DOI: 10.17587/prin.13.483-496

Reanimator: from Test Data to Code and Back

A. M. Abdullin, Post-Graduate Student, Senior Lecturer, azat.aam@gmail.com,
V. M. Itsykson, PhD, Associate Professor, vlad@icc.spbstu.ru,
Peter the Great St. Petersburg Polytechnic University (SPbPU), Saint-Petersburg, 195251, Russian Federation

Corresponding author:

Azat M. Abdullin, Post-Graduate Student, Senior Lecturer,
Peter the Great St.Petersburg Polytechnic University (SPbPU), Saint-Petersburg, 195251, Russian Federation
E-mail: azat.aam@gmail.com

Received on September 09, 2022
Accepted on October 10, 2022

State-of-the-art automatic testing tools can efficiently detect various kinds of errors in software projects; however, these errors often cannot be automatically extracted to a standalone reproducing test case, a feature valuable for the purposes of further software maintenance. Objects present the main difficulty, as generating code to construct an object given its state is hard, due to the encapsulation principle.

In this paper we present an approach called Reanimator that, given an object representation, is able to generate a valid code snippet which constructs this object using its publicly available API. Reanimator can be applied to any automatic testing tool to help it generate reproducing test cases for detected failures.

We implemented our approach as a part of an automatic testing tool called Kex and evaluated it on a number of open-source projects from GitHub. Evaluation results showed that our approach is compatible with the state of the art techniques.

Keywords: *automatic test generation, symbolic execution, software testing*

For citation:

Abdullin A. M., Itsykson V. M. Reanimator: from Test Data to Code and Back, *Programmnaya Ingeneria,* 2022, vol. 13, no. 10, pp. 483—496.

УДК 004.05

А. М. Абдуллин, аспирант, ст. преподаватель, azat.aam@gmail.com,
В. М. Ицыксон, канд. техн. наук, доц., vlad@icc.spbstu.ru,
Санкт-Петербургский политехнический университет Петра Великого

Реаниматор: от тестовых данных к коду и обратно

Современные инструментальные средства для автоматической генерации тестов способны эффективно находить различные виды ошибок в программном обеспечении. Однако эти ошибки зачастую не могут быть автоматически преобразованы в воспроизводимый тестовый пример. Такое преобразование позволило бы значительно упростить процесс поддержки разрабатываемого программного обеспечения. Наибольшее сложным вопросом решения этой задачи является восстановление экземпляров классов, так как в силу принципа инкапсуляции очень сложно сгенерировать программный код, который позволяет воспроизвести необходимое состояние объекта.

В статье описан оригинальный, именуемый Реаниматором, авторский подход, который, получив описание необходимого состояния экземпляра класса в каком-либо формате, формирует программный код, результатом которого является создание необходимого экземпляра в нужном состоянии с использованием исключительно публичного интерфейса класса. Такой подход может быть использован для создания тестов в любом инструментарии для автома-

тической генерации тестов, воспроизводящих найденные ошибки. Предлагаемый подход был разработан и реализован в составе инструментального средства Kex для автоматической генерации тестов и протестирован на наборе проектов с открытым исходным кодом. Тестирование показало применимость и конкурентоспособность предлагаемого подхода в сравнении с современными методами генерации тестовых примеров.

Ключевые слова: *автоматическая генерация тестов, символьное исполнение, тестирование программного обеспечения*

Introduction

Software affects almost every part of human life in the modern world. People are surrounded by computers and electronic devices which control transport, business, medicine, etc. Software defects in these computer systems may lead to fatal consequences, and software companies use various methods of software quality assurance [1, 2] to detect and fix errors on all stages of development.

One of the most popular methods of software quality assurance is software testing. Testing involves executing the program (or one of its components) on a set of predefined inputs to detect incorrect behavior. Software testing has proved its efficiency, but it has one significant weakness: creating software tests is a very tedious process requiring a lot of time-consuming manual work [3].

Automatic test generation is one of the attempts to overcome this weakness; being an extension of automatic software testing, it not only finds the software bugs, but also generates reproducing test cases, usable for further software maintenance. Generated tests are often included in the regression test suite to ensure that the found problems will not reappear in the future. There are a lot of methods of automatic test generation, based on random testing [4], symbolic execution [5], hybrid search-based approaches [6], etc. These methods may use different strategies for test generation, but in the end they all produce error-inducing inputs by either operating on *test code* or on *test data*.

Methods from the first group [6—8], which primarily use flavors of random search (e. g., evolutionary algorithms), operate on code in the form of function or method calls and work by extending the generated call sequence with additional calls on each search step. This allows these methods to produce a code snippet which can be immediately used in the testing environment; however, their random nature has a direct impact on their bug-revealing efficiency. Another downside of these approaches is that they are dynamic by nature and cannot be used in purely static setup.

Methods from the second group [5, 9, 10], based on random data generation or more complex approaches, such as symbolic execution, operate on data rather than code and try to generate interesting program inputs. Therefore, to turn the error-inducing input into a test case, one needs to generate a code snippet which constructs the required data. For programming languages following encapsulation principle (e. g., object-oriented ones), this leads to the following problem: one cannot directly build an object from its internal data (aka "encapsulation problem"). This forces the test generation tools to use alternative ways to create test cases, such as reflection [11].

Post-failure debugging techniques analyze the program data after the failure and try to lift the data into a test case. Given a bug report in some form (e. g., core dump, stack trace, etc.), they attempt to generate program inputs (execution traces, test cases, etc.) which reproduce the crash. However, they are also affected by the "encapsulation problem", if the program input is complex.

Record and replay [12, 13] crash replication approaches fall somewhere in between these two groups. These methods capture program interactions at runtime (via instrumentation) and store them as execution traces. Test cases can be later recreated from these traces, based on the sequences of program events, and these sequences correspond to test code fragments. However, one may need to generate the test data for these fragments, which brings us back to the "encapsulation problem". Additionally, the need for instrumentation often leads to a performance overhead.

This paper presents an approach we call Reanimator which solves the "encapsulation problem" and allows one to automatically generate valid code snippets to create a given target object, using its *publicly available API*; the generated snippets can be then used to build reproducing test cases. The approach is general and can be used not only to solve the "encapsulation problem" for data-based test generation methods, but also to enhance code-based test generation, e. g., by using the object creation snippets as primitive elements during the call sequence search process.

Reanimator is built on an original backward search algorithm, which works by gradually reducing the search space of applicable actions initializing one or more object fields. At each step of the search algorithm, we take the target object T and construct another source object S, so that all fields of S are equal to corresponding fields of T except for one, set to the default value of its type. We then use symbolic execution to check if it is possible to execute an action on S to produce the target object T. If it is possible, we save the information about found action

and its arguments, and recursively continue the search on *S*, until we find a constructor-like action, meaning we can create *T* without any source object. The resulting action sequence can be later transformed into a valid test case.

We implemented our approach as a prototype in an automatic testing tool provided in Kex [14] platform and evaluated it on a number of real-world projects. The evaluation consists of three scenarios: extracting errors found by the white-box fuzzer component of Kex to the reproducing test cases, creating test cases for generating random objects and integrating Reanimator approach with the TARDIS [15] tool. Evaluation results show Reanimator can successfully and efficiently generate 62.3 % of target objects on average and it can be compatible with the dynamic approach of TARDIS. Based on the evaluation we can say the proposed approach is applicable for automatic test case generation.

1. Related work

Automatic generation of reproducing code snippets (i. e., test cases) from error-inducing inputs is a problem that has been addressed in areas of automatic test generation, debugging and crash reproduction. Different approaches have different capabilities for generating test cases, here we talk about the most important ones.

Some of the approaches [6—8, 16] are based on the idea of constructing the test case by gradually adding new code expressions from the list of available expressions until the user-specified stop criterion is reached (not unlike a form of program synthesis [17]). While these approaches are great at generating test cases (which they perform by construction), the generation process is usually random to some extent and may not be able to create test cases that cover complex parts of the target program. Also, those approaches are based on dynamic analysis of the program, as they are usually coverage driven.

Symbolic execution based approaches differ on how exactly they use symbolic execution for test case generation. Some of them (i. e., JBSE [5]) find interesting object states via symbolic execution and generate test cases using reflection to reconstruct the target objects. While reflection does sidestep the "encapsulation problem", it has the following downsides:

- reflection-based tests are hard to comprehend and maintain;
- using reflection breaks encapsulation and allows generating an object that cannot be created via its publicly available API.

Other symbolic execution approaches (e. g., Symstra [9]) are exploring the test case search space for an arbitrarily selected subset of possible call sequences. During the exploration, they symbolically execute a call sequence and, if the execution is interesting, generate concrete values for the arguments. However, Symstra is limited in that it only supports primitive types; the authors argue constructing complex objects can be done either using reflection or by viewing it as a nested Symstra problem, which may increase the search space.

JBSE [5] symbolic execution engine is also used as a basis for SUSHI [10] and TARDIS [15] test generation tools. SUSHI uses JBSE to compute interesting path conditions of target program and converts them into a fitness function. This fitness function is then used in the search algorithm which generates test cases satisfying the fitness function and, therefore, the original path conditions; to implement search, SUSHI uses a custom version of EvoSuite [6]. The authors also propose a way to encode the program input properties as external specifications, which can help SUSHI to identify the unsatisfiable path conditions in advance and improve its efficiency.

TARDIS [15] is an extension of SUSHI that uses concolic testing [18] for exploring path conditions of the target program, which are then also handled via a fitness function. The use of concolic testing improves TARDIS performance (compared to SUSHI) in cases when there is no user provided specification. The main disadvantage of these test generation approaches is that the search algorithm is separated from the symbolic execution engine, making it extremely hard to efficiently evaluate the feasibility of interesting path conditions. The results presented in [10, 15] and our experiments show that most of the time is spent on search-based test case generation, compared to time taken by the symbolic execution used to explore interesting path conditions.

Crash replication is another area interested in test case generation. Approaches in this field can be divided into two main categories: record and replay approaches and post-failure techniques. The former (SCAPE [12], CR [19], JINSI [20], BugRedux [21], ReMinds [22], GenThat [13]) are all based on capturing program execution traces via instrumentation and replaying these traces later, either in a special execution environment or as generated unit tests. However, the unit tests created do not support complex data or utilize reflection; more so, program instrumentation can lead to additional performance overhead.

Post-failure techniques [23—27] analyze execution data (e. g., core dumps) after the program has already crashed. RECORE [24] uses core dumps to generate a fitness function and provides it to evolutionary search-based algorithm to generate the test cases, somewhat similar to SUSHI (and having the same disadvantages). SBFR [27] (search-based failure reproduction) takes a failing program, a grammar describing the complete program input, and a (partial) call sequence and uses genetic algorithm to generate failing inputs. While such approach works for functional tests, the need to create an input grammar for separate unit tests adds a significant overhead.

ESD [23] is a technique for automated debugging which, given a program and a bug report, uses symbolic execution to synthesize a program execution which reproduces the required bug. The execution can then be replayed in a special ESD playback environment to support classic debugging techniques, but it does not provide a way to generate a reproducing test case. DESCRY [25] is an automated tool for reproducing system-level concurrency failures based on log messages collected from the running program. Once again, it is aimed at functional-level analysis and does not provide utilities to generate separate unit tests.

Summary. As one can see, there already exists a large body of work on automatic generation of test cases. Existing methods belong to one of the following high-level groups:

- complete test code generation via random search;
- end-to-end test data generation, which primarily supports functional-level tests with primitive data types;
- unit-level test data generation, which comes in two flavors:
  - using reflection to create the required data;
  - using search-based approaches to synthesize code creating the required data.

For unit tests, reflection-based approaches sidestep the "encapsulation problem", but create hard to maintain and even incorrect (with reference to program API) tests. Search-based approaches do not have this problem, but sacrifice performance with the separation between symbolic execution (used for data exploration) and code synthesis (used for data generation).

Limitations of both approaches have inspired us to attempt to remove the separation and use symbolic execution for both data exploration and generation.

2. Motivating example

Let us show a motivating example of how current test generation tools support complex data generation. Consider a simple Java class ListExample given in Listing 1. It defines an inner class Point with two integer coordinates and defines a method foo which accepts ArrayList<Point> as its argument. This method fails on line 18, when argument *a* has a specific shape.

Imagine we want to generate a test case for this example. Automatic test generation tools based on random- and search-based code generation will have difficulties finding this failure, as the probability they will synthesize code, which creates an object of required shape to trigger the bug, is low. Tools using symbolic execution, on the other hand, will be able to find the error-inducing input, but will encounter some problems generating a standalone test case. As discussed in the previous section, one of the options is to use reflection. Listing 2 shows a part of the test suite for our example program generated by JBSE [5] tool. This example

```
1. package test;
2. import java.util.ArrayList;
3.
4. public class ListExample {
5.     public static class Point {
6.         private int x;
7.         private int y;
8.         public Point(int x, int y) {
9.             this.x = x; this.y = y;
10.        }
11.        public int getX() { return x; }
12.        public int getY() { return y; }
13.    }
14.    public void foo(ArrayList<Point> a) {
15.        if (a.size() == 2) {
16.            if (a.get(0).getX() == 10) {
17.                if (a.get(1).getY() == 11) {
18.                    throw new IllegalStateException();
19.                }
20.            }
21.        }
22.    }
23.}
```

Listing 1. A program with a hard-to-find bug

```
public class TestSuite {
    ...
    public void test4() {
        this.nullObjectFields = new HashSet<>();
        ...
        test.ListExample __ROOT_this = (test.ListExample)
            newInstance("test.ListExample");
        java.util.ArrayList __ROOT_a = (java.util.ArrayList)
            newInstance("java.util.ArrayList");
        new AccessibleObject(__ROOT_a)
            .set("java/util/ArrayList:size", 2L);
        new AccessibleObject(__ROOT_a)
            .set("java/util/ArrayList:elementData",
                newArray("java.lang.Object", 2L));
        new AccessibleObject(__ROOT_a)
            .set("java/util/ArrayList:elementData[0]", null);
        this.nullObjectFields.add(new ObjectField(
            __ROOT_a, "java/util/ArrayList:elementData[0]"));
        __ROOT_this.foo(__ROOT_a);
    }
    ...
}
```

Listing 2. Example of a test case generated by JBSE

highlights the problems with the use of reflection we mentioned earlier. The total size of a single generated test is more than 400 lines, and the generated code is complex and hard to understand and to maintain.

Tools, which use search-based approaches to create the test code for interesting data (such as SUSHI or TARDIS), encounter performance problems. Our experiments show SUSHI is not able to generate a test case triggering the bug in line 18 with a time budget of 20 min. TARDIS is more successful in this case and can generate a test case in 2 min (Listing 3). These tools require more time for test case generation; however, the created tests do not break encapsulation and are easier to comprehend and maintain if needed.

```
public class ListExample_0_Test extends ListExample_0_Test_scaffolding {
  @Test(timeout = 4000)
  public void test0() throws Throwable {
    ListExample listExample0 = new ListExample();
    ArrayList<Point> arrayList0 = new ArrayList<Point>();
    int int0 = 10;
    Point point0 = new Point(int0, int0);
    int int1 = 11;
    Point point1 = new Point(int1, int1);
    boolean boolean0 = arrayList0.add(point0);
    boolean boolean1 = arrayList0.add(point1);
    try {
      listExample0.foo(arrayList0);
    } catch (IllegalStateException e) {
      verifyException("org.example.ListExample", e);
    }
  }
}
```

Listing 3. Example of a test case generated by TARDIS

Reanimator attempts to combine the best of both worlds, by supporting direct generation of test code from interesting data without the need to use reflection.

3. Reanimator

Given a target object representation (as an error-inducing input from the testing tool), Reanimator generates a code snippet for creating the object. The approach was originally developed for the JVM platform; thus, its description contains several JVM-specific features, but it can be adapted for most general-purpose programming languages. We assume a *closed-world model*, i. e., we have full access to all types, functions, etc. Reanimator can be divided into three stages:

1) *descriptor* conversion: descriptors are the internal object representation used in Reanimator;

2) *action sequence* generation: each action is an operation (e. g., function call or array element access) in the target language;

3) *code snippet* generation: code snippet is a valid code sample which creates the target object.

3.1. Descriptor conversion

First, the target object should be converted to a Reanimator descriptor. Descriptors are used to represent the object shape; one may consider them to be trees which capture (nested) object states. The descriptor format for the JVM platform is given in Listing 4; if needed, the format can be extended to support other languages. To convert an object representation to a descriptor, one may follow its shape in a bottom-up fashion, converting object elements to their corresponding descriptors along the way.

3.2. Action sequence generation

At this stage Reanimator tries to create a sequence of valid actions which, if performed, create an object corresponding to the target descriptor. The list of supported actions in the current implementation for the JVM platform is as follows:

- constructor-like calls:
  - no-arg constructor calls;
  - constructor call with arguments;
  - external constructor-like call (static factory methods, etc.);
- (static) method calls;
- (static) field assignments;
- array creations;
- array element writes;
- primitive value creations;
- "unknown" actions.

```
<Descriptor> ::= "ConstantDescriptor"
     "ObjectDescriptor" fields:<ListOfFields>
     "ArrayDescriptor" elements:<ListOfElements>
     "StaticFieldDescriptor" field:<Field>

<ConstantDescriptor> ::= "NullDescriptor"
     "BoolDescriptor" value:Boolean
     "ByteDescriptor" value:Byte
     "ShortDescriptor" value:Short
     "CharDescriptor" value:Char
     "IntDescriptor" value:Int
     "LongDescriptor" value:Long
     "FloatDescriptor" value:Float
     "DoubleDescriptor" value:Double

<Field> ::= name:String klass:Class value:<Descriptor>

<Element> ::= index:Int value:<Descriptor>

<ListOfFields> ::= <Field> <ListOfFields> | <empty>

<ListOfElements> ::= <Element> <ListOfElements> | <empty>
```

Listing 4. JVM descriptor format

```
Input: d — target descriptor
Input: limit — action sequence length limit
Output: calls — generated action sequence
 1: function generate(d, limit)
 2:     if 0 == limit then
 3:         return unknown
 4:     end if
 5:     calls ← []
 6:     if d ∈ ConstantDescriptor then
 7:         calls += PrimitiveValue(d.value)
 8:     else if d ∈ StaticFieldDescriptor then
 9:         value ← generate(d.value, limit − 1)
10:         calls += StaticFieldSetter(d, value)
11:     else if d ∈ ArrayDescriptor then
12:         eType ← d.elementType
13:         length ← generate(d.length, limit − 1)
14:         arr ← NewArray(eType, length)
15:         calls += arr
16:         for (idx, ed) ∈ d.elements do
17:             value ← generate(ed, limit − 1)
18:             calls += ArrayWrite(arr, idx, value)
19:         end for
20:     else if d ∈ ObjectDescriptor then
21:         calls += generateObject(d, limit − 1)
22:     end if
23:     return calls
24: end function
```

Figure 1. Action sequence generation algorithm

Reanimator respects the encapsulation principle and uses only publicly available program actions, e. g., field assignments are allowed only for public fields. A high-level overview of action sequence generation is shown in Figure 1. Generation of action sequences for constant descriptors, array descriptors and static field descriptors is straightforward and self-explanatory. Generation of object descriptors, however, is more complex; let us discuss it in more detail.

As shown in Listing 4, object descriptor is represented as a list of fields with their types and values. If a field is not important in the context of current test generation (i. e., it is irrelevant with reference to error-inducing input), it is not included in the object descriptor.

Each field of the object descriptor imposes new constraints for the object generation. Generation of an object descriptor with n defined fields is strictly more complex than generation of another object descriptor with $m < n$ defined fields. This reduction-like intuition is the basis of the action sequence generation algorithm for object descriptors presented in Figure 2.

The algorithm gradually reduces the descriptor until it finds a constructor-like call, which can directly create the object. A naive approach to doing this is to check all possible valid action sequences, but such brute-force search is very inefficient and may not terminate in a reasonable time in some cases. To overcome this problem, we apply symbolic execution, using it to offer several optimizations to speed up the search of interesting object actions, and impose a hard limit to ensure generation termination.

As the first step, we perform descriptor concretization. The purpose of this is to replace all non-instantiable types in object descriptors with arbitrary instantiable ones. Currently, a type is considered non-instantiable if it is an abstract class or an interface, as they are not constructible directly. Finding compatible types can be done efficiently, as Reanimator operates under the closed-world assumption.

The next step is setter extraction, which is optional, and its main purpose is to speed up the search. We consider method a "setter", if it takes exactly one argument and changes exactly one of the object fields. The main idea is to preprocess available classes and find their setters (if present). Then, if an object descriptor contains fields with available setters, we shortcut and generate corresponding setter call actions, add them to the sequence and reduce these fields from the target descriptor. The details of how we check if a method is a setter are covered in more detail in the following section.

```
Input: d — target object descriptor
Input: limit — action sequence length limit
Output: calls — generated action sequence
 1: function generateObject(d, limit)
 2:     d ← concretize(d)
 3:     ctors ← d.class.ctorLikeCalls
 4:     methods ← d.class.methods
 5:     (d, setters) ← extractSetters(d)
 6:     query ← {d, setters}
 7:     calls ← []
 8:     while query ≠ ∅ do
 9:         if length(calls) > limit then
10:             return unknown
11:         end if
12:         (desc, calls) ← query.poll()
13:         for ctor ∈ ctors do
14:             (nDesc, args) ← execAsCtor(ctor, desc)
15:             if isFinal(nDesc) then
16:                 margs ← genArgs(args, limit)
17:                 calls += CtorCall(ctor, margs)
18:                 return calls
19:             end if
20:         end for
21:         for m ∈ interestingFor(desc, methods) do
22:             (nDesc, args) ← execAsMethod(m, desc)
23:             if nDesc ⩽ desc then
24:                 margs ← genArgs(args, limit)
25:                 nCalls ← calls + MethodCall(m, margs)
26:                 entry ← {nDesc, nCalls}
27:                 query.push(entry)
28:             end if
29:         end for
30:     end while
31: end function
```

Figure 2. Object descriptor processing

After that, the main search procedure is started. At each step, we first check for the termination condition; if we stop the search prematurely, we signal this with a special "unknown" action. Then we check if it is possible to create the given descriptor using any constructor-like call; if that is true, the action sequence generation is complete. We create a constructor call action, add it to the sequence and return the result.

If no constructor-like calls are applicable, the search continues to iterate over all interesting methods, attempting to reduce the descriptor. A method is interesting with reference to given descriptor, if it is public (i. e., accessible from tests) and changes at least one of the descriptor fields. If we were able to successfully reduce the descriptor using one of the methods, we generate arguments for that method with *genArgs* function, which recursively calls *generate* for all argument descriptors. Then we build a method call action, add it to the current sequence and schedule it for processing.

We should also note that *generate* and *generateObject* also save all information about generated descriptors to cache allowing them to support generation of cyclic descriptors.

Functions *execAsCtor* and *execAsMethod* are used to symbolically execute a callable (a constructor or a method) to check if it can be used to create or reduce the target descriptor. If the check is successful, they return the reduced descriptor and arguments descriptors needed to successfully call it. Internally, these functions use SMT solvers [28] to reason about the behavior of callables. Let us describe how these functions work in more detail.

3.2.1. Reducing descriptors using symbolic execution

Applying symbolic execution to reason about callables and their influence on object state is what allows Reanimator to efficiently reduce descriptors during action sequence generation, while respecting encapsula-

tion principle. Our instance of SMT based symbolic execution consists of the following steps:

- encode the callable and the descriptor as SMT formulae;
- perform an SMT query into the SMT solver;
- decode a new descriptor from the resulting SMT model.

To be able to use SMT solver for symbolic execution, we need to define a memory model suitable for representing the program and its variables as SMT formulae. The memory model used in Reanimator is inspired by the work on Kex automatic test generation tool [14].

JVM bytecode has several primitive data types: booleans, integers (short, int, etc.), *floating* point numbers (*float and double*). Each variable of a given type can be represented as an expression of corresponding SMT theory: *booleans* for boolean, *bitvectors* [29] for integers, *floating point numbers* [30] for float and double.

JVM also supports non-primitive data types in the form of reference types (objects and arrays). To represent references in the heap we use a "property-based" memory model [31]: memory is encoded as a collection of SMT arrays [32], each array corresponding to a disjoint partition of heap objects not aliasing object from other partitions. This allows to encode object references as 32-bit bitvector indices into their partition; arrays are represented as continuous chunks, with array reference pointing to its start index. Object fields are represented in a similar fashion, using "property memories": each field is mapped to a separate SMT array, indexed by object references; to access field *x.y* one needs to work with property memory *typeOf(x).y* by index *x*. This allows for precise modeling of heap structures while also reducing the complexity of solving the resulting formulae, as disjoint SMT arrays decrease the search space SMT solver needs to work with.

Runtime type information is encoded in a special "type" property memory: each reference may be used as an index to this property memory to get its type. As we analyze the program as a closed-world system, we can assign a constant to each type and encode subtyping via SMT axioms over *isSubtype* uninterpreted function, which encodes all the available type information.

All type-related operations in the program are expressed through *isSubtype*: casts and *instanceof* checks impose new constraints on the "type" property of the corresponding variable. That, together with the subtyping axioms, gives SMT solver enough information to correctly analyze types.

3.2.2. Encoding the SMT formulae

Methods are encoded as SMT formulae, using the described memory model. Encoding most JVM instructions is straightforward, as they can be directly mapped to corresponding SMT expressions (*iadd* to *bvadd*, *aaload* to *select*, etc.). However, there are some typical problems: loops and method calls cannot be encoded as SMT formula as easily.

To overcome these problems, we use the following processing. All loops are unrolled to a predefined bound, underapproximating the possible method behavior, similarly to how bounded model checking works [33]. This unrolling allows the methods to be converted into an SMT formula. Function calls are inlined if they can be statically resolved, otherwise, they are underapproximated as "noop" operations. For test case generation, our experiments show such underapproximations are good enough for most practical cases.

3.2.3. Performing an SMT query

After the methods are represented as SMT formulae, which symbolically capture their behavior, they are used together with the descriptors to answer different queries with reference to the action sequence generation algorithm. The purpose of these queries is to understand how a given method affects an object's state, i. e., how the final object state is changed in comparison with its initial state after the method invocation.

SMT queries to answer that are constructed as follows. The target descriptor defines the constraints for the final memory state: properties, corresponding to the defined descriptor fields, should be assigned their value in the final state. Method type defines the constraints for the initial memory state:

- constructor constraints, used to check if the target descriptor can be created by this constructor; they require the complete initial memory state to be uninitialized;
- setter constraints, used to check if a method can be used to set one or more target fields to their descriptor values; they require all target field values in the initial memory state to be uninitialized;
- method constraints, used to check how method execution affects the object state; they do not impose any constraints on the initial memory state.

Function *execAsCtor* is checking queries with constructor constraints, function *execAsMethod* — queries with setter and method constraints. Setter constraints are also used in the setter extraction.

We need to analyze method constraints as well as setter constraints, because some fields of an object may not have a direct setter but could be modified by other methods. For example, the size field of a collection cannot be directly manipulated, but it is changed after calls add or clear.

3.2.4. Decoding new descriptors

A successful SMT query returns an SMT model containing the initial memory state, the final memory state, and values of all method variables. To extract the

```
package test;
import java.lang.Throwable;
import java.lang.IllegalStateException;
import org.junit.Test;
import test.ListExample;
import test.ListExample.Point;
import java.util.ArrayList;

public class ListExample_foo {
  public <T> T unknown() {
    throw new IllegalStateException();
  }
  @Test
  public  void test_bb10() throws Throwable {
    ListExample listExample1 = new ListExample();
    ArrayList<ListExample.Point> al =
        new ArrayList<ListExample.Point>(2);
    ListExample.Point point1 = new ListExample.Point(10, 0);
    al.add(point1);
    ListExample.Point point2 = new ListExample.Point(0, 11);
    al.add(point2);
    listExample1.test(al);
  }
}
```

Listing 5. JVM descriptor format

information, we need to decode this model into new, reduced descriptors.

The decoding process consists of evaluating the values of all interesting fields from the initial memory state, to understand which constructor, setter or method constraints were not violated. Fields that have the uninitialized value in the initial memory state are considered successfully reduced and are not included in the resulting descriptor. If the constructor constraints were not violated (the complete initial memory state is uninitialized), the descriptor is considered successfully generated.

3.3. Code snippet generation

After we have created an action sequence for the target object, we need to convert it into a test case code snippet. As the action sequence is a list of callable actions with their arguments, which create the needed object, the transformation is straightforward.

To get a complete code snippet, we need additional information, specifically, resolved type and type parameter information, if we want to correctly use the public API and avoid using reflection. The required type information can be extracted by traversing the action sequence two times: in backward and forward direction. In the prototype implementation, backward traversal uses Java reflection to obtain type parameter types. Forward traversal uses the results of the backward one and resolves the final types for action calls and variables, which are used as follows:

- parameter types can provide type arguments for executable calls;
- explicit type casts are added if an argument variable has type incompatible with the parameter type.

The rest of code snippet generation involves mechanical extraction of sequence actions into code. The extracted Java snippet for our example is shown in Listing 5. The *unknown* function is generated to support the special unknown action, meaning Reanimator can create a test case even if the generation failed, supporting the option of manual developer intervention.

4. Implementation

We implemented the Reanimator approach in an automatic testing tool provided in Kex [14] platform. Kex is a white-box [34] testing tool targeting JVM bytecode, and uses Kfg library[1] to analyze .jar files and construct control flow graphs (CFG). Kfg is also used to perform various bytecode transformations and simplifications, e. g., loop unrolling's.

Kex works as a symbolic execution engine and uses SMT solvers to perform constraint solving. Instead of working directly with Kfg CFGs, Kex uses its own

[1] https://github.com/vorpal-research/kfg

intermediate representation called *predicate state*. Predicate state serves as an inter-layer between Kfg and SMT formulae, allowing it to easily support multiple solvers (Boolector [35], Z3 [36], STP [37]). It is also used to perform additional, SMT-specific transformations on the program code.

5. Evaluation

To evaluate Reanimator, we ran the prototype implementation on a set of open-source projects. First, we selected projects from JUnitContest 2020 [38] benchmark set: fescar-0.1.0, pdfbox-2.0.18 and spoon-7.2.0. Second, we selected several open-source projects from GitHub: authforce-core-13.3.0, exp4j-0.4.9, exposed-0.27.1, imixs-workflow-4.4.6, karg-0.1, koin-2.1.6, kotlinpoet-1.7.0, kfg-0.0.10. We tried to diversify the test projects by picking them from different application domains, such as command line parsing, SQL libraries, code generation utilities, etc.

All tests were run on a test system with 64-bit Arch Linux OS (kernel 5.8.13-arch1), Intel Core i7-4790 CPU @ 3.60GHz, 32GB of RAM and Samsung SSD 950 PRO 512GB storage. We used the following Reanimator configuration:

- Z3 solver for SMT solving;
- all loops are unrolled to bound $k\_l = 2$;
- the length of action sequences set to $limit = 5$.

We have chosen values for parameters k_l and *limit* based on the preliminary experiments with the following reasoning. As we discussed in Section 3, unrolling is used for underapproximating the method behavior, with larger unroll bound making it more precise, but also more difficult to solve. Our experiments showed larger unroll bounds do not significantly improve the generation quality; therefore, we selected k_l to be relatively small. The length of action sequences *limit* is directly affecting both Reanimator performance and the final generation quality. However, while increasing *limit* increases Reanimator generation time, the improvement in generation quality quickly decreases. We selected *limit*, so that the Reanimator can finish code snippet generation in reasonable time. However, one may select other values for k_l and *limit* if needed; it should not have any impact on the applicability of Reanimator, and influence only its performance.

In the evaluation, we address the following research questions (RQ).

RQ1. Can Reanimator be used in automatic test generation tool to create valid test cases? We want to understand what percentage of interesting inputs generated by Kex can be converted to valid code snippets using Reanimator.

RQ2. Can Reanimator be used to generate valid code snippets for random target objects? We want to see if our approach can create code snippets for randomly generated objects and analyze what causes it to fail.

RQ3. How does Reanimator compare to other existing approaches? We want to know how Reanimator performs in comparison with TARDIS automatic test generation tool.

5.1. RQ1: Using Reanimator as a test generator for Kex

We address RQ1 by using our prototype as a "backend" for test generation from the output of Kex. In its default mode, Kex uses Java reflection library to create objects from the symbolic output of an SMT solver (similarly to JBSE). In this part of the evaluation, instead of reflection we used the Reanimator to generate code snippets. Being a white-box tool, Kex tries to cover each basic block of each method by generating interesting input data. The achieved coverage is measured as instruction coverage.

The results of the evaluation are presented in Table 1. The first column contains the name of the tested project. The second column shows project line coverage, as a relative measure of the project complexity. The third column contains the percentage of successfully generated code snippets for all interesting inputs found by Kex (**AG**). The fourth column shows the average depth of descriptors (depth of a descriptor is the maximal nesting level of its elements) (**ADD**). As we found during evaluation, many descriptors generated by Kex are simple constant descriptors or empty object descriptors, for that reason we decided to additionally measure non-trivial descriptor parameters. The fifth column shows

Table 1

Results of using Reanimator as a part of Kex

Project	Coverage, %	AG, %	ADD	NTG, %	NTDD
authforce	13.9	71.8	1.5	48.5	2.5
exp4j	27.5	79.7	1.6	4.7	2.8
exposed	29.0	70.0	1.3	45.3	2.3
fescar	21.7	86.1	1.4	47.3	2.6
imixs	25.1	89.3	1.5	70.1	2.3
karg	15.8	66.0	1.5	38.6	2.3
kfg	21.7	58.1	1.4	42.0	2.4
koin	33.2	70.4	1.9	59.3	3.0
kotlinpoet	28.5	76.3	1.6	65.8	2.6
pdfbox	9.8	83.4	1.6	75.1	2.3
spoon	5.9	83.8	1.3	40.6	2.3
average	21.1	75.9	1.5	48.8	2.5

the percentage of successfully generated code snippets for non-trivial descriptors (**NTG**). The last column shows the average depth of non-trivial descriptors (**NTDD**).

The results show that on average Reanimator can successfully generate code snippets in 48.8 % of non-trivial cases and in 75.9 % of all cases. The average depth of non-trivial descriptors over all projects is 2.5. Manual analysis of the results has showed the main reasons for Reanimator failures are code complexity (w.r.t. our underapproximation), "impossible" objects (objects that cannot be created using the public API) and higher-order functions.

The results also show that Reanimator performed poorly on exp4j project. That happened because exp4j project contains a lot of package-private and anonymous classes which Reanimator cannot generate.

Based on that, we believe the proposed approach can be used in automatic test generation tools to create code snippets from input data.

5.2. RQ2: Generating code snippets for creating random objects

To address RQ2 we evaluated our prototype on generating random objects for classes from the test projects. We used the following evaluation technique. For each test project we generated a set of random objects using easy-random-4.2.0 library[1]. From this set we have selected a subset of "valid" objects, i. e. objects with public visibility and at least one public constructor-like callable. We then used our prototype to try and generate code snippets for valid objects, until we got 1000 successful generations for each project. To estimate the complexity of random objects we also measured the average depths of generated descriptors.

To check the correctness of the generation, we need to have an oracle: a way to ensure the generated object is structurally equal to the target one. We cannot use equals implementations because they are not required to test structural equality. To sidestep that, we implemented an external structural equality test.

- For a target object *a*, we convert it into a descriptor *a_d*.
- We generate an action sequence for *a_d* and use it to create a generated object *b*.
- The generated object *b* is converted into a descriptor *b_d*.
- We check the descriptors for equality: $a\_d = b\_d$.

By construction, if the generated object is structurally equal to the target object, their corresponding descriptors should be equal.

The results of random object generation are presented in Table 2. The first column shows the name of the project. The second and third columns show the sucess rate for generation of valid objects (**AG**) and average depth of the descriptors for valid objects (**ADD**). During the experiments we noticed the Reanimator cannot successfully generate unordered collections (sets and maps) in most cases, as their complex internal structure presents problems for the symbolic execution. For that reason, we decided to also measure success rate (column **VG**) and average depth (column **VDD**) for *viable descriptors*, i. e. descriptors without complex collections.

Table 2

Results of generating random objects

Project	AG, %	ADD	VG, %	VDD
authforce	32.4	3.2	40.4	3.0
exp4j	41.1	2.6	47.9	2.3
exposed	42.4	3.3	49.9	2.9
fescar	46.6	3.1	63.5	2.1
imixs	67.4	3.5	80.4	2.8
karg	75.5	3.4	75.5	3.4
kfg	66.3	3.8	95.3	3.1
koin	38.7	3.8	50.1	3.1
kotlinpoet	29.7	2.5	63.7	1.5
pdfbox	28.5	5.2	38.5	4.0
spoon	37.0	3.6	44.1	3.3
average	45.9	3.5	59.0	2.9

We can see that the average success rate is 45.9 % for valid objects and 59 % for viable descriptors; overall, the success rate is very dependent on the target project. For example, pdfbox project has a lot of classes that operate with complex PDF document structures. Reanimator is not always able to resolve these complex structures, therefore, the success rate is relatively low. On the other hand, project kfg has a lot of data classes: immutable classes with public "setter" constructors, allowing to initialize all object fields at once, thus, the success rate for kfg is high.

The amount of generated objects does not allow us to manually inspect each Reanimator failure and the complexity of such task does not allow performing it automatically. Therefore, we inspected a subset of 120 Reanimator failures to analyze their causes. 35 % of the failures were caused by impossible objects generated by easy-random. The main reason for other failures is the complexity of the generated objects (58 %). Among other reasons for the failure are the use of higher-order functions (4 %) and "builder" pattern (3 %), which are currently not supported by Reanimator.

[1] https://github.com/j-easy/easy-random

The last group of failures can be explained by the prototype implementation limitations. The approach can be easily extended to support the builder pattern. As for the higher order functions, it should be possible to search for existing functions with the required signature or even generate them on-the-fly. Another way to improve Reanimator success rate is to improve the symbolic execution by adding support of unordered collections or improving its underapproximation. We consider these tasks as future work.

5.3. RQ3: Comparing Reanimator to TARDIS approach

To address RQ3 we have integrated Reanimator to the TARDIS tool and compared it to the default approach used in TARDIS on the JUnitContest 2021 benchmark set. JUnitContest 2021 used the time budgets of 60 and 120 s, but we decided to also run TARDIS on the increased time budgets as the authors of TARDIS suggested that 120 s is too low. We have also evaluated Kex with Reanimator on the same benchmark set.

The results of the evaluation are presented in Table 3. The first column shows the name of the project and the approach used for code snippet generation. Other columns show average line coverage achieved by the tools on the benchmark classes. We can see that TARDIS with Reanimator performed similarly as TARDIS with Evosuite. Thus, we can say that Reanimator is an applicable approach for automatic test case generation tools.

Table 3

Results on JUnitContest 2021 benchmark, %

Tool	Time budget, s			
	60	120	300	600
TARDIS + Evosuite	14.0	15.7	18.5	19.6
TARDIS + Reanimator	13.9	16.0	17.8	19.3
Kex + Reanimator	24.6	25.3	25.4	27.6

Conclusion and future work

In this paper we presented an approach called Reanimator for generating valid code snippets to create a given target object, using only its publicly available API (i. e., respecting the "encapsulation principle"). This approach is applicable in automatic test generation tools to create correct and easy to maintain test cases. It is based on an original search algorithm, augmented with symbolic execution, which attempts to find an action sequence that creates the objects needed to exercise interesting executions found by the testing tool. The approach targets the JVM platform and considers several JVM-specific features but is general enough and can be extended to other programming languages.

The proposed approach was implemented as a module in an automatic testing tool provided in Kex platform and evaluated on a set of open-source projects. The results show it can successfully and efficiently generate 64.4 % of target objects on average, with the best result being 99.9 %, in a reasonable time. Based on the results, we can say Reanimator is applicable for automatic test case snippet generation.

In the future, we plan to explore the following directions:

- perform a more thorough investigation of Reanimator failures to understand how we can increase its success rate;
- improve support of complex types such as (unordered) collections and/or try using other symbolic execution engines;
- support generation of higher order functions;
- integrate our approach with unit-testing engines to generate additional assertions.

References

1. **Ayewah N., Pugh W., Hovemeyer D.** et al. Using static analysis to find bugs, *IEEE software,* 2008, vol. 25, no. 5, pp. 22—29. DOI: 10.1109/MS.2008.130.
2. **Calcagno C., Distefano D., Dubreil J.** et al. Moving fast with software verification, *NASA Formal Methods Symposium,* Springer, 2015, LNPSE, vol. 9058, pp. 3—11. DOI: 10.1007/978-3-319-17524-9_1.
3. **Sharma R. M.** Quantitative analysis of automation and manual testing, *International Journal of Engineering and Innovative Technology*, 2014, vol. 4, no. 1, pp. 252—257.
4. **Klees G., Ruef A., Cooper B.** et al. Evaluating fuzz testing, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123—2138. DOI: 10.1145/3243734.3243804.
5. **Braione P., Denaro G., Pezze M.** JBSE: A symbolic executor for Java programs with complex heap inputs, *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 1018—1022. DOI: 10.1145/2950290.2983940.
6. **Fraser G., Arcuri A.** EvoSuite: automatic test suite generation for object-oriented software, *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011, pp. 416—419. DOI: 10.1145/2025113.2025179.
7. **Csallner C., Smaragdakis Y.** JCrasher: an automatic robustness tester for Java, *Software: Practice and Experience*, 2004, vol. 34, no. 11, pp. 1025—1050. DOI: 10.1002/spe.602.
8. **Fraser G., Arcuri A.** Evolutionary generation of whole test suites, *2011 11th International Conference on Quality Software*, IEEE, 2011, pp. 31—40. DOI: 10.1109/QSIC.2011.19.
9. **Xie T., Marinov D., Schulte W., Notkin D.** Symstra: A framework for generating object-oriented unit tests using symbolic execution, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems,* Springer, 2005, pp. 365—381. DOI: 10.1007/978-3-540-31980-1_24.
10. **Braione P., Denaro G., Mattavelli A., Pezze M.** SUSHI: A test generator for programs with complex structured inputs, *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, IEEE, 2018, pp. 21—24. DOI: 10.1145/3183440.3183472.
11. **Demers F.-N., Malenfant J.** Reflection in logic, functional and object oriented programming: a short comparative study, *Proceedings of the IJCAI,* 1995, vol. 95, pp. 29—38.

12. **Orso A., Kennedy B.** Selective capture and replay of program executions, *ACM SIGSOFT Software Engineering Notes*, 2005, vol. 30, no. 4, pp. 1—7. DOI: 10.1145/1082983.1083251.

13. **Krikava F., Vitek J.** Tests from traces: automated unit test extraction for R, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 232—241. DOI: 10.1145/3213846.3213863.

14. **Abdullin A. M., Itsykson V. M.** Kex: A platform for analysis of JVM programs, *Information and control systems,* 2022, no. 1 (116), pp. 30—43. DOI: 10.31799/1684-8853-2022-1-30-43.

15. **Braione P., Denaro G.** SUSHI and TARDIS at the SBST2019 Tool Competition, *IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST),* IEEE, 2019, pp. 25—28. DOI: 10.1109/SBST.2019.00016.

16. **Pouria D.** Well-informed Test Case Generation and Crash Reproduction, *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST),* IEEE, 2020, pp. 424—426. DOI: 10.1109/ICST46399.2020.00054.

17. **Gulwani S.** Dimensions in program synthesis, *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, 2010, pp. 13—24. DOI: 0.1145/1836089.1836091.

18. **Koushik S.** Concolic testing, *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 571—572. DOI: 10.1145/1321631.1321746.

19. **Elbaum S., Chin H. N., Dwyer M. B., Dokulil J.** Carving differential unit test cases from system test cases, *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering,* 2006, pp. 253—264. DOI: 10.1145/1181775.1181806.

20. **Burger M., Zeller A.** Minimizing reproduction of software failures, *Proceedings of the 2011 International Symposium on Software Testing and Analysis,* 2011, pp. 221—231. DOI: 10.1145/2001420.2001447.

21. **Jin W., Orso A.** Automated support for reproducing and debugging field failures, *ACM Transactions on Software Engineering and Methodology (TOSEM),* 2015, vol. 24, no. 4, pp. 26:1—26:35. DOI: 10.1145/2774218.

22. **Thanhofer-Pilisch J., Rabiser R., Krismayer T.** et al. An event-based capture-and-compare approach to support the evolution of systems of systems, *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, 2017, pp. 261—270. DOI: 10.1145/3093742.3093909.

23. **Zamfir C., Candea G.** Execution synthesis: a technique for automated software debugging, *Proceedings of the 5th European Conference on Computer Systems,* 2010, pp. 321—334. DOI: 10.1145/1755913.1755946.

24. **Rößler J., Zeller A., Fraser G.** et al. Reconstructing core dumps, *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, IEEE, 2013, pp. 114—123. DOI: 10.1109/ICST.2013.18.

25. **Yu Tingting, Zaman Tarannum S., Wang Chao.** DESCRY: reproducing system-level concurrency failures, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 694—704. DOI: DOI:10.1145/3106237.3106266.

26. **Leitner A., Pretschner A., Mori S.** et al. On the effectiveness of test extraction without overhead, *2009 International Conference on Software Testing Verification and Validation,* IEEE, 2009, pp. 416—425. DOI: DOI:10.1109/ICST.2009.30.

27. **Kifetew F. M., Jin W., Tiella R.** et al. Reproducing field failures for programs with complex grammar-based input, *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, IEEE, 2014, pp. 163—172. DOI: 10.1109/ICST.2014.29.

28. **Barrett C., Tinelli C.** Satisfiability modulo theories, *Handbook of Model Checking,* Springer, 2018, pp. 305—343. DOI: 10.1007/978-3-319-10575-8_11.

29. **Jha Susmit, Limaye Rhishikesh, Seshia Sanjit A.** Beaver: Engineering an efficient SMT solver for bit-vector arithmetic, *International Conference on Computer Aided Verification,* Springer, 2009, pp. 668—674.

30. **Rummer P., Wahl T.** An SMT-LIB theory of binary floating-point arithmetic, *International Workshop on Satisfiability Modulo Theories (SMT),* 2010, pp. 151.

31. **Kapus T., Cadar C.** A segmented memory model for symbolic execution, *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 774—784. DOI: 10.1145/3338906.3338936.

32. **Stump A., Barrett C. W., Dill D. L., Levitt J.** A decision procedure for an extensional theory of arrays, *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science,* IEEE, 2001, pp. 29—37.

33. **Clarke E., Biere A., Raimi R., Zhu Y.** Bounded model checking using satisfiability solving, *Formal Methods in System Design*, 2001, vol. 19, no. 1, pp. 7—34. DOI: 10.1023/A:1011276507260.

34. **Godefroid P., Levin M. Y., Molnar D. A.** Automated Whitebox Fuzz Testing, *NDSS,* 2008, vol. 8, pp. 151—166.

35. **Brummayer R., Biere A.** Boolector: An efficient SMT solver for bit-vectors and arrays, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* 2009., Springer, 2009, LNTCS, vol. 5505, pp. 174—177. DOI: 10.1007/978-3-642-00768-2_16.

36. **De Moura L., Bjørner N.** Z3: An efficient SMT solver, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems 2008,* Springer-Verlag, 2008, LNTCS, vol. 4963, pp. 337—340. DOI: 10.1007/978-3-540-78800-3_24.

37. **Vijay G., Trevor H.** STP constraint solver: Simple theorem prover SMT solver, available at: https://stp.github.io/

38. **Devroey X., Panichella S., Gambi A.** Java Unit Testing Tool Competition: Eighth Round, *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 545—548. DOI: 10.1145/3387940.3392265.