library modelling method, we first selected 12 standard C library functions that are frequently used by C programs, and that have been studied in related works [31]. These functions include file systems related functions: `read`, `write`, `seek`, `chmod`, `chown` and `stat`, `malloc`, and string functions: `strtol`, `strlen`, `strcmp`. To synthesize models of these functions, we used predefined sketches specified in Table 2. Note that some of the functions (e.g. memory allocation) depend on OS kernel, that cannot be modelled using our method. For these function, we modelled only the part of functionality that does not rely on OS kernel behavior, such as computing the real allocation size for `malloc`.

To perform modelling, we selected programs from GNU Coreutils that rely on these functions. We run provided tests for all 89 programs from GNU Coreutils 6.11 with ltrace[5] in order to identify which of the selected library functions are used by these programs.

[1] Our second-order KLEE extension: http://angelix.io/second-order.html

[2] GNU Coreutils: https://www.gnu.org/software/coreutils/

[3] GNU Findutils: https://www.gnu.org/software/findutils/

[4] GNU Grep: https://www.gnu.org/software/grep/

[5] ltrace: http://www.ltrace.org/

⟨*Bool*⟩ ::= ⟨*Int*⟩ ‘<’ ⟨*Int*⟩ | ⟨*Int*⟩ ‘<=’ ⟨*Int*⟩ | ⟨*Int*⟩ ‘==’ ⟨*Int*⟩
| ⟨*Bool*⟩ ‘||’ ⟨*Bool*⟩ | ⟨*Bool*⟩ ‘&&’ ⟨*Bool*⟩ | ‘!’ ⟨*Bool*⟩

⟨*Int*⟩ ::= ⟨*Var*⟩ | ⟨*Constant*⟩
| ⟨*Int*⟩ ‘+’ ⟨*Int*⟩ | ⟨*Int*⟩ ‘-’ ⟨*Int*⟩ | ‘-’ ⟨*Int*⟩

(a) Boolean functions.

⟨*Term*⟩ ::= ⟨*Var*⟩ | ⟨*Constant*⟩
| ⟨*Term*⟩ ‘+’ ⟨*Term*⟩ | ⟨*Term*⟩ ‘-’ ⟨*Term*⟩ | ‘-’ ⟨*Term*⟩
| ⟨*Bool*⟩ ‘?’ ⟨*Term*⟩ ‘:’ ⟨*Term*⟩

(b) Integer functions.

Figure 6: Language of interpretations (search space).

Then, we chose 10 programs from GNU Coreutils that rely on different combinations of library functions as shown in Table 3.

To examine the effect of second-order constraints on the path explosion, we compared our approach with Angelix [26], a state-of-the-art program repair system that relies on first-order symbolic execution. Specifically, we used the following three configurations:

- **FO** Angelix that relies on first-order symbolic execution.
- **SO/CBS** SE-ESOC that uses the encoding by Jha et al. [15] for second-order constraint solving.
- **SO/PSE** SE-ESOC that uses the introduced propositional encoding for second-order constraint solving.

Since the proposed library modelling method is the first that synthesizes library models from the usage context, we compared it against the baseline approach (treating all outputs of unknown library functions as symbolic) and manually-written models. Specifically, we used the following configurations:

- **Baseline** Treating outputs of unknown functions as symbolic.
- **SE-ESOC** Using models synthesized by SE-ESOC.
- **Manual** Using models provided by KLEE.

We do not experimentally compare our approach with the modelling technique by Qi et al. [31], since their approach executes the library to collect output values, while our goal is to synthesize models when the library is not available.

We conducted all experiments on an Intel® Core™ i7-2600 CPU 3.40GHz machine running Ubuntu 14.04 with 8GB of memory.

5.2 Program Repair

To investigate the effect of second-order constraints on path explosion, we compared SE-ESOC configurations with Angelix (FO). Particularly, we executed repair algorithms described in Section 4.3 on the subjects of DBGBench, using developer-provided tests as the correctness criteria for patches. For each suspicious program location, we bounded[6] the number of explored paths to 400, and used a 10 minutes time limit. SE-ESOC configurations used the languages of interpretations given in Figure 6a and Figure 6b. The same languages were used by Angelix (FO) synthesizer. Besides, we specified the depth bound $D := 3$ for the synthesized functions.

Table 4 summarizes the results of our experiments. The column “Subject” lists subject program and their versions (commit hashes). The columns “Patch” show if a patch was generated by each configuration; we present only versions for which at least one configuration generated a repair. In these columns, “Correct” indicates that the generated patch is syntactically equivalent to the developer patch, otherwise the patch is classified as “Plausible”.

In order to investigate the cause of failures of some configurations to find a patch, we collected additional statistics of symbolic execution sessions. For each configuration, we collected data for the session in which the program is executed symbolically with the failing test and a symbolic variable is installed in the fix location. Specifically, the columns “Paths” in Table 4 denote how many paths were explored by each configuration during this symbolic execution session. The columns “Time” show the time taken by each configuration to explore these paths.

Overall, the configurations based on second-order constraints generated all the patches generated by the approach based on first-order constraints, and SO/PSE also found three additional patches for find.091557f6, find.dbcb10e9 and grep.54d55bba. For each of these three cases, FO reached the limit of 400 paths during exploration, while SO/PSE reduced the number of explored paths to 14-26, which led to the successful generation of patches. However, SO/PSE required slightly more time on average for path exploration compared with FO (1m 24s for SO/PSE, and 1m 5s for FO).

In all three cases for which SO/PSE exclusively generated patches, the modified statements are executed multiple times by the failing tests. To explain how the reduction of explored paths is achieved in this case, we consider the experiments with the bug find.091557f6 in greater details. This bug in `find` utility is caused by a wrong handling of symbolic link loops when searching for files in a directory. One of the possible correct fixes is to add the disjunct `ent->fts_errno == ELOOP` to the condition shown in Figure 7a.

To synthesize a patch, Angelix (FO) replaces the buggy condition with a first-order variable, and performs symbolic execution to infer synthesis specification. The condition is evaluated multiple time during an execution of the failing test, and Angelix (FO) fails to infer sufficient specification to synthesize a patch due to the path explosion (since it terminates after reaching the limit of 400 paths).

Contrary to Angelix, SE-ESOC injects a second-order variable ρ applied to local program variables as shown in Figure 7a. It also associates the language in Figure 6a and the bound $D := 3$ with ρ, that effectively define the search space of patches. This enables SE-ESOC to take advantage of the stronger notion of infeasibility (Definition 4.1) to prune irrelevant paths. Specifically, it determines that only 16 execution paths are feasible w.r.t. the considered language of interpretations, and synthesizes the interpretations $\rho_1, ..., \rho_{16}$ in Figure 7b corresponding to the 16 feasible paths.

In order to prune infeasible paths, SE-ESOC has to solve more complex constraints than FO, which results in additional performance overhead. However, the execution time of SE-ESOC is less dependent on the bounds of symbolic execution. To demonstrate this, we executed FO and SE/PSE with the example in Figure 7a, ranging the number of explored paths (`--max-forks` KLEE options) from 100 to 2000. The results of our experiment are presented in Figure 7c. As can be seen, the time taken by FO increases with the increase of the path bound, while the time taken by SE/PSE does not depend on the path bound, since it only has to explore 16 paths.

Overall, SE-ESOC helps alleviate path explosion when repaired expressions are executed *multiple times* during program execution.

[6] It is common for synthesis-based program repair techniques to rely on path bounds. For instance, an enumerative approach SPR [21] uses the bound of 11 paths.

Table 4: Program repair results.

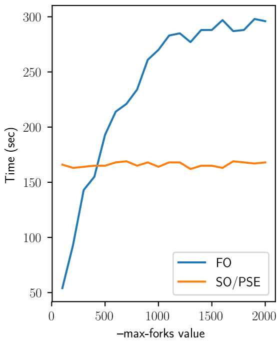
Subject	Patch			Paths			Time			SAT/UNSAT	
	FO	SO/CBS	SO/PSE	FO	SO/CBS	SO/PSE	FO	SO/CBS	SO/PSE	SO/CBS	SO/PSE
find.091557f6	-	-	Correct	400	0	16	2m 35s	Timeout	2m 29s	3.5s/Timeout	2.2s/3.8s
find.24bf33c0	Plausible	Plausible	Plausible	2	2	2	2s	3s	3s	1.3/-	0.5s/-
find.24e2271e	Correct	Correct	Correct	24	24	24	39s	1m 1s	1m 43s	2.1s/-	2.2s/-
find.07b941b1	Plausible	Plausible	Plausible	11	11	11	29s	41s	31s	0.7s/-	0.5s/-
find.e6680237	Correct	Correct	Correct	46	46	46	56s	2m 41s	2m 23s	1.2s/-	1.1s/-
find.dbcb10e9	-	-	Correct	400	0	14	3m 31s	Timeout	3m 1s	3.0s/Timeout	1.7s/5.8s
find.e1d0a991	Plausible	Plausible	Plausible	4	4	4	5s	5s	5s	2.2s/-	1.9s/-
grep.55cf7b6a	Correct	Correct	Correct	2	2	2	3s	3s	3s	0.8s/-	0.4s/-
grep.3220317a	Plausible	Plausible	Plausible	27	27	27	41s	1m 5s	1m 11s	1.2s/-	1.1s/-
grep.db9d6340	Plausible	Plausible	Plausible	2	2	2	2s	3s	2s	1.8s/-	0.6s/-
grep.c96b0f2c	Plausible	Plausible	Plausible	41	41	41	1m 19s	1m 59s	2m 11s	2.2s/-	2.3s/-
grep.5fa8c7c9	Correct	Correct	Correct	34	34	34	55s	2m 43s	1m 35s	4.9s/-	3.8s/-
grep.54d55bba	-	-	Plausible	400	0	26	2m 42s	Timeout	2m 59s	4.1s/Timeout	2.4s/5.2s
Overall	10	10	13	107.2	14.8	19.2	1m 5s	1m 21s	1m 24s	2.2s/-	1.6s/4.9s

```
...
else if (ent->fts_info == FTS_DC)
  {
    issue_loop_warning(ent);
    error_severity(1);
    return;
  }
// ρ(ent->fts_info, ent->fts_errno, prev_depth)
else if (ent->fts_info == FTS_SLNONE)
  {
    if (symlink_loop(ent->fts_accpath))
      {
        error(0, ELOOP, ent->fts_path);
        error_severity(1);
        return;
...
```

(a) Buggy condition in find.091557f6.

```
ρ1  :=(4 <= ent->fts_info)
ρ2  :=!(ent->fts_errno == prev_depth)
ρ3  :=((4 < ent->fts_info) && (prev_depth <= ent->fts_errno))
ρ4  :=!(0 == ent->fts_errno)
ρ5  :=((ent->fts_info == ent->fts_errno) || (9 <= prev_depth))
ρ6  :=(ent->fts_info == (7 + prev_depth))
ρ7  :=((prev_depth + ent->fts_errno) == (ent->fts_info - 6))
ρ8  :=((ent->fts_errno < prev_depth) || (6 == ent->fts_info))
ρ9  :=(ent->fts_info < (4 + ent->fts_errno))
ρ10 :=(ent->fts_info <= (ent->fts_errno + 6))
ρ11 :=!(ent->fts_info == 6)
ρ12 :=(0 <= prev_depth)
ρ13 :=!(4 < ent->fts_info)
ρ14 :=!(1 <= prev_depth)
ρ15 :=((ent->fts_errno < prev_depth) || (ent->fts_info <= 1))
ρ16 :=((ent->fts_errno < 32) || (prev_depth == ent->fts_info))
```

(b) Interpretations of ρ found along feasible paths.



(c) Time and explored paths.

Figure 7: Repairing wrong handling of symbolic link loops in `find`.

5.3 Library Modelling

To evaluate the effect of library modelling on the effectiveness of symbolic analysis, we computed the coverage of test suites generated by symbolic execution using the synthesized models. First, we used the available tests of GNU Coreutils to synthesize models of the unknown library functions using SE-ESOC.

To check the efficacy of our modelling technique, we used three sets of library templates: Baseline, SE-ESOC and Manual, that are defined in Section 5.1. Using these three approaches, we generated test suites using KLEE. We used the recommended configuration for testing GNU Coreutils available on KLEE website[7], and set the time limit of a half an hour for test generation. Then, to check the efficacy of the library models, we check the efficacy of the generated test suites in the three approaches, as follows. We linked application programs that use the libraries, with the *real* libraries (not the models). Then, we computed the line coverage of the generated test suites in the programs linked against the real libraries.

Table 5 summarizes the statistics of the generated test suites. The columns "Tests" contain the number of tests generated by KLEE using different models. The columns "Coverage" depict the line coverage produced by the test suites and computed using gcov.

The coverage achieved using models synthesized by SE-ESOC is 52.17% which is greater than 41.2% coverage achieved using baseline approach. However, the automatically generated library models using SE-ESOC achieve lower coverage than the carefully crafted hand-written models which achieve around 64.51% coverage.

Table 5: Library modelling results.

Program	Baseline		SE-ESOC		Manual	
	Tests	Coverage	Tests	Coverage	Tests	Coverage
base64	53	76.19%	62	89.32%	78	91.43%
cat	64	71.55%	60	86.10%	63	88.36%
cp	51	35.33%	154	54.73%	168	67.66%
head	44	23.76%	48	29.26%	90	70.32%
ls	35	27.10%	41	36.10%	55	46.07%
pr	22	40.07%	55	58.65%	75	63.36%
pwd	29	15.25%	20	16.61%	18	20.34%
sort	59	27.23%	90	36.60%	106	46.53%
test	12	29.08%	42	42.92%	111	68.63%
wc	54	66.41%	57	71.40%	69	82.44%
Overall	42.3	41.20%	62.9	52.17%	83.3	64.51%

5.4 Second-order Solving

Since our approach relies on the notion of infeasibility (Definition 4.1) to prune explored paths, it is critical that it can efficiently handle unsatisfiable second-order queries. To investigate how existing component-based synthesis (CBS) encoding (Section 3) and the introduced propositional (PSE) encoding (Section 4.1) perform

[7] KLEE configuration for Coreutils: http://klee.github.io/tutorials/testing-coreutils/

on unsatiafiable formulas that occur in the context of symbolic execution, we collected solver statistics in Table 4. The columns "SAT/UNSAT" demonstrate the average time taken by satisfiable and unsatisfiable queries during path exploration ("-" indicates that no such queries are performed). As can be seen, both CBS and PSE demonstrated comparable performance on satisfiable formulas (on average, CBS requires 2.2s, and PSE requires 1.6s). However, CBS did not solve any unsatisfiable queries within the time limit (10 minutes), while the average time taken by PSE on unsatisfiable queries is 4.9s, which is similar to the time taken by satisfiable queries. Thus, PSE is crucial in enabling SE-ESOC to avoid infeasible paths.

6 RELATED WORK

Symbolic execution. Godefroid [13] proposed to use higher order constraints to model imprecision of symbolic execution. Specifically, this approach replaces unknown/complex instructions with uninterpreted functions and generates inputs by solving *universal* constraints over *uninterpreted* functions. The main difference of our approach is that it solves *existential* constraints over functions whose interpretations are restricted by a user-defined language, which implies different methodology and applications. First, the approach with uninterpreted functions cannot reduce the number of explored paths as shown in Section 2.3 in the context of program repair, since this reduction is achieved by restricting of the space of interpretations. Second, the approach with uninterpreted functions cannot be used for library modelling as shown in Section 2.4, since it relies on input-output function sampling that cannot be performed when the library is not available. Palikareva et al. [29] proposed to test divergences between program versions by encoding two versions in a single symbolic execution session. Our approach differs in that it encodes a potentially infinite number of program versions inside a single symbolic execution session.

Program synthesis and second-order constraint solving. From the logical point of view, program synthesis is existential second-order constraint solving [11]. Several techniques have been proposed for solving such second-order constraints. Enumerative techniques [3, 4] that explicitly generate and test individual terms cannot be applied in the context of symbolic execution because they would require checking satisfiability of path constraint for each possible expression in the search space. Reynolds et al. [32] introduced an algorithm for synthesizing programs inside an SMT solver. This approach tends to synthesize complex solutions consisting of thousands of nodes[8] that cannot be understood by humans, which makes it unsuitable for program repair. The encoding of second-order formulas proposed by Jha et al. [15] relies on linear integer arithmetic constraints, which results in inefficient proofs of unsatisfiability. Symbolic execution requires checking unsatisfiability of path constraints to avoid infeasible paths, needing efficient unsatisfiability proofs. The second-order formula encoding introduced in this work addresses the above limitations of existing techniques.

Program repair algorithms. Various program repair techniques have been recently proposed [20, 22]. The most relevant works to our approach are techniques [23, 26, 28] that employ first-order symbolic execution to infer patch synthesis specification [23, 26]. Since these techniques rely on symbolic execution, they suffer from the path explosion problem as shown in Section 2.3. SE-ESOC alleviates the path explosion, which leads to repairing more defects in programs with loops.

Specification inference and modelling of libraries. In order to enable analysis of code written against libraries, a common approach is to provide a library model, summary or specification. KLEE [8] symbolic execution engine relies on manually-written POSIX libraries models to analyze system software. Qi et al. [31] proposed to synthesize a library model for symbolic execution by using the library as an oracle for program synthesis, i.e. it executes the library to collect output values. Our approach differs in that it is designed for the case when the library is not available. Several techniques have been proposed to derive specification by analyzing the library usage context, when library is not available. Bastani et al. [6] infer data-flow library specification from the usage context. Albarghouthi et al. [2] synthesize a maximal library specification to ensure the correctness of a given program that uses this library. Our library modelling technique differs from the approach by Albarghouthi et al. in that (1) we synthesize a code fragment that captures input-output relationship of library functions (to enable generation of tests that reach certain locations), rather than a specification for proving program properties for all inputs and (2) our technique is unsound (it relies on inductive generalization without verification and imprecise symbolic execution), but more applicable (it can be used with real-world software without formal specification).

Test-equivalence analysis. Test-equivalence is a property of two modifications of a program to behave equivalently (for some definition of equivalence) when executing a given test. This property has been applied in multiple domains such as mutation testing [17, 37], program repair [24], and compiler testing [19]. The analysis technique proposed in this paper can be considered as a generalization of previously used test-equivalence analyses. Specifically, a second-order path constraint captures the set of all modifications of a given program that would drive the test execution along the corresponding path (as in Section 2.3). Thus, it defines path-based test-equivalence relation on the space of program modifications.

7 CONCLUSION

We have proposed symbolic execution with existential second-order constraints. We developed a method for second-order constraint solving that provides efficient proofs of unsatisfiability, which is needed for efficient symbolic execution. We described two applications, namely automated program repair and library modelling. We experimentally showed that the proposed technique can reduce path explosion when applied for program repair, which helps to repair more bugs in programs with loops. Besides, it can be used to synthesize models of unavailable libraries. Our work is being made available as an extension of KLEE: http://angelix.io/second-order.html

ACKNOWLEDGEMENTS

This research is supported by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

[8] SyGuS-Comp 2017 results: http://sygus.seas.upenn.edu/SyGuS-COMP2017.html

REFERENCES

[1] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. 2013. A parametric approach for smaller and better encodings of cardinality constraints. In *CP*. Springer, 80–96.

[2] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal specification synthesis. In *POPL*. ACM, 789–801.

[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *FMCAD*. IEEE.

[4] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *TACAS*. Springer, 319–336.

[5] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, Cesare Tinelli, et al. 2009. Satisfiability modulo theories. *Handbook of satisfiability* 185 (2009), 825–885.

[6] Osbert Bastani, Saswat Anand, and Alex Aiken. 2015. Specification inference using context-free language reachability. In *POPL*. ACM, 553–566.

[7] Marcel Böhme, Ezekiel O Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the bug and how is it fixed? an experiment with practitioners. In *FSE*. ACM, 117–128.

[8] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[9] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT Solver.. In *TACAS*, Vol. 7795. Springer, 93–107.

[10] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. 2007. Bouncer: Securing software by blocking bad input. In *OSR*. ACM, 117–130.

[11] Cristina David and Daniel Kroening. 2017. Program synthesis: challenges and opportunities. *Phil. Trans. R. Soc. A* (2017), 20150403.

[12] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. *TACAS* (2008), 337–340.

[13] Patrice Godefroid. 2011. Higher-order test generation. In *PLDI*. ACM, 258–269.

[14] Joxan Jaffar, Vijayaraghavan Murali, Jorge Navas, and Andrew Santosa. 2012. TRACER: A symbolic execution tool for verification. In *CAV*. Springer, 758–766.

[15] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *ICSE*. ACM, 215–224.

[16] James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *ICSE*. ACM, 467–477.

[17] René Just, Michael D Ernst, and Gordon Fraser. 2014. Efficient mutation analysis by propagating and partitioning infected execution states. In *ISSTA*. ACM, 315–326.

[18] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* (1976), 385–394.

[19] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *PLDI*. ACM, 216–226.

[20] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *ICSE*. IEEE, 3–13.

[21] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *FSE*. ACM, 166–178.

[22] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. *POPL* 51, 1 (2016), 298–312.

[23] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2018. Semantic Program Repair Using a Reference Implementation. In *ICSE*.

[24] Sergey Mechtaev, Gao Xiang, Shin Hwei Tan, and Abhik Roychoudhury. 2018. Test-equivalence Analysis for Automatic Patch Generation. *TOSEM* (2018).

[25] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *ICSE*. IEEE Press, 448–458.

[26] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *ICSE*. IEEE, 691–701.

[27] Elliott Mendelson. 2009. *Introduction to mathematical logic*. CRC press.

[28] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *ICSE*. IEEE Press, 772–781.

[29] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. 2016. Shadow of a doubt: testing for divergences between software versions. In *ICSE*. ACM, 1181–1192.

[30] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. 2012. Darwin: An approach to debugging evolving programs. *TOSEM* (2012), 19.

[31] Dawei Qi, William N Sumner, Feng Qin, Mai Zheng, Xiangyu Zhang, and Abhik Roychoudhury. 2012. Modeling software execution environment. In *WCRE*. IEEE, 415–424.

[32] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. 2015. Counterexample-guided quantifier instantiation for synthesis in SMT. In *CAV*. Springer, 198–216.

[33] João P Marques Silva and Karem A Sakallah. 1997. GRASPâĂŤa new search algorithm for satisfiability. In *CAV*. IEEE Computer Society, 220–227.

[34] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *FSE*. ACM, 532–543.

[35] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *FSE*. ACM, 727–738.

[36] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex–white box test generation for .Net. *Tests and Proofs* (2008), 134–153.

[37] Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, and Dan Hao. 2017. Faster mutation analysis via equivalence modulo states. In *ISSTA*. ACM, 295–306.

[38] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *ICSE*. IEEE Press, 416–426.