Utrecht University

A Heuristic Approach to the Path Explosion Problem for Complex Heap Programs

Daniel van Vliet

Master Thesis

Supervisors: Dr. S.W.B. (Wishnu) Prasetya
Prof. dr. G.K. (Gabriele) Keller

July 2023

Abstract

Symbolic verification is a verification method for automatically verifying the correctness of programs, by collecting path constraints which are used to assert specifications and invariants using a third-party constraint solver. In this thesis we have extended OOX, a symbolic verification engine for object oriented programs, with support for subtyping through inheritance and dynamic dispatch, and have implemented a heuristic approach for reducing the path explosion problem. The verification engine operates on programs in the language OOX, which is one of the first intermediate verification languages with support for inheritance.

In the experiments we show that there is an exponential explosion in verification runtime as the number of subtypes of a symbolic object increase in a datastructure. With the aim to improve on this we have implemented an initial heuristic approach to reduce the path explosion problem, by selecting paths with a criterion. We have implemented heuristics based on maximising code coverage, on randomness and a round-robin combination. In our evaluation we found that the performance of the heuristic is dependent on the depth of the input program, with our first results showing that the maximising code coverage heuristic is not feasible for smaller programs.

Contents

Chapter 1

Introduction

As society relies more on software, bugs become increasingly noticeable and more attention is paid to finding and preventing them. The impact a bug has depends on the domain of the software. It can range from having unhappy users to potentially life threatening situations. Finding, solving and preventing bugs is desirable in either case.

Coralogix[2], a data analytics company, claims that on average developers write as many as 70 bugs per 1000 lines of code. Of these 70 bugs, 15 are not detected and find their way to end users. A large amount of development time is spent trying to find and fix them. Manually finding bugs can be time intensive. Companies resort to automated tools to replace much of their manual work [29].

In traditional automated testing the developer has to provide test input to the program. Given the input, the program is executed following a certain path. Usually the test is concluded with an assertion that asserts the program is correct for that input, i.e. that a property holds. To further increase the confidence in the correctness of the program, testers repeat this with a number of different inputs. Finding test input that exposes the bugs can be a difficult, time-intensive task. An alternative to testing is formal verification, which are methods to verify that a program is correct with a proof. This is can be applied to smaller programs but is not always practical in the software industry due to larger codebases.

In research there are some approaches to automatically verify a program. One of which is symbolic execution (SE), a program analysis technique. The goal of SE is to automatically explore many paths and inputs and assert that properties hold on those paths. Symbolic execution, unlike traditional testing, can verify a program without needing a testing input from the programmer.

In SE the concrete input values that are given to a program are replaced with symbolic input variables. A symbolic variable represents all values of the type at the same time. During the program execution, the values that the symbolic input variable can represent are constrained by branch statements such as *if* and *while*. At such branch statements, there is a branch condition; the guard boolean expression. The engine splits the path up, into a branch where the branch condition is true and one where it is false. The SE engine then constraints the symbolic variables with the branch condition on each path, reducing the values that the variable can represent accordingly.

Let's take a look at an example in listing 1.1. The symbolic execution of this program is visualized in figure 1.1. In the symbolic execution, we have multiple states which consist of a symbolic store σ, the path constraint π and the current statement. σ keeps track of the current variables and their symbolic or concrete representation. The path constraint π contains the constraints on the symbolic variables as a result of the branches it has taken.

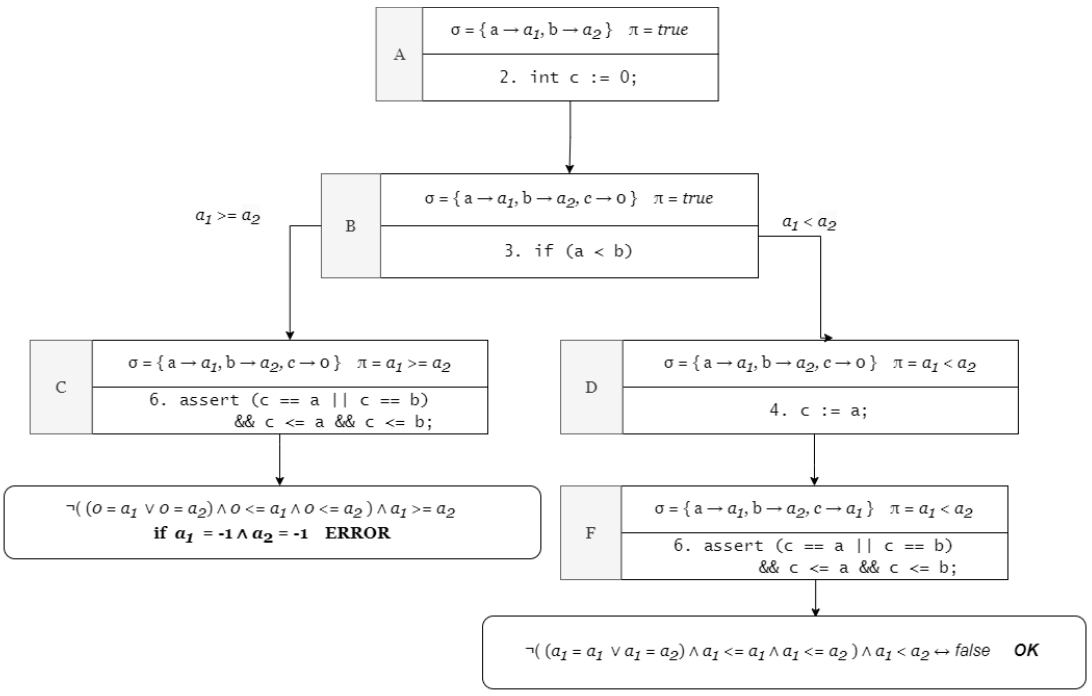

A
σ = { a → a1, b → a2 } π = true
2. int c := 0;
B
σ = { a → a1, b → a2, c → 0 } π = true
3. if (a < b)
a1 >= a2
a1 < a2
C
σ = { a → a1, b → a2, c → 0 } π = a1 >= a2
6. assert (c == a || c == b) && c <= a && c <= b;
D
σ = { a → a1, b → a2, c → 0 } π = a1 < a2
4. c := a;
¬((0 = a1 ∨ 0 = a2) ∧ 0 <= a1 ∧ 0 <= a2) ∧ a1 >= a2
if a1 = -1 ∧ a2 = -1 ERROR
F
σ = { a → a1, b → a2, c → a1 } π = a1 < a2
6. assert (c == a || c == b) && c <= a && c <= b;
¬((a1 = a1 ∨ a1 = a2) ∧ a1 <= a1 ∧ a1 <= a2) ∧ a1 < a2 ↔ false OK

Figure 1.1: Symbolic execution of function min

As can be seen in the example the path constraint initially is *true*, since there are no assumptions made yet and thus no constraints on the symbolic variables a_1 and a_2. After the if statement in state B, the path is split up into two states, one where the condition $a < b$ holds and one where it does not. The path constraints are updated to reflect this assumption. In the states with an assertion, C and F, the program is evaluated resulting in an error at C.

The programs are evaluated using a constraint solver like Z3 [23]. If the negated expression is satisfiable this means that there is a set of inputs such that the assertion does not hold. Z3 could provide us with an example of this input, in this case $a_1 = -1 \wedge a_2 = -1$. On the other branch at state F the expression is unsatisfiable and thus holds for all input a and b.

```
int min(int a, int b) {
    int c := 0;
    if (a < b) {
        c := a;
    }
    assert (c == a || c == b) && c <= a && c <= b;
    return c;
}
```

Listing 1.1: Invalid min function with property assertion

One of the largest issues with Symbolic Execution is the problem of path explosion. In the previous example of the min function there are only two paths, however the number of paths increases quickly as we introduce more if and while branches to the program. Take for instance Listing 1.2, a simple algorithm to compute the sum up to n numbers. When this program is run with a symbolic executor, the path is repeatedly divided at the while condition. Since n is symbolic, and represents any number up to 2^{32}, this simple program already explodes in 2^{31} paths. A program with many such loops or recursive functions, let alone nested loops, can quickly explode to a number of paths that is unfeasible to verify.

A lot of research has been done to reduce the path explosion problem, for instance through merging or summarizing states, or setting a limit for the depth of the loops taken. This usually results in sacrificing soundness or completeness. The latter means that the program is not verified for all paths and as a result there may be false positives.

```
int sumToN(int n) {
  if (n < 0) return -1;
  int r = 0;
  while (n != 0) {
    r += n;
    n -= 1;
  }

  assert (r == (n * (n + 1)) / 2)
  return r;
}
```

Listing 1.2: Path explosion due to while loop

Every year a competition, SV-COMP, is held with the goal to improve software verification tools[5]. In 2022, symbolic execution was used in the tools of 17 of the 47 participants. In this competition they have gathered a large set of benchmark programs which are used to compare the performance of the participants. The tools are compared on the number of bugs found and execution time, amongst other aspects.

Besides loops and recursion, another way for the number of paths and/or state space in a Symbolic Execution to explode is through heap objects of various shapes. For instance consider a function that is passed an array of elements of which the size is unknown. Similar to the loop, the symbolic executor can split up the paths such that in each path the array has a larger size than the previous. But as arrays can become as large as memory allows, the number of paths needed here explodes. For arrays this is a well known problem and a similar challenge exists when it comes to heap objects with references to other objects.

Consider the program in Listing 1.3. When an object is passed to a function, the SE engine must consider all the possible inputs in order to be complete. One way to achieve this is to create paths with increasingly larger `List` objects.

If every input shape becomes its own path in the Symbolic Execution engine this will lead to increased path explosion. Thus some of the recent research has been focused on eliminating inconsistent inputs efficiently.

A further extension in the verification would be to add support for inheritance, such that in the object shapes, references can also point to a subclass of a type. This will lead to yet more possible shapes and thus path explosion is inevitable when attempting all possibilities.

```java
class List {
  int size = 0;
  ListEntry head = null;

  void insert(int value) {
    this.size += 1;
    this.head = new ListEntry(this.head, value);
  }

  int popHead() {
    ListEntry temp = this.head;
    this.head = this.head.next;
    return temp.value;
  }
}

class ListEntry {
  ListEntry next;
  int value;
}

class Main {
  static void main(List list) {
    list.insert(0);
    int head = list.popHead();
  }
}
```

Listing 1.3: Complex referencing heap object

1.1 Contributions

In this project we will research the path explosion problem in object oriented programs. This will be done using the OOX language and tool[1]. With the addition of inheritance, the OOX language can represent complex heap programs. When verified these heap programs will result in path and state explosion. We will attempt to reduce this explosion with the use of a heuristic.

The main contributions of this project can be summarized as follows:

- Overview of the current state of the art approaches to verification of complex heap programs.
- Extend the OOX language to support inheritance
- Create and find a set of complex heap test programs as benchmarks.
- Develop and study a heuristic to reduce the number of object graphs that have to be explored.
- Study the impact of inheritance on the path explosion problem, and the performance of the heuristic.

The main goals of the thesis will be to study the path explosion problem in programs with complex heap objects. We conduct experiments to measure the impact of adding inheritance to heap input is on the explosion. This leads us to the first research question.

[1]https://github.com/DanielvanVliet/OOX

Research question 1: How does the addition of inheritance on the programs under verification impact the path explosion problem?

Next we research heuristic to reduce the number of object graphs that need to be verified in the symbolic execution. A heuristic can for example be used to reduce the number of object graphs by eliminating unfeasible structures.

Research question 2: What are suitable heuristic to effectively reduce the path explosion in verification?

1.2 Thesis structure

In this document we will first give some background information in chapter 2 on the topics that are discussed in the remaining chapters. Then in chapter 3 we give an overview of existing verification techniques for dealing with complex heap programs. After that we describe the addition of inheritance to OOX in terms of syntax and semantics in chapter 4. Then in chapter 5 we describe the heuristics that were implemented in OOX. In chapter 6 we describe the minimal statement distance algorithm that was needed for one of the heuristics in chapter 5. In chapter 7 we evaluate the addition of inheritance and the heuristics with an experiment. In chapter 8 we provide a chapter about related work and in chapter 9 we conclude and suggest future work.

1.3 Societal relevance

Software plays an important role in the current society. It is used on every digital device that we own. As a result of this, society relies more on software and its correctness. The correctness has become important, both in fields where there are lives at risk but also in fields where there are simply a lot of users that rely upon the service. Symbolic verification can assist in proving the correctness of the software.

A lot of software is written today with object oriented languages. Verifying object oriented patterns can be hard [12] due to their reliance on heap objects and references between them, resulting in large complex heap tree-like structures. This research project will attempt create a better understanding of the issue and improve symbolic execution on such programs with a heuristic. This may lead to improved verification on object oriented programs.

Existing research towards SE on complex heap objects does not consider the impact of inheritance on the path explosion and this is something that this project will attempt to show.

Chapter 2

Scientific background

2.1 Symbolic Execution

Symbolic execution is a technique for verifying whether certain properties hold in a program. This can be used to catch bugs such as dereferencing of nullpointers, to check whether two functions are equivalent or whether there are security issues. In short, it allows one to check for the correctness of the software. The technique can achieve this statically, meaning without necessarily executing the program. Contrary to executing a program where a single path is followed with concrete values for variables, symbolic execution considers multiple paths that the program takes and for each path considers all possible values for the variables. This is achieved through the use of symbolic variables which represent a range of values. For each path in the program, a boolean formula is maintained containing the conditions satisfied by the current path as a result of the branches that were taken. This formula constrains the possible values that the variables in the program could take.

In the symbolic memory store the program variables are mapped to symbolic expressions, which are updated through assignments. Depending on the implementation this can also be seen as a model for the stack, which stores local variables and parameters in each function. Using the branch conditions and the memory store the engine can for each path determine whether constraints are satisfied or if the path is unfeasible.

When the symbolic engine then encounters an assert statement containing a property, it invokes a theorem prover to verify whether the property holds for the current branch. An example of a theorem prover is the Satisfiability Modulo Theories (SMT) solver Z3[3]. This solver allows the process to be fully automatic, however in practice it is limited by increasing computation time as the formulas to check become larger.

In symbolic execution one can take the exhaustive approach and attempt to find and verify all program paths. This is sound and complete, meaning there are no false negatives or false positives. However, in practice due to program constructs such as loops and recursion, the number of paths that a program can take can become unfeasibly large. Thus often a limit is set on the depth of the loop/recursion as a workaround, but this sacrifices completeness, resulting in potentially missed bugs.

There exist other challenges for symbolic execution such as how to deal with software calls to external libraries of which the source code can be unavailable, or in programs where the input of the user is requested. These challenges are out of scope for this research project.

2.2 Memory model of Symbolic Execution

When we consider more complex programs than single function programs the need arises for dynamic allocation; i.e. objects in the heap. In Symbolic Execution a memory model must be chosen to represent a heap to allow such allocations. There are different approaches to modelling the memory of a program in Symbolic Execution. This is partially due to the fact that there are languages which simply have different memory models such as low level C and more object oriented languages such as Java, where the former allows pointers to memory addresses and the latter contains pointers to objects. Which one is chosen depends on what kind of programs have to be verified. In this project we will focus more on the high level memory model seen in object oriented languages such as Java and C#.

When considering pointers and arrays there are also different approaches among the existing symbolic execution frameworks [3]. The most precise is the fully symbolic approach. The fully symbolic memory approach may consider a memory address as fully symbolic. When an operation reads from or writes to a symbolic address the state is either forked to all possible states after the operation or the uncertainty of possible values is encoded through if-then-else formulas in the expression kept in the symbolic store.

Another approach is to model the heap such that pointers are restricted to be either null, or point to an allocated tracked heap object. This is recursively repeated for objects which itself contain pointers to other objects. This is also referred to as Heap modeling [3]. The number of addresses or objects that a pointer could point to can be reduced in this way. This is also the approach that is taken by OOX.

Partial approaches also exist which combine fully symbolic memory and heap modeling. This is a trade-off that existing verification frameworks make between performance and control over memory. [3]

2.3 Satisfiability Modulo Theories solvers

In order to check whether a path in a program satisfies the required properties, we resort to model checkers. A commonly used model checker is the SMT solver. SMT is a computerised method for finding solutions to business and industrial problems expressed mathematically by systems of constraints [4].

A constraint or equation is satisfiable if there exist values for the variables such that it can be made true. Given a program path and a property that must hold on that path, an SMT solver can be invoked to automatically compute whether the property holds.

Many existing symbolic verification engines resort to a third party SMT for constraint solving. A benefit of using a third party SMT instead of developing own ways of constraint solving is that it has a large research community already. Which allows SE engines to benefit from performance improvements and bugfixes and focus more on the path explosion problem.

2.4 Intermediate verification language

An intermediate verification language is a language that is understood by a verification tool and can describe programs of a certain class of programming languages. The benefit of having such an intermediate language is that multiple programming languages can be represented by this tool and that it can be kept simple. This is not the only approach to verification, as there are also tools that work directly on a specific language such as JBSE which operates on Java programs[6]. KLEE [8] is an example of a tool that does not verify on programs of a programming language but on output of the LLVM compiler infrastructure.

The IVL that will be used in this project is OOX [18]. OOX is an object oriented language that can represent Java and C# programs. The language supports classes and objects and has a strong-type system. There is also support for concurrency, but this will be left out of scope for this research project. The language currently does not have support for inheritance. However due to the well documented semantics we will be able to extend the language and add support for it.

Another IVL that can represent Java programs, without inheritance, is the SimpIL language[27]. This was later extended by Pham et al.[25] and used in their test generation engine based on symbolic execution.

2.5 Inheritance

Inheritance in object oriented programming languages is the ability for one class to extend another class. When class A extends B, the fields and methods of class B will be inherited in class A, along with its own defined fields and methods [15]. When a reference of class A is required in a language with inheritance, an object of type A or any of the sub-classes of A is accepted. Since in a typical program most classes will have subclasses, inheritance increases the burden of possible objects to explore during symbolic execution.

In order to add inheritance the semantics of OOX need to be extended. There have been attempts to add formal semantics for languages like Java [13], some of which include semantics for inheritance, and these can be used as a reference to extend OOX with inheritance.

2.6 Test generation or program verification

In the recent years the SE community has split themselves to either test generation or program proving. Most SE approaches focus on test generation rather than program verification [28]. Test generation is made possible by using the counterexamples that can be provided by Satisfiable Modulo Theories solvers. An SMT solver can provide counterexamples when a constraint is proven wrong. A counterexample can be concrete values for variables in a function that lead to a false assertion. These values can then be used to create test input.

The techniques for generation of heap input are independent of this and can be used for both purposes. A downside of this divide is that in the yearly software verification competition that is held, SV-COMP[5], many of the test generation focused SE approaches do not participate. Thus some of the recent improvements to heap generation are not applied to program verification and compared with existing methods.

The test generation research that uses symbolic execution has two major approaches; concolic testing and execution generated testing. In concolic testing, the symbolic execution is driven by concrete input. Path constraints are maintained and used to generate new input with an SMT solver that will travel a different branch. OOX is not driven by concrete input although it is possible to run the program without symbolic values. In this case the program behaves as a normal execution following a single path as all infeasible paths can be easily pruned.

In execution generated testing, the symbolic execution is like normal with the ability to switch to concrete execution when a path condition becomes unsolvable or when the instruction does not contain a symbolic value. Because this approach is more like traditional symbolic execution, it is easier to compare with OOX and similar heuristics can be applied to both.

Chapter 3

Overview verification of complex heap programs

In recent years research has been dedicated to the path explosion problem with focus on complex heap objects. Programs containing complex heap objects are often encouraged in object oriented languages like Java or C#. In these languages the objects can contain references to other objects, for instance in design patterns [12]. Each possible instance of such a complex object graph becomes its own path and contributes to the path explosion.

In this chapter we give an overview of the current state of the art approaches to verification of complex heap programs.

3.1 Lazy Initialization

Object oriented languages may contain objects which have many references to other objects on the heap, who themselves have references to other objects. This results in an object graph.

It would be unnecessary to explore a part of such a graph if the object is never accessed, and thus the lazy approach is sometimes used. When new objects are tracked, their fields are uninitialized until they are accessed by the program [17]. This further reduces the number of paths the SE engine has to take. Once the uninitialized object is used in the program, for example when a field or method is accessed, it will be initialised. An example is shown in 3.1. In this example, *value* is not lazy initialised because it is not an object but a primitive. It is initialised as soon as the object Foo itself is initialised. The field bar is initialised only after the first operation on it occurs at line 10.

When initialized it should be considered which objects it can point to and the paths must be split accordingly. It can point to either null, a fresh object or a previously initialised object of the same type. This way the aliasing case, where an object points to an existing object, is also handled.

This approach is also taken in OOX, all the (reference type) fields of a symbolic object are lazy-initialised.

```
class Foo {
  Bar bar;
  int value;
}

void main(Foo foo) {
  // value is a primitive symbolic value that is not lazily initialised.
  foo.value += 1;
  // bar is initialised to an existing Bar, null and a new instance of Bar
  foo.bar.f();
}
```

Listing 3.1: Lazy initialisation

3.2 Checkable preconditions

To further reduce the number of objects to generate, lazy initialization can be extended by describing preconditions for the structure and methods. An example of a precondition would be that a structure is a non-null acyclic binary tree. Such a precondition can be used to eliminate program paths early and improve the performance of the symbolic verification engine. A programmer can create a method to check whether an object satisfies such a precondition, this is usually referred to as a **repOk** routine [26].

These **repOk** routines can be extended to work with structures containing symbolic values. The strategies for what to do when encountering a symbolic value range from simply assuming validity or perform more sophisticated backtracking. The latter requires expertise from a programmer and is program specific, thus usually the first approach is taken.

3.3 Bounded lazy initialization

When dealing with objects that must follow certain constraints such as linked lists or binary trees, a further extension is to use bounded lazy initialization [26] (BLI). BLI analyses the structures involved and computes field bounds which are used to reduce the alternative instances of the structures that need to be explored in symbolic verification. Given a scope representing the upper maximum number of objects that may be involved in the structure, and any preconditions of the structure. A field bound represents the objects that a field of object N can point to in any well-formed structure, within a given scope and complying with preconditions. By annotating the partial structure with the field bounds, it allows to prevent the creation of invalid structures. One downside of this approach is that a scope is required which is usually set in a way that the verification can be executed in the limited time constraints.

3.4 HEX

Another approach was taken by Braione et al.[7]. They proposed a language called HEX for specifying invariants of partially initialized structures. With this language they can for instance, express that a datastructure must be circular and not contain any null references. For a linked list, the following HEX sentence would achieve this:

$$root.\text{list}(.\text{next})^{+}\ \textbf{not null} \wedge root.\text{list}(.\text{next})^{+}\ \textbf{aliases}\ root.\text{list}$$

This sentence states that (1) on any object in the linked list the field *.next* may not be a null reference and (2) that any *.next* field that refers to an alias, may only refer to the root node. The semantics are designed to work on data structures that are not completed yet, and this allows them to effectively combine it with lazy initialization.

They implemented a decision procedure for HEX that is able to incrementally evaluate the consistency of input structures. The benefit of achieving this incrementally is that they can reuse results from smaller structures and thus gain a large improvement in performance compared to existing methods. They have implemented their HEX incremental decision procedure in their Java Bytecode Symbolic Executor [6].

This technique still has some drawbacks. Some properties cannot be captured by the HEX language, for instance that nodes in a tree are sorted on their value [24]. A workaround for this is to allow user provided methods to still capture these properties.

3.5 Separation Logic

A recent approach to complex heap inputs for Symbolic Execution is the paper by Pham et al.[25]. By using the separating conjunction from separation logic, they are able to split up the global heap and enforce that references are not pointing to the same object. Their work is implemented in a test generator called Java StarFinder, focused on test generation rather than program verification. They have no support yet for inheritance and polymorphism.

In their work, the path condition is in separation logic and thus they have a specific solver S2SAT [19]. This in turn uses Z3 for the satisfiability problems over arithmetic.

3.6 Heuristic for path exploration

This approach does not attempt to reduce the path explosion by pruning paths, but instead by using a heuristic to select a subset of paths to explore. There have been several heuristic proposed, ranging from random path selection to selecting paths aimed to maximize coverage [11].

In this thesis project we are also taking a heuristic approach to reduce the path explosion and aim to improve bug finding performance.

Chapter 4

Inheritance

In this chapter we describe the first main contribution of this thesis project. We will describe the changes that were made to OOX to add support for subtyping through inheritance and dynamic dispatch.

4.1 Syntax

The syntactical structure of OOX is updated to support inheritance. The ⟨*class*⟩ rule was updated to allow inheriting other classes by extending a class and to implement one or more interfaces. The constructor was updated to support calling the constructor of the super class with the optional ⟨*superstatement*⟩.

A new grammar rule ⟨*interface*⟩ for declaring interfaces was added, which can extend from other interfaces. Unlike classes, interfaces cannot declare fields and instead have a ⟨*interface member*⟩ rule. This allows them to have default and non-default methods which do and do not have a method body. We will show only the updated syntax rules, a complete ruleset can be found in the original thesis for OOX [18].

⟨*class*⟩ ::= **class** ⟨*identifier*⟩ ⟨*extends*⟩? ⟨*implements*⟩? **{** ⟨*member*⟩* **}**

⟨*extends*⟩ ::= **extends** ⟨*identifier*⟩

⟨*implements*⟩ ::= **implements** ⟨*identifier*⟩ (**,** ⟨*identifier*⟩)+

⟨*constructor*⟩ ::= ⟨*identifier*⟩ **(** ⟨*parameters*⟩ **)** ⟨*specification*⟩ ⟨*constructorbody*⟩

⟨*constructorbody*⟩ ::= **{** ⟨*superstatement*⟩? ⟨*statement*⟩+ **}**

⟨*superstatement*⟩ ::= **super** **(** ⟨*parameters*⟩ **)** **;**

⟨*interface*⟩ ::= **interface** ⟨*identifier*⟩ ⟨*extends*⟩? **{** ⟨*interface-member*⟩* **}**

⟨*interface-member*⟩ ::= ⟨*interface-method*⟩ | ⟨*default-interface-method*⟩

⟨*interface-method*⟩ ::= ⟨*type*⟩ ⟨*identifier*⟩ **(** ⟨*parameters*⟩ **)** **;**

$\langle$*default-interface-method*$\rangle$ ::= $\langle$*type*$\rangle$ $\langle$*identifier*$\rangle$ ($\langle$*parameters*$\rangle$) $\langle$*body*$\rangle$

We have also extended the lexer with keywords **interface**, **extends**, **implements** and **super**.

4.2 Typing

With the added syntax support for inheritance and interfaces from object oriented programming, we can now discuss subtyping in OOX. In OOX there are primitive and reference types. Primitive types are allocated on the stack and reference types are allocated on the heap [18]. Currently there are the primitive `bool` and `int` types along with user definable reference types using the `class` keyword[1].

With the extension of inheritance, subtyping is now allowed between classes. Class A is a subtype of class B if A extends B. Class extension is transitive, meaning if A is a subtype of B and B is a subtype of C, then A is a subtype of C.

Aside from classes we also have support for interfaces. Interfaces are not initializable. Similarly to classes, an interface can extend one or more other interfaces. Interface A is a subtype of interface B if A extends B. Interface extension is also transitive, meaning if A is a subtype of B and B is a subtype of C, then A is a subtype of C.

A class can also be a subtype of an interface, but an interface cannot be a subtype of a class.

Class A is a subtype of interface B if A implements B. Class implementation of interfaces, using the **implements** keyword, is transitive. Meaning that if class A is a subtype of interface B and interface B is a subtype of interface C, then class A is a subtype of interface C. Similarly class implementation is transitive with respect to class extends. If class A extends class B and class B implements interface C, then class A implements C.

In other words, subtyping with respect to classes and interfaces is similar to object oriented languages such as Java and C#.

For all reference types defined by classes and interfaces, reflection also holds. Thus a type is also a subtype of itself.

4.3 Static semantics

With the added syntax support for object oriented programming, the next goal is to implement semantics for this. The OOX with support for inheritance should accept a superset of programs accepted by the old OOX.

Classes This iteration of OOX will accept classes that inherit from at most one class and implement one or more interfaces. The inherited must be defined in the program and not be (indirectly) the class itself. The implemented interfaces must be defined in the program, and the same interface cannot be implemented more than once. Each method must be uniquely defined in the interfaces, with no conflicts.

Interfaces Furthermore interfaces can extend interfaces, and they contain either methods or default methods, where the difference is that default methods have a body and are thus the same as class methods and interface methods do not have a body implementation.

[1]The semantics allow for more primitives and the string type, but they have not been implemented yet.

Method overriding When a class implements an interface, it must contain a method for all non-default methods in the interface. This is called method overriding. Furthermore a class can override a method from a super class with the same signature.

4.4 Method invocation resolving

In the Control Flow Graph (CFG) of OOX, all methods are disconnected from each other. Method resolving is the task of finding the target method in the Control Flow Graph when an invocation is encountered. A method has three properties that are used to identify which method should be resolved to:

1. Declaration name, of the class or interface
2. Method name
3. Formal parameters

In the previous iteration of OOX, there was no inheritance and no dynamic binding. It was therefore possible to use the declared type of the object together with the method name and arguments to resolve the targeted method.

After the implementation of inheritance there is more than one method that can be invoked if we consider only the declared type. We must therefore use the type at runtime of the object to find the exact method that must be called. It can be statically determined which methods can be called at runtime by an object due to the typing semantics of OOX. We know that a declared type can at runtime be the type itself or a subtype. Subtypes are able to override a method by declaring the same method and arguments as a supertype.

The set of methods that can be executed by an invocation is called the polymorphic call set [1]. The exact method that is called depends on the runtime type of the object. Therefore in OOX we define the polymorphic call map, a mapping from runtime type to declaration, method pair. Since interfaces cannot be instantiated to objects they will not appear in the mapping.

This mapping is computed at each invocation of a method, by using the declared object type. And looking at the closest overriden method closest to the runtime type in the hierarchy.

We will now give an example of a polymorphic call map (PCM). Consider the class hierarchy in figure 4.1 and the corresponding type families in table 4.1a. Given a type of I i, the invocation $i.f()$ would result in the polymorphic call map shown in table 4.1b. This means that if the type at runtime were to be A, then I::f() would be invoked.

This procedure is repeated at every encountered invocation in the program to resolve the method.

Type	Type Family
I	{A, B, C}
A	{A, C}
B	{B}
C	{C}
Foo	{Foo}

(a) Type family of fig. 4.1

Type	Method
A	I::f()
B	B::f()
C	C::f()

(b) PCM of i.f()

Type	Method
A	I::g()
B	I::g()
C	C::g()

(c) PCM of i.g()

Table 4.1: Type family and Polymorphic call map example

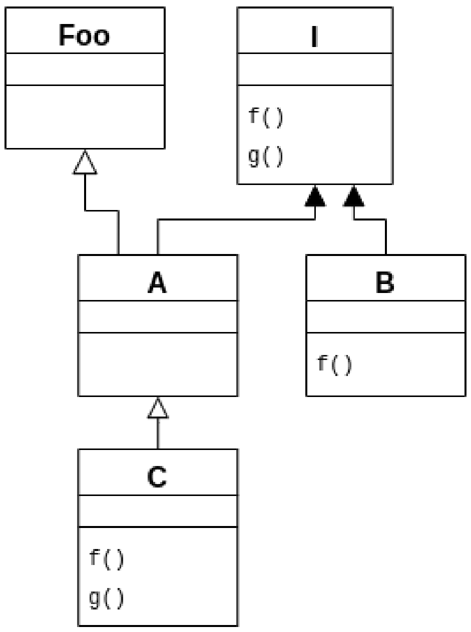


Figure 4.1: An example class hierarchy, where I is an interface and the other declarations are classes

4.5 Verification

During verification when an invocation is encountered on an object, for example i.g(), the variable i can be a concrete reference or a symbolic reference. In case it is a concrete reference we simply lookup the target method in the PCM table 4.1c for the type of i, and continue the verification. If it is a symbolic reference we may have to split the path. Whether this is needed depends on the types in the aliasmap of i.

For instance, if the aliasmap entry of i has aliases of the types {A, B, C}, then a path split is required since types A and B resolve to I::g() and type C to C::g(). This will result in two paths with an update to the aliasmap entry of symbolic reference i.

The first path will continue at the function entry of I::g(), and the aliasmap is updated such that it only contains aliases of types that resolve to I::g(), in this case A and B. The second path continues at function entry C::g(), and the aliasmap is updated to only include aliases of type C in the aliasmap entry of i.

In this way the polymorphic call map is used at each invocation to determine the targeted methods and required path splitting, if any.

4.6 Lazy initialisation

With the introduction of inheritance, the lazy initialisation of OOX has been changed to support subtyping. Instead of having only one type of object that must be considered, all (class) types in the type family of the declared object have to be considered.

When a symbolic reference is lazily initialised, a new entry is inserted in the aliasmap. The aliasmap is a mapping from symbolic references to sets of concrete references. The purpose of the aliasmap is to track which heap objects the symbolic reference can point to and to assist in the lazy initialisation of new variables [18].

Let us call this new aliasmap entry A, which consists of all concrete references that the symbolic reference could point to.

In OOX without inheritance, A was defined as follows:

$$A = A_{existing} \cup \{\text{null}, ref_{fresh}\}$$

Where $A_{existing}$ is the union of the sets of concrete references of all other symbolic references of the same type in the aliasmap. ref_{fresh} is a fresh concrete reference to a new heap object of the same type.

With the extension of inheritance, the definition of A has been updated:

$$A' = A_{existing'} \cup \{\text{null}\} \cup \{ref_a \mid a \in TypeFamily\}$$

Where TypeFamily is the type family set of the object type, as shown in table 4.1a. $A_{existing'}$ is similar to $A_{existing}$, but is now the union of the concrete references of all other symbolic references whose type is in the type family of the object.

When the type of the lazy initialised object has no inheritance, i.e. the type family is a singleton set, then $A' = A$.

4.7 Cast and instanceof operator

To allow more programs to be representable with OOX we have also added limited support for the casting and instanceof operators. We have extended the allowed *Rhs* operators with a ClassCast operator $rhs_{cast}(\tau, i)$, which can be used on the right hand side of assignments in OOX. The updated list of *Rhs* can be found below.

$$\begin{aligned} v \in Rhs ::= \; & rhs_{expr}(E) \\ & \mid rhs_{field}(i, F) \\ & \mid rhs_{call}(I) \\ & \mid rhs_{elem}(i, E) \\ & \mid rhs_{array}(\tau, E) \\ & \mid rhs_{cast}(\tau, i) \end{aligned}$$

In the type system we require that the type casted is in the type family of the cast type. For example take the cast assignment in 4.1. In this example, the type of b must be in the type family of Foo.

During the parsing phase we apply a syntactical transformation to this assignment, similar to how is done in invocations and field access [18]. The transformation is found in listing 4.1. This transformation prevents incorrect casting and allows us to catch the exceptional case.

```
Foo f := (Foo) b;
// becomes
if (b instanceof Foo) {
    assume b instanceof Foo;
    Foo f := (Foo) b;
} else {
    assume !b instanceof Foo;
    throw "exception";
}
```

Listing 4.1: Cast syntactical transformation

Furthermore we have added the instanceof expression. The instanceof operator was introduced under a separate, new type of expression called TypeExpression. The reason for this is that we wanted to limit the complexity of the expressions containing instanceof expressions. We only allow checking if an object is of type τ or if it is not. The reason for this is to disallow complex expressions containing instanceof that make the verification more difficult. In future work they may be merged with the existing expression type if the problem is better understood.

$$\begin{aligned} T \in \mathit{TypeExpression} ::= &\ \mathit{instanceof}(i, \tau) \\ &\mid \mathit{notinstanceof}(i, \tau) \end{aligned}$$

```
// Allowed operations
x instanceof Foo
!x instanceof Foo

//Not allowed
!(!x instanceof Foo)
x instanceof Foo && !instanceof Bar
```

The *ite* abstract syntax from OOX[18] was updated to allow either regular *Expression* or *TypeExpression*. Like with regular expressions, the parser will apply a syntax transformation to insert assume E; and assume !E; in the if and else body.

When such an assume statement is encountered in the engine we remove any objects in the aliasmap that are not instances of the type, likewise in the else assume we remove any objects in the aliasmap that are instances of the type. The path will be pruned when there are no concrete references that match the instanceof type.

When there is an assert statement with an instanceof expression, we assert that all concrete references of the object match the type.

Chapter 5

Heuristic approach for reducing the path explosion

The second contribution of this project is the addition of search heuristic to optimise path exploration. A search heuristic guides the symbolic execution towards certain paths according to its criterion[11]. This is useful because of the path explosion problem as it is infeasible to explore all paths.

In this project we have implemented three heuristic alongside the existing depth first search heuristic. Each heuristic is complete with respect to the original Depth First Search heuristic. This means that eventually they will reach the same result for a program depth limit, given enough time.

5.1 Depth first search

This is the default heuristic that was implemented and used in the previous iteration of OOX. The symbolic engine is guided in a depth first search manner, following a path until it is finished, before backtracking and choosing a different path at the last path junction. This approach works for small programs with finite paths, but will prioritise the path deep in the program and therefore may not reach a different part of the program within the required time. This effect is increased by programs with while loops.

In the current implementation with dynamic dispatch, sometimes there are multiple resolved methods and a path split is required. In this case we select the first one that we have found, but due to the usage of a non-deterministic hashmap this is not deterministic. As a result of this the depth first search will not behave exactly the same when repeatedly run. In a future work this can be solved by using determistic datastructures within OOX.

5.2 Random path

The random path heuristic is execution tree based, like the MD2U heuristic which is described in the next section. It makes use of an execution tree, which is a tree datastructure that keeps track of the traveled paths and branches of all states. In the tree, the nodes represent junctions in the program, and the leafs the current states. A visualisation can be seen in 5.1. The junctions can be any statement that results in branching of paths, such as if and while or dynamic dispatch invocation.

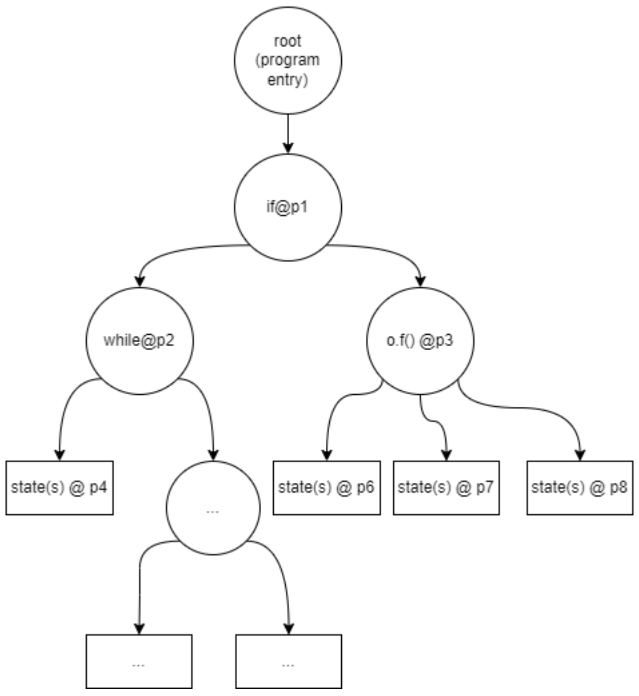


Figure 5.1: Execution tree visualized, the internal nodes are branching points in the program, the leafs contain one or more states. Both nodes and leafs keep a reference to their program point.

To select the next state to explore, we repeatedly make a random choice at junctions in the execution tree until we have reached a leaf with states. Program junctions are branching statements such as if and while statements, but also invocations with dynamic dispatch, resulting in multiple different method calls. We then execute one statement, update the execution tree, and repeat this process to find the next state.

This heuristic has the property that it more likely explores states higher up in the execution tree, since more junctions have to be crossed as we reach deeper.

5.3 Minimal distance to uncovered (MD2U)

This heuristic was inspired by the Coverage-Optimized Search in KLEE [8]. We have implemented this heuristic in OOX. Its goal is to achieve full statement coverage starting from the program entry method. Full statement coverage means that any statement in the entry method, and in every method (in)directly called during verification, have been visited at least once.

At every step in the symbolic execution, we first compute the minimal distance for each statement in the reachable program space, the algorithm for this is explained in chapter 6. Then a weighted stochastic choice is made at between all states in the leafs of the execution tree, where program paths closer to uncovered statements have a higher weight.

If there are no longer any paths with an uncovered statement, the heuristic makes a random choice between the remaining states.

5.4 Round robin

The last heuristic allows you to combine other heuristic in a round robin way. This means that we can for example alternately execute one step with the md2u heuristic and the next step with random path. This is more likely to target different execution states and may help to escape local searches.

Chapter 6

Computing the minimal distance to uncovered

Computing the minimal distance to uncovered (MD2U) for each statement is achieved on a method basis, meaning that it is computed for each method separately. When a method invocation is called, the procedure recursively computes the MD2U for that invocation first. This was done because the methods are separated in the control flow graph and to optimize caching.

In this heuristic we compute for each method and all their statements separately either:

1. The distance to the first uncovered statement if any (within this method).
2. Or otherwise the closest distance to the end of the method.

Distance is defined as follows:

$$\begin{aligned}\textbf{type}\ \mathit{Distance} &= \mathit{ToFirstUncovered}\\ &\mid \mathit{ToEndOfMethod}\end{aligned}$$

We make this distinction on the distance to make it possible to do this computation for each method separately, and thus easily cache the results for fully uncovered methods. If it is found that a method no longer contains any uncovered statements, we cache the shortest path to the end of the method. This can be reused to optimise the search for uncovered statements in further iterations.

Adding two distances together will result in ToFirstUncovered if either one is ToFirstUncovered. Otherwise it results in ToEndOfMethod.

Statement cost During the computation of the distance of each statement, we compute the statement cost. This is the cost of one statement in isolation. For any non-branching and non-invocation statements, the cost is 1. Because of the potential while loops and recursion in a method body we have to make the cost an expression that can be partially left unevaluated until a later point. The Cost is defined as follows:

$$\begin{aligned}\textbf{type } Cost = {} & Cost(Distance)\\ & \mid Cycle(ProgramCounter)\\ & \mid Cost + Cost\\ & \mid min(Cost, Cost)\\ & \mid MethodCall(Method)\end{aligned}$$

In the remainder of this chapter we describe the edge cases for the statement cost.

6.1 Sequence statement

The cost of the sequence statement is defined as the cost of the first statement added to the cost of the second statement.

6.2 Invocation statement

The cost of an invocation depends on whether the invocation is recursive, if it is a polymorphic invocation resulting in multiple methods or if it is an invocation for a (static) single method.

In case it is a recursive method we annotate the cost of that statement with MethodCall, and can substitute it once we know the minimal cost of the method call, more on this in section 6.5. If it is a polymorphic invocation with multiple methods we choose the minimal cost of those methods.

$$invocation_{cost} = \begin{cases} A_{cost} & \text{if invocation resolves to single (static) method } A \\ min(M) & \text{if invocation resolves to methods } M \end{cases}$$

Note that we assume that all methods that could be called by the object, can be called. This may not always be true if we have filtered the object instances using the **instanceof** operator.

6.3 If statement

If we encounter an if statement, the program splits into two paths and the cost is equal to the minimal of these paths. When the statement itself is uncovered for the first time, the cost is one instead.

$$if_{cost} = \begin{cases} 1 & \text{if uncovered} \\ 1 + min(Cost_{true}, Cost_{false}) & \text{otherwise} \end{cases}$$

6.4 While loops

The Cycle variant is needed for while loops. In while loops the cost of the while guard is the minimal cost between the while body and after the while loop. Listing 6.2 shows how the cost is computed for a while loop with n statements in the body. In this case the cost depends on whether the body cost is lower than the cost outside the body.

However when the while body has no uncovered statements, the statements will loop back to the while guard itself and thus depend on its cost, see listing 6.2.

```
...
while (b) {   // cost = 1 + min(n, <outside body>)
    s_n;      // cost = n
    ...
    s_1;      // uncovered statement! cost = 1
}
...
```

Listing 6.1: Cost computation in a while loop with uncovered statement s1 in body

```
...
while (b) {  // cost = 1 + min(<while cycle> + n, <outside body>)
    s_n;     // cost = <while cycle> + n
    ...
    s_1;     // cost = <while cycle> + 1
}
...
```

Listing 6.2: Cost computation in a while loop without uncovered statement in body

A similar approach is taken for nested while loops.

6.5 Resolving recursion

Recursion in OOX has to be addressed during the coverage computation. Recursion can occur in two ways:

1. Direct recursion, like the Factorial recursive function
2. Indirect recursion, for instance f() calls g() calls h() calls f() as shown in figure 6.1

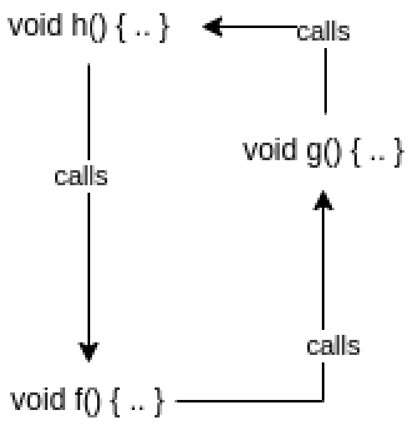


Figure 6.1: An example of indirect recursion

```
int factorial(int n) {      // cost = 3
   if(n == 0) {             // cost = 1 + min(1, 2 + cost(factorial))
      return 1;             // cost = 1
   } else {
      int n_min_1 =
        factorial(n-1);     // cost = 2 + cost(factorial)
      return n * n_min_1    // cost = 1 + cost(factorial)
   }
}
```

Listing 6.3: Example recursion cost of factorial

In both cases, a program point is reached that has been visited before. In other words, the cost depends on the cost of the function we are still investigating. When this is the case, we use the MethodCall variant of the Cost to indicate this. Eventually when the method is fully explored we can find out what the shortest path is of that method. With shortest path we mean the shortest path from the start of that method to either exiting the method (when no uncovered statement is found) or reaching a statement with Cost distance ToUncoveredStatement. Then we can replace the MethodCall in the cost to compute the minimal distance for that method itself.

In the listing 6.3 for instance, the shortest path to the end of the method is the base case which has 3 statements. Thus we can replace the recursive call with 3 to find the minimal distance to uncovered.

6.6 Throw and Catch

In the current version, the minimal distance to uncovered computation has not been implemented for throw and catch statements. The distance of a throw statement is always 1, the distance to a potential catch statement is not considered. In practice this means that the heuristic will not always guide the symbolic verification to the closest uncovered statement surrounding throw and catch statements. We leave this to future work to implement.

Chapter 7

Evaluation

In this chapter, we will evaluate our work by answering two research questions. The first research question will attempt to show how inheritance impacts the path explosion.

Research question 1: How does the addition of inheritance on the programs under verification impact the path explosion problem?

With the second research question we aim to avoid this path explosion by choosing paths more carefully with a heuristic.

Research question 2: What are suitable heuristic to effectively reduce the path explosion in verification?

In order to answer these two research questions we perform an experiment for each research question.

7.1 Impact of inheritance on the path explosion

Experiment 1: Scalability of the addition of inheritance In the first experiment we show the impact that subtyping through inheritance and interfaces and dynamic dispatch has on the path explosion problem. This will be an experiment to measure the scalability of the tool with respect subtyping and dynamic dispatch. In this experiment we will only consider the naive approach of attempting to explore all paths. In the second experiment we will consider a heuristic approach.

We expect that given a program with a symbolic class object with N subtypes, the number of paths to explore will increase exponentially as N increases.

The first experiment is a program containing a `LinkedList` with type `Node`, which has two fields: *value* and *next*. The goal of the program is to find the minimal element of the `LinkedList`. We start off with one `Node` class. Due to lazy initialisation, the *next* field is unitialised until it is accessed in a statement. When it is initialised it can point to any of the previously initialised objects, to null or to a fresh object added to the heap. If we have only one type of `Node`, i.e. it has no subtypes, it can only point to objects of type `Node` in the heap.

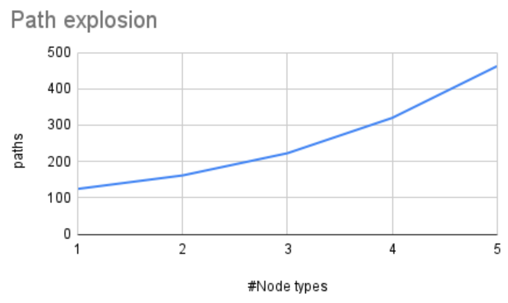
This program is shown in the appendix, listing A.1.

Next we add a subtype of `Node`, `NodeA`, to the program. Now a newly initialised object can point to any of the existing `Node` objects in the heap or any of the existing `NodeA` objects in the heap, or two new objects, one of type `Node` and one of type `NodeA`.

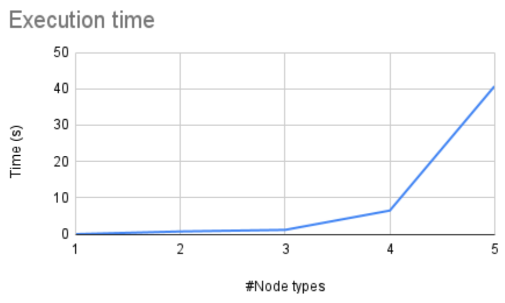
We repeat this process by adding more subtypes of `Node` and observe the path explosion. The experiment is run with a program depth k of 60, on the DepthFirstSearch heuristic. The results are

Experiment 1		
Program	Time (s)	#paths
One node	0.056	125
Two Nodes	0.794	162
Three Nodes	1.232	223
Four Nodes	6.536	320
Five Nodes	40.685	462

Table 7.1: Scalability of subtyping in OOX in a LinkedList



(a) Number of paths over the number of subtypes



(b) Execution time over the number of subtypes

Figure 7.1: Scalability of subtyping in OOX in a LinkedList with respect to paths and time

shown in table 7.1. The program for 3 Nodes is shown in the appendix listing A.2, similarly it is done for 2, 4 and 5 nodes.

The time was measured with a benchmark library Criterion, except for 'Five Nodes' which was run once to show the explosion. The results show that there is an exponential explosion in the execution time. The number of paths explored has a less visible exponential explosion. The reason for this is likely because the actual state explosion is captured in the aliasmap. The aliasmap keeps track for each symbolic object the possible objects it may point to and is able to combine a number of states into one. When an assertion is encountered, branching occurs and the alias map is concretized.

Given these results we can state that the addition of inheritance has an exponential impact on the execution time. In a different implementation without optimisations like the aliasmap we might see stronger exponential growth for the number of paths.

7.2 Reducing the path explosion with a heuristic

In the second experiment we compare the implemented heuristics, to find out if they can effectively reduce the path explosion. A heuristic guides the Symbolic Execution towards certain paths that meet its criterion. We try to answer the second research question; What are suitable heuristic to effectively reduce the path explosion in verification?

In the answering of the research question, we are mostly interested in programs with true positive verdicts, meaning those where there exists a bug that is found by OOX in the given time and program depth. We are unable to search programs up to any depth, within a time limit because of the memory explosion that occurs after a certain path depth in some programs.

We will compare the runtime of the four implemented heuristics of true positives verdicts in a set of programs. It should be noted that because we limit the verification by maximum program depth and not use time as a limitation, if it can be found by one heuristic then all heuristics can find it. All heuristics will eventually find the bug if it can be found within depth k, since each one will search the execution tree up to depth k. In this experiment we will first check the heuristics' ability to catch mutations in two complex object programs and secondly we will check the ability to catch bugs in two java programs which are converted to OOX.

7.2.1 Mutation testing

Experiment 2.1: List Sorting algorithms In this program we verify that three implementations of list sorting algorithms, BubbleSort, MergeSort and InsertionSort, are correctly implemented on Lists of type ArrayList and LinkedList. We apply object oriented features of inheritance and dynamic dispatch in the form of a List type and a Sorter type. The program is partially included in the appendix listing A.3.

The experiment is ran using a program depth k of 110 and a maximum symbolic array size of 5. We set a limit on program depth instead of time because of risk of memory overflow. The downside of this is that we reduce the amount of mutations that can be found.

We aim to compare the performance of the heuristics on invalid assertions, in other words bugs found. Since when a bug is not found by one heuristic, it won't be found by any of the others due to the program depth k limit.

First off we generate mutations for this program. The size of the program determines how many mutations can be generated. In the List Sorting algorithms experiment, up to 700 mutations are generated using the mutation generator from OOX[18]. The setup and assertion part of the program is excluded from the mutations. Due to time constraints we cannot verify all mutations of this experiment and so we randomly sample 20% of these mutations for the experiment, resulting in 129 mutations that compile and run. We run each heuristic on the mutations and measure the runtime.

Of the 129 mutations, 42 were caught invalid [1]. The mutations that were not caught can be explained by one of the following:

1. The allowed path depth is too shallow to find the mutation.
2. The specifications are not strong enough to find the mutation.
3. The program no longer terminates and does not reach an assertion.
4. There is a mistake in the OOX engine that fails to capture it.

We will focus on the mutations that were caught and compare the heuristics based on them. The runtime of each heuristic for the mutations are shown in figure 7.2.

The results show that for this program depth and size, the DFS and MD2U heuristics are outperformed by the RandomPath heuristic. The RoundRobin heuristic is a combination of the MD2U and RandomPath heuristic, and its performance can be explained because it is partially using the RandomPath heuristic.

[1] All heuristics are able to find these mutations eventually, because the exit condition is the maximum achieved program depth and not for example a time limit

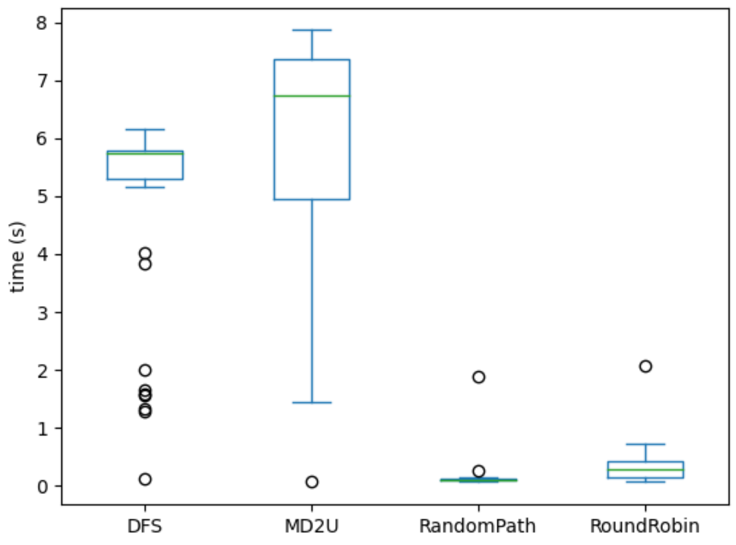


Figure 7.2: Heuristic runtime of the 42 caught mutations on List Sorting algorithms

Experiment 2.2: LinkedList comparison In this program we take a LinkedList with a dynamic value, where the value can be one of the following class types: {`Integer`, `Point2D`, `Point3D`, `LinkedList`}, that all implement the `Value` type. This means that each node in the list has a field 'value' of type `Value`, with at runtime one of the mentioned four class types. The program is included in appendices A.4, A.5.

In the program we take two symbolic `LinkedList` objects and assert that if they are equal they must have the same length. What makes this program challenging is that every value type has to be considered, which causes a combinatorial explosion as the list length increases.

The experiment is run using a program depth k of 90. There are no arrays in this program and so we do not set the symbolic array size. Beyond program depth 90, the program runs out of memory on a valid run.

Again we generate mutations of this program, up to 215 mutations for the experiment. We will not sample a sub-selection of the mutations in this case, as it is doable to experiment on all of these in the benchmark time constraints.

Of the 215 mutations, only 7 were caught invalid. More mutations may have been found if the program depth was increased but this was not possible due to memory explosion.

As mentioned in the previous experiment we focus on the runtime of the heuristics on the caught mutations. Figure 7.3 shows the runtime for the mutations that were caught. For mutations in this program it appears that DepthFirstSearch is performing better than the other heuristics. However we cannot conclude this with much confidence due to the low amount of mutations found.

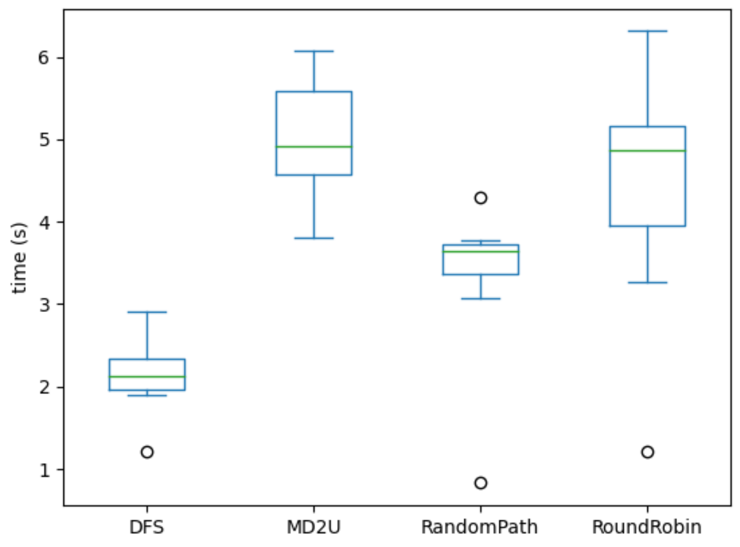


Figure 7.3: Heuristic runtime of the 7 caught mutations on LinkedList comparison

7.2.2 Defects4J bugs

Next we will run the engine with each heuristic on two Java bugs taken from the Defects4J repository[16]. First we describe how the two bugs were obtained, and then we compare each heuristic on the bugs.

Obtaining the benchmark programs The bugs can be found in the Defects4J repository[16], and were selected on feasibility of reproducing the bug in OOX. A program is less feasible if it contains many operations that cannot be reproduced in the current OOX. To still use those programs we have to replace them with operations OOX does support, for example replace strings with integers. This is only possible if the bug is not relying on the replaced operation, and all operations are possible to mimick without influencing the output.

We were able to convert two bugs. The Java bugs were isolated, meaning that any code unrelated to the bug was removed, and converted to OOX. Due to manual conversion work and limited expressions in OOX, the amount of code we had to remove is not ideal. The more code is removed the less authentic the bug becomes.

Experiment 2.3: Defects4J Collections 25 This experiment contains a Java Collections bug discovered by a test. The bug involves an Iterator, `CollatingIterator` whose `Comparator` field is null, which causes a `NullPointerException` when the iterator is converted to a List. The program is 350 lines after it was converted to OOX. We have changed the input of the test to be a symbolic array of up to size 5 and run the program with a maximum program depth k of 1000. This depth was chosen to not limit the engine in finding the bug, since there were no memory issues in this case.

The results are shown in table 7.2a. In this experiment the DepthFirstSearch heuristic is performing better than the other heuristics, with the MD2U heuristic performing worst. This shows that the tradeof of computing the minimal distance to each statement is not worth it in this case.

Experiment 2.4: Defects4J Compress 3 This program is a bug from the Compress project in Defects4J, exposed by a test. In the program an ArchiveOutputStream is constructed from a Factory, onto which an archive is put. The program is meant to throw an exception when finish() is called on the OutputStream. There are four different implementations of ArchiveOutputStream defined in the project, and one of them does not meet this specification.

We change the test case to verify that any of type ArchiveOutputStream created by the factory will throw the exception, and pass it a symbolic ArchiveOutputStream. We run the test case with program depth k of 1000, and a maximum symbolic array size of 5.

The results are shown in table 7.2b.

Heuristic	Time (s)
DepthFirstSearch	0.204
MD2U	1.721
RandomPath	0.367
RoundRobin	0.729

(a) Collections 25 heuristic runtime

Heuristic	Time (s)
DepthFirstSearch	0.195
MD2U	0.272
RandomPath	0.208
RoundRobin	0.247

(b) Compress 3 heuristic runtime

Table 7.2: Detection runtime of two Defect4J bugs in OOX

7.3 Discussion

The first experiment showed that increasing the number of subtypes in a LinkedList structure exponentially impacts the scalability of the program. We argue that a LinkedList is representative for other complex datastructures, and thus that increasing the subtypes of values in any (recursive) datastructure will exponentially increase the runtime, as the size of the datastructure increases.

In all benchmarks of the second experiment, either DepthFirstSearch or RandomPath outperformed the other heuristics in runtime. In the first mutation experiment 2.1, RandomPath far outperformed the other heuristics. Because RandomPath is more likely to search states higher in the execution tree, we think that the mutations that were found are shallow, meaning they are found quickly when looking in a breadth first search manner. The DepthFirstSearch heuristic may have spent more time looking for the bug somewhere deeper in the execution tree and is likely why it takes more time.

The MD2U heuristic seems to also not be favored in the mutation tests, potentially increasing the runtime cost by having to compute the minimal distance for each statement every execution step. Even though we have caching for the MD2U heuristic, this will only start to benefit the heuristic when multiple methods have been fully explored. At that point the bug may have already been found.

RoundRobin combines the RandomPath with the MD2U by alternating between them at every step. It is likely that the heuristic gains most of its performance from the RandomPath heuristic here.

In the second mutation experiment 2.2, DepthFirstSearch is better suited than RandomPath, this seems to indicate that the depth of the mutations are deeper and found earlier. We cannot conclude this with much confidence due to the low number of mutations found.

On the Defect4J bugs, the DFS heuristic performed best, followed by the RandomPath heuristic. The Collections 25 bug in experiment 2.3 shows that the distance computations for the MD2U heuristic are expensive on a larger program and in this case not worth the tradeoff. In the Compress 3 bug in experiment 2.4, the MD2U heuristic is not far behind the others. This can be explained by that the bug contains smaller methods than the Collections 25 bug and is easier to explore by the heuristic.

We can conclude that which heuristic is more effective in reducing the path explosion depends on the program size and depth of the bug. We measure reducing the path explosion by decreasing the time it takes to find bugs and mutations. We can conclude that the DFS and RandomPath are the more effective heuristics in the programs we have experimented on. The difference between the heuristics is the depth to which they explore earlier on, which makes us think that their performance relies on the shallowness (or depth) of the bug. The MD2U heuristic has not been shown to be worth the computing tradeoff for performance for these programs. In future work we may be able to explore larger programs where a variation of the MD2U heuristic can be put to better use.

Chapter 8

Related work

8.1 Object oriented IVL

To the best of our knowledge there has not been any intermediate verification language other than OOX with native support for inheritance. There have been other IVLs that do not directly support inheritance features but may be able to represent them. Examples of these are Boogie [20] by Microsoft and the ML dialect WhyML[14] which is used in Why3.

Having native support for inheritance may make it easier to have heuristics specifically aimed at reducing path explosion for inheritance.

Another IVL that can represent object oriented programs, without inheritance, is the SimpIL language[27]. This was later extended by Pham et al.[25] and used in their test generation engine based on symbolic execution.

8.2 Search heuristic

There have been many attempts at reducing the path explosion using heuristic. We will first distinguish two categories of heuristic which we have also implemented in OOX.

Random Path Random pathing seems straightforward but there are multiple approaches possible with different properties. One described in the survey of search heuristic by LIU et al.[22], follows a single path at once, making a random choice when a conditional branch is reached. This is more likely to explore different random paths deeper into the program.

The approach by KLEE and also by OOX is different because it does not track a single path, but maintains an execution tree. At every step the heuristic walks the execution tree from the root and at every conditional branch it makes a stochastic choice until an unfinished state is found. This approach is more likely to explore earlier states in the execution tree first.

Figure 8.1 shows how this might look, where the first approach follows fewer paths deeper into the program, and the latter approach randomly chooses from the states available, more likely to choose states higher in the execution tree.

Maximizing Coverage There have been multiple heuristic that aim to maximize code coverage, like the MD2U heuristic implemented in OOX in this project. KLEE has the Coverage-Optimized Search heuristic, in which they combine the minimum distance to an uncovered instruction with the

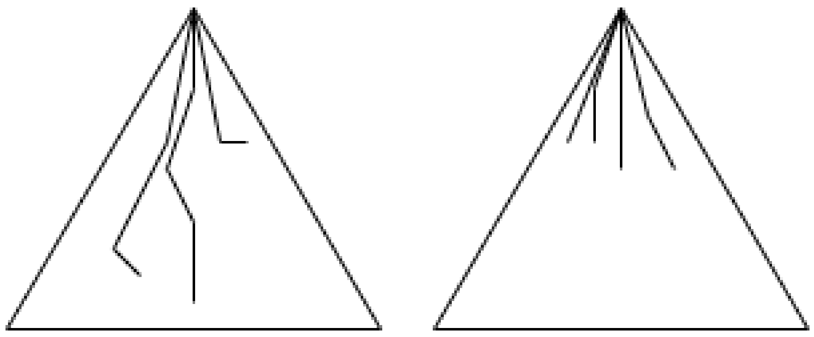


Figure 8.1: The execution tree visualised for two different random heuristic approaches

callstack and whether the state recently uncovered new code [8]. This is effectively a combination of MD2U with another heuristic to ensure that the engine does not repeatedly try to reach a single statement, ignoring the remainder of the program. In OOX the MD2U algorithm by itself may get stuck as described and should ideally be combined with another heuristic.

EXE[9] and Mayham[10] are two tools that make use of symbolic execution to find bugs in C and/or binary executables. These tools have both implemented a heuristic whose goal it is to maximize coverage, among others.

In the dynamic symbolic execution research area some heuristic for maximizing coverage have been used as well. For instance The Generational Strategy by SAGE[22].

A different approach to maximizing coverage is the approach by Li et al. (2013) [21], which prioritizes less traveled parts of the program using a frequency distributions of explored length-n subpaths.

Other heuristic variations A more general approach by Sooyoung et al. [11] is to automatically learn heuristics rather than manually craft them. They achieve this by defining a set of state features that say something about a path, such as how often a branch has been selected, whether it is a deep branch but also if it has been uncovered or not. They then present an algorithm to automatically learn a heuristic from these search heuristics. Their algorithm was able to automatically create a heuristic that increased the average branch coverage, while at the same time showing that there was no other heuristic that performed well on all of their test programs. They claim that manually crafted heuristics are likely to be suboptimal and unstable.

In our experiment we found that the heuristics were also not performing consistently across different programs, strenghtening this claim. A future work in OOX might therefore be to expand on the state features of OOX to allow creating such an automatically learned heuristic.

Chapter 9

Conclusion and future work

In this thesis project we have expanded OOX with subtyping through inheritance and have added initial work towards heuristics for reducing the path explosion problem. Implementing support for inheritance features such as class inheritance and interface implementation, allowed dynamic dispatch to occur. This allowed for more classes of programs from object oriented languages to be verified by OOX.

In our first experiment we have then shown that this extension of OOX further adds another dimension into which the path explosion can occur, namely the subtyping and dynamic dispatch of symbolic objects. Particularly in cases with repeated lazy initialisation of such objects like in (recursive) datastructures. We have shown that an exponential explosion occurs in the runtime of the verification as more subtypes are added to the value type in a datastructure.

Furthermore we have researched and implemented initial heuristics aiming to reduce the path explosion problem. We have implemented three heuristics inspired by existing research, on top of the already used depth-first-search heuristic. First we have implemented the minimal-distance-to-uncovered heuristic, whose criterion is to maximise statement coverage. Next we have implemented a random-path heuristic whose main usefulness was expected to be shown when combined with different heuristics in a round-robin manner. However it also performed well on its own in programs with shallow bugs. Finally we have implemented a way to combine multiple heuristics in a round-robin manner and applied this by combining our previous two heuristics.

We have evaluated the heuristics on two manually crafted programs and two bugs from the Defects4J repository. The first two programs were used to evaluate the heuristics by adding mutations. The Defects4J programs were converted to OOX and similarly tested by the heuristics. We show that all heuristics are complete with respect to the original heuristic when verified up to path depth k. The results showed that the original depth first search and random path heuristic outperform the minimal-distance-to-uncovered and round-robin heuristic on all tested programs. Due to memory limitations with OOX we were not able to verify programs deeply, which is where we believe a variation of MD2U would prove more useful.

9.1 Future work

Expand on the heuristics or implement new ones In future work we could implement new heuristics, or expand on the existing ones. We could also attempt to introduce one of the techniques other than heuristics to reduce path explosion that were mention in chapter 3, such as Separation Logic or BLISS.

Improve usability of OOX In future work we may extend OOX with support for more operators to allow more programs to be represented. This will allow for easier and more accurate comparison with other tools. Examples are String operations and Bitshifting operations. In many open source bug repositories these operations were found to be used often, limiting the number of bugs we could reproduce in OOX.

Similarly OOX would benefit from a tool to help convert Java (or any different object oriented language) to OOX. It would make reproducing bugs easier and make OOX more practical to use.

There is also the memory explosion limitation within OOX on lazy initialised recursive objects, this explosion is not necessarily caused by the path explosion entirely. With array initialised objects this issue does not occur, suggesting that the aliasmap may be the cause of the explosion. In future work we could remove the aliasmap or find an alternative. It would likely result in more states that would otherwise have been captured with the aliasmap but this may fix the memory issue.

Bibliography

[1] Paul Ammann and Jeff Offutt. "Introduction to Software Testing". In: New York: Cambridge University Press, 2008. Chap. 7, 1.

[2] Ariel Assaraf. *This is what your developers are doing 75% of the time, and this is the cost you pay.* 2015. URL: `https://coralogix.com/blog/this-is-what-your-developers-are-doing-75-of-the-time-and-this-is-the-cost-you-pay/` (visited on 10/18/2022).

[3] Roberto Baldoni et al. "A Survey of Symbolic Execution Techniques". In: *ACM Comput. Surv.* 51.3 (May 2018). ISSN: 0360-0300. DOI: `10.1145/3182657`. URL: `https://doi.org/10.1145/3182657`.

[4] Clark Barrett, Daniel Kroening, and Thomas Melham. *Problem Solving for the 21st Century: Efficient Solvers for Satisfiability Modulo Theories.* Tech. rep. 3. Knowledge Transfer Report. London Mathematical Society, Smith Institute for Industrial Mathematics, and System Engineering, June 2014. URL: `http://www.cs.stanford.edu/~barrett/pubs/BKM14.pdf`.

[5] Dirk Beyer. "Progress on Software Verification: SV-COMP 2022". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by Dana Fisman and Grigore Rosu. Cham: Springer International Publishing, 2022, pp. 375–402. ISBN: 978-3-030-99527-0.

[6] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. "JBSE: A Symbolic Executor for Java Programs with Complex Heap Inputs". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering.* FSE 2016. Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 1018–1022. ISBN: 9781450342186. DOI: `10.1145/2950290.2983940`. URL: `https://doi.org/10.1145/2950290.2983940`.

[7] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. "Symbolic Execution of Programs with Heap Inputs". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 602–613. ISBN: 9781450336758. DOI: `10.1145/2786805.2786842`. URL: `https://doi.org/10.1145/2786805.2786842`.

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation.* OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224.

[9] Cristian Cadar et al. "EXE: Automatically Generating Inputs of Death". In: *ACM Trans. Inf. Syst. Secur.* 12.2 (Dec. 2008). ISSN: 1094-9224. DOI: `10.1145/1455518.1455522`. URL: `https://doi.org/10.1145/1455518.1455522`.

[10] Sang Kil Cha et al. "Unleashing Mayhem on Binary Code". In: *2012 IEEE Symposium on Security and Privacy.* 2012, pp. 380–394. DOI: `10.1109/SP.2012.31`.

[11] Sooyoung Cha et al. "Enhancing Dynamic Symbolic Execution by Automatically Learning Search Heuristics". In: *IEEE Transactions on Software Engineering* 48.9 (2022), pp. 3640–3663. DOI: 10.1109/TSE.2021.3101870.

[12] Dino Distefano and Matthew J. Parkinson J. "JStar: Towards Practical Verification for Java". In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications.* OOPSLA '08. Nashville, TN, USA: Association for Computing Machinery, 2008, pp. 213–226. ISBN: 9781605582153. DOI: 10.1145/1449764.1449782. URL: https://doi.org/10.1145/1449764.1449782.

[13] Samuel Feitosa, Rodrigo Ribeiro, and André Du Bois. "Formal Semantics for Java-like Languages and Research Opportunities". In: *Revista de Informática Teórica e Aplicada* 25 (Sept. 2018), p. 62. DOI: 10.22456/2175-2745.80912.

[14] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3—where programs meet provers". In: *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22.* Springer. 2013, pp. 125–128.

[15] Ralph Johnson and Brian Foote. "Designing Reusable Classes". In: *Journal of Object-Oriented Programming* (1991). URL: https://www.cse.msu.edu/~cse870/Input/SS2002/MiniProject/Sources/DRC.pdf.

[16] René Just, Darioush Jalali, and Michael D. Ernst. "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis.* ISSTA 2014. San Jose, CA, USA: Association for Computing Machinery, 2014, pp. 437–440. ISBN: 9781450326452. DOI: 10.1145/2610384.2628055. URL: https://doi.org/10.1145/2610384.2628055.

[17] Sarfraz Khurshid, Corina S. PĂsĂreanu, and Willem Visser. "Generalized Symbolic Execution for Model Checking and Testing". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by Hubert Garavel and John Hatcliff. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 553–568. ISBN: 978-3-540-36577-8.

[18] Stefan Koppier. "The Path Explosion Problem in Symbolic Execution An Approach to the Effects of Concurrency and Aliasing". MA thesis. Utrecht University, 2020. URL: https://studenttheses.uu.nl/handle/20.500.12932/35856.

[19] Quang Loc Le et al. "A Decidable Fragment in Separation Logic with Inductive Predicates and Arithmetic". In: *Computer Aided Verification.* Ed. by Rupak Majumdar and Viktor Kunčak. Cham: Springer International Publishing, 2017, pp. 495–517. ISBN: 978-3-319-63390-9.

[20] K Rustan M Leino. "This is Boogie 2". In: (Jan. 2008).

[21] You Li et al. "Steering symbolic execution to less traveled paths". In: *ACM SIGPLAN Notices* 48 (Nov. 2013), pp. 19–32. DOI: 10.1145/2544173.2509553.

[22] Yu Liu, Xu Zhou, and Wei-Wei Gong. "A Survey of Search Strategies in the Dynamic Symbolic Execution". In: *ITM Web of Conferences* 12 (Jan. 2017), p. 03025. DOI: 10.1051/itmconf/20171203025.

[23] Leonardo de Moura and Nikolaj Bjørner. "Z3: an efficient SMT solver". In: vol. 4963. Apr. 2008, pp. 337–340. ISBN: 978-3-540-78799-0. DOI: 10.1007/978-3-540-78800-3_24.

[24] Long H. Pham et al. "Concolic Testing Heap-Manipulating Programs". In: *Formal Methods – The Next 30 Years.* Ed. by Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira. Cham: Springer International Publishing, 2019, pp. 442–461. ISBN: 978-3-030-30942-8.

[25] Long H. Pham et al. “Enhancing Symbolic Execution of Heap-Based Programs with Separation Logic for Test Input Generation”. In: *Automated Technology for Verification and Analysis.* Ed. by Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza. Cham: Springer International Publishing, 2019, pp. 209–227. ISBN: 978-3-030-31784-3.

[26] Nicolás Rosner et al. “BLISS: Improved Symbolic Execution by Bounded Lazy Initialization with SAT Support”. In: *IEEE Transactions on Software Engineering* 41.7 (2015), pp. 639–660. DOI: 10.1109/TSE.2015.2389225.

[27] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. “All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)”. In: *2010 IEEE Symposium on Security and Privacy.* 2010, pp. 317–331. DOI: 10.1109/SP.2010.26.

[28] Dominic Steinhöfel. “Symbolic Execution: Foundations, Techniques, Applications, and Future Perspectives”. In: *The Logic of Software. A Tasting Menu of Formal Methods: Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday.* Ed. by Wolfgang Ahrendt et al. Cham: Springer International Publishing, 2022, pp. 446–480. ISBN: 978-3-031-08166-8. DOI: 10.1007/978-3-031-08166-8_22. URL: https://doi.org/10.1007/978-3-031-08166-8_22.

[29] *The 2021 State of Testing report is out.* 2021. URL: https://www.practitest.com/resource/state-of-testing-report-2021/ (visited on 10/18/2022).

Appendix A

Benchmark Programs

```
class Node {
    int value ;
    Node next ;

    bool member(int x) {
        assume this != null;
        int v := this.value ;
        if(x==v) return true ;
        else {
            Node n := this.next ;
            if (n == null) {
                return false;
            } else {
                bool b := n.member(x) ;
                return b ;
            }
        }
    }

    Node min()
    {
       Node p := this ;
       Node min := p ;
       int minval := min.value ;

       while ( p != null) {
          int value := p.value ;
          if ( value < minval) {
               min := p ;
               minval := value ;
          }
          p := p.next;
       }
       return min ;
    }
}

class Main {
    static void test2(Node node)
        requires(node != null)
        exceptional(false)
    {
        Node min := node.min() ;
        int x := min.value ;
        bool ok := node.member(x) ;
        assert ok ;
    }
}
```

Listing A.1: Experiment 1 initial program

```
class Node {
    ...
}

class NodeB extends Node {
    int value2 ;
}

class NodeC extends Node {
    int value3 ;
}


class Main {
    ...
}
```

Listing A.2: Experiment 1 - 3Node program

```
interface List {
    int get(int i);
    void set(int i, int element);
    int length();
    int[] toArray();
    bool repOk();
}

class IntArray implements List {
    int[] ints;
   ...
}


class Node implements List {
    int value ;
    Node next ;
   ...
}

interface Sorter {
    int[] sort(int[] array);
}

class BubbleSort implements Sorter {
    List sort(List array)
        requires(array != null)
        ensures(true)
        exceptional(false)
    {
        ...
    }
}



class MergeSort implements Sorter {
    List sort(List array)
        requires(!(array == null))
        exceptional(false)
    {
        ...
    }

}

class InsertionSort implements Sorter {
    List sort(List array)
        requires(!(array == null))
        exceptional(false)
    {
       ...
    }
}

class Main
{
    static void test(List list, Sorter sorter)
        requires(list != null)
    {
        bool listIsOk := list.repOk();
        assume listIsOk;
        int N := 5;
        list := sorter.sort(list);

        int[] a := list.toArray();
        assert forall v , i : a : (forall w , j : a : i < j ==> v <= w);
    }
}
```
Listing A.3: Experiment 2.1 List sorting Algorithms

```
interface Value {
    bool equals(Value other);
}
class Integer implements Value {
    int value;

    bool equals(Value other) {
        if (other instanceof Integer) {
            Integer otherInt := (Integer) other;
            int thisValue := this.value;
            int otherValue := otherInt.value;
            return thisValue == otherValue;
        }
        return false;
    }
}
class Point implements Value {
    int x;
    int y;

    bool equals(Value other) {
        if (other instanceof Point) {
            Point otherPoint := (Point) other;
            int thisX := this.x;
            int otherX := otherPoint.x;
            int thisY := this.y;
            int otherY := otherPoint.y;
            return thisX == otherX && thisY == otherY;
        }
        return false;
    }
}

class Point3 extends Point {
    int z;

    bool equals(Value other) {
        if (other instanceof Point3) {
            Point3 otherPoint := (Point3) other;
            int thisX := this.x;
            int otherX := otherPoint.x;
            int thisY := this.y;
            int otherY := otherPoint.y;
            int thisZ := this.z;
            int otherZ := otherPoint.z;
            return thisX == otherX && thisY == otherY && thisZ == otherZ;
        }
        return false;
    }
}
```

Listing A.4: Experiment 2.2 LinkedList comparison part 1/2

```
class Node implements Value {
    Value value ;
    Node next ;

    Node(Value[] a, int i) {
        if (i >= 0 && i < #a) {
            this.value := a[i];
            if (i + 1 < #a) {
                this.next := new Node(a, i + 1);
            }
        }
    }
    bool member(Value x) {
        Value v := this.value ;
        bool xEqualsV := x.equals(v);
        if(xEqualsV) return true ;
        else {
            Node n := this.next ;
            if (n == null) {
                return false;
            } else {
                bool b := n.member(x) ;
                return b ;
            }
        }
    }
    int length() {
        Node n := this.next ;
        if (n==null) return 1 ;
        else {
            int k := n.length() ;
            return k+1 ;
        }
    }
    bool equals(Value other) {
        if (other instanceof Node) {
            Node a := this;
            Node b := (Node) other;

            while (!(a==null && b == null)) {
                if (a == null || b == null) {
                    return false;
                }
                Value aValue := a.value;
                Value bValue := b.value;
                bool aEqualsB := aValue.equals(bValue);
                if (!aEqualsB) {
                    return false;
                }
                a := a.next;
                b := b.next;
            }
            return true;
        }
        return false;
    }
}


class Main {

    static void test(Node a, Node b) {
        bool aEqualsB := a.equals(b);
        int aLength := a.length();
        int bLength := b.length();

        assert aEqualsB ==> aLength == bLength;
    }
}
```

Listing A.5: Experiment 2.2 LinkedList comparison part 2/2